CUDA Teaching Center

Manual de prácticas de programacón paralela y multicore

Liliana Ibeth Barbosa-Santillán
Manuel Alejandro Urbano-Alcalá

8 de julio de 2014

No. Registro: 03-2014-080110580800-01

Índice general

Índice de figuras

Capítulo 1

Introducción

El manual de prácticas de programación paralela y multicore presenta quince prácticas que se llevarán a cabo a lo largo de un semestre. El objetivo principal es introducir al alumno en el mundo de programación paralela y multicore. Cada una de las prácticas tiene una breve descripción, uno o varios objetivos, las actividades de desarrollo y el código fuente.

El lenguaje de programación utlizado es CUDA. La comunicación puede ser sincronizada o asincronizada; permite escribir código fuente en el host y el device. Al utilizar el lenguaje CUDA los programas se aceleran de una a n veces y algunas veces el tiempo de ejecución es muy cercano al tiempo real. Iniciaremos con prácticas de complejidad baja hasta llegar a complejidad media. Los alumnos serán capaces de poder utilizar ésta guía como un medio de autoaprendizaje ayudándoles en tiempo de diseño, compilación y ejecución.

Capítulo 2

Prácticas

Comenzaremos con lo más simple que es el paralelismo estático. Iniciaremos nuestro primer programa que permite enviarnos un saludo desde el host y el device. Una vez que tengamos comunicación tanto con el host como con el device nos introduciremos al manejo de vectores, éste será nuestro comienzo con la programación paralela. Ya visto el concepto de la programación paralela y el cómo trabaja pasaremos a algo más complejo como lo es la multiplicación de matrices. Como ya es de imaginarse, la programación en paralelo y el manejo de matrices van de la mano, de esta manera es como vemos la utilidad de ello. Para ver el verdadero potencial de CUDA tenemos que hacer uso de matrices muy grandes.

Así después de un manejo más de vectores veremos un algoritmo de optimización como lo es el algoritmo de optimización Jacobi. Posteriormente, haremos implementación de este manejo de matrices para su uso, como lo es en el algoritmo de ordenamiento Quicksort y con más "magia" de CUDA veremos un Quicksort avanzado más adelante. CUDA trabaja con los GPU y son ellos los que hacen la "magia" del paralelismo, lo que nos llevará a pensar más en la distribución de nuestros datos en estas GPU y sus complementos; así veremos el algoritmo para valoración de opciones Monte Carlo con múltiples GPU.

Veremos más usos de la programación paralela explotando las bondades

de la programación estructurada de los lenguajes C y C++ con programas como un generador de números casi aleatorios, manejo de muchas funciones a la vez por medio de punteros a funciones y el algoritmo Merge sort.

Adentrándonos a las más recientes capacidades de los dispositivos CUDA veremos un poco de paralelismo dinámico, el cual consiste en el hecho de que ya no son necesarias varias llamadas al dispositivo desde el CPU, sino que, el mismo GPU puede lanzar procesos y bucles anidados por sí mismo de manera dinámica. Con ayuda de ello realizaremos un árbol de cuadros y veremos como va construyendo más y más hijos para éstos con ayuda de la funcionalidad del paralelismo dinámico. Éste es el mejor ejemplo ya que el algoritmo consta de crear nuevos nodos (cuadros) y el paralelismo dinámico consiste en lanzar automáticamente estos nodos nuevos. Como ya habíamos mencionado con ayuda de ésta capacidad optimizaremos el algoritmo Quicksort, que consiste de igual manera en crear subprocesos, tantos como sea necesario, y éstos los podemos manejar con la capacidad de lanzarlos dinámicamente.

Después de lo anterior ya habrá bases necesarias para aprovechar el cómputo paralelo en otros diversos usos como lo es el manejo de imágenes. Hay que recordar que las imágenes son matrices con valores para cada pixel. Así podremos ver la convolución de una imagen, con los conocimientos necesarios sobre este método y las modificaciones con la programación en paralelo. Podremos ver la disparidad de dos imágenes y para finalizar, una sumatoria de todos los conocimientos aplicados en la simulación de una ola de océano, en la cual observaremos diversos factores afectando la imagen, como el aire, dirección y fuerza, apreciar la sombra de las olas y el choque del agua. También veremos que renderizar es más rápido gracias al procesamiento paralelo.

Con todo lo anterior habremos visto las grandes bondades de la programación en paralelo y cómo con el uso de ésta, combinado con los conocimientos en programación estructurada se pueden hacer mejoras bastante importantes y en tiempo real, lo cual brinda demasiados beneficios ya que tendremos resultados inmediatos y mayores beneficios a la hora de análisis y toma de decisiones.

A continuación podemos ver en la Figura 2.1 la taxonomía de los programas mencionados.

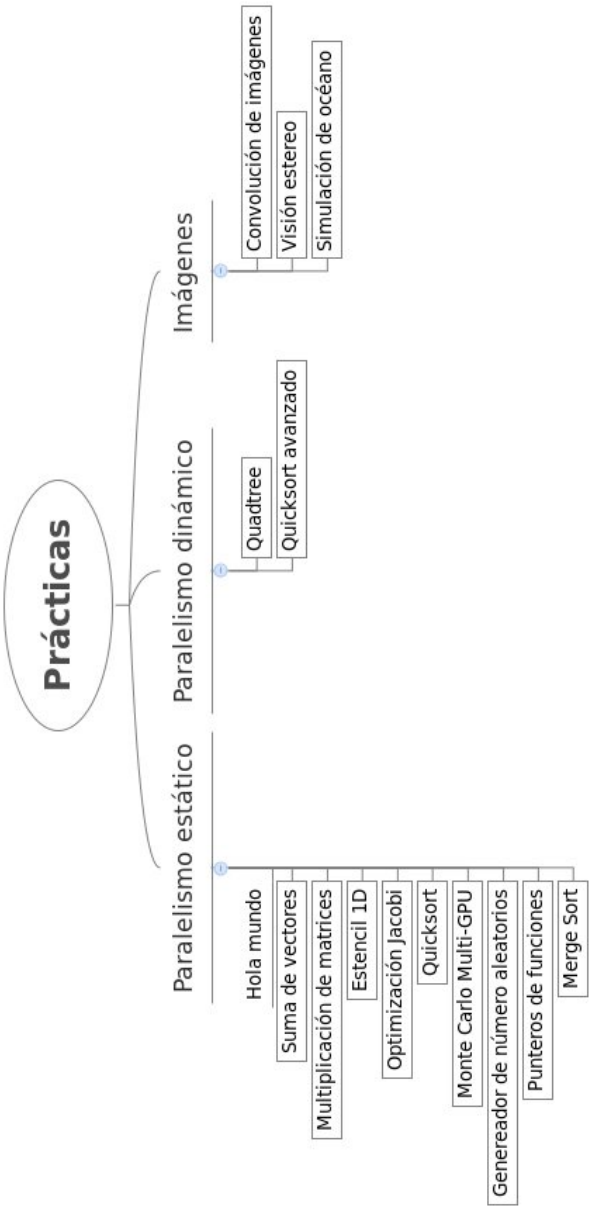

Figura 2.1: Taxonomía del curso.

2.1. Paralelismo estático

Es el comienzo y lo más básico en el paralelismo y consiste en la posibilidad de manejar todos los datos de un arreglo al mismo momento, en un solo ciclo de reloj. De la misma manera, manejando las funciones con punteros, podemos realizar varias funciones a la vez igualmente que con los datos.

2.1.1. Hola mundo.

Comenzaremos comprobando la comunicación entre el host y el dispositivo.

Descripción

Práctica sencilla para tener una primera interacción con el entorno de CUDA.

En la sección de anexos encontrarás el código completo.

Objetivos:

Conocer el ambiente de programación en el que se trabajará. Escribir unas primeras líneas de código. Aprender a declarar una función que se ejecute en el dispositivo. Compilar y ejecutar un primer programa.

Desarrollo

1. Conectar a una terminal remota con la dirección ip, el usuario y el password que el asesor le brindará:

   a) ssh usuario@ip

b) password: ******

2. Abrir un editor de consola (pico, nano, vi).

3. El programa deberá mandar un mensaje de saludo desde el host y otro desde el dispositivo.

4. Declarar una función que mande un mensaje de saludo usando el modificador __global__ para que pueda llamarse desde el host y ejecutarse en el dispositivo.

```
__global__ void device_greetings(void){
    cuPrintf("Hello, world from the device!\n");
}
```

   a) Para escribir en pantalla desde el dispositivo debe utilizarse la instrucción cuPrintf(), incluir "util/cuPrintf.cu".

```
#include "util/cuPrintf.cu"
#include <stdio.h>
```

5. En main:

   a) Mandar un mensaje de saludo.

```
printf("Hello, world from the host!\n");
```

   b) La función cuPrintf debe ser incializada mediante cudaPrintfInit().

```
cudaPrintfInit();
```

   c) Lanzar un kernel con un solo hilo mediante la instrucción funcion_saludo<<<1,1>>>.

```
device_greetings<<<1,1>>>();
```

   d) Para desplegar el mensaje del dispositivo usar cudaPrintfDisplay(), cuando se termina de usar cuPrintf debe agregarse la siguiente instrucción cudaPrintfEnd().

```
cudaPrintfDisplay();
cudaPrintfEnd();
```

6. Guardar el archivo en */home/usuario/CursoCuda/*.

7. Compilar el programa con *nvcc helloworld.cu -o helloworld.*

8. Ejecutar el programa Hello World mediante *./helloworld*.

Actividades

Modificar para que ejecute n hilos.

Código fuente

- Hello.cu.

Serán necesarias las siguientes librerías, descargar el archivo .zip y descomprimir donde se encuentra el archivo "Hello.cu", tendrá que quedar ahí la carpeta "util" con las dos librerías necesarias dentro: util.zip.

Compilar:
nvcc Hello.cu -o Hello

2.1.2. Suma de vectores.

Se realiza un sencilla suma de vectores, una suma lineal entre cada valor.

Descripción

Práctica en la cual se realizará una suma de vectores en el entorno CUDA C.

En la sección de anexos encontrarás el código completo.

Objetivos:

Suma de vectores: entender el uso de bloques y su codificación en CUDA C.

Desarrollo

En la Figura 2.2 se muestra el procedimiento para la suma de vectores.

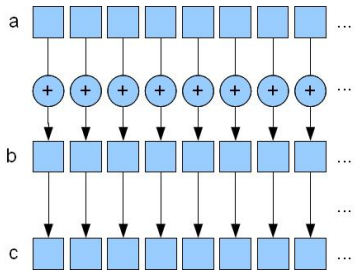

Figura 2.2: Procedimiento para la suma de vectores.

- Encabezados

```
#include <cuda.h>
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

#define length 64
```

- Función main

1. Asignar tres arreglos en device utilizando cudaMalloc() dos arreglos, dev_a y dev_b, para los vectores de entrada y un arreglo, dev_c, para el vector de resultado.

```
float a[length],b[length],c[length];
float *dev_a,*dev_b,*dev_c;

cudaMalloc((void**)&dev_a,length*sizeof(float));
cudaMalloc((void**)&dev_b,length*sizeof(float));
cudaMalloc((void**)&dev_c,length*sizeof(float));
```

2. Asignar valores a los arreglos a y b en la CPU.

```
time_t seconds;
time(&seconds);
srand((unsigned int) seconds);
for(int i=0; i<length; i++){
    a[i]=(float)rand()/(float)RAND_MAX;
    b[i]=(float)rand()/(float)RAND_MAX;
}
```

3. Al término del procedimiento se libera memoria con cudaFree().

```
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

4. Copiar los vectores de entrada al device utilizando cudaMemcpy(), utilizar el parámetro cudaMemcpyHostToDevice. Para copiar el vector resultado del device al host, utilizar el parámetro cudaMemcpyDeviceToHost.

   - Copiar los vectores de entrada:

     ```
     cudaMemcpy(dev_a,a,length*sizeof(float),
                cudaMemcpyHostToDevice);
     cudaMemcpy(dev_b,b,length*sizeof(float),
                cudaMemcpyHostToDevice);
     ```

   - La función add<<<,>>> se ejecuta desde el host en la función main() utilizando la sintaxis con pico-paréntesis triples.

     ```
     add<<<length,1>>>(dev_a,dev_b,dev_c);
     ```

   - Copiar los vectores de salida:

     ```
     cudaMemcpy(c,dev_c,length*sizeof(float),
                cudaMemcpyDeviceToHost);
     ```

   - Mostrar los valores de salida:

     ```
     for(int i=0; i<length; i++){
         printf("%.3f + %.3f = %.3f \n",a[i],b[i],c[i]);
     }
     ```

- Función device

  1. Agregar el calificador __global__ al nombre de la función.

     ```
     __global__ void add(float *a, float *b, float *c){
     }
     ```

  2. Llamada de la función device add desde la función main del host.

     - add<<<dimGrid,dimBlk>>>(dev_a,dev_b,dev_c); donde dimGrid se refiere al número de bloques y dimBlk se refiere al número de hilos por bloque.

       ```
       add<<<length,1>>>(dev_a,dev_b,dev_c);
       ```

- Ejemplo: add<<<N,1>>>, donde N es el número de bloques paralelos.
  - Variable empotrada de CUDA: blockIdx.x

```
int tid=blockIdx.x;
```

3. Suma de los vectores en el número de bloques.

```
if(tid < length)
    c[tid]=a[tid]+b[tid];
```

Actividades

1. Identificar los puntos explicados anteriormente en el código.
2. Cambiar el tamaño de los vectores a sumar.
3. En el código proporcionado se utilizan N número de bloques con un hilo en cada bloque. Realizar la suma de vectores variando el número de bloques y el número de hilos.

Código fuente

- vectAdd.cu

Compilar:
nvcc vectAdd.cu -o vectAdd

2.1.3. Multiplicación de matrices.

La multiplicacion de matrices implica multiplicaciones cruzadas y sumas, por lo que aquí veremos un par de trucos.

Descripción

Multiplicación de matrices con CUDA C.

En la sección de anexos encontrarás el código completo.

Objetivos:

Multiplicación de matrices: entender el uso de hilos y memoria global y su codificación en CUDA C.

Desarrollo

Multiplicación de matrices cuadradas utilizando memoria global $P = MxN$ de tamaño width x width. Como podemos ver en la Figura 2.3:

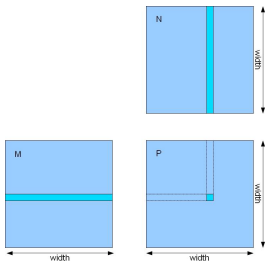

Figura 2.3: Multiplicación de matrices.

La multiplicación de matrices se realiza sin tiling.

- Cada hilo calcula un elemento de la matriz resultado P.
- Las matrices de entrada M y N se cargan de la memoria global un número de veces igual al ancho de la matriz (width).
- Multiplicación de matrices con memoria global en CUDA C.
- Función main:
  - Asignar memoria en device para las matrices M,N,P utilizando cudaMalloc(): dos arreglos, dev_M y dev_N, para los vectores de entrada y un arreglo, dev_P, para el vector resultado.
  - Al término del procedimiento se libera memoria con cudaFree().
  - Copiar las matrices de entrada al device utilizando cudaMemcpy(), utilizar el parámetro cudaMemcpyHostToDevice. Para copiar el vector resultado del device al host, utilizar el parámetro cudaMemcpyDeviceToHost.
  - La función matMultKernel<<<,>>> se ejecuta desde el host en la función main() utilizando la sintaxis con pico-paréntesis triples.
  - Función device.
  - Agregar el calificador __global__ al nombre de la función.
  - Llamada de la función device matMultKernel desde la función main del host.
  - matMultKernel<<<dimGrid,dimBlk>>>(dev_a,dev_b,dev_c); donde dimGrid se refiere al número de bloques y dimBlk se refiere al número de hilos por bloque.
  - Definir la configuración del kernel:
    - dim3 dimGrid(1,1); se declara el grid de una dimensión.
    - dim3 dimBlk(width,width); se declara el bloque de dos dimensiones del tamaño de las matrices.
    - Un bloque de hilos calcula la matriz resultado P.
    - Cada hilo:
      - Carga un renglón de la matriz M.

- ◇ Carga una columna de la matriz N.
- ◇ Multiplica y suma cada par de elementos de las matrices M y N.
- ◇ El tamaño de las matrices está limitado al número máximo de hilos permitidos en cada bloque (utilizar deviceQuery para saber el límite).

Actividades

1. Identificar los puntos explicados anteriormente en el código.
2. Cambiar las dimensiones del grid dividiendo el tamaño de la matriz en un número determinado de mosaicos (tamaño de matriz / tamaño de mosaico).
3. Cada bloque calculará el resultado parcial de los mosaicos, para el posterior cálculo total de los elementos de la matriz resultado P.

Código fuente

- findcudalib.mk
- Makefile
- matrixMul.cu

Compilar:
make

2.1.4. Estencil 1D.

Un estencil (vector) de cierta dimensión que se le agregará un radio definido a cada extremo.

Descripción

Práctica en la que se manipulará un arreglo.

En la sección de anexos encontrarás el código completo.

Objetivos:

Aplicar una máscara de una dimensión a un arreglo de elementos de una dimensión. Los elementos de salida son la suma de los elementos de entrada dentro de un radio. Si el radio es 3, entonces cada elemento de salida es la suma de 7 elementos de entrada como se muestra en la figura 2.4:

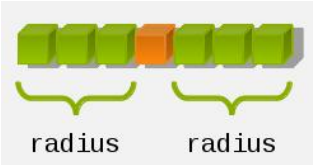

Figura 2.4: Radio aplicado a los elementos.

Desarrollo

Implementar dentro de un bloque:

- Cada hilo procesa un elemento de salida blockDim.x elementos por bloque.
- Los elementos de entrada se leen varias veces.
- Con un radio de 3, cada elemento de entrada es leído siete veces.
- Compartir datos entre hilos.
- Terminología: dentro de un bloque, los hilos comparten datos a través de memoria compartida.
- Extremadamente rápido de memoria en el chip, gestionado por el usuario.
- Declarar utilizando __ shared__, asignado por bloque.

```
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```

- Los datos no son visibles para los hilos en otros bloques.
- Implementación con memoria compartida.
- Datos de caché en la memoria compartida.
- Leer (blockDim.x + 2 * radio) los elementos de entrada de la memoria global a la memoria compartida.

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS){
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
```

- Calcular elementos de salida blockDim.x.

```
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

- Escribir los elementos de salida blockDim.x a la memoria global.

  ```
  out[gindex-RADIUS] = result;
  ```

- Cada bloque necesita un halo de elementos de radio en cada límite como se muestra en la Figura 2.5:

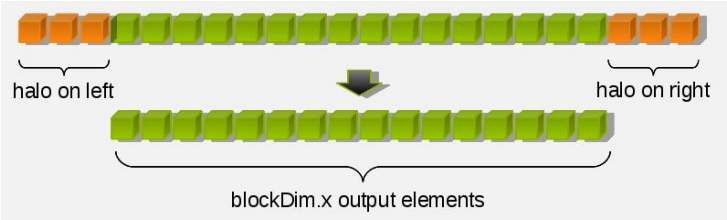

Figura 2.5: Bloque con halo en cada límite.

Stencil Kernel:

```
__global__ void stencil_1d(int *in, int *out){
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + (blockIdx.x * blockDim.x) +
                 RADIUS;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS){
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
```

```
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex-RADIUS] = result;
}
```

¡Data Race!
En el ejemplo de la plantilla no va a funcionar. Suponga que el hilo 15 lee el halo antes que el hilo 0.

```
// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS){
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
__syncthreads();
...
```

void __syncthreads()

- Sincroniza todos los hilos dentro de un bloque.
- Se utiliza para evitar los peligros RAW / WAR / WAW.
- Todos los hilos deben llegar a la barrera.
- En el código condicional, la condición debe ser uniforme en todo el bloque.

Actividades

1. Primero compilar y ejecutar el código sin utilizar cualquier memoria compartida. Asegúrese de que los estudiantes ejecuten el profiler y vean el tiempo que le toma al código.

2. Opcionalmente, puede mostrarles cómo utilizar el generador de perfiles para determinar cuál es el problema.

3. Haga que los estudiantes modifiquen el código para hacer uso de la memoria compartida. En este punto no mencionar la funcion __syncthreads() . Pídales que ejecuten el código varias veces y observen que obtienen errores semi-aleatorios en la salida.

4. Introducir el concepto __ syncthreads () y haga que los estudiantes traten y añadan en su código.

5. Una vez que se recibe la respuesta correcta, correr el código de nuevo para ver la aceleración alcanzada.

Código fuente

- 1DStencil.cu

Compilar:
nvcc 1DStencil.cu -o 1DStencil

2.1.5. Optimización Jacobi.

El método Jacobi consiste en la resolución de sistemas de ecuaciones mediante iteraciones de punto fijo.

Descripción

Paralelización del método de optimzación Jacobi.

En la sección de anexos encontrarás el código completo.

Objetivos:

En este ejercicio hay que implementar las partes faltantes del código. Finaliza cuando compiles y ejecutes el programa y se obtenga la salida "¡Correcto!"

Desarrollo

Planteamiento del problema: Método Jacobi.

- Declaración:
  - Parámetros:

    `Ni=512,Nj=512,Niterations=10000`

  - Dos matrices:

    `float t[Ni,Nj], t_prev[Ni,Nj]`

- Inicialización

```
for j = 1,Nj
  for i = 1,Ni
     t_prev(i,j) = 0.0
```

- Condiciones de límite

```
for i = 0,Ni-1{
    x = float(i)*h
    t_prev(i,0) = x*x
    t_prev(i,Nj-1) = x*x + 1.0
}

for j = 0,Nj-2{
    y = float(j)*h
    t_prev(0,j) = y*y
    t_prev(Ni-1,j) = 1.0 + y*y
}
```

- Ciclo de interacción

```
for  k = 1,Niterations{
  for  j = 1,Nj-2
    for  i = 1,Ni-2
      t(i,j) = 0.25 * (t_prev(i-1,j) + t_prev(i+1,j) +
       t_prev(i,j-1) +
       t_prev(i,j+1) - 4*h*h)

  for  j = 1,Nj-2
    for  i = 1,Ni-2
      t_prev(i,j) = t(i,j);
}

t(i,j) = 0.25 * (t_prev(i-1,j) +
t_prev(i+1,j) +
t_prev(i,j-1) +
t_prev(i,j+1) - 4*h*h)
```

Comencemos la optimización.

- ¿Es un límite de instrucción o límite de memoria?
- Vamos a empezar con un kernel sencillo – Kernel 1.
  - Transferencia normal a kernel de GPU.
  - Utilice la memoria del dispositivo.
- Vamos a mejorar aún más el uso de memoria compartida - Kernel 2.
- Otros consejos.
  - ¿Ha intentado la alineación a 128 byte?
  - ¿Ha intentado deshabilitar la Cache L1?
  - ¿Necesita cudaThreadSynchronize() dentro del bucle?
- ¿Ver si el perfilador nos da un consejo?

Compilar:
Argumentos del ejecutable:
−Ni=XX : Especifica XX número de elementos en Y dirección (opcional, default=512).
−Nj=XX : Especifica XX número de elementos en X dirección (opcional, default=512).
−iterations=XX: Especifica XX número de iteraciones del Kernel que será llamado (opcional, default=10000).
−kernel=XX : XX puede tomar los valores 1 o 2 (opcional, default=2).
−kernel=1 :: lanzará kernel desoptimizado de la GPU.
−kernel=2 :: lanzará kernel optimizado de la GPU.
Ponga el archivo Makefile en el directorio donde está el código.
Escribe “make” en la terminal

Código fuente

- definitions.cuh

- kernel.cu
- main.cu
- Makefile

Compilar:
make

2.1.6. Quicksort.

El algoritmo Quicksort es un algoritmo de ordenamiento que consiste en tomar la posición central como pivote, dejar los números menores del lado izquierdo y los mayores del lado derecho. Después de cada extremo toma otro pivote y repite los pasos hasta dejar el arreglo ordenado.

Descripción

Ordenamiento Quicksort con CUDA.

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo demuestra un simple ordenamineto rápido usando CUDA Dynamic Parallelism. Este ejemplo requiere dispositivos con capacidad de cálculo 3.5 o superior.

Desarrollo

Selection_sort es usado cuando el ancho es demasiado grande o el número de elementos cae por debajo del umbral.

```
__device__ void selection_sort(unsigned int *data, int left,
                               int right){
    for (int i = left ; i <= right ; ++i){
        unsigned min_val = data[i];
        int min_idx = i;

        // Encuentra el valor más pequeño en el rango
```

```
        // [left, right].
        for (int j = i+1 ; j <= right ; ++j){
            unsigned val_j = data[j];

            if (val_j < min_val){
                min_idx = j;
                min_val = val_j;
            }
        }

        // Cambia los valores.
        if (i != min_idx){
            data[min_idx] = data[i];
            data[i] = min_val;
        }
    }
}
```

Algoritmo muy básico de ordenamiento rápido, lanza la recursividad al siguiente nivel.

```
__global__ void cdp_simple_quicksort(unsigned int *data,
                                     int left, int right,
                                     int depth){
    // Si son demasiados elementos o quedan pocos elementos,
    // utilizamos ordenamiento por inserción.
    if (depth >= MAX_DEPTH || right-left <= INSERTION_SORT){
        selection\sort(data, left, right);
        return;
    }

    unsigned int *lptr = data+left;
    unsigned int *rptr = data+right;
    unsigned int  pivot = data[(left+right)/2];

    // Hacer el particionamiento.
```

```
while (lptr <= rptr){
    // Encuentra los próximos valores de izquierda y
    // derecha para cambiar
    unsigned int lval = *lptr;
    unsigned int rval = *rptr;

    // Mueve el puntero izquierdo, siempre y cuando
    // el elemento del puntero sea más pequeño que el
    // pivote.
    while (lval < pivot){
        lptr++;
        lval = *lptr;
    }

    // Mueva el puntero a la derecha, siempre y cuando
    // el elemento del puntero sea mayor que el pivote.
    while (rval > pivot){
        rptr--;
        rval = *rptr;
    }

    // Si los puntos de cambio son válidos, hacer el
    // cambio.
    if (lptr <= rptr){
        *lptr++ = rval;
        *rptr-- = lval;
    }
}

// Ahora la parte recursiva.
int nright = rptr - data;
int nleft  = lptr - data;

// Lance un nuevo bloque para ordenar la parte izquierda.
if (left < (rptr-data)){
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    cdp_simple_quicksort<<< 1, 1, 0, s >>>(data, left,
```

```
                                                nright, depth+1);
        cudaStreamDestroy(s);
    }

    // Lance un nuevo bloque para ordenar la parte derecha.
    if ((lptr-data) < right){
        cudaStream_t s1;
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        cdp_simple_quicksort<<< 1, 1, 0, s1 >>>(data, nleft,
                                                 right, depth+1);
        cudaStreamDestroy(s1);
    }
}
```

Llama al módulo de ordenamiento rápido desde el host.

```
void run_qsort(unsigned int *data, unsigned int nitems){
    // Prepara CDP para el máximo de elementos 'MAX_DEPTH'.
    checkCudaErrors(cudaDeviceSetLimit(
        cudaLimitDevRuntimeSyncDepth, MAX_DEPTH));

    // Inicia el dispositivo
    int left = 0;
    int right = nitems-1;
    std::cout << "Launching kernel on the GPU" << std::endl;
    cdp_simple_quicksort<<< 1, 1 >>>(data, left, right, 0);
    checkCudaErrors(cudaDeviceSynchronize());
}
```

Inicializar los datos en el host.

```
void initialize_data(unsigned int *dst, unsigned int nitems){
    // Semilla fija para la ilustración.
    srand(2047);
```

```
    // Rellena dst con valores aleatorios
    for (unsigned i = 0 ; i < nitems ; i++)
        dst[i] = rand() % nitems ;
}
```

Verifica los resultados.

```
void check_results(int n, unsigned int *results_d){
   unsigned int *results\_h = new unsigned[n];
    checkCudaErrors(cudaMemcpy(results_h, results_d,
        n*sizeof(unsigned),cudaMemcpyDeviceToHost));

    for (int i = 1 ; i $<$ n ; ++i)
        if (results\_h[i-1] $>$ results\_h[i]){
            std::cout << "Invalid item[" << i-1 << "]: " <<
            results_h[i-1] << " greater than " << results_h[i]
             << std::endl;
            exit(EXIT\_FAILURE);
        }

    std::cout << "OK" << std::endl;
    delete[] results_h;
}
```

Punto de entrada principal.

```
int main(int argc, char **argv){
    int num_items = 128;
    bool verbose = false;

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "h")){
        std::cerr << "Usage: " << argv[0] << " num_items=<
```

```
                num_items>\twhere num_items is the
                number of items to sort"
             << std::endl;
    exit(EXIT_SUCCESS);
}

if (checkCmdLineFlag(argc, (const char **)argv, "v")){
    verbose = true;
}

if (checkCmdLineFlag(argc,(const char**)argv, num_items"))
{
    num_items = getCmdLineArgumentInt(argc,
        (const char**)argv, "num_items");

    if (num_items < 1){
        std::cerr << "ERROR: num_items has to be greater
                     than 1"
                  << std::endl;
        exit(EXIT_FAILURE);
    }
}

// Obtiene las propiedades del dispositivo
int device_count = 0, device = -1;
checkCudaErrors(cudaGetDeviceCount(&device_count));

for (int i = 0 ; i < device_count ; ++i){
    cudaDeviceProp properties;
    checkCudaErrors(cudaGetDeviceProperties(&properties,i));

    if (properties.major > 3 || (properties.major == 3 &&
        properties.minor >= 5)){
        device = i;
        std::cout << "Running on GPU " << i << " (" <<
                     properties.name
                  << ")" << std::endl;
        break;
```

```
    }

    std::cout << "GPU " << i << " (" << properties.name <<
                 ") does not support CUDA Dynamic
                  Parallelism"
              << std::endl;
}

if (device == -1){
    std::cerr << "cdpSimpleQuicksort requires GPU devices
                  with compute SM 3.5 or higher.
                  Exiting..."
              << std::endl;
    exit(EXIT_SUCCESS);
}

cudaSetDevice(device);

// Crea datos de entrada
unsigned int *h_data = 0;
unsigned int *d_data = 0;

// Asignar memoria CPU e inicializar los datos.
std::cout << "Initializing data:" << std::endl;
h_data =(unsigned int *)
        malloc(num_items*sizeof(unsigned int));
initialize_data(h_data, num_items);

if (verbose){
    for (int i=0 ; i$<$num\_items ; i++)
        std::cout << "Data [" << i << "]: " << h_data[i]
                  << std::endl;
}

// Asignar memoria de la GPU.
checkCudaErrors(cudaMalloc((void **)&d_data, num_items *
    sizeof(unsigned int)));
checkCudaErrors(cudaMemcpy(d_data, h_data, num_items *
```

```
        sizeof(unsigned int), cudaMemcpyHostToDevice));

    // Ejecutar
    std::cout << "Running quicksort on " << num_items
              << " elements" << std::endl;
    run_qsort(d_data, num_items);

    // Verificar resultado
    std::cout << "Validating results: ";
    check_results(num_items, d_data);

    free(h_data);
    checkCudaErrors(cudaFree(d_data));
    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}
```

Código fuente:

- cdpSimpleQuickSort.cu
- findcudalib.mk
- Makefile

Compilar:
make

2.1.7. Monte Carlo con soporte Multi-GPU.

Éste algoritmo sirve para realizar una valoración de multiples opciones y aqui veremos un ejemplo con CUDA y viendo un uso con múltiples GPU.

Descripción

Monte Carlo con soporte Multi-GPU en CUDA C.

En la sección de anexos encontrarás el código completo.

Objetivos:

En este ejemplo se evalúa el precio de llamada justo para un determinado conjunto de opciones europeas que utilizan el método de Monte Carlo, aprovechando todas las GPU CUDA instaladas en el sistema. Este ejemplo utiliza el hardware de doble precisión si una GTX 200 GPU de clase está presente. La muestra también se aprovecha de la capacidad de CUDA 4.0 para apoyar el uso de un solo hilo de la CPU para controlar múltiples GPUs.

Desarrollo

Cabeceras

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda_runtime.h>
```

Includes del proyecto

```
#include <helper_functions.h>
// Funciones de ayuda (utilities, parsing, timing)
#include <helper_cuda.h>
//  Funciones de ayuda (cuda error checking and intialization)
#include <multithreading.h>

#include "MonteCarlo_common.h"

int *pArgc = NULL;
char **pArgv = NULL;

#ifdef WIN32
#define strcasecmp _strcmpi
#endif
```

Funciones comunes

```
float randFloat(float low, float high)
{
    float t = (float)rand() / (float)RAND_MAX;
    return (1.0f - t) * low + t * high;
}

// Función de utilidad para modificar problema con el tamaño
// pequeño para GPUs
int adjustProblemSize(int GPU_N, int default_nOptions)
{
    int nOptions = default_nOptions;

    // Seleccione el tamaño de problema
    for (int i=0; i<GPU_N; i++)
    {
        cudaDeviceProp deviceProp;
        checkCudaErrors(cudaGetDeviceProperties(&deviceProp,i));
        int cudaCores = _ConvertSMVer2Cores(deviceProp.major,
                                            deviceProp.minor)
```

```
                        * deviceProp.multiProcessorCount;

        if (cudaCores <= 32)
        {
            nOptions = (nOptions < cudaCores/2 ? nOptions :
                                    cudaCores/2);
        }
    }

    return nOptions;
}
```

Funciones de referencia de la CPU

```
extern "C" void MonteCarloCPU(
    TOptionValue   &callValue,
    TOptionData optionData,
    float *h_Random,
    int pathN
);
```

Fórmula Black-Scholes para las opciones de llamada

```
extern "C" void BlackScholesCall(
    float &CallResult,
    TOptionData optionData
);
```

Hilo anfitrion que conduce a GPU

```
//Temporizador
StopWatchInterface **hTimer = NULL;
```

```
static CUT_THREADPROC solverThread(TOptionPlan *plan)
{
    //Iniciar GPU
    checkCudaErrors(cudaSetDevice(plan->device));

    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp,
                                            plan->device));

    //Inicar temporizador
    sdkStartTimer(&hTimer[plan->device]);

    // Asignar memoria intermedia para MC integrador e
    // inicializar estados RNG
    initMonteCarloGPU(plan);

    // Cómputo principal
    MonteCarloGPU(plan);

    checkCudaErrors(cudaDeviceSynchronize());

    // Detener temporizador
    sdkStopTimer(&hTimer[plan->device]);

    // Apague este GPU
    closeMonteCarloGPU(plan);

    cudaStreamSynchronize(0);

    printf("solverThread() finished - GPU Device %d: %s\n",
           plan->device, deviceProp.name);
    cudaDeviceReset();
    CUT_THREADEND;
}

static void multiSolver(TOptionPlan *plan, int nPlans)
{
```

```
// Asignar e inicializar una matriz de identificadores de
// flujo
cudaStream_t *streams = (cudaStream_t *) malloc(nPlans *
                        sizeof(cudaStream_t));
cudaEvent_t *events = (cudaEvent_t *)malloc(nPlans *
                      sizeof(cudaEvent_t));

for (int i = 0; i < nPlans; i++)
{
    checkCudaErrors(cudaSetDevice(plan[i].device));
    checkCudaErrors(cudaStreamCreate(&(streams[i])));
    checkCudaErrors(cudaEventCreate(&(events[i])));
}

//Inicializar cada GPU
// En CUDA 4.0 podemos llamar cudaSetDevice varias veces
// para apuntar cada dispositivo
// Configure el dispositivo deseado, a continuación,
// realizar inicializaciones en ese dispositivo

for (int i=0 ; i<nPlans ; i++)
{
    // Configurar el dispositivo de destino para realizar
    // la inicialización
    checkCudaErrors(cudaSetDevice(plan[i].device));

    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp,
                    plan[i].device));

    // Asignar memoria intermedia para MC integrador e
    // inicializar estado RNG
    initMonteCarloGPU(&plan[i]);
}

//Comenzar el temporizador
sdkResetTimer(&hTimer[0]);
sdkStartTimer(&hTimer[0]);
```

```
    for (int i=0; i<nPlans; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));

        //Cálculos principales
        MonteCarloGPU(&plan[i], streams[i]);

        checkCudaErrors(cudaEventRecord(events[i]));
    }

    for (int i=0; i<nPlans; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));
        cudaEventSynchronize(events[i]);
    }

    //Detener temporizador
    sdkStopTimer(&hTimer[0]);

    for (int i=0 ; i<nPlans ; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));
        closeMonteCarloGPU(&plan[i]);
        checkCudaErrors(cudaStreamDestroy(streams[i]));
        checkCudaErrors(cudaEventDestroy(events[i]));
    }
}
```

Programa principal

```
#define DO_CPU
#undef DO_CPU

#define PRINT_RESULTS
#undef PRINT_RESULTS
```

```
void usage()
{
    printf("--method=[threaded,streamed]
            --scaling=[strong,weak] [--help]\n");
    printf("Method=threaded: 1 CPU thread for
            each GPU     [default]\n");
    printf("       streamed: 1 CPU thread handles all
            GPUs (requires CUDA 4.0 or newer)\n");
    printf("Scaling=strong : constant problem size\n");
    printf("        weak   : problem size scales with number
            of available GPUs [default]\n");
}


int main(int argc, char **argv)
{
    char *multiMethodChoice = NULL;
    char *scalingChoice = NULL;
    bool use_threads = true;
    bool bqatest = false;
    bool strongScaling = false;

    pArgc = &argc;
    pArgv = argv;

    printf("%s Starting...\n\n", argv[0]);

    if (checkCmdLineFlag(argc, (const char **)argv, "qatest"))
    {
        bqatest = true;
    }

    getCmdLineArgumentString(argc, (const char **)argv,
                             "method", &multiMethodChoice);
    getCmdLineArgumentString(argc, (const char **)argv,
                             "scaling", &scalingChoice);
```

```
if (checkCmdLineFlag(argc, (const char **)argv, "h") ||
    checkCmdLineFlag(argc, (const char **)argv, "help"))
{
    usage();
    exit(EXIT_SUCCESS);
}

if (multiMethodChoice == NULL)
{
    use_threads = true;
}
else
{
    if (!strcasecmp(multiMethodChoice, "threaded"))
    {
        use_threads = true;
    }
    else
    {
        use_threads = false;
    }
}

if (use_threads == false)
{
    printf("Using single CPU thread for multiple GPUs\n");
}

if (scalingChoice == NULL)
{
    strongScaling = false;
}
else
{
    if (!strcasecmp(scalingChoice, "strong"))
    {
        strongScaling = true;
```

```
    }
    else
    {
        strongScaling = false;
    }
}


//Número de GPU presente en el sistema
int GPU_N;
checkCudaErrors(cudaGetDeviceCount(&GPU_N));
int nOptions = 256;

nOptions = adjustProblemSize(GPU_N, nOptions);

// Seleccione el tamaño de problema
int scale = (strongScaling) ? 1 : GPU_N;
int OPT_N = nOptions * scale;
int PATH_N = 262144;
const unsigned long long SEED = 777;

// Inicializar temporizadores
hTimer = new StopWatchInterface*[GPU_N];

for (int i=0; i<GPU_N; i++)
{
    sdkCreateTimer(&hTimer[i]);
    sdkResetTimer(&hTimer[i]);
}

// Matriz de datos de entrada
TOptionData  *optionData   = new TOptionData[OPT_N];
// Los resultados finales GPU MC
TOptionValue *callValueGPU = new TOptionValue[OPT_N];
// Valores "teóricos" de llamada a la fórmula Black-Scholes
float *callValueBS = new float[OPT_N];
// Establecer configuración
TOptionPlan *optionSolver = new TOptionPlan[GPU_N];
```

```
// OS ID del hilo
CUTThread *threadID = new CUTThread[GPU_N];

int gpuBase, gpuIndex;
int i;

float time;

double delta, ref, sumDelta, sumRef, sumReserve;

printf("MonteCarloMultiGPU\n");
printf("==================\n");
printf("Parallelization method  = %s\n",
       use_threads ? "threaded" : "streamed");
printf("Problem scaling         = %s\n",
       strongScaling? "strong" : "weak");
printf("Number of GPUs          = %d\n", GPU_N);
printf("Total number of options = %d\n", OPT_N);
printf("Number of paths         = %d\n", PATH_N);


printf("main(): generating input data...\n");
srand(123);

for (i=0; i < OPT_N; i++)
{
    optionData[i].S = randFloat(5.0f, 50.0f);
    optionData[i].X = randFloat(10.0f, 25.0f);
    optionData[i].T = randFloat(1.0f, 5.0f);
    optionData[i].R = 0.06f;
    optionData[i].V = 0.10f;
    callValueGPU[i].Expected   = -1.0f;
    callValueGPU[i].Confidence = -1.0f;
}

printf("main(): starting %i host threads...\n", GPU_N);
```

```
//Obtenga opción contar para cada GPU
for (i = 0; i < GPU_N; i++)
{
    optionSolver[i].optionCount = OPT_N / GPU_N;
}

//Tome en cuenta los casos con opciones de contar "impares"
for (i = 0; i < (OPT_N % GPU_N); i++)
{
    optionSolver[i].optionCount++;
}

//Asignar rangos de opciones de GPU
gpuBase = 0;

for (i = 0; i < GPU_N; i++)
{
    optionSolver[i].device     = i;
    optionSolver[i].optionData = optionData   + gpuBase;
    optionSolver[i].callValue  = callValueGPU + gpuBase;
    // todos los dispositivos utilizan las mismas
    // semillas globales, pero comienzan la secuencia
    // en un desplazamiento diferente
    optionSolver[i].seed       = SEED;
    optionSolver[i].pathN      = PATH_N;
    gpuBase += optionSolver[i].optionCount;
}


if (use_threads || bqatest)
{
    //Comience hilo de CPU para cada GPU
    for (gpuIndex = 0; gpuIndex < GPU_N; gpuIndex++)
    {
        threadID[gpuIndex] = cutStartThread((
        CUT_THREADROUTINE)solverThread, &optionSolver[
        gpuIndex]);
    }
```

```
printf("main(): waiting for GPU results...\n");
cutWaitForThreads(threadID, GPU_N);

printf("main(): GPU statistics, threaded\n");

for (i = 0; i < GPU_N; i++)
{
    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(
        &deviceProp, optionSolver[i].device));
    printf("GPU Device #%i: %s\n", optionSolver[i].
        device, deviceProp.name);
    printf("Options         : %i\n", optionSolver[i].
        optionCount);
    printf("Simulation paths: %i\n", optionSolver[i].
        pathN);
    time = sdkGetTimerValue(&hTimer[i]);
    printf("Total time (ms.): %f\n", time);
    printf("Options per sec.: %f\n", OPT_N /
            (time * 0.001));
}

printf("main(): comparing Monte Carlo and
        Black-Scholes results...\n");
sumDelta   = 0;
sumRef     = 0;
sumReserve = 0;

for (i = 0; i < OPT_N; i++)
{
    BlackScholesCall(callValueBS[i], optionData[i]);
    delta     = fabs(callValueBS[i] -
                callValueGPU[i].Expected);
    ref       = callValueBS[i];
    sumDelta += delta;
    sumRef   += fabs(ref);
```

```
            if (delta > 1e-6)
            {
                sumReserve += callValueGPU[i].Confidence /
                              delta;
            }

#ifdef PRINT_RESULTS
            printf("BS: %f; delta: %E\n", callValueBS[i],
                   delta);
#endif

        }

        sumReserve /= OPT_N;
    }

    if (!use_threads || bqatest)
    {
        multiSolver(optionSolver, GPU_N);

        printf("main(): GPU statistics, streamed\n");

        for (i = 0; i < GPU_N; i++)
        {
            cudaDeviceProp deviceProp;
            checkCudaErrors(cudaGetDeviceProperties(
                &deviceProp, optionSolver[i].device));
            printf("GPU Device #%i: %s\n",
                optionSolver[i].device, deviceProp.name);
            printf("Options         : %i\n",
                optionSolver[i].optionCount);
            printf("Simulation paths: %i\n",
                optionSolver[i].pathN);
        }

        time = sdkGetTimerValue(&hTimer[0]);
        printf("\nTotal time (ms.): %f\n", time);
        printf("\tNote: This is elapsed time for all to
```

```
                compute.\n");
        printf("Options per sec.: %f\n", OPT_N /
               (time * 0.001));

        printf("main(): comparing Monte Carlo and
                Black-Scholes results...\n");
        sumDelta   = 0;
        sumRef     = 0;
        sumReserve = 0;

        for (i = 0; i < OPT_N; i++)
        {
            BlackScholesCall(callValueBS[i], optionData[i]);
            delta     = fabs(callValueBS[i] -
                        callValueGPU[i].Expected);
            ref       = callValueBS[i];
            sumDelta += delta;
            sumRef   += fabs(ref);

            if (delta > 1e-6)
            {
                sumReserve += callValueGPU[i].Confidence /
                              delta;
            }

#ifdef PRINT_RESULTS
            printf("BS: %f; delta: %E\n", callValueBS[i],
                   delta);
#endif
        }

        sumReserve /= OPT_N;
    }

#ifdef DO_CPU
    printf("main(): running CPU MonteCarlo...\n");
    TOptionValue callValueCPU;
    sumDelta = 0;
```

```
    sumRef   = 0;

    for (i = 0; i < OPT_N; i++)
    {
        MonteCarloCPU(
            callValueCPU,
            optionData[i],
            NULL,
            PATH_N
        );
        delta = fabs(callValueCPU.Expected -
                callValueGPU[i].Expected);
        ref       = callValueCPU.Expected;
        sumDelta += delta;
        sumRef   += fabs(ref);
        printf("Exp : %f | %f\t", callValueCPU.Expected,
               callValueGPU[i].Expected);
        printf("Conf: %f | %f\n", callValueCPU.Confidence,
               callValueGPU[i].Confidence);
    }

    printf("L1 norm: %E\n", sumDelta / sumRef);
#endif

    printf("Shutting down...\n");

    for (int i=0; i<GPU_N; i++)
    {
        sdkStartTimer(&hTimer[i]);
        checkCudaErrors(cudaSetDevice(i));
        cudaDeviceReset();
    }

    delete[] optionSolver;
    delete[] callValueBS;
    delete[] callValueGPU;
    delete[] optionData;
    delete[] threadID;
```

```
    delete[] hTimer;

    printf("Test Summary...\n");
    printf("L1 norm        : %E\n", sumDelta / sumRef);
    printf("Average reserve: %f\n", sumReserve);
    printf(sumReserve>1.0f?"Test passed\n":"Test failed!\n");
    exit(sumReserve > 1.0f ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

Código fuente

- findcudalib.mk
- Makefile
- MonteCarlo_common.h
- MonteCarlo_gold.cpp
- MonteCarlo_kernel.cu
- MonteCarloMultiGPU.cpp
- MonteCarlo_reduction.cuh
- multithreading.h
- multithreading.cpp
- realtype.h

Compilar:
make

2.1.8. Generador de números casi-aleatorios.

Éste algoritmo es un simple generador de números de manera aleatoria, solo que en éste caso serán casi-aleatorios, lo que podemos destacar es la increible rapidez de este algoritmo en CUDA.

Descripción

Generador de secuencias casi aleatorias con Niederreiter Quasirandom Sequence Generator en CUDA quasirandomGenerator_kernel (archivo de módulos)

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo implementa Niederreiter Quasirandom generador de secuencias y funciones de distribución normal acumulativa inversa para la generación estándar de distribuciones normales.

Desarrollo

Cabeceras:

```
#ifndef QUASIRANDOMGENERATOR_KERNEL_CUH
#define QUASIRANDOMGENERATOR_KERNEL_CUH

#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include "realtype.h"
```

```
#include "quasirandomGenerator_common.h"

//Rápida multiplicación de enteros
#define MUL(a, b) __umul24(a, b)
```

Módulo generador de números casi aleatorios Niederreiter

```
static __constant__
    unsigned int c_Table[QRNG_DIMENSIONS][QRNG_RESOLUTION];

static __global__ void quasirandomGeneratorKernel(
    float *d_Output,
    unsigned int seed,
    unsigned int N
){
    unsigned int *dimBase = &c_Table[threadIdx.y][0];
    unsigned int tid=MUL(blockDim.x,blockIdx.x)+threadIdx.x;
    unsigned int  threadN = MUL(blockDim.x, gridDim.x);

    for (unsigned int pos = tid; pos < N; pos += threadN){
        unsigned int result = 0;
        unsigned int data = seed + pos;

        for (int bit=0;bit<QRNG_RESOLUTION;bit++,data>>=1)
            if (data & 1){
                result ^= dimBase[bit];
            }

        d_Output[MUL(threadIdx.y,N)+pos]=(float)(result+1)*
                                        INT_SCALE;
    }
}
```

Tabla de inicialización de rutina.

```
static void initTableGPU(unsigned int
                  tableCPU[QRNG_DIMENSIONS][QRNG_RESOLUTION])
{
    checkCudaErrors(cudaMemcpyToSymbol(
                        c_Table,
                        tableCPU,
                        QRNG_DIMENSIONS * QRNG_RESOLUTION *
                        sizeof(unsigned int)
                    ));
}
```

Interfaz del lado anfitrión.

```
static void quasirandomGeneratorGPU(float *d_Output,
                                    unsigned int seed,
                                    unsigned int N){
    dim3 threads(128, QRNG_DIMENSIONS);
    quasirandomGeneratorKernel<<<128, threads>>>
                               (d_Output, seed, N);
    getLastCudaError("quasirandomGeneratorKernel()
                      execution failed.\n");
}
```

Función inversa de distribución normal acumulativa de aproximación de Moro.

```
#ifndef DOUBLE_PRECISION
__device__ inline float MoroInvCNDgpu(unsigned int x){
    const float a1 = 2.50662823884f;
    const float a2 = -18.61500062529f;
    const float a3 = 41.39119773534f;
    const float a4 = -25.44106049637f;
    const float b1 = -8.4735109309f;
    const float b2 = 23.08336743743f;
```

```
const float b3 = -21.06224101826f;
const float b4 = 3.13082909833f;
const float c1 = 0.337475482272615f;
const float c2 = 0.976169019091719f;
const float c3 = 0.160797971491821f;
const float c4 = 2.76438810333863E-02f;
const float c5 = 3.8405729373609E-03f;
const float c6 = 3.951896511919E-04f;
const float c7 = 3.21767881768E-05f;
const float c8 = 2.888167364E-07f;
const float c9 = 3.960315187E-07f;

float z;

bool negate = false;

// Asegúrese de que la conversión a punto flotante dará
// un valor (0,0.5] en el intervalo mediante la
// restricción de la entrada a la mitad inferior
// del dominio de entrada. Vamos a reflexionar más tarde
// el resultado si la entrada estaba originalmente en la
// mitad superior del dominio de entrada
if (x >= 0x80000000UL){
    x = 0xffffffffUL - x;
    negate = true;
}

// x está ahora en el rango [0,0x80000000)
// (i.e. [0,0x7fffffff])
// Convertir a punto flotante en  (0,0.5]
const float x1 = 1.0f / static_cast<float>(0xffffffffUL);
const float x2 = x1 / 2.0f;
float p1 = x * x1 + x2;
// Convertir a punto flotante en (-0.5,0]
float p2 = p1 - 0.5f;

// La entrada de la inversa de Moro es p2 que está
// en el rango (-0.5,0].
```

```
    // Esto significa que nuestra salida será el lado
    // negativo de la curva de campana
    // (que vamos a reflexionar si es "negativo").

    // El cuerpo principal de la curva de campana
    // para | p | <0,42
    if (p2 > -0.42f){
        z = p2 * p2;
        z = p2 * (((a4 * z + a3) * z + a2) * z + a1) /
            ((((b4 * z + b3) * z + b2) * z + b1) * z + 1.0f);
    }
    // Caso especial (Chebychev) para la cola
    else{
        z = __logf(-__logf(p1));
        z = - (c1 + z * (c2 + z * (c3 + z * (c4 + z *
           (c5 + z * (c6 + z * (c7 + z * (c8 + z * c9))))))));
    }

    // Si la entrada original (x) se encontraba en la mitad
    // superior de la gama, reflejar para conseguir el lado
    // positivo de la curva de la campana
    return negate ? -z : z;
}
#else
__device__ inline double MoroInvCNDgpu(unsigned int x){
    const double a1 = 2.50662823884;
    const double a2 = -18.61500062529;
    const double a3 = 41.39119773534;
    const double a4 = -25.44106049637;
    const double b1 = -8.4735109309;
    const double b2 = 23.08336743743;
    const double b3 = -21.06224101826;
    const double b4 = 3.13082909833;
    const double c1 = 0.337475482272615;
    const double c2 = 0.976169019091719;
    const double c3 = 0.160797971491821;
    const double c4 = 2.76438810333863E-02;
    const double c5 = 3.8405729373609E-03;
```

```
const double c6 = 3.951896511919E-04;
const double c7 = 3.21767881768E-05;
const double c8 = 2.888167364E-07;
const double c9 = 3.960315187E-07;

double z;

bool negate = false;

// Asegúrese de que la conversión a punto flotante dará
// un valor (0,0.5] en el intervalo mediante la
// restricción de la entrada a la mitad inferior del
// dominio de entrada. Vamos a reflexionar más tarde el
// resultado si la entrada estaba originalmente en la
// mitad superior del dominio de entrada
if (x >= 0x80000000UL){
    x = 0xffffffffUL - x;
    negate = true;
}

// x está ahora en el rango [0,0x80000000)
// (i.e. [0,0x7fffffff])
// Convertir a punto flotante en  (0,0.5]
const double x1 = 1.0 / static_cast<double>(0xffffffffUL);
const double x2 = x1 / 2.0;
double p1 = x * x1 + x2;
// Convertir a punto flotante en (-0.5,0]
double p2 = p1 - 0.5;

// La entrada de la inversa de Moro es p2 que está
// en el rango (-0.5,0].
// Esto significa que nuestra salida será el lado
// negativo de la curva de campana (que vamos a
// reflexionar si es "negativo").

// El cuerpo principal de la curva de campana
// para | p | <0,42
if (p2 > -0.42){
```

```
        z = p2 * p2;
        z = p2 * (((a4 * z + a3) * z + a2) * z + a1) /
        ((((b4 * z + b3) * z + b2) * z + b1) * z + 1.0);
    }
    // Caso especial (Chebychev) para la cola
    else{
        z = log(-log(p1));
        z = - (c1 + z * (c2 + z * (c3 + z * (c4 + z *
           (c5 + z * (c6 + z * (c7 + z * (c8 + z * c9))))))));
    }
    // Si la entrada original (x) se encontraba en la mitad
    // superior de la gama, reflejar para conseguir el lado
    // positivo de la curva de la campana.
    return negate ? -z : z;
}
#endif
```

Núcleo principal. Elija entre la transformación de secuencia de entrada y secuencia ascendente uniforme (0, 1)

```
static __global__ void inverseCNDKernel(
    float *d_Output,
    unsigned int *d_Input,
    unsigned int pathN
){
    unsigned int distance = ((unsigned int)-1) / (pathN + 1);
    unsigned int tid=MUL(blockDim.x,blockIdx.x) + threadIdx.x;
    unsigned int threadN = MUL(blockDim.x, gridDim.x);

    // Transformar el número de secuencia de entrada si se
    // suministra
    if (d_Input){
        for (unsigned int pos=tid;pos<pathN;pos+=threadN){
            unsigned int d = d_Input[pos];
            d_Output[pos] = (float)MoroInvCNDgpu(d);
        }
```

```
    }
    // Si no genera muestras de entrada se coloca
    // uniformemente sobre la marcha y escribir en el destino.
    else{
        for (unsigned int pos=tid;pos<pathN;pos+=threadN){
            unsigned int d = (pos + 1) * distance;
            d_Output[pos] = (float)MoroInvCNDgpu(d);
        }
    }
}
static void inverseCNDgpu(float *d_Output,
                          unsigned int *d_Input,
                          unsigned int N){
    inverseCNDKernel<<<128, 128>>>(d_Output, d_Input, N);
    getLastCudaError("inverseCNDKernel()
                     execution failed.\n");
}
#endif
```

Código fuente

- findcudalib.mk
- Makefile
- quasirandomGenerator.cpp
- quasirandomGenerator_common.h
- quasirandomGenerator_gold.cpp
- quasirandomGenerator_kernel.cuh
- quasirandomGenerator_SM10.cu
- quasirandomGenerator_SM13.cu
- realtype.h

Compilar:
make

2.1.9. Punteros de función.

Como bien se sabe, las funciones regresan un valor, propiedad que también tienen las variables, es por eso que podemos deducir que, si una variable puede manejarse con un puntero también lo puede hacer una función. Es aquí donde aprovecharemos esa propiedad para manejar diversas funciones al mismo tiempo, como veníamos haciendo con los valores de las variables y así optimizar mucho más el código.

Descripción

Práctica para aplicar filtro en imágenes monocromáticas de 8 bits. FunctionPointers_kernels.cu (Módulos)

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo muestra cómo utilizar punteros de función y aplica el filtro Sobel Detección de bordes de las imágenes monocromas de 8 bits.

Desarrollo

```
#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include "FunctionPointers_kernels.h"
```

Referencia textura para una imagen de lectura

```
texture<unsigned char, 2> tex;
```

```
extern __shared__ unsigned char LocalBlock[];
static cudaArray *array = NULL;

#define RADIUS 1
```

Valor de píxel utilizado para la función, funciona bien con la imagen de muestra 'llena'

```
#define THRESHOLD 150.0f

#ifdef FIXED_BLOCKWIDTH
#define BlockWidth 80
#define SharedPitch 384
#endif
```

Un puntero de función se puede declarar explícitamente como esta línea: __device__ unsigned char (* pointFunction) (unsigned char, float) = NULL; o usando de typedef como se muestra a continuación:

```
typedef unsigned char(*blockFunction_t)(
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    float);

typedef unsigned char(*pointFunction_t)(
    unsigned char, float);

__device__ blockFunction_t blockFunction;

__device__ unsigned char
ComputeSobel(unsigned char ul, // upper left
             unsigned char um, // upper middle
             unsigned char ur, // upper right
```

```
            unsigned char ml, // middle left
            unsigned char mm, // middle (unused)
            unsigned char mr, // middle right
            unsigned char ll, // lower left
            unsigned char lm, // lower middle
            unsigned char lr, // lower right
            float fScale)
{
    short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
    short Vert = ul + 2*um + ur - ll - 2*lm - lr;
    short Sum = (short)(fScale*(abs((int)Horz)+abs((int)Vert)));
    return (unsigned char)((Sum<0)?0:((Sum>255)?255:Sum));
}
```

Definir un puntero a función e inicializar a NULL

```
__device__ unsigned char(*varFunction)(
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    float x
) = NULL;

__device__ unsigned char
ComputeBox(unsigned char ul,   // upper left
           unsigned char um, // upper middle
           unsigned char ur, // upper right
           unsigned char ml, // middle left
           unsigned char mm, // middle...middle
           unsigned char mr, // middle right
           unsigned char ll, // lower left
           unsigned char lm, // lower middle
           unsigned char lr, // lower right
           float fscale
          )
{
```

```
    short Sum = (short)(ul+um+ur + ml+mm+mr + ll+lm+lr)/9;
    Sum *= fscale;
    return (unsigned char)((Sum<0)?0:((Sum>255)?255:Sum));
}
__device__ unsigned char
Threshold(unsigned char in, float thresh)
{
    if (in > thresh)
    {
        return 0xFF;
    }
    else
    {
        return 0;
    }
}
```

Declarar tablas de funciones, una para la función del puntero elegido, y una para la función de bloque elegido. El número de entradas es determinado por la enumeración en FunctionPointers_kernels.h

```
__device__ blockFunction_t
    blockFunction_table[LAST_BLOCK_FILTER];
__device__ pointFunction_t
    pointFunction_table[LAST_POINT_FILTER];
```

Declarar los punteros a funciones laterales del dispositivo. Les consultamos más tarde con cudaMemcpyFromSymbol() establecer nuestras tablas de funciones por encima de alguna orden particular especificadó en tiempo de ejecución.

```
__device__ blockFunction_t pComputeSobel = ComputeSobel;
__device__ blockFunction_t pComputeBox   = ComputeBox;
__device__ pointFunction_t pComputeThreshold = Threshold;
```

Asignar tablas del lado del host para reflejar el lado del dispositivo, y más tarde, cargamos estas tablas con los punteros a funciones. Esto nos permite enviar los punteros al kernel en la invocación, como método de elección para que funcione.

```
blockFunction_t h_blockFunction_table[2];
pointFunction_t h_pointFunction_table[2];
```

Lleva a cabo una operación de filtrado de los datos, utilizando la memoria compartida. La operación actual se determina por el puntero de función "blockFunction" y se selecciona por el argumento entero "blockOperation" y tiene acceso a un pixel a la redonda del píxel actual que se está procesando. Después de la operación de bloque, una operación por píxel, apuntado por pPointFunction se lleva a cabo antes de que se produzca el último píxel.

```
__global__ void
SobelShared(uchar4 *pSobelOriginal, unsigned short SobelPitch,
#ifndef FIXED_BLOCKWIDTH
            short BlockWidth, short SharedPitch,
#endif
            short w, short h, float fScale,
            int blockOperation, pointFunction_t pPointFunction
           )
{
    short u = 4*blockIdx.x*BlockWidth;
    short v = blockIdx.y*blockDim.y + threadIdx.y;
    short ib;

    int SharedIdx = threadIdx.y * SharedPitch;

    for (ib=threadIdx.x;ib<BlockWidth+2*RADIUS;ib+=blockDim.x)
    {
        LocalBlock[SharedIdx+4*ib+0] =
          tex2D(tex,(float)(u+4*ib-RADIUS+0),(float)(v-RADIUS));
        LocalBlock[SharedIdx+4*ib+1] =
```

```
      tex2D(tex,(float)(u+4*ib-RADIUS+1),(float)(v-RADIUS));
    LocalBlock[SharedIdx+4*ib+2] =
      tex2D(tex,(float)(u+4*ib-RADIUS+2),(float)(v-RADIUS));
    LocalBlock[SharedIdx+4*ib+3] =
      tex2D(tex,(float)(u+4*ib-RADIUS+3),(float)(v-RADIUS));
}

if (threadIdx.y < RADIUS*2)
{
    // copia arrastrando RADIUS * 2 filas de píxeles en
    // común
    SharedIdx = (blockDim.y+threadIdx.y) * SharedPitch;

    for (ib=threadIdx.x;ib<BlockWidth+2*RADIUS;ib+=blockDim.x)
    {
        LocalBlock[SharedIdx+4*ib+0] =
          tex2D(tex,(float)(u+4*ib-RADIUS+0),
          (float)(v+blockDim.y-RADIUS));
        LocalBlock[SharedIdx+4*ib+1] =
          tex2D(tex,(float)(u+4*ib-RADIUS+1),
          (float)(v+blockDim.y-RADIUS));
        LocalBlock[SharedIdx+4*ib+2] =
          tex2D(tex,(float)(u+4*ib-RADIUS+2),
          (float)(v+blockDim.y-RADIUS));
        LocalBlock[SharedIdx+4*ib+3] =
          tex2D(tex,(float)(u+4*ib-RADIUS+3),
          (float)(v+blockDim.y-RADIUS));
    }
}

__syncthreads();

u >>= 2;    // indexa como uchar4 desde aqui
uchar4 *pSobel =
    (uchar4 *)(((char *) pSobelOriginal)+v*SobelPitch);
SharedIdx = threadIdx.y * SharedPitch;

blockFunction = blockFunction_table[blockOperation];
```

```
for (ib = threadIdx.x; ib < BlockWidth; ib += blockDim.x)
{

    uchar4 out;

    unsigned char pix00 =
        LocalBlock[SharedIdx+4*ib+0*SharedPitch+0];
    unsigned char pix01 =
        LocalBlock[SharedIdx+4*ib+0*SharedPitch+1];
    unsigned char pix02 =
        LocalBlock[SharedIdx+4*ib+0*SharedPitch+2];
    unsigned char pix10 =
        LocalBlock[SharedIdx+4*ib+1*SharedPitch+0];
    unsigned char pix11 =
        LocalBlock[SharedIdx+4*ib+1*SharedPitch+1];
    unsigned char pix12 =
        LocalBlock[SharedIdx+4*ib+1*SharedPitch+2];
    unsigned char pix20 =
        LocalBlock[SharedIdx+4*ib+2*SharedPitch+0];
    unsigned char pix21 =
        LocalBlock[SharedIdx+4*ib+2*SharedPitch+1];
    unsigned char pix22 =
        LocalBlock[SharedIdx+4*ib+2*SharedPitch+2];

    out.x = (*blockFunction)(pix00, pix01, pix02,
                             pix10, pix11, pix12,
                             pix20, pix21, pix22, fScale);

    pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+3];
    pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+3];
    pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+3];
    out.y = (*blockFunction)(pix01, pix02, pix00,
                             pix11, pix12, pix10,
                             pix21, pix22, pix20, fScale);

    pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+4];
    pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+4];
```

```
        pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+4];
        out.z = (*blockFunction)(pix02, pix00, pix01,
                                 pix12, pix10, pix11,
                                 pix22, pix20, pix21, fScale);

        pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+5];
        pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+5];
        pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+5];
        out.w = (*blockFunction)(pix00, pix01, pix02,
                                 pix10, pix11, pix12,
                                 pix20, pix21, pix22, fScale);

        if (pPointFunction != NULL)
        {
            out.x = (*pPointFunction)(out.x, THRESHOLD);
            out.y = (*pPointFunction)(out.y, THRESHOLD);
            out.z = (*pPointFunction)(out.z, THRESHOLD);
            out.w = (*pPointFunction)(out.w, THRESHOLD);
        }

        if (u+ib < w/4 && v < h)
        {
            pSobel[u+ib] = out;
        }

    }

    __syncthreads();
}

__global__ void
SobelCopyImage(Pixel *pSobelOriginal, unsigned int Pitch,
               int w, int h, float fscale)
{
    unsigned char *pSobel =
        (unsigned char *)(((char *) pSobelOriginal)+
        blockIdx.x*Pitch);
```

```
    for (int i = threadIdx.x; i < w; i += blockDim.x)
    {
        pSobel[i] = min(max((tex2D(tex, (float) i,
                    (float) blockIdx.x)*fscale), 0.f), 255.f);
    }
}
```

Realice filtrado de bloque y de puntero mediante las búsquedas de textura. Las operaciones de bloque y de punteros se determinan mediante el argumento de entrada (Véase el comentario de la función "SobelShared").

```
__global__ void
SobelTex(Pixel *pSobelOriginal, unsigned int Pitch,
         int w, int h, float fScale, int blockOperation,
         pointFunction_t pPointOperation)
{
    unsigned char *pSobel =
        (unsigned char *)(((char *) pSobelOriginal)+
        blockIdx.x*Pitch);
    unsigned char tmp = 0;

    for (int i = threadIdx.x; i < w; i += blockDim.x)
    {
        unsigned char pix00 =
            tex2D(tex, (float) i-1, (float) blockIdx.x-1);
        unsigned char pix01 =
            tex2D(tex, (float) i+0, (float) blockIdx.x-1);
        unsigned char pix02 =
            tex2D(tex, (float) i+1, (float) blockIdx.x-1);
        unsigned char pix10 =
            tex2D(tex, (float) i-1, (float) blockIdx.x+0);
        unsigned char pix11 =
            tex2D(tex, (float) i+0, (float) blockIdx.x+0);
        unsigned char pix12 =
            tex2D(tex, (float) i+1, (float) blockIdx.x+0);
        unsigned char pix20 =
```

```
            tex2D(tex, (float) i-1, (float) blockIdx.x+1);
        unsigned char pix21 =
            tex2D(tex, (float) i+0, (float) blockIdx.x+1);
        unsigned char pix22 =
            tex2D(tex, (float) i+1, (float) blockIdx.x+1);
        tmp = (*(blockFunction_table[blockOperation]))(pix00,
            pix01, pix02, pix10, pix11, pix12, pix20, pix21,
            pix22, fScale);

        if (pPointOperation != NULL)
        {
            tmp = (*pPointOperation)(tmp, 150.0);
        }

        pSobel[i] = tmp;
    }
}

extern "C" void setupTexture(int iw, int ih, Pixel *data,
                             int Bpp)
{
    cudaChannelFormatDesc desc;

    if (Bpp == 1)
    {
        desc = cudaCreateChannelDesc<unsigned char>();
    }
    else
    {
        desc = cudaCreateChannelDesc<uchar4>();
    }

    checkCudaErrors(cudaMallocArray(&array, &desc, iw, ih));
    checkCudaErrors(cudaMemcpyToArray(array, 0, 0, data,
        Bpp*sizeof(Pixel)*iw*ih, cudaMemcpyHostToDevice));
}

extern "C" void deleteTexture(void)
```

```
{
    checkCudaErrors(cudaFreeArray(array));
}
```

Copia los punteros de las tablas de función a el lado del host

```
void setupFunctionTables()
{
    //Asignar la tabla de funciones dinámicamente.
    // Copie los punteros de función a su ubicación apropiada
    // de acuerdo con la enumeración
    checkCudaErrors(cudaMemcpyFromSymbol(
        &h_blockFunction_table[SOBEL_FILTER], pComputeSobel,
        sizeof(blockFunction_t)));
    checkCudaErrors(cudaMemcpyFromSymbol(
        &h_blockFunction_table[BOX_FILTER], pComputeBox,
        sizeof(blockFunction_t)));

    // Hacer lo mismo para la función del puntero, donde la
    // segunda función es NULL (filtro de "no-op",
    // Omitido en el código del módulo)
    checkCudaErrors(cudaMemcpyFromSymbol(
        &h_pointFunction_table[THRESHOLD_FILTER],
        pComputeThreshold, sizeof(pointFunction_t)));
    h_pointFunction_table[NULL_FILTER] = NULL;

    // Ahora copie las tablas de función de nuevo al
    // dispositivo, por lo que si queremos podemos utilizar
    // un índice en la tabla para elegirlos
    // Hemos puesto a la orden en la tabla de funciones de
    // acuerdo a nuestra enumeración.
    checkCudaErrors(cudaMemcpyToSymbol(blockFunction_table,
        h_blockFunction_table, sizeof(blockFunction_t) *
        LAST_BLOCK_FILTER));
    checkCudaErrors(cudaMemcpyToSymbol(pointFunction_table,
        h_pointFunction_table, sizeof(pointFunction_t) *
```

```
        LAST_POINT_FILTER));
}
```

Contenedor para la llamada __global__ que configura la textura y los hilos. A continuación se muestran dos métodos para seleccionar la función de procesamiento de imágenes. BlockOperation es un argumento kernel entero utilizado como un índice en el blockFunction_table en el lado del dispositivo pPointOp que es en sí mismo un puntero de función que se pasa como argumento kernel, recuperado de una copia del host de la tabla de funciones.

```
extern "C" void sobelFilter(Pixel *odata, int iw, int ih,
 enum SobelDisplayMode mode, float fScale,
 int blockOperation, int pointOperation)
{
    checkCudaErrors(cudaBindTextureToArray(tex, array));
    pointFunction_t pPointOp =
        h_pointFunction_table[pointOperation];

    switch (mode)
    {
        case SOBELDISPLAY_IMAGE:
            SobelCopyImage<<<ih, 384>>>(odata, iw, iw, ih,
                                        fScale);
            break;
        case SOBELDISPLAY_SOBELTEX:
            SobelTex<<<ih, 384>>>(odata, iw, iw, ih, fScale,
                                  blockOperation, pPointOp);
            break;
        case SOBELDISPLAY_SOBELSHARED:
            {
                dim3 threads(16,4);
#ifndef FIXED_BLOCKWIDTH
                // debe ser divisible por 16 de coalescencia
                int BlockWidth = 80;
#endif
                dim3 blocks = dim3(iw/(4*BlockWidth)+
```

```
                    (0!=iw%(4*BlockWidth)), ih/threads.y+
                    (0!=ih%threads.y));
                int SharedPitch = ~0x3f&(4*
                    (BlockWidth+2*RADIUS)+0x3f);
                int sharedMem = SharedPitch*
                    (threads.y+2*RADIUS);

                // para el núcleo común, el ancho debe ser
                // divisible por 4
                iw &= ~3;

                SobelShared<<<blocks, threads, sharedMem>>>
                    ((uchar4 *) odata, iw,
                    #ifndef FIXED_BLOCKWIDTH
                        BlockWidth, SharedPitch,
                    #endif
                    iw, ih, fScale, blockOperation, pPointOp);
            }
            break;
    }

    checkCudaErrors(cudaUnbindTexture(tex));
}
```

Código fuente

- FunctionPointers.zip

Compilar:
make

2.1.10. Merge Sort.

El algoritmo Merge sort es un algoritmo de ordenamiento externo, podemos resumir su funcionamiento con la frase "divide y vencerás". Además éste ordenamiento es muy similar al Quicksort en el que se centra en dividir el arreglo por la mitad, pero aquí no toma un pivote, sino que en verdad divide y ordena, las veces que sean necesarias, así hasta terminar y al final mezclar y dejar una sola lista ordenada.

Descripción

Merge Sort con CUDA

En la sección de anexos encontrarás el código completo.

Objetivos:

Éste ejemplo implementa un mergesort, que es un algoritmo que pertenece a los de clasificación de redes. Aun que en general es muy poco eficiente en secuencias largas en comparación con otros algoritmos. Pueden ser los algoritmos de decisión para la clasificación de lotes de corto y mediano tamaño (clave, valor) pares de la matriz

Desarrollo

Cabeceras:

```
#include <assert.h>
#include <helper_cuda.h>
#include "mergeSort_common.h"
```

Las funciones auxiliares

```
static inline __host__ __device__ uint iDivUp(uint a, uint b)
{
    return ((a % b) == 0) ? (a / b) : (a / b + 1);
}

static inline __host__ __device__ uint getSampleCount(uint
dividend)
{
    return iDivUp(dividend, SAMPLE_STRIDE);
}

#define W (sizeof(uint) * 8)
static inline __device__ uint nextPowerOfTwo(uint x)
{
    /*
        --x;
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
        return ++x;
    */
    return 1U << (W - __clz(x - 1));
}

template<uint sortDir> static inline __device__ uint
binarySearchInclusive(uint val, uint *data, uint L,
uint stride)
{
    if (L == 0)
    {
        return 0;
    }
```

```
    uint pos = 0;

    for (; stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] <= val)) ||
         (!sortDir && (data[newPos - 1] >= val)))
        {
            pos = newPos;
        }
    }

    return pos;
}

template<uint sortDir> static inline __device__ uint
binarySearchExclusive(uint val, uint *data, uint L,
uint stride)
{
    if (L == 0)
    {
        return 0;
    }

    uint pos = 0;

    for (; stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] < val)) ||
         (!sortDir && (data[newPos - 1] > val)))
        {
            pos = newPos;
        }
    }
```

```
    return pos;
}
```

Nivel inferior a ordenar por combinación (basado en búsquedas en binario)

```
template<uint sortDir> __global__ void mergeSortSharedKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint arrayLength
)
{
    __shared__ uint s_key[SHARED_SIZE_LIMIT];
    __shared__ uint s_val[SHARED_SIZE_LIMIT];

    d_SrcKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_SrcVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    s_key[threadIdx.x + 0] = d_SrcKey[0];
    s_val[threadIdx.x + 0] = d_SrcVal[0];
    s_key[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] =
        d_SrcKey[(SHARED_SIZE_LIMIT / 2)];
    s_val[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] =
        d_SrcVal[(SHARED_SIZE_LIMIT / 2)];

    for (uint stride = 1; stride < arrayLength; stride <<= 1)
    {
        uint     lPos = threadIdx.x & (stride - 1);
        uint *baseKey = s_key + 2 * (threadIdx.x - lPos);
        uint *baseVal = s_val + 2 * (threadIdx.x - lPos);

        __syncthreads();
        uint keyA = baseKey[lPos +      0];
        uint valA = baseVal[lPos +      0];
```

```
        uint keyB = baseKey[lPos + stride];
        uint valB = baseVal[lPos + stride];
        uint posA = binarySearchExclusive<sortDir>(keyA,
                    baseKey + stride, stride, stride) + lPos;
        uint posB = binarySearchInclusive<sortDir>(keyB,
                    baseKey +      0, stride, stride) + lPos;

        __syncthreads();
        baseKey[posA] = keyA;
        baseVal[posA] = valA;
        baseKey[posB] = keyB;
        baseVal[posB] = valB;
    }

    __syncthreads();
    d_DstKey[0] = s_key[threadIdx.x + 0];
    d_DstVal[0] = s_val[threadIdx.x + 0];
    d_DstKey[(SHARED_SIZE_LIMIT / 2)] =
        s_key[threadIdx.x + (SHARED_SIZE_LIMIT / 2)];
    d_DstVal[(SHARED_SIZE_LIMIT / 2)] =
        s_val[threadIdx.x + (SHARED_SIZE_LIMIT / 2)];
}

static void mergeSortShared(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint batchSize,
    uint arrayLength,
    uint sortDir
)
{
    if (arrayLength < 2)
    {
        return;
    }
```

```
    assert(SHARED_SIZE_LIMIT % arrayLength == 0);
    assert(((batchSize*arrayLength)%SHARED_SIZE_LIMIT) == 0);
    uint  blockCount = batchSize * arrayLength /
                       SHARED_SIZE_LIMIT;
    uint threadCount = SHARED_SIZE_LIMIT / 2;

    if (sortDir)
    {
        mergeSortSharedKernel<1U>
        <<<blockCount, threadCount>>>(d_DstKey, d_DstVal,
        d_SrcKey, d_SrcVal, arrayLength);
        getLastCudaError("mergeSortShared<1>
        <<<>>> failed\n");
    }
    else
    {
        mergeSortSharedKernel<0U>
            <<<blockCount, threadCount>>>(d_DstKey, d_DstVal,
            d_SrcKey, d_SrcVal, arrayLength);
        getLastCudaError("mergeSortShared<0>
        <<<>>> failed\n");
    }
}
```

Combinar, el paso 1: generar filas de muestra

```
template<uint sortDir> __global__ void
generateSampleRanksKernel(
    uint *d_RanksA,
    uint *d_RanksB,
    uint *d_SrcKey,
    uint stride,
    uint N,
    uint threadCount
)
{
```

```
uint pos = blockIdx.x * blockDim.x + threadIdx.x;

if (pos >= threadCount)
{
    return;
}

const uint i = pos & ((stride / SAMPLE_STRIDE) - 1);
const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);
d_SrcKey += segmentBase;
d_RanksA += segmentBase / SAMPLE_STRIDE;
d_RanksB += segmentBase / SAMPLE_STRIDE;

const uint segmentElementsA = stride;
const uint segmentElementsB = umin(stride, N -
                              segmentBase - stride);
const uint  segmentSamplesA =
    getSampleCount(segmentElementsA);
const uint  segmentSamplesB =
    getSampleCount(segmentElementsB);

if (i < segmentSamplesA)
{
    d_RanksA[i] = i * SAMPLE_STRIDE;
    d_RanksB[i] = binarySearchExclusive<sortDir>(
                      d_SrcKey[i * SAMPLE_STRIDE],
                      d_SrcKey + stride,
                      segmentElementsB, nextPowerOfTwo(
                      segmentElementsB)
                  );
}

if (i < segmentSamplesB)
{
    d_RanksB[(stride / SAMPLE_STRIDE) + i] =
        i * SAMPLE_STRIDE;
    d_RanksA[(stride / SAMPLE_STRIDE) + i] =
        binarySearchInclusive<sortDir>(d_SrcKey[stride +
```

```
                i * SAMPLE_STRIDE], d_SrcKey + 0,
                segmentElementsA, nextPowerOfTwo(
                segmentElementsA));
    }
}

static void generateSampleRanks(
    uint *d_RanksA,
    uint *d_RanksB,
    uint *d_SrcKey,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint threadCount = (lastSegmentElements > stride) ?
        (N + 2 * stride - lastSegmentElements) /
        (2 * SAMPLE_STRIDE) : (N - lastSegmentElements) /
        (2 * SAMPLE_STRIDE);

    if (sortDir)
    {
        generateSampleRanksKernel<1U>
        <<<iDivUp(threadCount, 256), 256>>>(d_RanksA,
            d_RanksB, d_SrcKey, stride, N, threadCount);
        getLastCudaError("generateSampleRanksKernel<1U>
            <<<>>> failed\n");
    }
    else
    {
        generateSampleRanksKernel<0U>
            <<<iDivUp(threadCount, 256), 256>>>(d_RanksA,
            d_RanksB, d_SrcKey, stride, N, threadCount);
        getLastCudaError("generateSampleRanksKernel<0U>
            <<<>>> failed\n");
    }
}
```

Combinar, el paso 2: generar filas de muestra e índices

```
__global__ void mergeRanksAndIndicesKernel(
    uint *d_Limits,
    uint *d_Ranks,
    uint stride,
    uint N,
    uint threadCount
)
{
    uint pos = blockIdx.x * blockDim.x + threadIdx.x;

    if (pos >= threadCount)
    {
        return;
    }

    const uint i = pos & ((stride / SAMPLE_STRIDE) - 1);
    const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);
    d_Ranks  += (pos - i) * 2;
    d_Limits += (pos - i) * 2;

    const uint segmentElementsA = stride;
    const uint segmentElementsB =
        umin(stride, N - segmentBase - stride);
    const uint  segmentSamplesA =
        getSampleCount(segmentElementsA);
    const uint  segmentSamplesB =
        getSampleCount(segmentElementsB);

    if (i < segmentSamplesA)
    {
        uint dstPos = binarySearchExclusive<1U>(d_Ranks[i],
        d_Ranks + segmentSamplesA, segmentSamplesB,
        nextPowerOfTwo(segmentSamplesB)) + i;
        d_Limits[dstPos] = d_Ranks[i];
    }
```

```
    if (i < segmentSamplesB)
    {
        uint dstPos =
            binarySearchInclusive<1U>(d_Ranks[segmentSamplesA+
                i], d_Ranks, segmentSamplesA,
                nextPowerOfTwo(segmentSamplesA)) + i;
        d_Limits[dstPos] = d_Ranks[segmentSamplesA + i];
    }
}

static void mergeRanksAndIndices(
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint *d_RanksA,
    uint *d_RanksB,
    uint stride,
    uint N
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint threadCount = (lastSegmentElements > stride) ? (N +
         2 * stride - lastSegmentElements) / (2 *
         SAMPLE_STRIDE) : (N - lastSegmentElements) / (2 *
         SAMPLE_STRIDE);

    mergeRanksAndIndicesKernel<<<iDivUp(threadCount,256),256>>>
    (
        d_LimitsA,
        d_RanksA,
        stride,
        N,
        threadCount
    );
    getLastCudaError("mergeRanksAndIndicesKernel(A)<<<
    >>> failed\n");

    mergeRanksAndIndicesKernel<<<iDivUp(threadCount,256),256>>>
```

```
    (
        d_LimitsB,
        d_RanksB,
        stride,
        N,
        threadCount
    );
    getLastCudaError("mergeRanksAndIndicesKernel(B)<<<
    >>> failed\n");
}
```

Combinar, el paso 3: combinar intervalos elementales

```
template<uint sortDir> inline __device__ void merge(
    uint *dstKey,
    uint *dstVal,
    uint *srcAKey,
    uint *srcAVal,
    uint *srcBKey,
    uint *srcBVal,
    uint lenA,
    uint nPowTwoLenA,
    uint lenB,
    uint nPowTwoLenB
)
{
    uint keyA, valA, keyB, valB, dstPosA, dstPosB;

    if (threadIdx.x < lenA)
    {
        keyA = srcAKey[threadIdx.x];
        valA = srcAVal[threadIdx.x];
        dstPosA = binarySearchExclusive<sortDir>(keyA,
                  srcBKey, lenB, nPowTwoLenB) + threadIdx.x;
    }
```

```
    if (threadIdx.x < lenB)
    {
        keyB = srcBKey[threadIdx.x];
        valB = srcBVal[threadIdx.x];
        dstPosB = binarySearchInclusive<sortDir>(keyB,
                  srcAKey, lenA, nPowTwoLenA) + threadIdx.x;
    }

    __syncthreads();

    if (threadIdx.x < lenA)
    {
        dstKey[dstPosA] = keyA;
        dstVal[dstPosA] = valA;
    }

    if (threadIdx.x < lenB)
    {
        dstKey[dstPosB] = keyB;
        dstVal[dstPosB] = valB;
    }
}

template<uint sortDir> __global__ void
mergeElementaryIntervalsKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N
)
{
    __shared__ uint s_key[2 * SAMPLE_STRIDE];
    __shared__ uint s_val[2 * SAMPLE_STRIDE];
```

```
const uint   intervalI = blockIdx.x & ((2 * stride) /
                         SAMPLE_STRIDE - 1);
const uint segmentBase = (blockIdx.x - intervalI) *
                         SAMPLE_STRIDE;
d_SrcKey += segmentBase;
d_SrcVal += segmentBase;
d_DstKey += segmentBase;
d_DstVal += segmentBase;

//Establecer parámetros de todo el bloque de hilos
__shared__ uint startSrcA, startSrcB, lenSrcA, lenSrcB,
               startDstA, startDstB;

if (threadIdx.x == 0)
{
    uint segmentElementsA = stride;
    uint segmentElementsB = umin(stride, N - segmentBase -
                            stride);
    uint  segmentSamplesA =
        getSampleCount(segmentElementsA);
    uint  segmentSamplesB =
        getSampleCount(segmentElementsB);
    uint   segmentSamples =
        segmentSamplesA + segmentSamplesB;

    startSrcA    = d_LimitsA[blockIdx.x];
    startSrcB    = d_LimitsB[blockIdx.x];
    uint endSrcA = (intervalI + 1 < segmentSamples) ?
        d_LimitsA[blockIdx.x + 1] : segmentElementsA;
    uint endSrcB = (intervalI + 1 < segmentSamples) ?
        d_LimitsB[blockIdx.x + 1] : segmentElementsB;
    lenSrcA      = endSrcA - startSrcA;
    lenSrcB      = endSrcB - startSrcB;
    startDstA    = startSrcA + startSrcB;
    startDstB    = startDstA + lenSrcA;
}

//Cargue los datos de entrada principal
```

```
__syncthreads();

if (threadIdx.x < lenSrcA)
{
    s_key[threadIdx.x + 0] =
        d_SrcKey[0 + startSrcA + threadIdx.x];
    s_val[threadIdx.x + 0] =
        d_SrcVal[0 + startSrcA + threadIdx.x];
}

if (threadIdx.x < lenSrcB)
{
    s_key[threadIdx.x + SAMPLE_STRIDE] =
        d_SrcKey[stride + startSrcB + threadIdx.x];
    s_val[threadIdx.x + SAMPLE_STRIDE] =
        d_SrcVal[stride + startSrcB + threadIdx.x];
}

//Combinar datos en la memoria compartida
__syncthreads();
merge<sortDir>(
    s_key,
    s_val,
    s_key + 0,
    s_val + 0,
    s_key + SAMPLE_STRIDE,
    s_val + SAMPLE_STRIDE,
    lenSrcA, SAMPLE_STRIDE,
    lenSrcB, SAMPLE_STRIDE
);

//Almacenar datos combinados
__syncthreads();

if (threadIdx.x < lenSrcA)
{
    d_DstKey[startDstA + threadIdx.x] = s_key[threadIdx.x];
    d_DstVal[startDstA + threadIdx.x] = s_val[threadIdx.x];
```

```
    }

    if (threadIdx.x < lenSrcB)
    {
        d_DstKey[startDstB + threadIdx.x] =
            s_key[lenSrcA + threadIdx.x];
        d_DstVal[startDstB + threadIdx.x] =
            s_val[lenSrcA + threadIdx.x];
    }
}

static void mergeElementaryIntervals(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint mergePairs = (lastSegmentElements > stride) ?
         getSampleCount(N) : (N - lastSegmentElements) /
         SAMPLE_STRIDE;

    if (sortDir)
    {
        mergeElementaryIntervalsKernel<1U>
        <<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
```

```
            stride,
            N
        );
        getLastCudaError("
            mergeElementaryIntervalsKernel<1> failed\n");
    }
    else
    {
        mergeElementaryIntervalsKernel<0U>
        <<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
            stride,
            N
        );
        getLastCudaError("
            mergeElementaryIntervalsKernel<0> failed\n");
    }
}



extern "C" void bitonicSortShared(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint batchSize,
    uint arrayLength,
    uint sortDir
);

extern "C" void bitonicMergeElementaryIntervals(
    uint *d_DstKey,
```

```
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N,
    uint sortDir
);



static uint *d_RanksA, *d_RanksB, *d_LimitsA, *d_LimitsB;
static const uint MAX_SAMPLE_COUNT = 32768;

extern "C" void initMergeSort(void)
{
    checkCudaErrors(cudaMalloc((void **)&d_RanksA,
        MAX_SAMPLE_COUNT * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_RanksB,
        MAX_SAMPLE_COUNT * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_LimitsA,
        MAX_SAMPLE_COUNT * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_LimitsB,
        MAX_SAMPLE_COUNT * sizeof(uint)));
}

extern "C" void closeMergeSort(void)
{
    checkCudaErrors(cudaFree(d_RanksA));
    checkCudaErrors(cudaFree(d_RanksB));
    checkCudaErrors(cudaFree(d_LimitsB));
    checkCudaErrors(cudaFree(d_LimitsA));
}

extern "C" void mergeSort(
    uint *d_DstKey,
    uint *d_DstVal,
```

```
    uint *d_BufKey,
    uint *d_BufVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint N,
    uint sortDir
)
{
    uint stageCount = 0;

    for (uint stride = SHARED_SIZE_LIMIT; stride < N;
         stride <<= 1, stageCount++);

    uint *ikey, *ival, *okey, *oval;

    if (stageCount & 1)
    {
        ikey = d_BufKey;
        ival = d_BufVal;
        okey = d_DstKey;
        oval = d_DstVal;
    }
    else
    {
        ikey = d_DstKey;
        ival = d_DstVal;
        okey = d_BufKey;
        oval = d_BufVal;
    }

    assert(N <= (SAMPLE_STRIDE * MAX_SAMPLE_COUNT));
    assert(N % SHARED_SIZE_LIMIT == 0);
    mergeSortShared(ikey, ival, d_SrcKey, d_SrcVal, N /
        SHARED_SIZE_LIMIT, SHARED_SIZE_LIMIT, sortDir);

    for (uint stride = SHARED_SIZE_LIMIT; stride < N;
         stride <<= 1)
    {
```

```
        uint lastSegmentElements = N % (2 * stride);

        //Encuentra filas de muestra y prepara para los
        // límites de combinación.
        generateSampleRanks(d_RanksA, d_RanksB, ikey, stride,
                            N, sortDir);

        //Combinar filas e índices
        mergeRanksAndIndices(d_LimitsA, d_LimitsB, d_RanksA,
                             d_RanksB, stride, N);

        //Combinar intervalos elementales
        mergeElementaryIntervals(okey, oval, ikey, ival,
            d_LimitsA, d_LimitsB, stride, N, sortDir);

        if (lastSegmentElements <= stride)
        {
            // Último segmento de combinación, consiste en una
            // matriz única que sólo tiene que ser pasada
            checkCudaErrors(cudaMemcpy(okey + (N -
                lastSegmentElements), ikey + (N -
                lastSegmentElements), lastSegmentElements *
                sizeof(uint), cudaMemcpyDeviceToDevice));
            checkCudaErrors(cudaMemcpy(oval + (N -
                lastSegmentElements), ival + (N -
                lastSegmentElements), lastSegmentElements *
                sizeof(uint), cudaMemcpyDeviceToDevice));
        }

        uint *t;
        t = ikey;
        ikey = okey;
        okey = t;
        t = ival;
        ival = oval;
        oval = t;
    }
}
```

Código fuente

- bitonic.cu
- findcudalib.mk
- main.cpp
- Makefile
- mergeSort.cu
- mergeSort_common.h
- mergeSort_host.cpp
- mergeSort_validate.cpp

Compilar:
make

2.2. Paralelismo dinámico

Consiste en que ya no son necesarias varias llamadas al dispositivo desde el CPU, sino que, el mismo GPU puede lanzar procesos y bucles anidados por sí mismo de manera dinámica.

2.2.1. Quad Tree (CUDA Dynamic Parallelism).

Éste algoritmo realiza un árbol de cuadros (Quadtree) y como éste va construyendo más y más hijos para cada rama y cuadro utilizaremos ayuda de la funcionalidad del paralelismo dinámico. Éste es el mejor ejemplo ya que el algoritmo consta de crear nuevos nodos (cuadros) y el paralelismo dinámico consiste en lanzar automáticamente estos nodos nuevos de manera automática.

Descripción

Implementación de quadtrees con CUDA Dynamic Parallelism.

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo muestra árboles de cuadrángulos implementados usando CUDA Dynamic Parallelism. Este ejemplo requiere dispositivos con capacidad de cálculo 3.5 o superior.

Desarrollo

Cabeceras

```
#include <thrust/random.h>
#include <thrust/device_vector.h>
#include <helper_cuda.h>
```

Una estructura de puntos 2D (estructura de matrices).

```
class Points
{
        float *m_x;
        float *m_y;

    public:
        // Constructor.
        __host__ __device__ Points() : m_x(NULL), m_y(NULL) {}

        // Constructor.
        __host__ __device__ Points(float *x,float *y):m_x(x),
            m_y(y) {}

        // Obtener un punto.
        __host__ __device__ __forceinline__ float2 get_point
            (int idx) const
        {
            return make_float2(m_x[idx], m_y[idx]);
        }

        // Establecer un punto.
        __host__ __device__ __forceinline__ void set_point
            (int idx, const float2 &p)
        {
            m_x[idx] = p.x;
            m_y[idx] = p.y;
        }

        // Establecer los punteros.
        __host__ __device__ __forceinline__ void set
```

```
            (float *x, float *y)
        {
            m_x = x;
            m_y = y;
        }
};
```

Un cuadro de límite 2D

```
class Bounding_box
{
        // Puntos extremos del cuadro de selección.
        float2 m_p_min;
        float2 m_p_max;

    public:
        // Constructor. Crear un cuadro de unidad.
        __host__ __device__ Bounding_box()
        {
            m_p_min = make_float2(0.0f, 0.0f);
            m_p_max = make_float2(1.0f, 1.0f);
        }

        // Calcular el centro del límite del cuadro.
        __host__ __device__ void compute_center
            (float2 &center) const
        {
            center.x = 0.5f * (m_p_min.x + m_p_max.x);
            center.y = 0.5f * (m_p_min.y + m_p_max.y);
        }

        // Los puntos del cuadro.
        __host__ __device__ __forceinline__ const float2
        &get_max() const
        {
            return m_p_max;
```

```
        }

        __host__ __device__ __forceinline__ const float2
        &get_min() const
        {
            return m_p_min;
        }

        // El cuadro contiene un punto.
        __host__ __device__ bool contains(const float2 &p)
        constv
        {
            return p.x >= m_p_min.x && p.y < m_p_max.x && p.y
                >= m_p_min.y && p.y < m_p_max.y;
        }

        // Definir el limite del cuadro.
        __host__ __device__ void set(float min_x,
        float min_y, float max_x, float max_y)
        {
            m_p_min.x = min_x;
            m_p_min.y = min_y;
            m_p_max.x = max_x;
            m_p_max.y = max_y;
        }
};
```

Un nodo de un árbol de cuadrángulos.

```
class Quadtree_node
{
        // El identificador del nodo.
        int m_id;
        // El límite del cuadro del árbol.
        Bounding_box m_bounding_box;
        // El rango de puntos.
```

```
    int m_begin, m_end;


public:
    // Constructor.
    __host__ __device__ Quadtree_node():m_id(0),m_begin(0),
    m_end(0){}

    // El ID de un nodo en su nivel.
    __host__ __device__ int id() const
    {
        return m_id;
    }

    // El ID de un nodo en su nivel.
    __host__ __device__ void set_id(int new_id)
    {
        m_id = new_id;
    }

    // El límite del cuadro.
    __host__ __device__ __forceinline__ const
    Bounding_box &bounding_box() const
    {
        return m_bounding_box;
    }

    // Establecer el límite del cuadro.
    __host__ __device__ __forceinline__ void
    set_bounding_box(float min_x, float min_y,
    float max_x, float max_y)
    {
        m_bounding_box.set(min_x, min_y, max_x, max_y);
    }

    // El número de puntos en el árbol.
    __host__ __device__ __forceinline__ int num_points()
    const
```

```
        {
            return m_end - m_begin;
        }

        // El rango de puntos en el árbol.
        __host__ __device__ __forceinline__ int points_begin()
        const
        {
            return m_begin;
        }

        __host__ __device__ __forceinline__ int points_end()
        const
        {
            return m_end;
        }

        // Definir el rango para ese nodo.
        __host__ __device__ __forceinline__ void set_range
            (int begin, int end)
        {
            m_begin = begin;
            m_end = end;
        }
};
```

Parámetros del algoritmo.

```
struct Parameters
{
    // Seleccione el conjunto adecuado de puntos a usar como
    // entrada / salida.
    int point_selector;
    // El número de nodos en un nivel dado (2 ^ k para el
    // nivel k).
    int num_nodes_at_this_level;
```

```
    // El nivel de recursividad.
    int depth;
    // El valor máximo por nivel.
    const int max_depth;
    // El número mínimo de puntos en un nodo para parar la
    // recursión.
    const int min_points_per_node;

    // Constructor ajustado a los valores por defecto.
    __host__ __device__ Parameters(int max_depth,
    int min_points_per_node) :
        point_selector(0),
        num_nodes_at_this_level(1),
        depth(0),
        max_depth(max_depth),
        min_points_per_node(min_points_per_node)
    {}

    // Copia constructor. Cambia los valores para la próxima
    // iteración.
    __host__ __device__ Parameters(const Parameters &params,
    bool) :
        point_selector((params.point_selector+1) % 2),
        num_nodes_at_this_level(4 *
        params.num_nodes_at_this_level),
        depth(params.depth+1),
        max_depth(params.max_depth),
        min_points_per_node(params.min_points_per_node)
    {}
};
```

Construye un árbol de cuadrángulos en el GPU. Usa CUDA Dynamic Parallelism.

El algoritmo funciona de la siguiente manera. El host (CPU) lanza un bloque de hilos NUM_THREADS_PER_BLOCK. Ese bloque va a hacer los siguientes pasos:

1. Compruebe el número de puntos y su ancho.

   Imponemos un ancho máximo del árbol y un número mínimo de puntos por nodo. Esto, si la profundidad máxima se supera o se alcanza el número mínimo de puntos. Los hilos del bloque salen. Antes de salir, llevan a cabo un intercambio de buffer, si es necesario. De hecho, el algoritmo utiliza dos buffers para permutar los puntos y asegurarse de que están distribuidos correctamente en el árbol de cuadrángulos. Por diseño queremos que todos los puntos estén en el primer buffer de puntos al final del algoritmo. Es la razón por la cual es posible que tengamos que cambiar el búfer antes de dejarlo (si los puntos están en el segundo buffer).

2. Cuenta el número de puntos en cada hijo (cuadro).

   Si el ancho no es demasiado alto y el número de puntos es suficiente, el bloque tiene que enviar los puntos en cuatro cubos geométricos: Sus hijos. Para ello, se calcula el centro del cuadro delimitador y contamos el número de puntos en cada cuadrante.

   El conjunto de puntos se divide en secciones. Cada sección se da a un sesgo de hilos (32 hilos). Los sesgos utilizan __ballot y __popc intrínsecamente para contar los puntos. Consulte la Guía de programación para obtener más información acerca de estas funciones.

3. Escanea los resultados de los sesgos para conocer los números "globales".

   Los sesgos trabajan independientemente uno de otro. Al final, cada uno sabe el número de puntos en su sección. Para conocer los números para el bloque, el bloque realiza un escaneo/reducción en el nivel de bloque. Esto es un enfoque tradicional. La puesta en práctica en esa muestra no está tan optimizada como lo que se puede encontrar en las ordenaciones rápidas radix, por ejemplo, sino que se basa en la misma idea.

4. Mover puntos.

Ahora que el bloque sabe cuántos puntos va en cada uno de sus 4 hijos, sigue despachar los puntos. Es sencillo.

5. Lanzar nuevos bloques.

   El bloque lanza cuatro nuevos bloques: uno por hijo. Cada uno de los cuatro bloques aplicará el mismo algoritmo.

```
template< int NUM_THREADS_PER_BLOCK >
__global__
void build_quadtree_kernel(Quadtree_node *nodes,
Points *points, Parameters params)
{
    // El número de sesgos en un bloque.
    const int NUM_WARPS_PER_BLOCK = NUM_THREADS_PER_BLOCK /
                                    warpSize;

    // La memoria compartida para almacenar el número de
    // puntos.
    extern __shared__ int smem[];

    // s_num_pts[4][NUM_WARPS_PER_BLOCK];
    // Las direcciones de memoria compartida.
    volatile int *s_num_pts[4];

    for (int i = 0 ; i < 4 ; ++i)
        s_num_pts[i] = (volatile int *)
                       &smem[i*NUM_WARPS_PER_BLOCK];

    // Calcular las coordenadas de los hilos en el bloque.
    const int warp_id = threadIdx.x / warpSize;
    const int lane_id = threadIdx.x % warpSize;

    // Máscara para la compactación.
    int lane_mask_lt = (1 << lane_id) - 1;
    // Same as: asm( "mov.u32 %0, %%lanemask_lt;" : "=r"(
    // lane_mask_lt) );
```

```
// El nodo actual.
Quadtree_node &node = nodes[blockIdx.x];
node.set_id(node.id() + blockIdx.x);

// El número de puntos en el nodo.
int num_points = node.num_points();
```

1- Compruebe el número de puntos y su ancho.

```
// Detenga la recursividad aquí. Asegúrese de que
// points[0] contiene todos los puntos.
if (params.depth >= params.max_depth ||
    num_points <= params.min_points_per_node)
{
    if (params.point_selector == 1)
    {
        int it = node.points_begin(),
            end = node.points_end();

        for (it += threadIdx.x ; it < end ; it +=
        NUM_THREADS_PER_BLOCK)
            if (it < end)
                points[0].set_point(it, points[1].
                get_point(it));
    }

    return;
}

// Calcular el centro del límite de los puntos del cuadro .
const Bounding_box &bbox = node.bounding_box();
float2 center;
bbox.compute_center(center);
```

```
// Encontrar el número de puntos para dar a cada sesgo.
int num_points_per_warp = max(warpSize,
    (num_points + NUM_WARPS_PER_BLOCK-1) /
    NUM_WARPS_PER_BLOCK);

// Cada sesgo de hilos calculará el número de puntos para
// mover a cada cuadrante.
int range_begin = node.points_begin() + warp_id *
                  num_points_per_warp;
int range_end   = min(range_begin + num_points_per_warp,
                      node.points_end());
```

2- Cuenta el número de puntos en cada hijo (cuadro).

```
// Restablecer la cuenta de puntos por hijo.
if (lane_id == 0)
{
    s_num_pts[0][warp_id] = 0;
    s_num_pts[1][warp_id] = 0;
    s_num_pts[2][warp_id] = 0;
    s_num_pts[3][warp_id] = 0;
}

// Puntos de entrada.
const Points &in_points = points[params.point_selector];

// Calcula el número de puntos.
for (int range_it = range_begin + lane_id; __any(range_it
< range_end);
     range_it += warpSize)
{
    // ¿Sigue siendo un hilo activo?
    bool is_active = range_it < range_end;

    // Cargar las coordenadas del punto.
    float2 p = is_active ? in_points.get_point(range_it)
```

```
        : make_float2(0.0f, 0.0f);

        // Cuenta los  puntos superior izquierdos.
        int num_pts = __popc(__ballot(is_active && p.x <
                      center.x && p.y >= center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[0][warp_id] += num_pts;

        // Cuenta los puntos superior derechos.
        num_pts = __popc(__ballot(is_active && p.x >=
                  center.x && p.y >= center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[1][warp_id] += num_pts;

        // Cuenta los puntos inferior izquierdos.
        num_pts = __popc(__ballot(is_active && p.x <
                  center.x && p.y < center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[2][warp_id] += num_pts;

        // Cuenta los puntos inferior derechos.
        num_pts = __popc(__ballot(is_active && p.x >=
                  center.x && p.y < center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[3][warp_id] += num_pts;
    }

    // Hacer sesgos si es seguro que han terminado de contar.
    __syncthreads();
```

3- Escanea los resultados de los sesgos para conocer los números “globales”.

```
    // Los 4 primeros sesgos escanean el número de puntos por
    // hijo (exploración completa).
    if (warp_id < 4)
    {
        int num_pts = lane_id < NUM_WARPS_PER_BLOCK ?
        s_num_pts[warp_id][lane_id] : 0;
#pragma unroll

        for (int offset = 1 ; offset < NUM_WARPS_PER_BLOCK ;
        offset *= 2)
        {
            int n = __shfl_up(num_pts, offset,
                              NUM_WARPS_PER_BLOCK);

            if (lane_id >= offset)
                num_pts += n;
        }

        if (lane_id < NUM_WARPS_PER_BLOCK)
            s_num_pts[warp_id][lane_id] = num_pts;
    }

    __syncthreads();

    // Calcula las compensaciones globales.
    if (warp_id == 0)
    {
        int sum = s_num_pts[0][NUM_WARPS_PER_BLOCK-1];

        for (int row = 1 ; row < 4 ; ++row)
        {
            int tmp = s_num_pts[row][NUM_WARPS_PER_BLOCK-1];

            if (lane_id < NUM_WARPS_PER_BLOCK)
                s_num_pts[row][lane_id] += sum;

            sum += tmp;
        }
```

```
}

__syncthreads();

// Hacer el análisis exclusivo.
if (threadIdx.x < 4*NUM_WARPS_PER_BLOCK)
{
    int val = threadIdx.x == 0 ? 0 : smem[threadIdx.x-1];
    val += node.points_begin();
    smem[threadIdx.x] = val;
}

__syncthreads();
```

4- Mover puntos.

```
// Puntos de salida.
Points &out_points = points[(params.point_selector+1) % 2];

// Reordenar puntos.
for (int range_it = range_begin + lane_id ; __any(
range_it < range_end) ; range_it += warpSize)
{
    // ¿Sigue siendo un hilo activo?
    bool is_active = range_it < range_end;

    // Cargar las coordenadas del punto.
    float2 p = is_active ? in_points.get_point(range_it)
    : make_float2(0.0f, 0.0f);

    // Cuenta los  puntos superior izquierdos.
    bool pred = is_active && p.x < center.x && p.y >=
                center.y;
    int vote = __ballot(pred);
    int dest = s_num_pts[0][warp_id] + __popc(vote &
               lane_mask_lt);
```

```
if (pred)
    out_points.set_point(dest, p);

if (lane_id == 0)
    s_num_pts[0][warp_id] += __popc(vote);

// Cuenta los puntos superior derechos.
pred=is_active && p.x >= center.x && p.y >= center.y;
vote = __ballot(pred);
dest = s_num_pts[1][warp_id] + __popc(vote &
       lane_mask_lt);

if (pred)
    out_points.set_point(dest, p);

if (lane_id == 0)
    s_num_pts[1][warp_id] += __popc(vote);

// Cuenta los puntos inferior izquierdos.
pred = is_active && p.x < center.x && p.y < center.y;
vote = __ballot(pred);
dest = s_num_pts[2][warp_id] + __popc(vote &
       lane_mask_lt);

if (pred)
    out_points.set_point(dest, p);

if (lane_id == 0)
    s_num_pts[2][warp_id] += __popc(vote);

// Cuenta los puntos inferior derechos.
pred = is_active && p.x >= center.x && p.y < center.y;
vote = __ballot(pred);
dest = s_num_pts[3][warp_id] + __popc(vote &
       lane_mask_lt);

if (pred)
```

```
            out_points.set_point(dest, p);

        if (lane_id == 0)
            s_num_pts[3][warp_id] += __popc(vote);
    }

    __syncthreads();
```

5- Lanzar nuevos bloques.

```
    // El último hilo lanza nuevos bloques.
    if (threadIdx.x == NUM_THREADS_PER_BLOCK-1)
    {
        // El hijo
        Quadtree_node *children =
            &nodes[params.num_nodes_at_this_level];

        // Los desplazamientos de los hijos en su nivel.
        int child_offset = 4*node.id();

        // Establecer IDs.
        children[child_offset+0].set_id(4*node.id()+ 0);
        children[child_offset+1].set_id(4*node.id()+ 4);
        children[child_offset+2].set_id(4*node.id()+ 8);
        children[child_offset+3].set_id(4*node.id()+12);

        // Puntos del límite del cuadro.
        const float2 &p_min = bbox.get_min();
        const float2 &p_max = bbox.get_max();

        // Establezca los límites de los cuadros de los hijos.
        children[child_offset+0].set_bounding_box(p_min.x ,
            center.y, center.x, p_max.y);    // Top-left.
        children[child_offset+1].set_bounding_box(center.x,
            center.y, p_max.x , p_max.y);    // Top-right.
        children[child_offset+2].set_bounding_box(p_min.x ,
```

```
            p_min.y , center.x, center.y);   // Bottom-left.
        children[child_offset+3].set_bounding_box(center.x,
            p_min.y , p_max.x , center.y);   // Bottom-right.

        // Establecer los rangos de los hijos.
        children[child_offset+0].set_range(node.points_begin(),
          s_num_pts[0][warp_id]);
        children[child_offset+1].set_range(s_num_pts[0]
          [warp_id], s_num_pts[1][warp_id]);
        children[child_offset+2].set_range(s_num_pts[1]
          [warp_id], s_num_pts[2][warp_id]);
        children[child_offset+3].set_range(s_num_pts[2]
          [warp_id], s_num_pts[3][warp_id]);

        // Inicia 4 hijos.%-----------------------------
        build_quadtree_kernel<NUM_THREADS_PER_BLOCK>
          <<<4, NUM_THREADS_PER_BLOCK, 4 *NUM_WARPS_PER_BLOCK
          *sizeof(int)>>>(children, points, Parameters(
          params, true));
    }
}
```

Asegúrese que el árbol de cuadrángulos (Quadtree) está bien definido.

```
bool check_quadtree(const Quadtree_node *nodes, int idx,
int num_pts, Points *pts, Parameters params)
{
    const Quadtree_node &node = nodes[idx];
    int num_points = node.num_points();

    if (params.depth == params.max_depth ||
        num_points <= params.min_points_per_node)
    {
        int num_points_in_children = 0;

        num_points_in_children +=
```

```
            nodes[params.num_nodes_at_this_level +
            4*idx+0].num_points();
        num_points_in_children +=
            nodes[params.num_nodes_at_this_level +
            4*idx+1].num_points();
        num_points_in_children +=
            nodes[params.num_nodes_at_this_level +
            4*idx+2].num_points();
        num_points_in_children +=
            nodes[params.num_nodes_at_this_level +
            4*idx+3].num_points();

        if (num_points_in_children != node.num_points())
            return false;

        return check_quadtree(&nodes[params.
          num_nodes_at_this_level], 4*idx+0, num_pts, pts,
          Parameters(params, true)) &&
                check_quadtree(&nodes[params.
                num_nodes_at_this_level], 4*idx+1, num_pts,
                pts, Parameters(params, true)) &&
                check_quadtree(&nodes[params.
                num_nodes_at_this_level], 4*idx+2, num_pts,
                pts, Parameters(params, true)) &&
                check_quadtree(&nodes[params.
                num_nodes_at_this_level], 4*idx+3, num_pts,
                pts, Parameters(params, true));
    }

    const Bounding_box &bbox = node.bounding_box();

    for (int it=node.points_begin();it<node.points_end();++it)
    {
        if (it >= num_pts)
            return false;

        float2 p = pts->get_point(it);
```

```
        if (!bbox.contains(p))
            return false;
    }

    return true;
}
```

Generador de números aleatorios en paralelo.

```
struct Random_generator
{
    __host__ __device__ unsigned int hash(unsigned int a)
    {
        a = (a+0x7ed55d16) + (a<<12);
        a = (a^0xc761c23c) ^ (a>>19);
        a = (a+0x165667b1) + (a<<5);
        a = (a+0xd3a2646c) ^ (a<<9);
        a = (a+0xfd7046c5) + (a<<3);
        a = (a^0xb55a4f09) ^ (a>>16);
        return a;
    }

    __host__ __device__ __forceinline__ thrust::tuple<float,
    float> operator()()
    {
        unsigned seed=hash(blockIdx.x*blockDim.x+threadIdx.x);
        thrust::default_random_engine rng(seed);
        thrust::random::uniform_real_distribution<float>
                distrib;
        return thrust::make_tuple(distrib(rng), distrib(rng));
    }
};
```

Punto de entrada principal.

```
int main(int argc, char **argv)
{
    // Constantes para controlar el algoritmo.
    const int num_points = 1024;
    const int max_depth  = 8;
    const int min_points_per_node = 16;

    // Buscar/configurar el dispositivo.
    // La prueba requiere una arquitectura SM35 o mayor (CDP
    // capable).
    int warp_size = 0;
    int cuda_device=findCudaDevice(argc, (const char **)argv);
    cudaDeviceProp deviceProps;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProps,
                                            cuda_device));
    int cdpCapable = (deviceProps.major == 3 &&
                      deviceProps.minor >= 5) ||
                      deviceProps.major >=4;

    printf("GPU device %s has compute capabilities (SM %d.%d)
      \n", deviceProps.name, deviceProps.major, deviceProps.
      minor);

    if (!cdpCapable)
    {
        std::cerr << "cdpQuadTree requires SM 3.5 or higher
          to use CUDA Dynamic Parallelism.  Exiting...\n" <<
          std::endl;
        exit(EXIT_SUCCESS);
    }

    warp_size = deviceProps.warpSize;

    // Asignar memoria para puntos.
    thrust::device_vector<float> x_d0(num_points);
    thrust::device_vector<float> x_d1(num_points);
    thrust::device_vector<float> y_d0(num_points);
    thrust::device_vector<float> y_d1(num_points);
```

```
// Generar puntos aleatorios.
Random_generator rnd;
thrust::generate(
    thrust::make_zip_iterator(thrust::make_tuple(x_d0.
      begin(), y_d0.begin())),
    thrust::make_zip_iterator(thrust::make_tuple(x_d0.
      end(), y_d0.end())),rnd);

// Estructuras del host para analizar los dispositivos.
Points points_init[2];
points_init[0].set(thrust::raw_pointer_cast(&x_d0[0]),
                   thrust::raw_pointer_cast(&y_d0[0]));
points_init[1].set(thrust::raw_pointer_cast(&x_d1[0]),
                   thrust::raw_pointer_cast(&y_d1[0]));

// Asignar memoria para almacenar puntos.
Points *points;
checkCudaErrors(cudaMalloc((void **) &points, 2*sizeof(
                Points)));
checkCudaErrors(cudaMemcpy(points, points_init, 2*sizeof(
                Points), cudaMemcpyHostToDevice));

// Hacer una forma cerrada
int max_nodes = 0;

for (int i = 0, num_nodes_at_level = 1 ; i < max_depth ;
++i, num_nodes_at_level *= 4)
    max_nodes += num_nodes_at_level;

// Asignar memoria para almacenar el árbol.
Quadtree_node root;
root.set_range(0, num_points);
Quadtree_node *nodes;
checkCudaErrors(cudaMalloc((void **) &nodes,
    max_nodes*sizeof(Quadtree_node)));
checkCudaErrors(cudaMemcpy(nodes, &root, sizeof(
    Quadtree_node), cudaMemcpyHostToDevice));
```

```
// Hemos establecido el límite de recursividad para CDP a
// max_depth.
cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth,max_depth);

// Construir el árbol de cuadrángulos (Quadtree).
Parameters params(max_depth, min_points_per_node);
std::cout << "Launching CDP kernel to build the quadtree"
          << std::endl;
// No usar menos de 128 hilos
const int NUM_THREADS_PER_BLOCK = 128;
const int NUM_WARPS_PER_BLOCK = NUM_THREADS_PER_BLOCK /
                                warp_size;
const size_t smem_size = 4*NUM_WARPS_PER_BLOCK*sizeof(int);
build_quadtree_kernel<NUM_THREADS_PER_BLOCK> <<<1,
NUM_THREADS_PER_BLOCK, smem_size>>>(nodes, points,params);
checkCudaErrors(cudaGetLastError());

// Copiar puntos al CPU.
thrust::host_vector<float> x_h(x_d0);
thrust::host_vector<float> y_h(y_d0);
Points host_points;
host_points.set(thrust::raw_pointer_cast(&x_h[0]),
                thrust::raw_pointer_cast(&y_h[0]));

// Copiar nodos al CPU.
Quadtree_node *host_nodes = new Quadtree_node[max_nodes];
checkCudaErrors(cudaMemcpy(host_nodes, nodes,
max_nodes *sizeof(Quadtree_node), cudaMemcpyDeviceToHost));

// Validar los resultados.
bool ok = check_quadtree(host_nodes, 0, num_points,
                         &host_points, params);
std::cout << "Results: " << (ok ? "OK" : "FAILED")
          << std::endl;

// Liberar memoria del CPU.
delete[] host_nodes;
```

```
    // Liberar memoria.
    checkCudaErrors(cudaFree(nodes));
    checkCudaErrors(cudaFree(points));

    cudaDeviceReset();

    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

Código fuente

- cdpQuadtree.cu
- findcudalib.mk
- Makefile

Compilar:
make

2.2.2. Quicksort avanzado (CUDA Dynamic Parallelism)

Como ya vimos el algoritmo Quicksort es un algoritmo de ordenamiento que consiste en tomar la posición central como pivote, dejar los números menores del lado izquierdo y los mayores del lado derecho. Después de cada extremo toma otro pivote y repite los pasos hasta dejar el arreglo ordenado.

Con la ventaja del paralelismo dinámico, lanzaremos cada pivote de manera independiente y dinámica, haciendo así más rápido nuestro ordenamiento.

Descripción

QuickSort avanzado con CUDA Dynamic Parallelism

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo demuestra un simple ordenamineto rápido usando CUDA Dynamic Parallelism. Requiere dispositivos con capacidad de cálculo 3.5 o superior.

Desarrollo

Implementación de un ordenamiento rápido paralelo en CUDA. Se presenta en varias partes:

1. Un pequeño conjunto de la ordenación por inserción. Esto lo hacemos en cualquier conjunto <= 32 elementos.

2. Un núcleo particionado, que - dado un pivote - separa una matriz de entrada en elementos <= pivote, y > pivote. Dos ordenamientos rápidos son lanzados para resolver cada uno de estos.

3. Un coordinador de ordenamiento rápido, que se da cuenta de los núcleos a lanzar y cuándo lanzarlos.

Cabeceras

```
#include <thrust/random.h>
#include <thrust/device_vector.h>
#include <helper_cuda.h>
#include <helper_string.h>
#include "cdpQuicksort.h"
```

Llama en línea PTX para devolver el índice de bit más alto no-cero en una palabra

```
static __device__ __forceinline__ unsigned int __qsflo(
unsigned int word)
{
    unsigned int ret;
    asm volatile("bfind.u32 %0, %1;" : "=r"(ret) : "r"(word));
    return ret;
}
```

ringbufAlloc

Asignación de un ringbuffer. Permite no fallar cuando nos quedamos sin pila para el seguimiento de las cuentas de compensación para cada subsección de ordenación.
Nosotros usamos el truco atomicMax para permitir fuera del orden de retiro. Si estamos en el límite de tamaño en el buffer circular, entonces esperamos la vuelta para completar.

```
template< typename T >
static __device__ T *ringbufAlloc(qsortRingbuf *ringbuf)
{
    // Espera para que haya espacio en el ringbuffer. Vamos a
    // recomprobar sólo un número determinado de veces y
    // luego fallamos, para evitar un punto muerto fuera de la
    // memoria.
    unsigned int loop = 10000;

    while (((ringbuf->head - ringbuf->tail) >=
        ringbuf->stacksize) && (loop-- > 0));

    if (loop == 0)
        return NULL;

    // Tenga en cuenta que el elemento incluye un índice
    // pequeño contable, para liberar más tarde.
    unsigned int index = atomicAdd(
        (unsigned int *)&ringbuf->head, 1);
    T *ret = (T *)(ringbuf->stackbase) +
        (index & (ringbuf->stacksize-1));
    ret->index = index;

    return ret;
}
```

ringBufFree

Libera un elemento del ringbuffer. Ahora si cada elemento se libera incluyendo el presente, podemos avanzar en la cola para indicar que el espacio está disponible.

```
template< typename T >
static __device__ void ringbufFree(qsortRingbuf *ringbuf,
T *data)
{
    // Liberar el índice no envuelto
```

```
    unsigned int index = data->index;
    unsigned int count = atomicAdd(
        (unsigned int *)&(ringbuf->count), 1) + 1;
    unsigned int max = atomicMax(
        (unsigned int *)&(ringbuf->max), index + 1);

    // Actualización de la cola si es necesario. Nota
    // actualizamos "max" para ser el nuevo valor
    // en ringbuf->max
    if (max < (index+1)) max = index+1;

    if (max == count)
        atomicMax((unsigned int *)&(ringbuf->tail), count);
}
```

qsort_warp

La implementación más sencilla posible, hace una ordenación rápida por sesgo sin comunicación entre sesgos. Esto tiene una alta tasa de emisión atómica, pero el resto debería ser en realidad bastante rápido debido a la baja de trabajo por hilo.

Un sesgo encuentra su sección de los datos, luego escribe todos los datos <pivote a un búfer y todos los datos >pivote para el otro. Atomics se utilizan para obtener una sección única de la memoria intermedia.

Optimización obvia: hacer varios trozos por sesgo, para aumentar cargas en vuelo y cubrir los gastos generales de instrucciones.

```
__global__ void qsort_warp(unsigned *indata,
                           unsigned *outdata,
                           unsigned int offset,
                           unsigned int len,
                           qsortAtomicData *atomicData,
                           qsortRingbuf *atomicDataStack,
                           unsigned int source_is_indata,
                           unsigned int depth)
{
    // Encuentra los datos compensados, basado en el ID del
```

```
// sesgo
unsigned int thread_id = threadIdx.x +
    (blockIdx.x << QSORT_BLOCKSIZE_SHIFT);
// Usa solo para depurar
// unsigned int warp_id = threadIdx.x >> 5;
unsigned int lane_id = threadIdx.x & (warpSize-1);

// Salir si estoy fuera del rango de ordenamiento
if (thread_id >= len)
    return;

// Primera parte del algoritmo. Cada sesgo cuenta el
// número de elementos que son mayores/menores que el
// pivote.

// Cuando un sesgo conoce su conteo, se actualiza un
// contador atómico.

// Leer los datos y el pivote. Selección de pivotes
// arbitraria por ahora.
unsigned pivot = indata[offset + len/2];
unsigned data  = indata[offset + thread_id];

// Cuenta cuántos son <= y cuántos son > pivote.
// Si todos son <= pivote entonces ajustamos la
// comparación porque de lo contrario no pasará
// nada y vamos a iterar siempre.
unsigned int greater = (data > pivot);
unsigned int gt_mask = __ballot(greater);

if (gt_mask == 0)
{
    greater = (data >= pivot);
    // Debe volver a votación en comparación ajustada
    gt_mask = __ballot(greater);
}

unsigned int lt_mask = __ballot(!greater);
```

```
unsigned int gt_count = __popc(gt_mask);
unsigned int lt_count = __popc(lt_mask);

// Ajustar atómicamente el lt_ y gt_ por esta cantidad.
// Sólo un hilo necesita hacer esto.
// Comparte el resultado con shfl
unsigned int lt_offset, gt_offset;

if (lane_id == 0)
{
    if (lt_count > 0)
        lt_offset = atomicAdd(
          (unsigned int *)&atomicData->lt_offset,lt_count);

    if (gt_count > 0)
        gt_offset = len - (atomicAdd(
          (unsigned int *)&atomicData->gt_offset,gt_count)+
          gt_count);
}

// Todos toman los offsets del lugar 0
lt_offset = __shfl((int)lt_offset, 0);
gt_offset = __shfl((int)gt_offset, 0);

__syncthreads();

// Ahora calcular desplazamiento dentro del propio
// offset. Es necesario saber cuántos hilos con un ID
// menos que el actual va a escribir en el mismo buffer.
// Podemos utilizar popc para implementar una
// exploración de sesgo, en este caso solo operación.

unsigned lane_mask_lt;
asm("mov.u32 %0, %%lanemask_lt;" : "=r"(lane_mask_lt));
unsigned int my_mask = greater ? gt_mask : lt_mask;
unsigned int my_offset = __popc(my_mask & lane_mask_lt);

// Mover datos.
```

```
my_offset += greater ? gt_offset : lt_offset;
outdata[offset + my_offset] = data;


// Cuenta si son el último sesgo. Si es así, entonces
// Kepler pondrá en marcha el próximo conjunto de clases
// directamente desde aquí.
if (lane_id == 0)
{
    // Cuenta  los "elementos escritos". Si escribiera el
    // último, y dispara los próximos qsort
    unsigned int mycount = lt_count + gt_count;

    if (atomicAdd((unsigned int *)&atomicData->
    sorted_count, mycount) + mycount == len)
    {
        // Son los últimos sesgos para hacer cualquier
        // clasificación. Por lo tanto, depende de
        // éstos para poner en marcha la siguiente
        // etapa.
        unsigned int lt_len = atomicData->lt_offset;
        unsigned int gt_len = atomicData->gt_offset;

        cudaStream_t lstream, rstream;
        cudaStreamCreateWithFlags(&lstream,
          cudaStreamNonBlocking);
        cudaStreamCreateWithFlags(&rstream,
          cudaStreamNonBlocking);

        // Comience por la liberación de nuestro
        // almacenamiento atomicData. Es mejor para el
        // algoritmo de ringbuffer si nos liberamos
        // cuando hayamos terminado, en lugar de volver
        // a utilizar (hace menos de fragmentación).
        ringbufFree<qsortAtomicData>(atomicDataStack,
          atomicData);

        // Caso excepcional: si "lt_len" es cero,
```

```
// entonces todos los valores en el lote son
// iguales.
// Entonces hemos terminado (sin embargo , puede
// ser necesario copiar en búfer correcto)
if (lt_len == 0)
{
    if (source_is_indata)
        cudaMemcpyAsync(indata+offset,
        outdata+offset, gt_len*sizeof(unsigned),
        cudaMemcpyDeviceToDevice, lstream);

    return;
}

// Comience con la mitad inferior primero
if (lt_len > BITONICSORT_LEN)
{
    // Si hemos superado el ancho máximo,
    // descender a través de la copia de seguridad
    // big_bitonicsort
    if (depth >= QSORT_MAXDEPTH)
    {
        // Los ordenamientos finales de etapa
        // bitónica en lugar de "OutData". Por lo
        // tanto, la reutilización "InData" como
        // el seguimiento fuera de rango.
        // Para (2^n)1 elementos que necesitamos
        // (2^(n+1)) bytes de memoria intermedia.
        // El buffer qsort copia de seguridad es
        // al menos tan grande cuando
        // sizeof (QTYPE)> = 2.
        big_bitonicsort<<< 1, BITONICSORT_LEN, 0,
          lstream >>>(outdata, source_is_indata ?
          indata : outdata, indata, offset, lt_len);
    }
    else
    {
        // Ínicie otro quicksort. Tenemos que
```

```
            // asignar más espacio de almacenamiento
            // para los datos atómicos.
            if ((atomicData = ringbufAlloc<
            qsortAtomicData>(atomicDataStack)) == NULL)
                printf("Stack-allocation error.
                  Failing left child launch.\n");
            else
            {
                atomicData->lt_offset = atomicData->
                 gt_offset=atomicData->sorted_count=0;
                unsigned int numblocks =
                  (unsigned int)(lt_len+
                  (QSORT_BLOCKSIZE-1)) >>
                  QSORT_BLOCKSIZE_SHIFT;
                qsort_warp<<< numblocks,
                  QSORT_BLOCKSIZE, 0, lstream >>>(
                  outdata, indata, offset, lt_len,
                  atomicData, atomicDataStack,
                  !source_is_indata, depth+1);
            }
        }
    }
    else if (lt_len > 1)
    {
        // Etapa final utiliza una especie de lugar
        // bitónico. Es importante asegurarse de que
        // la etapa final termina en el buffer
        // correcto (original).
        // Lanzamos el menor número de hilos que
        // podemos, potencias de 2.
        unsigned int bitonic_len = 1 << (__qsflo(
            lt_len-1U)+1);
        bitonicsort<<< 1, bitonic_len, 0, lstream >>>(
            outdata, source_is_indata ? indata :
            outdata, offset, lt_len);
    }
    // Por último, si ordenamos sólo un elemento
    // único, todavía debemos asegurarnos de que se
```

```
// termina en el lugar correcto.
else if (source_is_indata && (lt_len == 1))
    indata[offset] = outdata[offset];

if (cudaPeekAtLastError() != cudaSuccess)
    printf("Left-side launch fail: %s\n",
      cudaGetErrorString(cudaGetLastError()));


// Ahora la mitad superior.
if (gt_len > BITONICSORT_LEN)
{
    // Si hemos superado el ancho máximo,
    // descender a través de la copia de seguridad
    // big_bitonicsort
    if (depth >= QSORT_MAXDEPTH)
        big_bitonicsort<<< 1, BITONICSORT_LEN, 0,
         rstream >>>(outdata, source_is_indata ?
         indata : outdata, indata, offset+lt_len,
         gt_len);
    else
    {
        // Asignar un nuevo almacenamiento
        // atómico para este lanzamiento
        if ((atomicData = ringbufAlloc<
         qsortAtomicData>(atomicDataStack))==NULL)
            printf("Stack allocation error!
                Failing right-side launch.\n");
        else
        {
            atomicData->lt_offset = atomicData->
              gt_offset = atomicData->
              sorted_count = 0;
            unsigned int numblocks = (unsigned int
              )(gt_len+(QSORT_BLOCKSIZE-1)) >>
              QSORT_BLOCKSIZE_SHIFT;
            qsort_warp<<< numblocks,
              QSORT_BLOCKSIZE, 0, rstream >>>(
```

```
                        outdata, indata, offset+lt_len,
                        gt_len, atomicData, tomicDataStack,
                        !source_is_indata, depth+1);
                }
            }
        }
        else if (gt_len > 1)
        {
            unsigned int bitonic_len = 1 << (__qsflo(
              gt_len-1U)+1);
            bitonicsort<<< 1, bitonic_len, 0, rstream >>>(
              outdata, source_is_indata ? indata :
              outdata, offset+lt_len, gt_len);
        }
        else if (source_is_indata && (gt_len == 1))
            indata[offset+lt_len] = outdata[offset+lt_len];

        if (cudaPeekAtLastError() != cudaSuccess)
            printf("Right-side launch fail: %s\n",
                cudaGetErrorString(cudaGetLastError()));
    }
}
}
```

run_quicksort

Código para ejecutar la versión de Kepler de quicksort del lado del host. Es bastante simple, porque todo el control de lanzamiento se maneja en el dispositivo a través de CDP.

Todos los ordenamientos rápidos paralelos requieren un buffer de tamaño igual a cero. Esto se debe pasar antes de tiempo.

Devuelve el tiempo transcurrido para la especie.

```
float run_quicksort_cdp(unsigned *gpudata,
unsigned *scratchdata, unsigned int count, cudaStream_t stream)
{
    unsigned int stacksize = QSORT_STACK_ELEMS;
```

```
// Esta es la pila, para el seguimiento atómico del
// estado de cada ordenamiento.
qsortAtomicData *gpustack;
checkCudaErrors(cudaMalloc((void**)&gpustack, stacksize *
    sizeof(qsortAtomicData)));
// Establecer la primera entrada a 0
checkCudaErrors(cudaMemset(gpustack, 0,
    sizeof(qsortAtomicData)));

// Cree el ringbuffer de memoria utilizada para el manejo
// de la pila.
// Inicializar todo en donde tiene que estar.
qsortRingbuf buf;
qsortRingbuf *ringbuf;
checkCudaErrors(cudaMalloc((void **)&ringbuf,
    sizeof(qsortRingbuf)));
buf.head = 1;           // Comenzar con una asignación
buf.tail = 0;
buf.count = 0;
buf.max = 0;
buf.stacksize = stacksize;
buf.stackbase = gpustack;
checkCudaErrors(cudaMemcpy(ringbuf, &buf, sizeof(buf),
    cudaMemcpyHostToDevice));


// Eventos de temporización...
cudaEvent_t ev1, ev2;
checkCudaErrors(cudaEventCreate(&ev1));
checkCudaErrors(cudaEventCreate(&ev2));
checkCudaErrors(cudaEventRecord(ev1));

// Ahora trivialmente lanzamos el kernel qsort
if (count > BITONICSORT_LEN)
{
    unsigned int numblocks = (unsigned int)(count+
        (QSORT_BLOCKSIZE-1)) >> QSORT_BLOCKSIZE_SHIFT;
```

```
    qsort_warp<<<numblocks, QSORT_BLOCKSIZE, 0, stream>>>
      (gpudata, scratchdata, 0U, count, gpustack,
      ringbuf, true, 0);
}
else
{
    bitonicsort<<< 1, BITONICSORT_LEN >>>(gpudata,
      gpudata, 0, count);
}

checkCudaErrors(cudaGetLastError());
checkCudaErrors(cudaEventRecord(ev2));
checkCudaErrors(cudaDeviceSynchronize());

float elapse=0.0f;

if (cudaPeekAtLastError() != cudaSuccess)
    printf("Launch failure: %s\n",
        cudaGetErrorString(cudaGetLastError()));
else
    checkCudaErrors(cudaEventElapsedTime(&elapse,ev1,ev2));

// Comprobación de validez que el asignador de pila está
// haciendo lo correcto
checkCudaErrors(cudaMemcpy(&buf, ringbuf,
    sizeof(*ringbuf), cudaMemcpyDeviceToHost));

if (count > BITONICSORT_LEN && buf.head != buf.tail)
{
    printf("Stack allocation error!\nRingbuf:\n");
    printf("\t head = %u\n", buf.head);
    printf("\t tail = %u\n", buf.tail);
    printf("\tcount = %u\n", buf.count);
    printf("\t  max = %u\n", buf.max);
}

// Suelte los datos de la pila una vez que hemos terminado
checkCudaErrors(cudaFree(ringbuf));
```

```
    checkCudaErrors(cudaFree(gpustack));

    return elapse;
}

int run_qsort(unsigned int size, int seed, int debug,
int loop, int verbose)
{
    if (seed > 0)
        srand(seed);

    // Creación y configuración de la prueba
    unsigned *gpudata, *scratchdata;
    checkCudaErrors(cudaMalloc((void **)&gpudata,
        size*sizeof(unsigned)));
    checkCudaErrors(cudaMalloc((void **)&scratchdata,
        size*sizeof(unsigned)));

    // Creación de datos CPU.
    unsigned *data = new unsigned[size];
    unsigned int min = loop ? loop : size;
    unsigned int max = size;
    loop = (loop == 0) ? 1 : loop;

    for (size=min; size<=max; size+=loop)
    {
        if (verbose)
            printf(" Input: ");

        for (unsigned int i=0; i<size; i++)
        {
            // Construir datos de 8 bits a la vez
            data[i] = 0;
            char *ptr = (char *)&(data[i]);

            for (unsigned j=0; j<sizeof(unsigned); j++)
            {
                // Leer datos fácil en modo de depuración
```

```
            if (debug)
            {
                *ptr++ = (char)(rand() % 10);
                break;
            }

            *ptr++ = (char)(rand() & 255);
        }

        if (verbose)
        {
            if (i && !(i%32))
                printf("\n        ");

            printf("%u ", data[i]);
        }
    }

    if (verbose)
        printf("\n");

    checkCudaErrors(cudaMemcpy(gpudata, data,
        size*sizeof(unsigned), cudaMemcpyHostToDevice));

    // Así que está ahora completo y listo! Clasificamos
    // nuestro lanzamiento como bloques de hasta
    // BLOCKSIZE hilos, y tamaño de la cuadrícula
    // correspondiente.
    // Uno de los hilos es lanzado por elemento.
    float elapse;
    elapse = run_quicksort_cdp(gpudata, scratchdata,
             size, NULL);

    //run_bitonicsort<SORTTYPE>(gpudata, scratchdata,
    //size, verbose);
    checkCudaErrors(cudaDeviceSynchronize());

    // Copiar de nuevo los datos y verificar la clasificación
```

```
// correcta
checkCudaErrors(cudaMemcpy(data, gpudata,
    size*sizeof(unsigned), cudaMemcpyDeviceToHost));

if (verbose)
{
    printf("Output: ");

    for (unsigned int i=0; i<size; i++)
    {
        if (i && !(i%32)) printf("\n        ");

        printf("%u ", data[i]);
    }

    printf("\n");
}

unsigned int check;

for (check=1; check<size; check++)
{
    if (data[check] < data[check-1])
    {
        printf("FAILED at element: %d\n", check);
        break;
    }
}

if (check != size)
{
    printf("    cdp_quicksort FAILED\n");
    exit(EXIT_FAILURE);
}
else
    printf("    cdp_quicksort PASSED\n");

// Muestra el tiempo entre grabaciones de eventos
```

```
        printf("Sorted %u elems in %.3f ms (%.3f Melems/sec)
          \n", size, elapse, (float)size/(elapse*1000.0f));
        fflush(stdout);
    }

    // Soltar todo y listo
    checkCudaErrors(cudaFree(scratchdata));
    checkCudaErrors(cudaFree(gpudata));
    delete(data);
    return 0;
}

static void usage()
{
    printf("Syntax: qsort [-size=<num>] [-seed=<num>] [-debug]
      [-loop-step=<num>] [-verbose]\n");
    printf("If loop_step is non-zero, will run from 1->
      array_len in steps of loop_step\n");
}

// Host de entrada
int main(int argc, char *argv[])
{
    int size = 5000;     // TODO: make this 1e6
    unsigned int seed = 100;    // TODO: make this 0
    int debug = 0;
    int loop = 0;
    int verbose = 0;

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "h"))
    {
        usage();
        printf("&&&& cdp_quicksort WAIVED\n");
        exit(EXIT_WAIVED);
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "size"))
```

```
{
    size = getCmdLineArgumentInt(argc, (const char **)
                                 argv, "size");
}

if (checkCmdLineFlag(argc, (const char **)argv, "seed"))
{
    seed = getCmdLineArgumentInt(argc, (const char **)
                                 argv, "seed");
}

if (checkCmdLineFlag(argc, (const char **)argv,
                     "loop-step"))
{
    loop = getCmdLineArgumentInt(argc, (const char **)
                                 argv, "loop-step");
}

if (checkCmdLineFlag(argc, (const char **)argv,
                     "loop-step"))
{
    debug = 1;
}

if (checkCmdLineFlag(argc, (const char **)argv, "verbose"))
{
    verbose = 1;
}

// Recibir propiedades del dispositivo
int cuda_device = findCudaDevice(argc, (const char **)argv);
cudaDeviceProp properties;
checkCudaErrors(cudaGetDeviceProperties(&properties,
                                        cuda_device));
int cdpCapable = (properties.major == 3 &&
                  properties.minor >= 5) ||
                  properties.major >=4;
```

```
    printf("GPU device %s has compute capabilities (SM %d.%d)
     \n", properties.name, properties.major, properties.minor);

    if (!cdpCapable)
    {
        printf("cdpLUDecomposition requires SM 3.5 or higher
            to use CUDA Dynamic Parallelism.  Exiting...\n");
        exit(EXIT_SUCCESS);
    }

    printf("Running qsort on %d elements with seed %d, on
        %s\n", size, seed, properties.name);

    run_qsort(size, seed, debug, loop, verbose);
    checkCudaErrors(cudaDeviceReset());
    exit(EXIT_SUCCESS);
}
```

Código fuente

- cdpAdvancedQuicksort.cu
- cdpBitonicSort.cu
- cdpQuicksort.h
- findcudalib.mk
- Makefile

Compilar:
make

2.3. Imágenes

El manejo de imágenes es algo muy fácil, pues hay que recordar que las imágenes son matrices con valores para cada pixel. Con la programación en paralelo esto se convierte en un proceso mucho mas fácil, en lugar de esperar a que el programa haga tantos ciclos de reloj como pixeles de la imagen, es posible, incluso, procesar toda una imagen pequeña al mismo tiempo, o una imagen considerablemente grande en cuestión de milisegundos.

2.3.1. Convolución

Éste algoritmo es un filtro de convolución, consiste en formar matrices imaginarias de 3x3 y con estas afectar el valor centra mediante algunos cálculos matemáticos. Con una imagen grande irla segmentando en matrices de 3x3 sería una tarea muy tardada, pero con la programación en paralelo podremos trabajar con todas estas matrices de 3x3 al mismo tiempo teniendo un resultado mucho más rápido y a tiempo real.

Descripción

Con ayuda de la programación en parlelo veremos como implementar la manera de funcionar de éste algoritmo con el cómputo en paralelo y así tener un proceso más acelerado.

En la sección de anexos encontrarás el código completo.

Objetivos:

Implementar un filtro de convolución separable de una señal 2D con un núcleo gaussiano.

Desarrollo

Encabezados

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string.h>
#include <math.h>
#include <cutil_inline.h>
#include <helper_cuda.h>
#include <helper_timer.h>
#include <stopwatch.h>
#include <cmath>
#include "convolution_kernel.cu"

using namespace std;
```

Copia una textura 3D de un arrgelo del host (float*) a un cudaArray en el dispositivo. La medida se debe especificar con todas las dimensiones en unidades de *elementos*.

```
void prepareCudaTexture(float* h_src,
                        cudaArray *d_dst,
                        cudaExtent const texExtent);
```

Suponer que la memoria de destino ya ha sido asignada, nPixels es impar.

```
void createGaussian1D(float* targPtr,
                      int    nPixels,
                      float  sigma,
                      float  ctr=0.0f);
```

Suponer que la memoria de destino ya ha sido asignada, nPixels es impar.

```
void createGaussian2D(float* targPtr,
                      int    nPixelsCol,
                      int    nPixelsRow,
                      float  sigmaCol,
                      float  sigmaRow,
                      float  ctrCol=0.0f,
                      float  ctrRow=0.0f);
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada.

```
void createBinaryCircle(float* targPtr,
                        int&   seNonZero,
                        int    diameter);
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada.

```
void createBinaryCircle(int* targPtr
                        int& seNonZero
                        int  diameter);
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada
Este filtro se utiliza para la detección de bordes. Convoluciona con el núcleo creado por esta función y, a continuación, busca los cruces con cero.
Como siempre, esperamos un diámetro impar. Para los núcleos del registro, asumimos que son cuadrados y simétricos, por lo que no hay opciones para diferentes dimensiones.

```
void createLaplacianOfGaussianKernel(float* targPtr,
                                     int    diameter);
```

Programa principal
TODO: Retire las llamadas a CUTIL, libcutil no se requiere para compilar/ejecutar.

```
int main( int argc, char** argv)
{
   cout << endl << "Executing GPU-accelerated convolution..."
        << endl;

   ////////////////////////////////////////////////////////
   // Consultar los dispositivos en el sistema y seleccionar el
   // más rápido
   int deviceCount = 0;
   if (cudaGetDeviceCount(&deviceCount) != cudaSuccess)
   {
      cout << "cudaGetDeviceCount() FAILED." << endl;
      cout << "CUDA Driver and Runtime version may be
               mismatched.\n";
      return -1;
   }

   // Asegurar que hay por lo menos un dispositivo
   // CUDA
   if( deviceCount == 0)
   {
      cout << "No CUDA devices available.  Exiting." << endl;
      return -1;
  }

   // Seleccionar automáticamente el dispositivo más rápido
   // Se puede anular con lo siguiente
   //int fastestDeviceID = cutGetMaxGflopsDeviceId() ;
   int fastestDeviceID = 0;
   cudaSetDevice(fastestDeviceID);

   cudaDeviceProp gpuProp;
   cout << "CUDA-enabled devices on this system:  "
```

```
     << deviceCount <<  endl;
for(int dev=0; dev<deviceCount; dev++)
{
   cudaGetDeviceProperties(&gpuProp, dev);
   char* devName = gpuProp.name;
   int mjr = gpuProp.major;
   int mnr = gpuProp.minor;
   if( dev==fastestDeviceID )
      cout << "\t* ";
   else
      cout << "\t  ";

   printf("(%d) %20s \tCUDA Capability %d.%d\n", dev,
          devName, mjr, mnr);
}
// Fin de la consulta/selección del dispositivo CUDA
////////////////////////////////////////////////////////


// Se concede. Hacer todo en COL-MAJOR
// Utilizar 17x17 porque es más grande que el tamaño del bloque,
// por lo que hace hincapié en la COPY_LIN_ARRAY_TO_SHMEM
// macro de bucle
unsigned int imgW  = 512;
unsigned int imgH  = 512;
unsigned int psfW  = 3;
unsigned int psfH  = 3;
unsigned int nPix  = imgH*imgW;
unsigned int nPsf  = psfW*psfH;
// Se hubiera esperado que 32x8 reduzca los conflictos de
// banco, pero 8x32 es un 30% más rápido, y ambos son mucho
// más rápidos que 16x16
unsigned int blockDimX = 8;               // X ~ COL
unsigned int blockDimY = 32;              // Y ~ ROW
unsigned int gridDimX = imgW/blockDimX;   // X ~ COL
unsigned int gridDimY = imgH/blockDimY;   // Y ~ ROW

unsigned int imgBytes = nPix*FLOAT_SZ;
```

```
unsigned int psfBytes = nPsf*FLOAT_SZ;

cout << endl;
printf("Executing convolution on %dx%d
        image with %dx%d PSF\n", imgW,imgH,psfW,psfH);

// Alojar memomoria del lado del host
float* imgIn  = (float*)malloc(imgBytes);
float* imgOut = (float*)malloc(imgBytes);
float* imgPsf = (float*)malloc(psfBytes);


// Leer la imagen [muy grande]
// Los datos se almacenan en la fila principal, por lo que
// hay que invertir el bucle para leer correctamente col-major,
// por lo que hay invertir el orden de los bucles
ifstream fpIn( "salt512.txt", ios::in);
cout << "Reading image from file..." << endl;
for(int r=0; r<imgH; r++)
   for(int c=0; c<imgH; c++)
       fpIn >> imgIn[c*imgH+r];
fpIn.close();


// Leer el PSF de un archivo
// Aquí se utiliza un PSF altamente asimétrico para que
// podamos verificar coordinar sistemas. PSF se lee como
// datos de las filas, pero nosotros hacemos todo nuestro
// procesamiento en col-major, por lo que hay que invertir
// el orden de los bucles

//createGaussian2D(imgPsf, psfW, psfH, (float)psfW/5.5,
                   (float)psfH/5.5);
int seNonZero;
createBinaryCircle(imgPsf, seNonZero, psfW);
cout << " SE Non Zero Elements: " << seNonZero << endl;

// Escribir el PSF por lo que puede ser comprobado más tarde
```

```
ofstream psfout("psf.txt", ios::out);
cout << endl << "Point Spread Function:" << endl;
for(int r=0; r<psfH; r++)
{
   cout << "\t";
   for(int c=0; c<psfW; c++)
   {
      printf("%0.3f ", imgPsf[c*psfH+r]);
      psfout << imgPsf[c*psfH+r] << " ";
   }
   cout << endl;
   psfout << endl;
}
cout << endl;

// Alojar memoria en el dispositivo y copiar los datos ahí
float* devIn;
float* devOut;
float* devPsf;
cudaMalloc((void**)&devIn,  imgBytes);
cudaMalloc((void**)&devOut, imgBytes);
cudaMalloc((void**)&devPsf, psfBytes);

dim3  GRID(  gridDimX,  gridDimY,   1);
dim3  BLOCK( blockDimX, blockDimY,  1);
printf("Grid  Dimensions: (%d, %d)\n", gridDimX,  gridDimY);
printf("Block Dimensions: (%d, %d)\n\n", blockDimX,
       blockDimY);

// GPU Funciones de tiempo
//unsigned int timer = 0;
//cutilCheckError( cutCreateTimer( &timer));
//cutilCheckError( cutStartTimer( timer));

cudaMemcpy(devIn,  imgIn,   imgBytes,
           cudaMemcpyHostToDevice);
cudaMemcpy(devPsf, imgPsf,  psfBytes,
           cudaMemcpyHostToDevice);
```

```
// Configurar geometría de ejecución del kernel
// *****************************************************
// Los datos están en el host, hacer un cálculo completo de
// ida y vuelta con copias de memoria
cout << "Data loaded into HOST mem, executing kernel..."
     << endl;
kernelDilate<<<GRID, BLOCK>>>(devIn, devOut, devPsf,
                              imgW, imgH, psfW/2, psfH/2,
                              seNonZero);
// Comprobar si la ejecución del kernel no falló
//cutilCheckMsg("Kernel execution failed");

cudaThreadSynchronize();

// Copiar resultados del dispositivo al host
cudaMemcpy(imgOut, devOut, nPix*sizeof(float),
           cudaMemcpyDeviceToHost);

//cutilCheckError( cutStopTimer( timer));
//float gpuTime_w_copy = cutGetTimerValue(timer);
//cutilCheckError( cutDeleteTimer( timer));
// *****************************************************


// *****************************************************
// Con los datos que ya están en el dispositivo, el tiempo
// de los cálculos está sobre 100 corridas
int NITER = 20;
printf("Data already on DEVICE, running %d times...\n",
       NITER);
//cutilCheckError( cutCreateTimer( &timer));
//cutilCheckError( cutStartTimer( timer));
for(int i=0; i<NITER; i++)
{
   kernelDilate<<<GRID, BLOCK>>>(devIn, devOut, devPsf,
                              imgW, imgH, psfW/2, psfH/2,
                              seNonZero);
```

```
   // Comprobar si la ejecución del kernel no falló
   //cutilCheckMsg("Kernel execution failed");

   cudaThreadSynchronize();
}
//cutilCheckError( cutStopTimer( timer));
//float gpuTime_compute_only = cutGetTimerValue(timer)/
                               (float)NITER;
//cutilCheckError( cutDeleteTimer( timer));
// ******************************************************

//float memCopyTime = gpuTime_w_copy -
                      gpuTime_compute_only;
//float cpySpeed = 2.0 * (float)imgBytes /
                  (float)(memCopyTime/1000. * 1024 * 1024);
cout << "Final Timing Results:" << endl << endl;

printf("\t-----------------------------------------\n");

// La escritura de datos de E/S en un archivo, para que pueda
// comprobarlo con MATLAB
// Una vez más, los datos se almacenan en col-major, pero los
// archivos son r/w en row-major, así que invierta los bucles
cout << "Writing before/after image to file..." << endl;
ofstream fpOrig("origImage.txt", ios::out);
ofstream fpOut("gpu_solution.txt", ios::out);
// Escribe fpOut
for(int r=0; r<imgH; r++)
{
   for(int c=0; c<imgW; c++)
   {
      fpOrig << imgIn[c*imgW+r] << " ";
      fpOut << imgOut[c*imgW+r] << " ";
   }
   fpOrig << endl;
   fpOut << endl;
}
fpOrig.close();
```

```
   fpOut.close();

   // Liberar memoria
   free(imgIn);
   free(imgOut);
   free(imgPsf);
   //free(imgPsfIntens);
   cudaFree(devIn);
   cudaFree(devOut);
   cudaFree(devPsf);

   cudaThreadExit();

   //cutilExit(argc, argv);
}
```

Suponer que la memoria de destino ya ha sido asignada, nPixels es impar.

```
void createGaussian1D(float* targPtr,
                      int    nPixels,
                      float  sigma,
                      float  ctr)
{
   if(nPixels%2 != 1)
   {
      cout << "***Warning: createGaussian(...) only defined
               for odd pixel"  << endl;
      cout << "            dimensions.  Undefined behavior
               for even sizes." << endl;
   }
   float pxCtr = (float)(nPixels/2 + ctr);
   float sigmaSq = sigma*sigma;
   float denom = sqrt(2*M_PI*sigmaSq);
   float dist;
   for(int i=0; i<nPixels; i++)
   {
```

```
      dist = (float)i - pxCtr;
      targPtr[i] = exp(-0.5 * dist * dist / sigmaSq) / denom;
   }
}
```

Suponer que la memoria de destino ya ha sido asignada, nPixels es impar. Usar col-row (D00_UL_ES)

```
void createGaussian2D(float* targPtr,
                      int    nPixelsCol,
                      int    nPixelsRow,
                      float  sigmaCol,
                      float  sigmaRow,
                      float  ctrCol,
                      float  ctrRow)
{
   if(nPixelsCol%2 != 1 || nPixelsRow != 1)
   {
      cout << "***Warning: createGaussian(...) only defined
               for odd pixel"  << endl;
      cout << "            dimensions.  Undefined behavior
               for even sizes." << endl;
   }

   float pxCtrCol = (float)(nPixelsCol/2 + ctrCol);
   float pxCtrRow = (float)(nPixelsRow/2 + ctrRow);
   float distCol, distRow, distColSqNorm, distRowSqNorm;
   float denom = 2*M_PI*sigmaCol*sigmaRow;
   for(int c=0; c<nPixelsCol; c++)
   {
      distCol = (float)c - pxCtrCol;
      distColSqNorm = distCol*distCol / (sigmaCol*sigmaCol);
      for(int r=0; r<nPixelsRow; r++)
      {
         distRow = (float)r - pxCtrRow;
         distRowSqNorm = distRow*distRow / (sigmaRow*sigmaRow);
```

```
         targPtr[c*nPixelsRow+r] = exp(-0.5*(distColSqNorm +
                                   distRowSqNorm)) / denom;
      }
   }
}
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada.

```
void createBinaryCircle(float* targPtr,
                        int&   seNonZero,
                        int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float rad;
   seNonZero = 0;
   for(int c=0; c<diameter; c++)
   {
      for(int r=0; r<diameter; r++)
      {
         rad = sqrt((c-pxCtr)*(c-pxCtr) + (r-pxCtr)*(r-pxCtr));
         if(rad <= pxCtr+0.5)
         {
            targPtr[c*diameter+r] = 1.0f;
            seNonZero++;
         }
         else
         {
            targPtr[c*diameter+r] = 0.0f;
         }
      }
   }
}
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada.

```
void createBinaryCircle(int*   targPtr,
                        int&   seNonZero,
                        int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float rad;
   for(int c=0; c<diameter; c++)
   {
      for(int r=0; r<diameter; r++)
      {
         rad = sqrt((c-pxCtr)*(c-pxCtr) + (r-pxCtr)*(r-pxCtr));
         if(rad <= pxCtr+0.5)
            targPtr[c*diameter+r] = 1;
         else
            targPtr[c*diameter+r] = 0;
      }
   }
}
```

Suponer que la memoria de destino de $diameter^2$ ya ha sido asignada
Este filtro se utiliza para la detección de bordes. Convoluciona con el núcleo creado por esta función y, a continuación, busca los cruces con cero.
Como siempre, esperamos un diámetro impar. Para los núcleos del registro, siempre asumimos que son cuadrados y simétricos, por lo que no hay opciones para diferentes dimensiones.

```
void createLaplacianOfGaussianKernel(float* targPtr,
                                     int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float dc, dr, dcSq, drSq;
   float sigma = diameter/10.0f;
   float sigmaSq = sigma*sigma;
   for(int c=0; c<diameter; c++)
   {
      dc = (float)c - pxCtr;
```

```
      dcSq = dc*dc;
      for(int r=0; r<diameter; r++)
      {
         dr = (float)r - pxCtr;
         drSq = dr*dr;

         float firstTerm  = (dcSq + drSq - 2*sigmaSq) /
                            (sigmaSq * sigmaSq);
         float secondTerm = exp(-0.5 * (dcSq + drSq) /
                            sigmaSq);
         targPtr[c*diameter+r] = firstTerm * secondTerm;
      }
   }
}
void prepareCudaTexture(float* h_src,
                        cudaArray *d_dst,
                        cudaExtent const texExtent)
{
   cudaMemcpy3DParms copyParams = {0};
   cudaPitchedPtr cppImgPsf =
      make_cudaPitchedPtr((void*)h_src,
      texExtent.width*FLOAT_SZ, texExtent.width,
      texExtent.height);
   copyParams.srcPtr   = cppImgPsf;
   copyParams.dstArray = d_dst;
   copyParams.extent   = texExtent;
   copyParams.kind     = cudaMemcpyHostToDevice;
   checkCudaErrors( cudaMemcpy3D(&copyParams) );
}
```

Código fuente

- convolution.zip

Compilar:
make

2.3.2. Disparidad éstero (Visión éstereo)

La Visión Éstero o Disparidad Éstereo es una técnica utilizada entre otras cosas, para estimar la distancia a los objetos, utilizando dos cámaras separadas, tratando de imitar la visión humana. El problema es la gran cantidad de recursos de cómputo que implican los cálculos necesarios en el tratamiento de las imágenes y fórmulas para obtener el resultado. Esto vuelve a las aplicaciones convencionales que emplean estos algoritmos, poco funcionales, pues requieren mucho tiempo de procesamiento donde se necesita obtener resultados en tiempo real.

Descripción

Un programa CUDA que muestra cómo calcular un mapa de disparidad estéreo utilizando SIMD SAD (Suma de diferencia absoluta) intrínsecos

En la sección de anexos encontrarás el código completo.

Objetivos:

Demostrar el funcionamiento de un algoritmo de Visión Estéreo que utilice paralelización con CUDA y observar su rendimiento. Obtener un performance de procesamiento cercano al tiempo real.

Desarrollo

En el caso del hombre y muchos otros animales, el cerebro percibe la realidad tridimensional a través de un sistema de visión, que consta de dos ojos desplazados ligeramente uno respecto del otro en un plano horizontal.

Basados en este hecho los sistemas artificiales de visión, cuentan de dos cámaras desplazadas entre sí una cierta distancia B, gracias a lo cual se puede

recuperar la tercera dimensión y la profundidad perdidas en las imágenes bidimensionales, calculando:

$$D = Bf/d$$

donde B es la distancia entre las cámaras, f es la longitud focal de las cámaras y d es la disparidad; que está dada por la diferencia de ubicación de un punto en ambas imágenes.

El principio es que cada cámara toma la imagen con un ángulo ligeramente distinto, donde los objetos más cercanos, tendrán una diferencia de ángulo con respecto a cada sensor, mayor, que los objetos más alejados.

La correspondencia de imágenes es un proceso mediante el cual se trata de identificar la misma característica espacial (matriz o máscara de N x N pixeles) en ambas imágenes del sistema estéreo. Este proceso que resulta sencillo para el humano, es enormente complejo en términos de computación, puesto que para un píxel o característica en una imagen pueden existir multitud de candidatos en la otra.

Como las cámaras que forman el sistema estéreo tienen ángulos diferentes, el mismo punto se proyecta en localizaciones diferentes; lo que se busca es encontrar el pico más alto de correspondencia para identificar el mismo punto en el espacio que corresponde para cada píxel de la imagen izquierda en la imagen derecha como podemos ver en la Figura 2.6.

Figura 2.6: Imágenes a tomar para probar la disparidad.

Suma de diferencia de cuadrados (SSD)

Existen diferentes métodos para encontrar la correspondencia, en este algoritmo se emplea la diferencia de cuadrados.

El éxito de los métodos basados en correlación depende de que la ventana utilizada capture la estructura que rodea al punto que se está estudiando para establecer la correspondencia.

¿Cómo elegir el tamaño de la ventana? Una ventana demasiado pequeña puede no capturar suficiente estructura de la imagen, y puede ser demasiado sensible al ruido. Una ventana demasiado grande es menos sensible al ruido, pero a cambio está sujeta a posibles variaciones de intensidad entre las dos imágenes del par estereoscópico.

Realizar la correlación sobre todos y cada uno de los píxeles de la imagen homóloga requiere un elevado tiempo de computación. Para mejorar el tiempo de respuesta de la correlación añadimos la restricción epipolar, que consiste en limitar la búsqueda sobre el epipolo de la imagen homóloga. El epipolo es el resultado de la proyección en la cámara homóloga de la línea de proyección que causó el píxel a emparejar.

De tal suerte, si se toma por ejemplo un punto de referencia en la imagen izquierda, se buscará su correspondencia en la imagen derecha, recorriendo hacia la izquierda en el mismo eje x a partir de la ubicación del punto de referencia.

En la Figura 2.7 podrémos ver la imagen resultante.

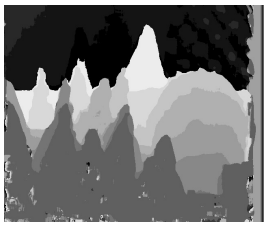

Figura 2.7: Imagen resultante de la disparidad.

Algoritmo

Entradas:

- Imagen izquierda (Ii) e imagen derecha (Id)
- Anchura de la ventana N, siendo N=2W+1.
- Región de búsqueda en la imagen derecha R(pi) asociado con un píxel pi en la imagen izquierda

Para cada píxel pi = (i, j) en la imagen izquierda:

- Para cada desplazamiento ( ) ( ) i d = d d Î R p 1 2 , calcular la similitud entre la ventana en la imagen izquierda y la ventana en la imagen derecha (c(d))
- La disparidad de pi es el vector ( ) 1 2 d = d ,d que maximiza c(d) sobre R(pd) d argmax [c(d)]

Librerías del sistema.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
```

Librerías de núcleos.

```
#include <cuda_runtime.h>
#include "stereoDisparity_kernel.cuh"
```

Librerías del proyecto.

```
#include <helper_functions.h>
// Ayudante para compartir los que son comunes a los ejemplos de
// CUDA SDK
#include <helper_cuda.h>
// Ayudante para comprobar inicialización cuda y comprobación de
// errores
#include <helper_string.h>
// Funciones de ayuda para la conversión de cadenas


static char *sSDKsample = "[stereoDisparity]\0";

int iDivUp(int a, int b){
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}
```

Declaraciones adelantadas.

```
void runTest(int argc, char **argv);
```

Programa principal.

```
int main(int argc, char **argv){
    runTest(argc, argv);
}
```

Muestra de CUDA para el cálculo de mapas de profundidad.

```
void runTest(int argc, char **argv){
    cudaDeviceProp deviceProp;
```

```
deviceProp.major = 0;
deviceProp.minor = 0;
int dev = 0;

// Elejir el mejor dispositivo CUDA
dev = findCudaDevice(argc, (const char **)argv);

checkCudaErrors(cudaGetDeviceProperties(&deviceProp,dev));

// Estadísticas sobre el dispositivo GPU
printf("> GPU device has %d Multi-Processors, SM %d.%d
        compute capabilities\n\n",
       deviceProp.multiProcessorCount, deviceProp.major,
       deviceProp.minor);

int version=(deviceProp.major * 0x10 + deviceProp.minor);

if (version < 0x20){
    printf("%s: requires a minimum CUDA compute 2.0
    capability\n",
           sSDKsample);
    exit(EXIT_SUCCESS);
}

StopWatchInterface *timer;
sdkCreateTimer(&timer);

// Buscar parámetros
int minDisp = -16;
int maxDisp = 0;

// Cargar datos de la imagen, asignar memoria para las
// imágenes de host, inicializar punteros a NULL para
// solicitar la llamada a lib y asignar según sea necesario
// Las imágenes PPM se cargan a 4 bytes/píxel de memoria
// (RGBX)
unsigned char *h_img0 = NULL;
unsigned char *h_img1 = NULL;
```

```
unsigned int w, h;
char *fname0 = sdkFindFilePath("stereo.im0.640x533.ppm",
                               argv[0]);
char *fname1 = sdkFindFilePath("stereo.im1.640x533.ppm",
                               argv[0]);

printf("Loaded <%s> as image 0\n", fname0);

if (!sdkLoadPPM4ub(fname0, &h_img0, &w, &h)){
    fprintf(stderr, "Failed to load <%s>\n", fname0);
}

printf("Loaded <%s> as image 1\n", fname1);

if (!sdkLoadPPM4ub(fname1, &h_img1, &w, &h)){
    fprintf(stderr, "Failed to load <%s>\n", fname1);
}

dim3 numThreads = dim3(blockSize_x, blockSize_y, 1);
dim3 numBlocks = dim3(iDivUp(w, numThreads.x), iDivUp(h,
                      numThreads.y));
unsigned int numData = w*h;
unsigned int memSize = sizeof(int) * numData;

// Asignar memoria para el resultado en la parte de host
unsigned int *h_odata = (unsigned int *)malloc(memSize);

// Inicializar la memoria
for (unsigned int i = 0; i < numData; i++)
    h_odata[i] = 0;

// Asignar memoria del dispositivo para el resultado
unsigned int *d_odata, *d_img0, *d_img1;

checkCudaErrors(cudaMalloc((void **) &d_odata, memSize));
checkCudaErrors(cudaMalloc((void **) &d_img0, memSize));
checkCudaErrors(cudaMalloc((void **) &d_img1, memSize));
```

```
// Copiar la memoria del host al dispositivo para
// inicializar a cero
checkCudaErrors(cudaMemcpy(d_img0,  h_img0, memSize,
                cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_img1,  h_img1, memSize,
                cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_odata, h_odata, memSize,
                cudaMemcpyHostToDevice));

size_t offset = 0;
cudaChannelFormatDesc ca_desc0 = cudaCreateChannelDesc
                                 <unsigned int>();
cudaChannelFormatDesc ca_desc1 = cudaCreateChannelDesc
                                 <unsigned int>();

tex2Dleft.addressMode[0] = cudaAddressModeClamp;
tex2Dleft.addressMode[1] = cudaAddressModeClamp;
tex2Dleft.filterMode     = cudaFilterModePoint;
tex2Dleft.normalized     = false;
tex2Dright.addressMode[0] = cudaAddressModeClamp;
tex2Dright.addressMode[1] = cudaAddressModeClamp;
tex2Dright.filterMode     = cudaFilterModePoint;
tex2Dright.normalized     = false;
checkCudaErrors(cudaBindTexture2D(&offset, tex2Dleft,
                                  d_img0, ca_desc0, w,
                                  h, w*4));
assert(offset == 0);

checkCudaErrors(cudaBindTexture2D(&offset, tex2Dright,
                                  d_img1, ca_desc1, w,
                                  h, w*4));
assert(offset == 0);

// First run the warmup kernel (which we'll use to get
// the GPU in the correct max power state
// En primer lugar ejecutar el núcleo de calentamiento (que vamos a util
stereoDisparityKernel<<<numBlocks, numThreads>>>
    (d_img0, d_img1, d_odata, w, h, minDisp, maxDisp);
```

```
cudaDeviceSynchronize();

// Allocate CUDA events that we'll use for timing
cudaEvent_t start, stop;
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));

printf("Launching CUDA stereoDisparityKernel()\n");

// Record the start event
checkCudaErrors(cudaEventRecord(start, NULL));

// launch the stereoDisparity kernel
stereoDisparityKernel<<<numBlocks, numThreads>>>
    (d_img0, d_img1, d_odata, w, h, minDisp, maxDisp);

// Record the stop event
checkCudaErrors(cudaEventRecord(stop, NULL));

// Wait for the stop event to complete
checkCudaErrors(cudaEventSynchronize(stop));

// Check to make sure the kernel didn't fail
getLastCudaError("Kernel execution failed");

float msecTotal = 0.0f;
checkCudaErrors(cudaEventElapsedTime(&msecTotal, start,
                                     stop));

//Copy result from device to host for verification
checkCudaErrors(cudaMemcpy(h_odata, d_odata, memSize,
                cudaMemcpyDeviceToHost));

printf("Input Size  [%dx%d], ", w, h);
printf("Kernel size [%dx%d], ", (2*RAD+1), (2*RAD+1));
printf("Disparities [%d:%d]\n", minDisp, maxDisp);

printf("GPU processing time : %.4f (ms)\n", msecTotal);
```

```
printf("Pixel throughput    : %.3f Mpixels/sec\n",
       ((float)(w *h*1000.f)/msecTotal)/1000000);

// calculate sum of resultant GPU image
unsigned int checkSum = 0;

for (unsigned int i=0 ; i<w *h ; i++){
    checkSum += h_odata[i];
}
printf("GPU Checksum = %u, ", checkSum);

// write out the resulting disparity image.
unsigned char *dispOut = (unsigned char *)malloc(numData);
int mult = 20;
char *fnameOut = "output_GPU.pgm";

for (unsigned int i=0; i<numData; i++){
    dispOut[i] = (int)h_odata[i]*mult;
}
printf("GPU image: <%s>
\n", fnameOut);
sdkSavePGM(fnameOut, dispOut, w, h);

//compute reference solution
printf("Computing CPU reference...\n");
cpu_gold_stereo((unsigned int *)h_img0, (unsigned int *)
                h_img1, (unsigned int *)h_odata, w, h,
                minDisp, maxDisp);
unsigned int cpuCheckSum = 0;

for (unsigned int i=0 ; i<w *h ; i++){
    cpuCheckSum += h_odata[i];
}
printf("CPU Checksum = %u, ", cpuCheckSum);
char *cpuFnameOut = "output_CPU.pgm";

for (unsigned int i=0; i<numData; i++){
    dispOut[i] = (int)h_odata[i]*mult;
```

```
    }
    printf("CPU image: <%s>
    \n", cpuFnameOut);
    sdkSavePGM(cpuFnameOut, dispOut, w, h);

    // cleanup memory
    checkCudaErrors(cudaFree(d_odata));
    checkCudaErrors(cudaFree(d_img0));
    checkCudaErrors(cudaFree(d_img1));

    if (h_odata != NULL) free(h_odata);
    if (h_img0 != NULL) free(h_img0);
    if (h_img1 != NULL) free(h_img1);
    if (dispOut != NULL) free(dispOut);
    sdkDeleteTimer(&timer);

    cudaDeviceReset();

    exit((checkSum == cpuCheckSum)?EXIT_SUCCESS:EXIT_FAILURE);
}
```

Código fuente

- stereoDisparity.zip

Compilar:
make

2.3.3. CUDA FFT Simulación de oceano.

Simulación de un pedazo de océano, con ayuda de la librería FFT, pero no la normal, sino que es una librería ya pasada a CUDA, llamada CUFFT, con ella es posible realizar la misma cantidad de cálculos que con la original, pero de manera paralela.

Descripción

Simulación de una ola del océano.

En la sección de anexos encontrarás el código completo.

Objetivos:

Este ejemplo simula una ola del océano utilizando la librería CUFFT y hace que el resultado se renderize con OpenGL.

Desarrollo

```
#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  define NOMINMAX
#  include <windows.h>
#endif
```

Includes:

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <math.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <cufft.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#if defined(__APPLE__) |
| defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

#include <rendercheck_gl.h>

const char *sSDKsample = "CUDA FFT Ocean Simulation";

#define MAX_EPSILON 0.10f
#define THRESHOLD   0.15f
#define REFRESH_DELAY     10 //ms
```

Constantes:

```
unsigned int windowW = 512, windowH = 512;

const unsigned int meshSize = 256;
const unsigned int spectrumW = meshSize + 4;
const unsigned int spectrumH = meshSize + 1;
```

```
const int frameCompare = 4;
```

Buffers de vértices OpenGL:

```
GLuint posVertexBuffer;
GLuint heightVertexBuffer, slopeVertexBuffer;
struct cudaGraphicsResource *cuda_posVB_resource,
       *cuda_heightVB_resource, *cuda_slopeVB_resource;
       // maneja el intercambio de OpenGL con CUDA

GLuint indexBuffer;
GLuint shaderProg;
char *vertShaderPath = 0, *fragShaderPath = 0;
```

Controles del ratón:

```
int mouseOldX, mouseOldY;
int mouseButtons = 0;
float rotateX = 20.0f, rotateY = 0.0f;
float translateX=0.0f, translateY = 0.0f, translateZ = -2.0f;

bool animate = true;
bool drawPoints = false;
bool wireFrame = false;
bool g_hasDouble = false;
```

Datos FFT:

```
cufftHandle fftPlan;
float2 *d_h0 = 0;   // campo de altura en el tiempo 0
float2 *h_h0 = 0;
float2 *d_ht = 0;   // campo de altura en el tiempo t
float2 *d_slope = 0;
```

Punteros a objeto dispositivo:

```
float *g_hptr = NULL;
float2 *g_sptr = NULL;
```

Parámetros de simulación:

```
const float g = 9.81f;          // constante gravitacional
const float A = 1e-7f;          // factor de escala de la ola
const float patchSize = 100;    // tamaño del parche
float windSpeed = 100.0f;
float windDir = CUDART_PI_F/3.0f;
float dirDepend = 0.07f;

StopWatchInterface *timer = NULL;
float animTime = 0.0f;
float prevTime = 0.0f;
float animationRate = -0.001f;
```

Código de Auto-Verificación:

```
const int frameCheckNumber = 4;
int fpsCount = 0;        // Recuento de FPS para promediar
int fpsLimit = 1;        // Límite de FPS para el muestreo
unsigned int frameCount = 0;
unsigned int g_TotalErrors = 0;
```

Kernels:

```
#include <oceanFFT_kernel.cu>

extern "C"
```

```
void cudaGenerateSpectrumKernel(float2 *d_h0,
                                float2 *d_ht,
                                unsigned int in_width,
                                unsigned int out_width,
                                unsigned int out_height,
                                float animTime,
                                float patchSize);

extern "C"
void cudaUpdateHeightmapKernel(float  *d_heightMap,
                               float2 *d_ht,
                               unsigned int width,
                               unsigned int height);

extern "C"
void cudaCalculateSlopeKernel(float *h, float2 *slopeOut,
                              unsigned int width, unsigned
                              int height);
```

Declaraciones adelantadas:

```
void runAutoTest(int argc, char **argv);
void runGraphicsTest(int argc, char **argv);
```

Funcionalidad GL:

```
bool initGL(int *argc, char **argv);
void createVBO(GLuint *vbo, int size);
void deleteVBO(GLuint *vbo);
void createMeshIndexBuffer(GLuint *id, int w, int h);
void createMeshPositionVBO(GLuint *id, int w, int h);
GLuint loadGLSLProgram(const char *vertFileName,
                       const char *fragFileName);
```

Renderizando llamadas de vuelta:

```
void display();
void keyboard(unsigned char key, int x, int y);
void mouse(int button, int state, int x, int y);
void motion(int x, int y);
void reshape(int w, int h);
void timerEvent(int value);
```

Funcionalidad Cuda:

```
void runCuda();
void runCudaTest(bool bHasDouble, char *exec_path);
void generate_h0(float2 *h0);
void generateFftInput();
```

Programa principal:

```
int main(int argc, char **argv)
{
    // Comprobar si hay argumentos de la línea de comandos
    if (checkCmdLineFlag(argc, (const char **)argv,"qatest"))
    {
        animate       = false;
        fpsLimit = frameCheckNumber;
        runAutoTest(argc, argv);
    }
    else
    {
        printf("[%s]\n\n"
               "Left mouse button          - rotate\n"
               "Middle mouse button        - pan\n"
               "Right mouse button         - zoom\n"
               "'w' key                    - toggle
               wireframe\n", sSDKsample);
```

```
        runGraphicsTest(argc, argv);
    }

    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}
```

Ejecute la prueba:

```
void runAutoTest(int argc, char **argv)
{
    printf("%s Starting...\n\n", argv[0]);

    // Cuda inicialización
    int dev = findCudaDevice(argc, (const char **)argv)};

    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp,dev));
    printf("Compute capability %d.%d\n", deviceProp.major,
                                         deviceProp.minor);
    int version = deviceProp.major*10 + deviceProp.minor;
    g_hasDouble = (version >= 13);

    // Crear plano FFT
    checkCudaErrors(cufftPlan2d(&fftPlan, meshSize,
                    meshSize, CUFFT_C2C));

    // Asignar memoria
    int spectrumSize = spectrumW*spectrumH*sizeof(float2);
    checkCudaErrors(cudaMalloc((void **)&d_h0, spectrumSize));
    h_h0 = (float2 *) malloc(spectrumSize);
    generate_h0(h_h0);
    checkCudaErrors(cudaMemcpy(d_h0, h_h0, spectrumSize,
                    cudaMemcpyHostToDevice));

    int outputSize =  meshSize*meshSize*sizeof(float2);
```

```
    checkCudaErrors(cudaMalloc((void**)&d_ht, outputSize));
    checkCudaErrors(cudaMalloc((void**)&d_slope,outputSize));

    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);
    prevTime = sdkGetTimerValue(&timer);

    runCudaTest(g_hasDouble, argv[0]);

    checkCudaErrors(cudaFree(d_ht));
    checkCudaErrors(cudaFree(d_slope));
    checkCudaErrors(cudaFree(d_h0));
    checkCudaErrors(cufftDestroy(fftPlan));
    free(h_h0);

    cudaDeviceReset();

    exit(g_TotalErrors==0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

Ejecute la prueba:

```
void runGraphicsTest(int argc, char **argv)
{
    printf("[%s] ", sSDKsample);
    printf("\n");

    if (checkCmdLineFlag(argc,(const char **)argv, "device"))
    {
        printf("[%s]\n", argv[0]);
        printf("   Does not explicitly support -device=n in
                OpenGL mode\n");
        printf("   To use -device=n, the sample must be
                running w/o OpenGL\n\n");
        printf(" > %s -device=n -qatest\n", argv[0]);
        printf("exiting...\n");
```

```
    exit(EXIT_SUCCESS);
}

// Primero inicializar el contexto de OpenGL, por lo que
// podemos configurar correctamente el GL para CUDA.
// Esto es necesario a fin de lograr un rendimiento
// óptimo con OpenGL/CUDA, interoperabilidad.
if (false == initGL(&argc, argv))
{
    cudaDeviceReset();
    return;
}

findCudaGLDevice(argc, (const char **)argv);

// Crear plano FFT.
checkCudaErrors(cufftPlan2d(&fftPlan, meshSize,
                            meshSize, CUFFT_C2C));

// Asignar memoria
int spectrumSize = spectrumW*spectrumH*sizeof(float2);
checkCudaErrors(cudaMalloc((void**)&d_h0, spectrumSize));
h_h0 = (float2 *) malloc(spectrumSize);
generate_h0(h_h0);
checkCudaErrors(cudaMemcpy(d_h0, h_h0, spectrumSize,
                cudaMemcpyHostToDevice));

int outputSize =  meshSize*meshSize*sizeof(float2);
checkCudaErrors(cudaMalloc((void**)&d_ht, outputSize));
checkCudaErrors(cudaMalloc((void**)&d_slope,outputSize));

sdkCreateTimer(&timer);
sdkStartTimer(&timer);
prevTime = sdkGetTimerValue(&timer);

// Crear búferes de vértice y registros con CUDA
createVBO(&heightVertexBuffer,
          meshSize*meshSize*sizeof(float));
```

```
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(
        &cuda_heightVB_resource, heightVertexBuffer,
        cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&slopeVertexBuffer, outputSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(
        &cuda_slopeVB_resource, slopeVertexBuffer,
        cudaGraphicsMapFlagsWriteDiscard));

    // Crear vértice y búfer de índice para la malla
    createMeshPositionVBO(&posVertexBuffer, meshSize,
                          meshSize);
    createMeshIndexBuffer(&indexBuffer, meshSize, meshSize);

    runCuda();

    // Registrar llamadas de vuelta
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);

    // Comenzar a renderizar el bucle principal
    glutMainLoop();
    cudaDeviceReset();
}

float urand()
{
    return rand() / (float)RAND_MAX;
}
```

Genera números aleatorios Gaussian con media 0 y desviación estándar 1:

```
float gauss()
{
    float u1 = urand();
    float u2 = urand();

    if (u1 < 1e-6f)
    {
        u1 = 1e-6f;
    }

    return sqrtf(-2 * logf(u1)) * cosf(2*CUDART_PI_F * u2);
}
```

Espectro Phillips:
(Kx, Ky) - normalizado vector de ola.
Vdir - ángulo de viento en radianes.
V - velocidad del viento.
A - constante.

```
float phillips(float Kx, float Ky, float Vdir, float V,
               float A, float dir_depend)
{
    float k_squared = Kx * Kx + Ky * Ky;

    if (k_squared == 0.0f)
    {
        return 0.0f;
    }

    // Ola más grande posible de viento constante a la
    // velocidad v
    float L = V * V / g;

    float k_x = Kx / sqrtf(k_squared);
    float k_y = Ky / sqrtf(k_squared);
    float w_dot_k = k_x * cosf(Vdir) + k_y * sinf(Vdir);
```

```
    float phillips = A * expf(-1.0f / (k_squared * L * L)) /
                 (k_squared * k_squared) * w_dot_k * w_dot_k;

    // Filtrar las olas que se mueven frente al viento
    if (w_dot_k < 0.0f)
    {
        phillips *= dir_depend;
    }

    // Amortiguar las ondas de muy pequeña longitud w << l
    float w = L / 10000;
    //phillips *= expf(-k_squared * w * w);

    return phillips;
}
```

Generar campo de altura de la base en el espacio de frecuencia:

```
void generate_h0(float2 *h0)
{
    for (unsigned int y = 0; y<=meshSize; y++)
    {
        for (unsigned int x = 0; x<=meshSize; x++)
        {
            float kx = (-(int)meshSize / 2.0f + x) * (2.0f *
                       CUDART_PI_F / patchSize);
            float ky = (-(int)meshSize / 2.0f + y) * (2.0f *
                       CUDART_PI_F / patchSize);

            float P = sqrtf(phillips(kx, ky, windDir,
                      windSpeed, A, dirDepend));

            if (kx == 0.0f && ky == 0.0f)
            {
                P = 0.0f;
```

```
            }

            //float Er = urand()*2.0f-1.0f;
            //float Ei = urand()*2.0f-1.0f;
            float Er = gauss();
            float Ei = gauss();

            float h0_re = Er * P * CUDART_SQRT_HALF_F;
            float h0_im = Ei * P * CUDART_SQRT_HALF_F;

            int i = y*spectrumW+x;
            h0[i].x = h0_re;
            h0[i].y = h0_im;
        }
    }
}
```

Ejecute los módulos de Cuda:

```
void runCuda()
{
    size_t num_bytes;

    // Generar espectro de ondas en el dominio de la
    // frecuencia
    cudaGenerateSpectrumKernel(d_h0, d_ht, spectrumW,
        meshSize, meshSize, animTime, patchSize);

    // Ejecutar FFT inversa para convertir al dominio
    // espacial
    checkCudaErrors(cufftExecC2C(fftPlan, d_ht, d_ht,
                                 CUFFT_INVERSE));

    // Actualizar los valores del mapa de altura en búfer de
    // vértice
    checkCudaErrors(cudaGraphicsMapResources(1,
```

```
                        &cuda_heightVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((
        void **)&g_hptr, &num_bytes, cuda_heightVB_resource));

    cudaUpdateHeightmapKernel(g_hptr, d_ht, meshSize,
                              meshSize);

    checkCudaErrors(cudaGraphicsUnmapResources(1,
                        &cuda_heightVB_resource, 0));

    // Calcular la inclinación para el sombreado
    checkCudaErrors(cudaGraphicsMapResources(1,
                        &cuda_slopeVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((
        void **)&g_sptr, &num_bytes, cuda_slopeVB_resource));

    cudaCalculateSlopeKernel(g_hptr, g_sptr, meshSize,
                             meshSize);

    checkCudaErrors(cudaGraphicsUnmapResources(1,
                        &cuda_slopeVB_resource, 0));
}

const char *sSpatialDomain[] =
{
    "ref_spatialDomain.bin",
    "ref_spatialDomain_sm13.bin",
    NULL
};

const char *sSlopeShading[] =
{
    "ref_slopeShading.bin",
    "ref_slopeShading_sm13.bin",
    NULL
};

void runCudaTest(bool bHasDouble, char *exec_path)
```

```
{
    checkCudaErrors(cudaMalloc((void **)&g_hptr,
                        meshSize*meshSize*sizeof(float)));
    checkCudaErrors(cudaMalloc((void **)&g_sptr,
                        meshSize*meshSize*sizeof(float2)));

    // Generar espectro de ondas en el dominio de la
    // frecuencia
    cudaGenerateSpectrumKernel(d_h0, d_ht, spectrumW,
                    meshSize, meshSize, animTime, patchSize);

    // Ejecutar FFT inversa para convertir al dominio
    // espacial
    checkCudaErrors(cufftExecC2C(fftPlan, d_ht, d_ht,
                                 CUFFT_INVERSE));

    // Actualizar los valores de alturas
    cudaUpdateHeightmapKernel(g_hptr, d_ht, meshSize,
                              meshSize);

    {
        float *hptr = (float *)malloc(
                      meshSize*meshSize*sizeof(float));
        cudaMemcpy((void *)hptr, (void *)g_hptr,
                    meshSize*meshSize*sizeof(float),
                    cudaMemcpyDeviceToHost);
        sdkDumpBin((void *)hptr, meshSize * meshSize *
                    sizeof(float), "spatialDomain.bin");

        if (!sdkCompareBin2BinFloat("spatialDomain.bin",
            sSpatialDomain[bHasDouble],
            meshSize*meshSize*sizeof(float),
            MAX_EPSILON, THRESHOLD, exec_path))
        {
            g_TotalErrors++;
        }

        free(hptr);
```

```
    }

    // Calcular la inclinación para el sombreado
    cudaCalculateSlopeKernel(g_hptr, g_sptr, meshSize,
                             meshSize);

    {
        float2 *sptr = (float2 *)malloc(
        meshSize*meshSize*sizeof(float2));
        cudaMemcpy((void *)sptr, (void *)g_sptr,
                    meshSize*meshSize*sizeof(float2),
                    cudaMemcpyDeviceToHost);
        sdkDumpBin(sptr, meshSize*meshSize*sizeof(float2),
                   "slopeShading.bin");

        if (!sdkCompareBin2BinFloat("slopeShading.bin",
            sSlopeShading[bHasDouble],
            meshSize*meshSize*sizeof(float2), MAX_EPSILON,
            THRESHOLD, exec_path))
        {
            g_TotalErrors++;
        }

        free(sptr);
    }

    checkCudaErrors(cudaFree(g_hptr));
    checkCudaErrors(cudaFree(g_sptr));
}


//void computeFPS()
//{
//    frameCount++;
//    fpsCount++;
//
//    if (fpsCount == fpsLimit) {
//        fpsCount = 0;
```

```
//    }
//}
```

Pantalla de devolución de llamada:

```
void display()
{
    // Ejecutar kernel CUDA para generar posiciones de los
    // vértices
    if (animate)
    {
        runCuda();
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Vista conjunto de matriz
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(translateX, translateY, translateZ);
    glRotatef(rotateX, 1.0, 0.0, 0.0);
    glRotatef(rotateY, 0.0, 1.0, 0.0);

    // Render de la vbo
    glBindBuffer(GL_ARRAY_BUFFER, posVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, heightVertexBuffer);
    glClientActiveTexture(GL_TEXTURE0);
    glTexCoordPointer(1, GL_FLOAT, 0, 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, slopeVertexBuffer);
    glClientActiveTexture(GL_TEXTURE1);
    glTexCoordPointer(2, GL_FLOAT, 0, 0);
```

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

glUseProgram(shaderProg);

// Establece los parámetros uniformes por defecto de las
// variables del vértice del sombreado
GLuint uniHeightScale, uniChopiness, uniSize;

uniHeightScale = glGetUniformLocation(shaderProg,
                                      "heightScale");
glUniform1f(uniHeightScale, 0.5f);

uniChopiness   = glGetUniformLocation(shaderProg,
                                      "chopiness");
glUniform1f(uniChopiness, 1.0f);

uniSize = glGetUniformLocation(shaderProg, "size");
glUniform2f(uniSize, (float) meshSize, (float) meshSize);

// Establece los parámetros uniformes por defecto de las
// variables del sombreado de píxeles.
GLuint uniDeepColor, uniShallowColor, uniSkyColor,
       uniLightDir;

uniDeepColor = glGetUniformLocation(shaderProg,
                                    "deepColor");
glUniform4f(uniDeepColor, 0.0f, 0.1f, 0.4f, 1.0f);

uniShallowColor = glGetUniformLocation(shaderProg,
                                       "shallowColor");
glUniform4f(uniShallowColor, 0.1f, 0.3f, 0.3f, 1.0f);

uniSkyColor=glGetUniformLocation(shaderProg,"skyColor");
glUniform4f(uniSkyColor, 1.0f, 1.0f, 1.0f, 1.0f);

uniLightDir=glGetUniformLocation(shaderProg, "lightDir");
glUniform3f(uniLightDir, 0.0f, 1.0f, 0.0f);
// End of uniform settings
```

```
    glColor3f(1.0, 1.0, 1.0);

    if (drawPoints)
    {
        glDrawArrays(GL_POINTS, 0, meshSize * meshSize);
    }
    else
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);

        glPolygonMode(GL_FRONT_AND_BACK,
                      wireFrame?GL_LINE:GL_FILL);
        glDrawElements(GL_TRIANGLE_STRIP, ((meshSize*2)+2)*
                      (meshSize-1), GL_UNSIGNED_INT, 0);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

    glDisableClientState(GL_VERTEX_ARRAY);
    glClientActiveTexture(GL_TEXTURE0);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glClientActiveTexture(GL_TEXTURE1);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(0);

    glutSwapBuffers();

    //computeFPS();
}

void timerEvent(int value)
{
    float time = sdkGetTimerValue(&timer);

    if (animate)
```

```
    {
        animTime += (time - prevTime) * animationRate;
    }

    glutPostRedisplay();
    prevTime = time;

    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
}

void cleanup()
{
    sdkDeleteTimer(&timer);
    checkCudaErrors(cudaGraphicsUnregisterResource(
                        cuda_heightVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(
                        cuda_slopeVB_resource));

    deleteVBO(&posVertexBuffer);
    deleteVBO(&heightVertexBuffer);
    deleteVBO(&slopeVertexBuffer);

    checkCudaErrors(cudaFree(d_h0));
    checkCudaErrors(cudaFree(d_slope));
    free(h_h0);
    cufftDestroy(fftPlan);
}
```

Los eventos de teclado controlador:

```
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    switch (key)
    {
        case (27) :
            cleanup();
```

```
            exit(EXIT_SUCCESS);

        case 'w':
            wireFrame = !wireFrame;
            break;

        case 'p':
            drawPoints = !drawPoints;
            break;

        case ' ':
            animate = !animate;
            break;
    }
}
```

Controladores de eventos del ratón:

```
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
    {
        mouseButtons |= 1<<button;
    }
    else if (state == GLUT_UP)
    {
        mouseButtons = 0;
    }

    mouseOldX = x;
    mouseOldY = y;
    glutPostRedisplay();
}

void motion(int x, int y)
{
```

```
    float dx, dy;
    dx = (float)(x - mouseOldX);
    dy = (float)(y - mouseOldY);

    if (mouseButtons == 1)
    {
        rotateX += dy * 0.2f;
        rotateY += dx * 0.2f;
    }
    else if (mouseButtons == 2)
    {
        translateX += dx * 0.01f;
        translateY -= dy * 0.01f;
    }
    else if (mouseButtons == 4)
    {
        translateZ += dy * 0.01f;
    }

    mouseOldX = x;
    mouseOldY = y;
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double) w / (double) h, 0.1, 10.0);

    windowW = w;
    windowH = h;
}
```

Inicializa GL:

```
bool initGL(int *argc, char **argv)
{
    // Crear contexto GL.
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(windowW, windowH);
    glutCreateWindow("CUDA FFT Ocean Simulation");

    vertShaderPath = sdkFindFilePath("ocean.vert", argv[0]);
    fragShaderPath = sdkFindFilePath("ocean.frag", argv[0]);

    if (vertShaderPath == NULL |
    | fragShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL vertex
                         and fragment shaders!\n");
        exit(EXIT_FAILURE);
    }

    // Inicializar extensiones OpenGL necesarias-
    glewInit();

    if (! glewIsSupported("GL_VERSION_2_0 "
                          ))
    {
        fprintf(stderr, "ERROR: Support for necessary OpenGL
                         extensions missing.");
        fflush(stderr);
        return false;
    }

    if (!glewIsSupported("GL_VERSION_1_5
     GL_ARB_vertex_buffer_object GL_ARB_pixel_buffer_object"))
    {
        fprintf(stderr, "Error: failed to get minimal
                         extensions for demo\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "  OpenGL version 1.5\n");
```

```
        fprintf(stderr, "  GL_ARB_vertex_buffer_object\n");
        fprintf(stderr, "  GL_ARB_pixel_buffer_object\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    // Default initialization
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);

    // Load shader
    shaderProg = loadGLSLProgram(vertShaderPath,
                                 fragShaderPath);

    SDK_CHECK_ERROR_GL();
    return true;
}
```

Crear VBO:

```
void createVBO(GLuint *vbo, int size)
{
    // Crear objeto buffer
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    SDK_CHECK_ERROR_GL();
}
```

Borrar VBO:

```
void deleteVBO(GLuint *vbo)
```

```
{
    glDeleteBuffers(1, vbo);
    *vbo = 0;
}
```

Crear búfer de índice para la prestación de malla quad:

```
void createMeshIndexBuffer(GLuint *id, int w, int h)
{
    int size = ((w*2)+2)*(h-1)*sizeof(GLuint);

    // Crear índice de buffer
    glGenBuffersARB(1, id);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *id);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER, size, 0,
                    GL_STATIC_DRAW);

    // Llenar con índices para la representación de malla
    // como tiras de triángulos
    GLuint *indices = (GLuint *) glMapBuffer(
                     GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!indices)
    {
        return;
    }

    for (int y=0; y<h-1; y++)
    {
        for (int x=0; x<w; x++)
        {
            *indices++ = y*w+x;
            *indices++ = (y+1)*w+x;
        }

        // Iniciar una nueva tira con triángulo degenerado
```

```
        *indices++ = (y+1)*w+(w-1);
        *indices++ = (y+1)*w;
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

Crear búfer de vértice fijo para almacenar vértices de la malla:

```
void createMeshPositionVBO(GLuint *id, int w, int h)
{
    createVBO(id, w*h*4*sizeof(float));

    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER,
                                       GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }

    for (int y=0; y<h; y++)
    {
        for (int x=0; x<w; x++)
        {
            float u = x / (float)(w-1);
            float v = y / (float)(h-1);
            *pos++ = u*2.0f-1.0f;
            *pos++ = 0.0f;
            *pos++ = v*2.0f-1.0f;
            *pos++ = 1.0f;
        }
    }
```

```
    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Adjuntar sombreado a un programa:

```
int attachShader(GLuint prg, GLenum type, const char *name)
{
    GLuint shader;
    FILE *fp;
    int size, compiled;
    char *src;

    fp = fopen(name, "rb");

    if (!fp)
    {
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    src = (char *)malloc(size);

    fseek(fp, 0, SEEK_SET);
    fread(src, sizeof(char), size, fp);
    fclose(fp);

    shader = glCreateShader(type);
    glShaderSource(shader, 1, (const char **)&src,
                   (const GLint *)&size);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS,
                  (GLint *)&compiled);

    if (!compiled)
```

```
    {
        char log[2048];
        int len;

        glGetShaderInfoLog(shader,2048,(GLsizei *)&len,log);
        printf("Info log: %s\n", log);
        glDeleteShader(shader);
        return 0;
    }

    free(src);

    glAttachShader(prg, shader);
    glDeleteShader(shader);

    return 1;
}
```

Crear programa de sombreado de vértices y fragmentos de archivos de sombreado:

```
GLuint loadGLSLProgram(const char *vertFileName,
                       const char *fragFileName)
{
    GLint linked;
    GLuint program;

    program = glCreateProgram();

    if (!attachShader(program,GL_VERTEX_SHADER,vertFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach vertex shader from
                         file %s\n", vertFileName);
        return 0;
    }
```

```
    if (!attachShader(program, GL_FRAGMENT_SHADER,
                      fragFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach fragment shader
                         from file %s\n", fragFileName);
        return 0;
    }

    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);

    if (!linked)
    {
        glDeleteProgram(program);
        char temp[256];
        glGetProgramInfoLog(program, 256, 0, temp);
        fprintf(stderr, "Failed to link program: %s\n", temp);
        return 0;
    }

    return program;
}
```

Código fuente

- oceanFFT.zip

Compilar:
make

Apéndices

Apéndice A

Paralelismo estático

A.1. Hola mundo

Hello.cu

```
#include "util/cuPrintf.cu"
#include <stdio.h>

__global__ void device_greetings(void)
{
  cuPrintf("Hello, world from the device!\n");
}

int main(void)
{
  // Saludos desde el Host
  printf("Hello, world from the host!\n");

  // inicializar cuPrintf
  cudaPrintfInit();

  // Lanzar un kernel con un solo hilo
  device_greetings<<<1,1>>>();

  // desplegar el saludo del dispositivo
  cudaPrintfDisplay();

  // limpiar después de  cuPrintf
  cudaPrintfEnd();

  return 0;
}
```

A.2. Suma de vectores

VectAdd.cu

```
// suma de vectores  utilizando un hilo por bloque en n bloques
#include<cuda.h>
#include<stdio.h>
#include <iostream>
#include <cstdlib>
#include <time.h>

#define length 64

__global__ void add(float *a, float *b, float *c){
      int tid=blockIdx.x; // handle data at this index
      if(tid < length)
            c[tid]=a[tid]+b[tid];
}

int main(void){
      float a[length],b[length],c[length];
      float *dev_a,*dev_b,*dev_c;

      time_t seconds;
      // allocate memory - GPU
      cudaMalloc((void**)&dev_a,length*sizeof(float));
      cudaMalloc((void**)&dev_b,length*sizeof(float));
      cudaMalloc((void**)&dev_c,length*sizeof(float));

      //fill arrays a and b on the CPU

      time(&seconds);
      srand((unsigned int) seconds);

      for(int i=0; i<length; i++){
            a[i]=(float)rand()/(float)RAND_MAX;
            b[i]=(float)rand()/(float)RAND_MAX;
      }

      //copy arrays from host to device
      cudaMemcpy(dev_a,a,length*sizeof(float),cudaMemcpyHostToDevice);
      cudaMemcpy(dev_b,b,length*sizeof(float),cudaMemcpyHostToDevice);

      add<<<length,1>>>(dev_a,dev_b,dev_c);

      //copy results back - device to host
      cudaMemcpy(c,dev_c,length*sizeof(float),cudaMemcpyDeviceToHost);

      //display results
      for(int i=0; i<length; i++){
            printf("%.3f + %.3f = %.3f \n",a[i],b[i],c[i]);
      }

      //free memory - GPU
      cudaFree(dev_a);
      cudaFree(dev_b);
      cudaFree(dev_c);

   return 0;
}
```

A.3. Multiplicación de matrices

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(GENCODE_SM30)
```

```
################################################################################

# Target rules
all: build

build: matrixMul

matrixMul.o: matrixMul.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

matrixMul: matrixMul.o
	$(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./matrixMul

clean:
	rm -f matrixMul matrixMul.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/matrixMul

clobber: clean
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Matrix multiplication: C = A * B.
 * Host code.
 *
 * This sample implements matrix multiplication as described in Chapter 3
 * of the programming guide.
 * It has been written for clarity of exposition to illustrate various CUDA
 * programming principles, not with the goal of providing the most
 * performant generic kernel for matrix multiplication.
 *
 * See also:
 * V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra,"
 * in Proc. 2008 ACM/IEEE Conf. on Superconducting (SC '08),
 * Piscataway, NJ: IEEE Press, 2008, pp. Art. 31:1-11.
 */

// System includes
#include <stdio.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>

// Helper functions and utilities to work with CUDA
#include <helper_functions.h>

/**
 * Matrix multiplication (CUDA Kernel) on the device: C = A * B
 * wA is A's width and wB is B's width
 */
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
```

```
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {

        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}

void constantInit(float *data, int size, float val)
{
```

```
    for (int i = 0; i < size; ++i)
    {
        data[i] = val;
    }
}

/**
 * Run a simple test of matrix multiplication using CUDA
 */
int matrixMultiply(int argc, char **argv, int block_size, dim3 &dimsA, dim3 &dimsB)
{
    // Allocate host memory for matrices A and B
    unsigned int size_A = dimsA.x * dimsA.y;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float *h_A = (float *)malloc(mem_size_A);
    unsigned int size_B = dimsB.x * dimsB.y;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float *h_B = (float *)malloc(mem_size_B);

    // Initialize host memory
    const float valB = 0.01f;
    constantInit(h_A, size_A, 1.0f);
    constantInit(h_B, size_B, valB);

    // Allocate device memory
    float *d_A, *d_B, *d_C;

    // Allocate host matrix C
    dim3 dimsC(dimsB.x, dimsA.y, 1);
    unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
    float *h_C = (float *) malloc(mem_size_C);

    if (h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host matrix C!\n");
        exit(EXIT_FAILURE);
    }

    cudaError_t error;

    error = cudaMalloc((void **) &d_A, mem_size_A);

    if (error != cudaSuccess)
    {
        printf("cudaMalloc d_A returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMalloc((void **) &d_B, mem_size_B);

    if (error != cudaSuccess)
    {
        printf("cudaMalloc d_B returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMalloc((void **) &d_C, mem_size_C);

    if (error != cudaSuccess)
    {
```

```
        printf("cudaMalloc d_C returned error code %d, line(%d)\n", error, __LINE__);
        exit(EXIT_FAILURE);
    }

    // copy host memory to device
    error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (d_A,h_A) returned error code %d, line(%d)\n", error,
__LINE__);
        exit(EXIT_FAILURE);
    }

    error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (d_B,h_B) returned error code %d, line(%d)\n", error,
__LINE__);
        exit(EXIT_FAILURE);
    }

    // Setup execution parameters
    dim3 threads(block_size, block_size);
    dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);

    // Create and start timer
    printf("Computing result using CUDA Kernel...\n");

    // Performs warmup operation using matrixMul CUDA kernel
    if (block_size == 16)
    {
        matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
    }
    else
    {
        matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
    }

    printf("done\n");

    cudaDeviceSynchronize();

    // Allocate CUDA events that we'll use for timing
    cudaEvent_t start;
    error = cudaEventCreate(&start);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    cudaEvent_t stop;
    error = cudaEventCreate(&stop);

    if (error != cudaSuccess)
    {
```

```
        fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    // Record the start event
    error = cudaEventRecord(start, NULL);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    // Execute the kernel
    int nIter = 300;

    for (int j = 0; j < nIter; j++)
    {
        if (block_size == 16)
        {
            matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
        }
        else
        {
            matrixMulCUDA<32><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
        }
    }

    // Record the stop event
    error = cudaEventRecord(stop, NULL);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    // Wait for the stop event to complete
    error = cudaEventSynchronize(stop);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to synchronize on the stop event (error code %s)!\n",
cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }

    float msecTotal = 0.0f;
    error = cudaEventElapsedTime(&msecTotal, start, stop);

    if (error != cudaSuccess)
    {
        fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
        exit(EXIT_FAILURE);
    }
```

```
    // Compute and print the performance
    float msecPerMatrixMul = msecTotal / nIter;
    double flopsPerMatrixMul = 2.0 * (double)dimsA.x * (double)dimsA.y *
(double)dimsB.x;
    double gigaFlops = (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
    printf(
        "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops, WorkgroupSize= %u
threads/block\n",
        gigaFlops,
        msecPerMatrixMul,
        flopsPerMatrixMul,
        threads.x * threads.y);

    // Copy result from device to host
    error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

    if (error != cudaSuccess)
    {
        printf("cudaMemcpy (h_C,d_C) returned error code %d, line(%d)\n", error,
__LINE__);
        exit(EXIT_FAILURE);
    }

    printf("Checking computed result for correctness: ");
    bool correct = true;

    // test relative error by the formula
    //     |<x, y>_cpu - <x,y>_gpu|/<|x|, |y|>  < eps
    double eps = 1.e-6 ; // machine zero
    for (int i = 0; i < (int)(dimsC.x * dimsC.y); i++)
    {
        double abs_err = fabs(h_C[i] - (dimsA.x * valB));
        double dot_length = dimsA.x;
        double abs_val = fabs(h_C[i]);
        double rel_err = abs_err/abs_val/dot_length ;
        if (rel_err > eps)
        {
            printf("Error! Matrix[%05d]=%.8f, ref=%.8f error term is > %E\n", i,
h_C[i], dimsA.x*valB, eps);
            correct = false;
        }
    }

    printf("%s\n", correct ? "Result = PASS" : "Result = FAIL");

    // Clean up memory
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    printf("\nNote: For peak performance, please refer to the matrixMulCUBLAS
example.\n");

    cudaDeviceReset();

    if (correct)
    {
```

```
        return EXIT_SUCCESS;
    }
    else
    {
        return EXIT_FAILURE;
    }
}


/**
 * Program main
 */
int main(int argc, char **argv)
{
    printf("[Matrix Multiply Using CUDA] - Starting...\n");

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "?"))
    {
        printf("Usage -device=n (n >= 0 for deviceID)\n");
        printf("      -wA=WidthA -hA=HeightA (Width x Height of Matrix A)\n");
        printf("      -wB=WidthB -hB=HeightB (Width x Height of Matrix B)\n");
        printf("  Note: Outer matrix dimensions of A & B matrices must be equal.\n");

        exit(EXIT_SUCCESS);
    }

    // By default, we use device 0, otherwise we override the device ID based on what
is provided at the command line
    int devID = 0;

    if (checkCmdLineFlag(argc, (const char **)argv, "device"))
    {
        devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
        cudaSetDevice(devID);
    }

    cudaError_t error;
    cudaDeviceProp deviceProp;
    error = cudaGetDevice(&devID);

    if (error != cudaSuccess)
    {
        printf("cudaGetDevice returned error code %d, line(%d)\n", error, __LINE__);
    }

    error = cudaGetDeviceProperties(&deviceProp, devID);

    if (deviceProp.computeMode == cudaComputeModeProhibited)
    {
        fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
        exit(EXIT_SUCCESS);
    }

    if (error != cudaSuccess)
    {
        printf("cudaGetDeviceProperties returned error code %d, line(%d)\n", error,
__LINE__);
    }
```

```
    else
    {
        printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
    }

    // Use a larger block size for Fermi and above
    int block_size = (deviceProp.major < 2) ? 16 : 32;

    dim3 dimsA(5*2*block_size, 5*2*block_size, 1);
    dim3 dimsB(5*4*block_size, 5*2*block_size, 1);

    // width of Matrix A
    if (checkCmdLineFlag(argc, (const char **)argv, "wA"))
    {
        dimsA.x = getCmdLineArgumentInt(argc, (const char **)argv, "wA");
    }

    // height of Matrix A
    if (checkCmdLineFlag(argc, (const char **)argv, "hA"))
    {
        dimsA.y = getCmdLineArgumentInt(argc, (const char **)argv, "hA");
    }

    // width of Matrix B
    if (checkCmdLineFlag(argc, (const char **)argv, "wB"))
    {
        dimsB.x = getCmdLineArgumentInt(argc, (const char **)argv, "wB");
    }

    // height of Matrix B
    if (checkCmdLineFlag(argc, (const char **)argv, "hB"))
    {
        dimsB.y = getCmdLineArgumentInt(argc, (const char **)argv, "hB");
    }

    if (dimsA.x != dimsB.y)
    {
        printf("Error: outer matrix dimensions must be equal. (%d != %d)\n",
               dimsA.x, dimsB.y);
        exit(EXIT_FAILURE);
    }

    printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", dimsA.x, dimsA.y, dimsB.x, dimsB.y);

    int matrix_result = matrixMultiply(argc, argv, block_size, dimsA, dimsB);

    exit(matrix_result);
}
```

A.4. Estencil 1D

```
#include <stdio.h>

#define RADIUS        3
#define BLOCK_SIZE    256
#define NUM_ELEMENTS  (4096*2)

// CUDA API error checking macro
#define cudaCheck(error) \
  if (error != cudaSuccess) { \
    printf("Fatal error: %s at %s:%d\n", \
      cudaGetErrorString(error), \
      __FILE__, __LINE__); \
    exit(1); \
  }

__global__ void stencil_1d(int *in, int *out)
{
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + (blockIdx.x * blockDim.x) + RADIUS;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS)
    {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Make sure all threads get to this point before proceeding!
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex-RADIUS] = result;
}

int main()
{
  unsigned int i;
  int h_in[NUM_ELEMENTS + 2 * RADIUS], h_out[NUM_ELEMENTS];
  int *d_in, *d_out;

  // Initialize host data
  for( i = 0; i < (NUM_ELEMENTS + 2*RADIUS); ++i )
    h_in[i] = 1; // With a value of 1 and RADIUS of 3, all output values should be
7

  // Allocate space on the device
  cudaCheck( cudaMalloc( &d_in, (NUM_ELEMENTS + 2*RADIUS) * sizeof(int)) );
  cudaCheck( cudaMalloc( &d_out, NUM_ELEMENTS * sizeof(int)) );

  // Copy input data to device
```

```
  cudaCheck( cudaMemcpy( d_in, h_in, (NUM_ELEMENTS + 2*RADIUS) * sizeof(int),
cudaMemcpyHostToDevice) );

  stencil_1d<<< (NUM_ELEMENTS + BLOCK_SIZE - 1)/BLOCK_SIZE, BLOCK_SIZE >>> (d_in,
d_out);

  cudaCheck( cudaMemcpy( h_out, d_out, NUM_ELEMENTS * sizeof(int),
cudaMemcpyDeviceToHost) );

  // Verify every out value is 7
  for( i = 0; i < NUM_ELEMENTS; ++i )
    if (h_out[i] != 7)
    {
      printf("Element h_out[%d] == %d != 7\n", i, h_out[i]);
      break;
    }

  if (i == NUM_ELEMENTS)
    printf("SUCCESS!\n");

  // Free out memory
  cudaFree(d_in);
  cudaFree(d_out);

  return 0;
}
```

A.5. Optimización Jacobi

```
//Number of Threads per bolock
#define THREADS_PER_BLOCK_X 32
#define THREADS_PER_BLOCK_Y 16
#define RADIUS 1

#ifndef STRNCASECMP
#ifdef _WIN32
#define STRNCASECMP _strnicmp
#else
#define STRNCASECMP strncasecmp
#endif
#endif

int initializeCPU(float **t, float **t_prev);
int initializeGPU(float **d_t, float **d_t_prev);

void unInitializeCPU(float **t, float **t_prev);
void unInitializeGPU(float **d_t, float **d_t_prev);

void performCPUCFD(float *t, float *t_prev, float  *cpuTime);
int performGPUCFD(float *d_t, float *d_t_prev, float *t, float *t_prev, float
*gpuTime);

//Utility functions
int checkHostEqualsDevice(float* o_host, float* o_device);
void parseCommandLineArguments(int argc,char*argv[]);

//Kernel Declaration
 __global__ void calculateCFD_V1( float* input,  float* output, unsigned int Ni,
unsigned int Nj, float h);
 __global__ void calculateCFD_V2( float* input,  float* output, unsigned int Ni,
unsigned int Nj, float h);
```

```
#include "definitions.cuh"

//Performs CFD calculation on global memory. This code does not use any advance
optimization technique on GPU
// But still acheives many fold performance gain
__global__ void calculateCFD_V1( float* input,  float* output, unsigned int Ni,
unsigned int Nj,
                                                    float h)
{
      unsigned int i = blockDim.x * blockIdx.x + threadIdx.x; // Y - ID
      unsigned int j = blockDim.y * blockIdx.y + threadIdx.y; // X - ID

      unsigned int iPrev = i-1; // Previous Y element
      unsigned int iNext = i+1; // Next Y element

      unsigned int jPrev = j-1; //Previous X element
      unsigned int jNext = j+1; // Next X element


      unsigned int index = i * Nj + j;

      if( i > 0 && j > 0 && i < (Ni-1) && j <(Nj-1))
            output[index] = 0.25f * (input[iPrev * Nj + j] + input[iNext* Nj + j] +
input[i * Nj+ jPrev]
                  + input[i* Nj + jNext] - 4*h*h);
}

//This version of Kernel uses optimization by copying the data into shared memory
and hence results in better performance
__global__ void calculateCFD_V2( float* input,  float* output, unsigned int Ni,
unsigned int Nj,
                                                    float h)
{

      //Current Global ID
      int i = blockDim.y * blockIdx.y + threadIdx.y; // Y - ID
      int j = blockDim.x * blockIdx.x + threadIdx.x; // X - ID

      //Current Local ID (lXX --> refers to local ID i.e. inside a block)
      int li = threadIdx.y;
      int lj = threadIdx.x;

      // e_XX --> variables refers to expanded shared memory location in order to
accomodate halo elements
      //Current Local ID with radius offset.
      int e_li = li + RADIUS;
      int e_lj = lj + RADIUS;

      // Variable pointing at top and bottom neighbouring location
      int e_li_prev = e_li - 1;
      int e_li_next = e_li + 1;

      // Variable pointing at left and right neighbouring location
      int e_lj_prev = e_lj - 1;
      int e_lj_next = e_lj + 1;

      __shared__ float sData [THREADS_PER_BLOCK_Y + 2 * RADIUS]
```

```
[THREADS_PER_BLOCK_X + 2 * RADIUS];

      unsigned int index = (i)* Nj + (j) ;

      if( li<RADIUS ) // copy top and bottom halo
      {
            //Copy Top Halo Element
            if(blockIdx.y > 0) // Boundary check
                  sData[li][e_lj] = input[index - RADIUS * Nj];

            //Copy Bottom Halo Element
            if(blockIdx.y < (gridDim.y-1)) // Boundary check
                  sData[e_li+THREADS_PER_BLOCK_Y][e_lj] = input[index +
THREADS_PER_BLOCK_Y * Nj];

      }

      if( lj<RADIUS ) // copy left and right halo
      {
            if( blockIdx.x > 0) // Boundary check
                  sData[e_li][lj] = input[index - RADIUS];

            if(blockIdx.x < (gridDim.x-1)) // Boundary check
                  sData[e_li][e_lj+THREADS_PER_BLOCK_X] = input[index +
THREADS_PER_BLOCK_X];
      }

      // copy current location
      sData[e_li][e_lj] = input[index];

      __syncthreads( );

      if( i > 0 && j > 0 && i < (Ni-1) && j <(Nj-1))
            output[index] = 0.25f * (sData[e_li_prev][e_lj] + sData[e_li_next]
[e_lj] + sData[e_li][e_lj_prev]
                  + sData[e_li][e_lj_next] - 4*h*h);

}
```

main.cu

```
#include <iostream>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "definitions.cuh"
#include <time.h>
#include <stdio.h>

//Number of elements on which to perform CFD
unsigned int Ni = 512; // Y elements
unsigned int Nj = 512; // X elements
unsigned int nIterations = 10000; // No Of Iterations
unsigned int kernelVersion =2; // Decides which GPU kernel version to call (Set it
to 1 or 2)

int main(int argc, char** argv)
{

	//Variables for Timing
	float cpuTime, gpuTime;

	// CPU and GPU Pointers ( d_XX : refers to pointer pointing to GPU memory.
This is just a convention)
	float *t = NULL, *t_prev = NULL;
	float *d_t = NULL,*d_t_prev= NULL;

	parseCommandLineArguments(argc, (char **)argv);
	printf("\n Ni= %d, Nj=%d nIteration=%d",Ni,Nj,nIterations);

	unsigned int size = Ni * Nj * sizeof(float);

	if(!initializeCPU(&t, &t_prev) )
	{
		printf("\n Error in allocating memory on CPU!!!");
		unInitializeCPU(&t, &t_prev);
		getchar();
		return 0;
	}

	if (!initializeGPU(&d_t, &d_t_prev))
	{
		printf("\n Error in allocating memory on GPU!!!");
		unInitializeCPU(&t, &t_prev);
		unInitializeGPU(&d_t, &d_t_prev);
		return 0;
	}

	//Perform CFD on CPU
	performCPUCFD(t,t_prev, &cpuTime);

	// To temporarily store CPU data. This is just for comparing with GPU output
	float *tempBuffer = (float*) calloc(Ni*Nj, sizeof(float));
	memcpy(tempBuffer, t_prev, size);

	//Perform CFD on GPU
	if(!performGPUCFD(d_t,d_t_prev, t, t_prev, &gpuTime))
	{
		printf("\n GPU Kernel failed !!!");
```

```
            unInitializeCPU(&t, &t_prev);
            unInitializeGPU(&d_t, &d_t_prev);
            if(tempBuffer !=NULL)
                  free(tempBuffer);
            return 0;
      }

      printf("\n Is host equal to device = %d",
checkHostEqualsDevice(tempBuffer,t));
      printf("\n Speedup = %fx", (float)(cpuTime/gpuTime));

      unInitializeCPU(&t, &t_prev);
      unInitializeGPU(&d_t, &d_t_prev);

      if(tempBuffer !=NULL)
            free(tempBuffer);

      printf("\n Finished Processing!!!");
      getchar();

}


void parseCommandLineArguments(int argc, char**argv)
{
    if (argc >= 1)
      {
        for (int i=1; i < argc; i++)
            {
                  int bFirstArgIsParam = false;
                  int string_start = 0;
                  while (argv[i][string_start] == '-')
                        string_start++;
                  char *string_argv = &argv[i][string_start];

                  if (!STRNCASECMP(string_argv, "Ni=", 3))
                  {
                bFirstArgIsParam = true;
                        Ni = atoi(&string_argv[3]);
                        continue;
                  }
                  if (!STRNCASECMP(string_argv, "Nj=", 3))
                  {
                bFirstArgIsParam = true;
                        Nj = atoi(&string_argv[3]);
                        continue;
                  }
                  if (!STRNCASECMP(string_argv, "iterations=", 11))
                  {
                bFirstArgIsParam = true;
                        nIterations = atoi(&string_argv[11]);
                        continue;
                  }
                  if (!STRNCASECMP(string_argv, "kernel=", 7))
                  {
                bFirstArgIsParam = true;
                        kernelVersion = atoi(&string_argv[7]);
                        continue;
```

```
                }

                if (!bFirstArgIsParam)
                {
                            printf("Invalid arguments\n");
                            for (int n=0; n < argc; n++)
                            {
                                  printf("argv[%d] = %s\n", n, argv[n]);
                            }
                            printf("\n");
                            exit(0);
                }
          }
      }

      if(( Ni % THREADS_PER_BLOCK_Y != 0)  || (Nj % THREADS_PER_BLOCK_X != 0))
      {
            fprintf(stderr, "Please specify Ni & Nj as multiple of 16 !!!!");
            getchar();
            exit(0);
      }
}

int initializeCPU(float **t, float **t_prev)
{
      *t = (float*) calloc(Ni*Nj, sizeof(float));
      *t_prev = (float*) calloc(Ni*Nj, sizeof(float));

      if((*t)==NULL || (*t_prev) == NULL)
            return 0;
      else
            return 1;
}

void unInitializeCPU(float **t, float **t_prev)
{
      if((*t) !=NULL)
            free(*t);
      if((*t_prev) != NULL)
            free(*t_prev);
}

int initializeGPU(float **d_t, float **d_t_prev)
{

      unsigned int size = Ni * Nj * sizeof(float);



      // Choose which GPU to run on, change this on a multi-GPU system.
    cudaError_t cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU
installed?");
            getchar();
        return 0;
    }
       // Allocate GPU buffers.
```

```
    cudaStatus = cudaMalloc((void**)&(*d_t), size);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
            getchar();
        return 0;
    }

       // Allocate GPU buffers   .
    cudaStatus = cudaMalloc((void**)&(*d_t_prev), size);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
            getchar();
        return 0;
    }

       // Memset GPU buffers
    cudaStatus = cudaMemset((*d_t),0, size);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemset failed!");
            getchar();
        return 0;
    }

      // Memset GPU buffers
    cudaStatus = cudaMemset((*d_t_prev),0, size);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemset failed!");
            getchar();
        return 0;
    }

      return 1;
}


void unInitializeGPU(float **d_t, float **d_t_prev)
{
      cudaError_t cudaStatus;

      if((*d_t)!=NULL)
    cudaStatus = cudaFree((*d_t));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaFree failed!");
        return;
    }

      if((*d_t_prev)!=NULL)
    cudaStatus = cudaFree((*d_t_prev));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaFree failed!");
        return;
    }

      cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
            getchar();
        return;
```

```
    }
}

void performCPUCFD(float *t, float *t_prev, float *cpuTime)
{

      float h,x,y;

      h = 1.0f/(Ni-1);

      for(unsigned int i=0;i<Ni;i++)
      {
            x = i*h;
            t_prev[i*Nj+0] = x*x;
            t_prev[i*Nj+(Nj-1)] = x*x + 1.0f;
      }

      for(unsigned int j=0;j < Nj; j++)
      {
            y = j*h;
            t_prev[0*Nj+j] = y*y;
            t_prev[((Ni-1) * Nj) + j] = 1.0f + y*y;

      }

    float elapsedTimeInMs = 0.0f;

      clock_t start = clock();

      for(unsigned int k=0;k<nIterations;k++)
      {
            for(unsigned int j=1;j<(Nj-1);j++)
            {
                  for(unsigned int i=1;i<(Ni-1);i++)
                  {
                        t[i*Nj+j] = 0.25f * (t_prev[(i-1)*Nj+j] +
t_prev[(i+1)*Nj+j] + t_prev[i*Nj+(j-1)] +
                              t_prev[i*Nj+(j+1)] - 4*h*h);
                  }
            }

            float* pingPong = t_prev;
        t_prev = t;
        t = pingPong;
      }

      clock_t end = clock();
      elapsedTimeInMs = (float)((end - start) * 1000 / CLOCKS_PER_SEC);

      printf("\n CPU Time:: %f ms", elapsedTimeInMs);
      *cpuTime = elapsedTimeInMs;
}

int performGPUCFD(float *d_t, float *d_t_prev, float *t, float *t_prev,
float*gpuTime)
{

      float h,x,y;
```

```
	const char *str = (char*) malloc(1024); // To store error string

	//Decide how many blocks per thread and how many blocks per grid
	dim3 dimBlock(THREADS_PER_BLOCK_X,THREADS_PER_BLOCK_Y);
	dim3 dimGrid(Nj/dimBlock.x,Ni/dimBlock.y);

	h = 1.0f/(Ni-1);
	memset(t_prev, 0, sizeof(float) * Ni * Nj);

	for(unsigned int i=0;i<Ni;i++)
	{
		x = i*h;
		t_prev[i*Nj+0] = x*x;
		t_prev[i*Nj+(Nj-1)] = x*x + 1.0f;
	}

	for(unsigned int j=0;j < Nj; j++)
	{
		y = j*h;
		t_prev[0*Nj+j] = y*y;
		t_prev[((Ni-1) * Nj) + j] = 1.0f + y*y;

	}

	//Copy data to device
	cudaMemcpy(d_t_prev, t_prev, sizeof(float) * Ni * Nj ,
cudaMemcpyHostToDevice);

	//Insert event to calculate time
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);


	//This calls Version 1 of kernel which uses Global memory
	if(kernelVersion ==1)
	{
		cudaEventRecord(start, 0);

		for(unsigned int k=0;k<nIterations;k++)
		{
			// Launch a kernel on the GPU with one thread for each element.
			calculateCFD_V1<<<dimGrid,dimBlock>>>(d_t_prev,d_t, Ni, Nj, h);

			float* pingPong = d_t_prev;
			d_t_prev = d_t;
			d_t = pingPong;
		}
		cudaEventRecord(stop, 0);
		cudaEventSynchronize(stop);
	}
	//This calls Version 2 of kernel which uses optimization by copying data to
shared memory
	else if(kernelVersion ==2)
	{
		cudaEventRecord(start, 0);

		for(unsigned int k=0;k<nIterations;k++)
```

```
            {
                  // Launch a kernel on the GPU with one thread for each element.
                  calculateCFD_V2<<<dimGrid,dimBlock>>>(d_t_prev,d_t, Ni, Nj, h);

                  float* pingPong = d_t_prev;
                  d_t_prev = d_t;
                  d_t = pingPong;
            }
            cudaEventRecord(stop, 0);
            cudaEventSynchronize(stop);

      }

      float elapsedTime;
      cudaEventElapsedTime(&elapsedTime, start, stop);
      printf("\n GPU Time:: %f ms", elapsedTime);

      *gpuTime = elapsedTime;

      cudaError_t cudaStatus = cudaMemcpy(t, d_t_prev, sizeof(float) * Ni * Nj ,
cudaMemcpyDeviceToHost);
      if (cudaStatus != cudaSuccess) {
            fprintf(stderr, "cudaMemcpy failed!");
            str = cudaGetErrorString(cudaStatus);
            fprintf(stderr, "CUDA Error!:: %s\n", str);
            getchar();
            return 0;
      }

      return 1;
}

int checkHostEqualsDevice(float* o_host, float* o_device)
{
      int flag =1;

      float tolerance = 0.0001f;
      //Compare the results
      for(unsigned int j=0;j<Nj;j++)
      {
            for(unsigned int i=0;i<Ni;i++)
            {
                  if( (o_host[i*Nj+j] - o_device[i*Nj+j]) >= tolerance ||
(o_host[i*Nj+j] - o_device[i*Nj+j]) <= -tolerance)
                  {
                        printf("\n D=[%f]!=H=[%f] since Diff  > tol %f for [%d]
[%d]",o_device[i*Nj+j],  o_host[i*Nj+j],tolerance, i, j);
                        flag =0;
                        //getchar();
                  }
            }
      }

      return flag;
}
```

Makefile

```
all:  cfd

cfd:  kernel.o main.o
      nvcc kernel.o main.o -o cfd

main.o:     main.cu
      nvcc -c main.cu -o main.o

kernel.o: kernel.cu
      nvcc -c kernel.cu -o kernel.o

clean:
      rm -f *.o cfd
```

A.6. QuickSort

```
#include <iostream>
#include <cstdio>
#include <helper_cuda.h>
#include <helper_string.h>

#define MAX_DEPTH       16
#define INSERTION_SORT  32

////////////////////////////////////////////////////////////////////////////
// Selection sort used when depth gets too big or the number of elements drops
// below a threshold.
////////////////////////////////////////////////////////////////////////////
__device__ void selection_sort(unsigned int *data, int left, int right)
{
    for (int i = left ; i <= right ; ++i)
    {
        unsigned min_val = data[i];
        int min_idx = i;

        // Find the smallest value in the range [left, right].
        for (int j = i+1 ; j <= right ; ++j)
        {
            unsigned val_j = data[j];

            if (val_j < min_val)
            {
                min_idx = j;
                min_val = val_j;
            }
        }

        // Swap the values.
        if (i != min_idx)
        {
            data[min_idx] = data[i];
            data[i] = min_val;
        }
    }
}

////////////////////////////////////////////////////////////////////////////
// Very basic quicksort algorithm, recursively launching the next level.
////////////////////////////////////////////////////////////////////////////
__global__ void cdp_simple_quicksort(unsigned int *data, int left, int right, int
depth)
{
    // If we're too deep or there are few elements left, we use an insertion
```

```
sort...
    if (depth >= MAX_DEPTH || right-left <= INSERTION_SORT)
    {
        selection_sort(data, left, right);
        return;
    }

    unsigned int *lptr = data+left;
    unsigned int *rptr = data+right;
    unsigned int  pivot = data[(left+right)/2];

    // Do the partitioning.
    while (lptr <= rptr)
    {
        // Find the next left- and right-hand values to swap
        unsigned int lval = *lptr;
        unsigned int rval = *rptr;

        // Move the left pointer as long as the pointed element is smaller than
the pivot.
        while (lval < pivot)
        {
            lptr++;
            lval = *lptr;
        }

        // Move the right pointer as long as the pointed element is larger than
the pivot.
        while (rval > pivot)
        {
            rptr--;
            rval = *rptr;
        }

        // If the swap points are valid, do the swap!
        if (lptr <= rptr)
        {
            *lptr++ = rval;
            *rptr-- = lval;
        }
    }

    // Now the recursive part
    int nright = rptr - data;
    int nleft  = lptr - data;

    // Launch a new block to sort the left part.
    if (left < (rptr-data))
    {
        cudaStream_t s;
        cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
        cdp_simple_quicksort<<< 1, 1, 0, s >>>(data, left, nright, depth+1);
        cudaStreamDestroy(s);
    }

    // Launch a new block to sort the right part.
    if ((lptr-data) < right)
    {
```

```
        cudaStream_t s1;
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        cdp_simple_quicksort<<< 1, 1, 0, s1 >>>(data, nleft, right, depth+1);
        cudaStreamDestroy(s1);
    }
}

////////////////////////////////////////////////////////////////////////////
// Call the quicksort kernel from the host.
////////////////////////////////////////////////////////////////////////////
void run_qsort(unsigned int *data, unsigned int nitems)
{
    // Prepare CDP for the max depth 'MAX_DEPTH'.
    checkCudaErrors(cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, MAX_DEPTH));

    // Launch on device
    int left = 0;
    int right = nitems-1;
    std::cout << "Launching kernel on the GPU" << std::endl;
    cdp_simple_quicksort<<< 1, 1 >>>(data, left, right, 0);
    checkCudaErrors(cudaDeviceSynchronize());
}

////////////////////////////////////////////////////////////////////////////
// Initialize data on the host.
////////////////////////////////////////////////////////////////////////////
void initialize_data(unsigned int *dst, unsigned int nitems)
{
    // Fixed seed for illustration
    srand(2047);

    // Fill dst with random values
    for (unsigned i = 0 ; i < nitems ; i++)
        dst[i] = rand() % nitems ;
}

////////////////////////////////////////////////////////////////////////////
// Verify the results.
////////////////////////////////////////////////////////////////////////////
void check_results(int n, unsigned int *results_d)
{
    unsigned int *results_h = new unsigned[n];
    checkCudaErrors(cudaMemcpy(results_h, results_d, n*sizeof(unsigned),
cudaMemcpyDeviceToHost));

    for (int i = 1 ; i < n ; ++i)
        if (results_h[i-1] > results_h[i])
        {
            std::cout << "Invalid item[" << i-1 << "]: " << results_h[i-1] << "
greater than " << results_h[i] << std::endl;
            exit(EXIT_FAILURE);
        }

    std::cout << "OK" << std::endl;
    delete[] results_h;
}

////////////////////////////////////////////////////////////////////////////
```

```
// Main entry point.
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    int num_items = 128;
    bool verbose = false;

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "h"))
    {
        std::cerr << "Usage: " << argv[0] << " num_items=<num_items>\twhere
num_items is the number of items to sort" << std::endl;
        exit(EXIT_SUCCESS);
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "v"))
    {
        verbose = true;
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "num_items"))
    {
        num_items = getCmdLineArgumentInt(argc, (const char **)argv, "num_items");

        if (num_items < 1)
        {
            std::cerr << "ERROR: num_items has to be greater than 1" << std::endl;
            exit(EXIT_FAILURE);
        }
    }

    // Get device properties
    int device_count = 0, device = -1;
    checkCudaErrors(cudaGetDeviceCount(&device_count));

    for (int i = 0 ; i < device_count ; ++i)
    {
        cudaDeviceProp properties;
        checkCudaErrors(cudaGetDeviceProperties(&properties, i));

        if (properties.major > 3 || (properties.major == 3 && properties.minor >=
5))
        {
            device = i;
            std::cout << "Running on GPU " << i << " (" << properties.name << ")"
<< std::endl;
            break;
        }

        std::cout << "GPU " << i << " (" << properties.name << ") does not support
CUDA Dynamic Parallelism" << std::endl;
    }

    if (device == -1)
    {
        std::cerr << "cdpSimpleQuicksort requires GPU devices with compute SM 3.5
or higher.  Exiting..." << std::endl;
        exit(EXIT_SUCCESS);
```

```
    }

    cudaSetDevice(device);

    // Create input data
    unsigned int *h_data = 0;
    unsigned int *d_data = 0;

    // Allocate CPU memory and initialize data.
    std::cout << "Initializing data:" << std::endl;
    h_data =(unsigned int *)malloc(num_items*sizeof(unsigned int));
    initialize_data(h_data, num_items);

    if (verbose)
    {
        for (int i=0 ; i<num_items ; i++)
            std::cout << "Data [" << i << "]: " << h_data[i] << std::endl;
    }

    // Allocate GPU memory.
    checkCudaErrors(cudaMalloc((void **)&d_data, num_items * sizeof(unsigned
int)));
    checkCudaErrors(cudaMemcpy(d_data, h_data, num_items * sizeof(unsigned int),
cudaMemcpyHostToDevice));

    // Execute
    std::cout << "Running quicksort on " << num_items << " elements" << std::endl;
    run_qsort(d_data, num_items);

    // Check result
    std::cout << "Validating results: ";
    check_results(num_items, d_data);

    free(h_data);
    checkCudaErrors(cudaFree(d_data));
    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
GENCODE_SM35    := -gencode arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   :=  $(GENCODE_SM35)

ALL_CCFLAGS += -dc

LIBRARIES += -lcudadevrt
```

```
################################################################################

# Target rules
all: build

build: cdpSimpleQuicksort

cdpSimpleQuicksort.o: cdpSimpleQuicksort.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

cdpSimpleQuicksort: cdpSimpleQuicksort.o
	$(NVCC) $(ALL_LDFLAGS) $(GENCODE_FLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./cdpSimpleQuicksort

clean:
	rm -f cdpSimpleQuicksort cdpSimpleQuicksort.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/cdpSimpleQuicksort

clobber: clean
```

A.7. MonteCarloMultiGPU

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(GENCODE_SM30)
```

```
LIBRARIES += -lcurand

################################################################################

# Target rules
all: build

build: MonteCarloMultiGPU

MonteCarlo_kernel.o: MonteCarlo_kernel.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

MonteCarlo_gold.o: MonteCarlo_gold.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

multithreading.o: multithreading.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

MonteCarloMultiGPU.o: MonteCarloMultiGPU.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

MonteCarloMultiGPU: MonteCarloMultiGPU.o multithreading.o MonteCarlo_gold.o
MonteCarlo_kernel.o
	$(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./MonteCarloMultiGPU

clean:
	rm -f MonteCarloMultiGPU MonteCarloMultiGPU.o multithreading.o
MonteCarlo_gold.o MonteCarlo_kernel.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/MonteCarloMultiGPU

clobber: clean
```

```
#ifndef MONTECARLO_COMMON_H
#define MONTECARLO_COMMON_H
#include "realtype.h"
#include "curand_kernel.h"

////////////////////////////////////////////////////////////////////////////////
// Global types
////////////////////////////////////////////////////////////////////////////////
typedef struct
{
    float S;
    float X;
    float T;
    float R;
    float V;
} TOptionData;

typedef struct
        //#ifdef __CUDACC__
        //__align__(8)
        //#endif
{
    float Expected;
    float Confidence;
} TOptionValue;

//GPU outputs before CPU postprocessing
typedef struct
{
    real Expected;
    real Confidence;
} __TOptionValue;



typedef struct
{
    //Device ID for multi-GPU version
    int device;
    //Option count for this plan
    int optionCount;

    //Host-side data source and result destination
    TOptionData  *optionData;
    TOptionValue *callValue;
```

```
    //Temporary Host-side pinned memory for async + faster data transfers
    __TOptionValue *h_CallValue;


    //Intermediate device-side buffers
    void *d_Buffer;

    //random number generator states
    curandState *rngStates;

    //Pseudorandom samples count
    int pathN;

    //Time stamp
    float time;

    //random number generator seed.
    unsigned long long seed;
} TOptionPlan;


extern "C" void initMonteCarloGPU(TOptionPlan *plan);
extern "C" void MonteCarloGPU(TOptionPlan *plan, cudaStream_t stream=0);
extern "C" void closeMonteCarloGPU(TOptionPlan *plan);

#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <curand.h>

//#include "curand_kernel.h"
#include "helper_cuda.h"



////////////////////////////////////////////////////////////////////////////////
// Common types
////////////////////////////////////////////////////////////////////////////////
#include "MonteCarlo_common.h"


////////////////////////////////////////////////////////////////////////////////
// Black-Scholes formula for Monte Carlo results validation
////////////////////////////////////////////////////////////////////////////////
#define A1 0.31938153
#define A2 -0.356563782
#define A3 1.781477937
#define A4 -1.821255978
#define A5 1.330274429
#define RSQRT2PI 0.39894228040143267793994605993438

//Polynomial approxiamtion of
//cumulative normal distribution function
double CND(double d)
{
    double
    K = 1.0 / (1.0 + 0.2316419 * fabs(d));

    double
    cnd = RSQRT2PI * exp(- 0.5 * d * d) *
          (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))));

    if (d > 0)
        cnd = 1.0 - cnd;

    return cnd;
}

//Black-Scholes formula for call value
```

```
extern "C" void BlackScholesCall(
    float &callValue,
    TOptionData optionData
)
{
    double     S = optionData.S;
    double     X = optionData.X;
    double     T = optionData.T;
    double     R = optionData.R;
    double     V = optionData.V;

    double sqrtT = sqrt(T);
    double    d1 = (log(S / X) + (R + 0.5 * V * V) * T) / (V * sqrtT);
    double    d2 = d1 - V * sqrtT;
    double CNDD1 = CND(d1);
    double CNDD2 = CND(d2);
    double expRT = exp(- R * T);

    callValue = (float)(S * CNDD1 - X * expRT * CNDD2);
}


////////////////////////////////////////////////////////////////////////////////
// CPU Monte Carlo
////////////////////////////////////////////////////////////////////////////////
static double endCallValue(double S, double X, double r, double MuByT, double
VBySqrtT)
{
    double callValue = S * exp(MuByT + VBySqrtT * r) - X;
    return (callValue > 0) ? callValue : 0;
}

extern "C" void MonteCarloCPU(
    TOptionValue    &callValue,
    TOptionData optionData,
    float *h_Samples,
    int pathN
)
{
    const double        S = optionData.S;
    const double        X = optionData.X;
    const double        T = optionData.T;
    const double        R = optionData.R;
    const double        V = optionData.V;
    const double    MuByT = (R - 0.5 * V * V) * T;
    const double VBySqrtT = V * sqrt(T);

    float *samples;
    curandGenerator_t gen;

    checkCudaErrors(curandCreateGeneratorHost(&gen, CURAND_RNG_PSEUDO_DEFAULT));
    unsigned long long seed = 1234ULL;
    checkCudaErrors(curandSetPseudoRandomGeneratorSeed(gen,  seed));

    if (h_Samples != NULL)
    {
        samples = h_Samples;
    }
```

```
    else
    {
        samples = (float *) malloc(pathN * sizeof(float));
        checkCudaErrors(curandGenerateNormal(gen, samples, pathN, 0.0, 1.0));
    }

    // for(int i=0; i<10; i++) printf("CPU sample = %f\n", samples[i]);

    double sum = 0, sum2 = 0;

    for (int pos = 0; pos < pathN; pos++)
    {

        double    sample = samples[pos];
        double callValue = endCallValue(S, X, sample, MuByT, VBySqrtT);
        sum  += callValue;
        sum2 += callValue * callValue;
    }

    if (h_Samples == NULL) free(samples);

    checkCudaErrors(curandDestroyGenerator(gen));

    //Derive average from the total sum and discount by riskfree rate
    callValue.Expected = (float)(exp(-R * T) * sum / (double)pathN);
    //Standart deviation
    double stdDev = sqrt(((double)pathN * sum2 - sum * sum)/ ((double)pathN *
(double)(pathN - 1)));
    //Confidence width; in 95% of all cases theoretical value lies within these
borders
    callValue.Confidence = (float)(exp(-R * T) * 1.96 * stdDev /
sqrt((double)pathN));
}
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

////////////////////////////////////////////////////////////////////////////////
// Global types
////////////////////////////////////////////////////////////////////////////////
#include <stdlib.h>
#include <stdio.h>
#include <helper_cuda.h>
#include <curand_kernel.h>
#include "MonteCarlo_common.h"



////////////////////////////////////////////////////////////////////////////////
// Helper reduction template
// Please see the "reduction" CUDA SDK sample for more information
////////////////////////////////////////////////////////////////////////////////
#include "MonteCarlo_reduction.cuh"

////////////////////////////////////////////////////////////////////////////////
// Internal GPU-side data structures
////////////////////////////////////////////////////////////////////////////////
#define MAX_OPTIONS 512

//Preprocessed input option data
typedef struct
{
    real S;
    real X;
    real MuByT;
    real VBySqrtT;
} __TOptionData;
static __device__ __constant__ __TOptionData d_OptionData[MAX_OPTIONS];

static __device__ __TOptionValue d_CallValue[MAX_OPTIONS];

////////////////////////////////////////////////////////////////////////////////
// Overloaded shortcut payoff functions for different precision modes
////////////////////////////////////////////////////////////////////////////////
__device__ inline float endCallValue(float S, float X, float r, float MuByT, float
VBySqrtT)
{
    float callValue = S * __expf(MuByT + VBySqrtT * r) - X;
    return (callValue > 0) ? callValue : 0;
}

#define THREAD_N 256
```

```
////////////////////////////////////////////////////////////////////////////////
// This kernel computes the integral over all paths using a single thread block
// per option. It is fastest when the number of thread blocks times the work per
// block is high enough to keep the GPU busy.
////////////////////////////////////////////////////////////////////////////////
static __global__ void MonteCarloOneBlockPerOption(
    curandState *rngStates,
    int pathN)
{
    const int SUM_N = THREAD_N;
    __shared__ real s_SumCall[SUM_N];
    __shared__ real s_Sum2Call[SUM_N];

    const int optionIndex = blockIdx.x;
    const real        S = d_OptionData[optionIndex].S;
    const real        X = d_OptionData[optionIndex].X;
    const real    MuByT = d_OptionData[optionIndex].MuByT;
    const real VBySqrtT = d_OptionData[optionIndex].VBySqrtT;

    // determine global thread id
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    // Copy random number state to local memory for efficiency
    curandState localState = rngStates[tid];

    //Cycle through the entire samples array:
    //derive end stock price for each path
    //accumulate partial integrals into intermediate shared memory buffer
    for (int iSum = threadIdx.x; iSum < SUM_N; iSum += blockDim.x)
    {
        __TOptionValue sumCall = {0, 0};

        for (int i = iSum; i < pathN; i += SUM_N)
        {
            real              r = curand_normal(&localState);
            real      callValue = endCallValue(S, X, r, MuByT, VBySqrtT);
            sumCall.Expected   += callValue;
            sumCall.Confidence += callValue * callValue;
        }

        s_SumCall[iSum]  = sumCall.Expected;
        s_Sum2Call[iSum] = sumCall.Confidence;
    }

    // store random number state back to global memory
    rngStates[tid] = localState;

    //Reduce shared memory accumulators
    //and write final result to global memory
    sumReduce<real, SUM_N, THREAD_N>(s_SumCall, s_Sum2Call);

    if (threadIdx.x == 0)
    {
        __TOptionValue t = {s_SumCall[0], s_Sum2Call[0]};
        d_CallValue[optionIndex] = t;
    }
}
```

```
static __global__ void rngSetupStates(
    curandState *rngState,
    unsigned long long seed,
    unsigned long long offset)
{
    // determine global thread id
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    // Each thread gets the same seed, a different
    // sequence number. A different offset is used for
    // each device.
    curand_init(seed, tid, offset, &rngState[tid]);
}



////////////////////////////////////////////////////////////////////////////////
// Host-side interface to GPU Monte Carlo
////////////////////////////////////////////////////////////////////////////////

extern "C" void initMonteCarloGPU(TOptionPlan *plan)
{
    //Allocate internal device memory
    checkCudaErrors(cudaMallocHost(&plan->h_CallValue,
sizeof(__TOptionValue)*MAX_OPTIONS));
    //Allocate states for pseudo random number generators
    checkCudaErrors(cudaMalloc((void **) &plan->rngStates,
                               plan->optionCount * THREAD_N *
sizeof(curandState)));

    // place each device pathN random numbers apart on the random number sequence
    unsigned long long offset = plan->device * plan->pathN;
    rngSetupStates<<<plan->optionCount, THREAD_N>>>(plan->rngStates, plan->seed,
offset);
    getLastCudaError("rngSetupStates kernel failed.\n");
}

//Compute statistics and deallocate internal device memory
extern "C" void closeMonteCarloGPU(TOptionPlan *plan)
{
    for (int i = 0; i < plan->optionCount; i++)
    {
        const double    RT = plan->optionData[i].R * plan->optionData[i].T;
        const double   sum = plan->h_CallValue[i].Expected;
        const double  sum2 = plan->h_CallValue[i].Confidence;
        const double pathN = plan->pathN;
        //Derive average from the total sum and discount by riskfree rate
        plan->callValue[i].Expected = (float)(exp(-RT) * sum / pathN);
        //Standart deviation
        double stdDev = sqrt((pathN * sum2 - sum * sum)/ (pathN * (pathN - 1)));
        //Confidence width; in 95% of all cases theoretical value lies within
these borders
        plan->callValue[i].Confidence = (float)(exp(-RT) * 1.96 * stdDev /
sqrt(pathN));
    }

    checkCudaErrors(cudaFree(plan->rngStates));
    checkCudaErrors(cudaFreeHost(plan->h_CallValue));
}
```

```
//Main computations
extern "C" void MonteCarloGPU(TOptionPlan *plan, cudaStream_t stream)
{
    __TOptionData h_OptionData[MAX_OPTIONS];
    __TOptionValue *h_CallValue = plan->h_CallValue;

    if (plan->optionCount <= 0 || plan->optionCount > MAX_OPTIONS)
    {
        printf("MonteCarloGPU(): bad option count.\n");
        return;
    }

    for (int i = 0; i < plan->optionCount; i++)
    {
        const double           T = plan->optionData[i].T;
        const double           R = plan->optionData[i].R;
        const double           V = plan->optionData[i].V;
        const double       MuByT = (R - 0.5 * V * V) * T;
        const double    VBySqrtT = V * sqrt(T);
        h_OptionData[i].S        = (real)plan->optionData[i].S;
        h_OptionData[i].X        = (real)plan->optionData[i].X;
        h_OptionData[i].MuByT    = (real)MuByT;
        h_OptionData[i].VBySqrtT = (real)VBySqrtT;
    }

    checkCudaErrors(cudaMemcpyToSymbolAsync(
                        d_OptionData,
                        h_OptionData,
                        plan->optionCount * sizeof(__TOptionData),
                        0, cudaMemcpyHostToDevice, stream
                    ));

    MonteCarloOneBlockPerOption<<<plan->optionCount, THREAD_N, 0, stream>>>(
        plan->rngStates,
        plan->pathN
    );
    getLastCudaError("MonteCarloOneBlockPerOption() execution failed\n");


    checkCudaErrors(cudaMemcpyFromSymbolAsync(
                        h_CallValue,
                        d_CallValue,
                        plan->optionCount * sizeof(__TOptionValue), (size_t)0,
cudaMemcpyDeviceToHost, stream
                    ));

    //cudaDeviceSynchronize();

}
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/*
 * This sample evaluates fair call price for a
 * given set of European options using Monte Carlo approach.
 * See supplied whitepaper for more explanations.
 */



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda_runtime.h>

// includes, project
#include <helper_functions.h> // Helper functions (utilities, parsing, timing)
#include <helper_cuda.h>      // helper functions (cuda error checking and
intialization)
#include <multithreading.h>

#include "MonteCarlo_common.h"

int *pArgc = NULL;
char **pArgv = NULL;

#ifdef WIN32
#define strcasecmp _strcmpi
#endif

////////////////////////////////////////////////////////////////////////////////
// Common functions
////////////////////////////////////////////////////////////////////////////////
float randFloat(float low, float high)
{
    float t = (float)rand() / (float)RAND_MAX;
    return (1.0f - t) * low + t * high;
}

/// Utility function to tweak problem size for small GPUs
int adjustProblemSize(int GPU_N, int default_nOptions)
{
    int nOptions = default_nOptions;

    // select problem size
    for (int i=0; i<GPU_N; i++)
    {
        cudaDeviceProp deviceProp;
        checkCudaErrors(cudaGetDeviceProperties(&deviceProp, i));
```

```
        int cudaCores = _ConvertSMVer2Cores(deviceProp.major, deviceProp.minor)
                        * deviceProp.multiProcessorCount;

        if (cudaCores <= 32)
        {
            nOptions = (nOptions < cudaCores/2 ? nOptions : cudaCores/2);
        }
    }

    return nOptions;
}



////////////////////////////////////////////////////////////////////////////////
// CPU reference functions
////////////////////////////////////////////////////////////////////////////////
extern "C" void MonteCarloCPU(
    TOptionValue   &callValue,
    TOptionData optionData,
    float *h_Random,
    int pathN
);

//Black-Scholes formula for call options
extern "C" void BlackScholesCall(
    float &CallResult,
    TOptionData optionData
);


////////////////////////////////////////////////////////////////////////////////
// GPU-driving host thread
////////////////////////////////////////////////////////////////////////////////
//Timer
StopWatchInterface **hTimer = NULL;

static CUT_THREADPROC solverThread(TOptionPlan *plan)
{
    //Init GPU
    checkCudaErrors(cudaSetDevice(plan->device));

    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, plan->device));

    //Start the timer
    sdkStartTimer(&hTimer[plan->device]);

    // Allocate intermediate memory for MC integrator and initialize
    // RNG states
    initMonteCarloGPU(plan);

    // Main commputation
    MonteCarloGPU(plan);

    checkCudaErrors(cudaDeviceSynchronize());

    //Stop the timer
    sdkStopTimer(&hTimer[plan->device]);
```

```
    //Shut down this GPU
    closeMonteCarloGPU(plan);

    cudaStreamSynchronize(0);

    printf("solverThread() finished - GPU Device %d: %s\n", plan->device,
deviceProp.name);
    cudaDeviceReset();
    CUT_THREADEND;
}

static void multiSolver(TOptionPlan *plan, int nPlans)
{

    // allocate and initialize an array of stream handles
    cudaStream_t *streams = (cudaStream_t *) malloc(nPlans * sizeof(cudaStream_t));
    cudaEvent_t *events = (cudaEvent_t *)malloc(nPlans * sizeof(cudaEvent_t));

    for (int i = 0; i < nPlans; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));
        checkCudaErrors(cudaStreamCreate(&(streams[i])));
        checkCudaErrors(cudaEventCreate(&(events[i])));
    }

    //Init Each GPU
    // In CUDA 4.0 we can call cudaSetDevice multiple times to target each device
    // Set the device desired, then perform initializations on that device

    for (int i=0 ; i<nPlans ; i++)
    {
        // set the target device to perform initialization on
        checkCudaErrors(cudaSetDevice(plan[i].device));

        cudaDeviceProp deviceProp;
        checkCudaErrors(cudaGetDeviceProperties(&deviceProp, plan[i].device));

        // Allocate intermediate memory for MC integrator
        // and initialize RNG state
        initMonteCarloGPU(&plan[i]);
    }

    //Start the timer
    sdkResetTimer(&hTimer[0]);
    sdkStartTimer(&hTimer[0]);

    for (int i=0; i<nPlans; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));

        //Main computations
        MonteCarloGPU(&plan[i], streams[i]);

        checkCudaErrors(cudaEventRecord(events[i]));
    }

    for (int i=0; i<nPlans; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));
        cudaEventSynchronize(events[i]);
```

```
    }

    //Stop the timer
    sdkStopTimer(&hTimer[0]);

    for (int i=0 ; i<nPlans ; i++)
    {
        checkCudaErrors(cudaSetDevice(plan[i].device));
        closeMonteCarloGPU(&plan[i]);
        checkCudaErrors(cudaStreamDestroy(streams[i]));
        checkCudaErrors(cudaEventDestroy(events[i]));
    }
}



////////////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////////////
#define DO_CPU
#undef DO_CPU

#define PRINT_RESULTS
#undef PRINT_RESULTS


void usage()
{
    printf("--method=[threaded,streamed] --scaling=[strong,weak] [--help]\n");
    printf("Method=threaded: 1 CPU thread for each GPU     [default]\n");
    printf("       streamed: 1 CPU thread handles all GPUs (requires CUDA 4.0 or
newer)\n");
    printf("Scaling=strong : constant problem size\n");
    printf("        weak   : problem size scales with number of available GPUs
[default]\n");
}


int main(int argc, char **argv)
{
    char *multiMethodChoice = NULL;
    char *scalingChoice = NULL;
    bool use_threads = true;
    bool bqatest = false;
    bool strongScaling = false;

    pArgc = &argc;
    pArgv = argv;

    printf("%s Starting...\n\n", argv[0]);

    if (checkCmdLineFlag(argc, (const char **)argv, "qatest"))
    {
        bqatest = true;
    }

    getCmdLineArgumentString(argc, (const char **)argv, "method", &multiMethodChoice);
    getCmdLineArgumentString(argc, (const char **)argv, "scaling", &scalingChoice);

    if (checkCmdLineFlag(argc, (const char **)argv, "h") ||
```

```
    checkCmdLineFlag(argc, (const char **)argv, "help"))
{
    usage();
    exit(EXIT_SUCCESS);
}

if (multiMethodChoice == NULL)
{
    use_threads = true;
}
else
{
    if (!strcasecmp(multiMethodChoice, "threaded"))
    {
        use_threads = true;
    }
    else
    {
        use_threads = false;
    }
}

if (use_threads == false)
{
    printf("Using single CPU thread for multiple GPUs\n");
}

if (scalingChoice == NULL)
{
    strongScaling = false;
}
else
{
    if (!strcasecmp(scalingChoice, "strong"))
    {
        strongScaling = true;
    }
    else
    {
        strongScaling = false;
    }
}


//GPU number present in the system
int GPU_N;
checkCudaErrors(cudaGetDeviceCount(&GPU_N));
int nOptions = 256;

nOptions = adjustProblemSize(GPU_N, nOptions);

// select problem size
int scale = (strongScaling) ? 1 : GPU_N;
int OPT_N = nOptions * scale;
int PATH_N = 262144;
const unsigned long long SEED = 777;

// initialize the timers
hTimer = new StopWatchInterface*[GPU_N];
```

```
for (int i=0; i<GPU_N; i++)
{
    sdkCreateTimer(&hTimer[i]);
    sdkResetTimer(&hTimer[i]);
}

//Input data array
TOptionData  *optionData   = new TOptionData[OPT_N];
//Final GPU MC results
TOptionValue *callValueGPU = new TOptionValue[OPT_N];
//"Theoretical" call values by Black-Scholes formula
float *callValueBS = new float[OPT_N];
//Solver config
TOptionPlan *optionSolver = new TOptionPlan[GPU_N];
//OS thread ID
CUTThread *threadID = new CUTThread[GPU_N];

int gpuBase, gpuIndex;
int i;

float time;

double delta, ref, sumDelta, sumRef, sumReserve;

printf("MonteCarloMultiGPU\n");
printf("==================\n");
printf("Parallelization method  = %s\n", use_threads ? "threaded" : "streamed");
printf("Problem scaling         = %s\n", strongScaling? "strong" : "weak");
printf("Number of GPUs          = %d\n", GPU_N);
printf("Total number of options = %d\n", OPT_N);
printf("Number of paths         = %d\n", PATH_N);


printf("main(): generating input data...\n");
srand(123);

for (i=0; i < OPT_N; i++)
{
    optionData[i].S = randFloat(5.0f, 50.0f);
    optionData[i].X = randFloat(10.0f, 25.0f);
    optionData[i].T = randFloat(1.0f, 5.0f);
    optionData[i].R = 0.06f;
    optionData[i].V = 0.10f;
    callValueGPU[i].Expected   = -1.0f;
    callValueGPU[i].Confidence = -1.0f;
}

printf("main(): starting %i host threads...\n", GPU_N);


//Get option count for each GPU
for (i = 0; i < GPU_N; i++)
{
    optionSolver[i].optionCount = OPT_N / GPU_N;
}

//Take into account cases with "odd" option counts
for (i = 0; i < (OPT_N % GPU_N); i++)
{
    optionSolver[i].optionCount++;
```

```
    }

    //Assign GPU option ranges
    gpuBase = 0;

    for (i = 0; i < GPU_N; i++)
    {
        optionSolver[i].device     = i;
        optionSolver[i].optionData = optionData   + gpuBase;
        optionSolver[i].callValue  = callValueGPU + gpuBase;
        // all devices use the same global seed, but start
        // the sequence at a different offset
        optionSolver[i].seed       = SEED;
        optionSolver[i].pathN      = PATH_N;
        gpuBase += optionSolver[i].optionCount;
    }


    if (use_threads || bqatest)
    {
        //Start CPU thread for each GPU
        for (gpuIndex = 0; gpuIndex < GPU_N; gpuIndex++)
        {
            threadID[gpuIndex] = cutStartThread((CUT_THREADROUTINE)solverThread,
&optionSolver[gpuIndex]);
        }

        printf("main(): waiting for GPU results...\n");
        cutWaitForThreads(threadID, GPU_N);

        printf("main(): GPU statistics, threaded\n");

        for (i = 0; i < GPU_N; i++)
        {
            cudaDeviceProp deviceProp;
            checkCudaErrors(cudaGetDeviceProperties(&deviceProp,
optionSolver[i].device));
            printf("GPU Device #%i: %s\n", optionSolver[i].device, deviceProp.name);
            printf("Options         : %i\n", optionSolver[i].optionCount);
            printf("Simulation paths: %i\n", optionSolver[i].pathN);
            time = sdkGetTimerValue(&hTimer[i]);
            printf("Total time (ms.): %f\n", time);
            printf("Options per sec.: %f\n", OPT_N / (time * 0.001));
        }

        printf("main(): comparing Monte Carlo and Black-Scholes results...\n");
        sumDelta   = 0;
        sumRef     = 0;
        sumReserve = 0;

        for (i = 0; i < OPT_N; i++)
        {
            BlackScholesCall(callValueBS[i], optionData[i]);
            delta     = fabs(callValueBS[i] - callValueGPU[i].Expected);
            ref       = callValueBS[i];
            sumDelta += delta;
            sumRef   += fabs(ref);

            if (delta > 1e-6)
            {
```

```
                sumReserve += callValueGPU[i].Confidence / delta;
            }

#ifdef PRINT_RESULTS
            printf("BS: %f; delta: %E\n", callValueBS[i], delta);
#endif

        }

        sumReserve /= OPT_N;
    }

    if (!use_threads || bqatest)
    {
        multiSolver(optionSolver, GPU_N);

        printf("main(): GPU statistics, streamed\n");

        for (i = 0; i < GPU_N; i++)
        {
            cudaDeviceProp deviceProp;
            checkCudaErrors(cudaGetDeviceProperties(&deviceProp,
optionSolver[i].device));
            printf("GPU Device #%i: %s\n", optionSolver[i].device, deviceProp.name);
            printf("Options         : %i\n", optionSolver[i].optionCount);
            printf("Simulation paths: %i\n", optionSolver[i].pathN);
        }

        time = sdkGetTimerValue(&hTimer[0]);
        printf("\nTotal time (ms.): %f\n", time);
        printf("\tNote: This is elapsed time for all to compute.\n");
        printf("Options per sec.: %f\n", OPT_N / (time * 0.001));

        printf("main(): comparing Monte Carlo and Black-Scholes results...\n");
        sumDelta   = 0;
        sumRef     = 0;
        sumReserve = 0;

        for (i = 0; i < OPT_N; i++)
        {
            BlackScholesCall(callValueBS[i], optionData[i]);
            delta     = fabs(callValueBS[i] - callValueGPU[i].Expected);
            ref       = callValueBS[i];
            sumDelta += delta;
            sumRef   += fabs(ref);

            if (delta > 1e-6)
            {
                sumReserve += callValueGPU[i].Confidence / delta;
            }

#ifdef PRINT_RESULTS
            printf("BS: %f; delta: %E\n", callValueBS[i], delta);
#endif
        }

        sumReserve /= OPT_N;
    }

#ifdef DO_CPU
```

```
    printf("main(): running CPU MonteCarlo...\n");
    TOptionValue callValueCPU;
    sumDelta = 0;
    sumRef   = 0;

    for (i = 0; i < OPT_N; i++)
    {
        MonteCarloCPU(
            callValueCPU,
            optionData[i],
            NULL,
            PATH_N
        );
        delta     = fabs(callValueCPU.Expected - callValueGPU[i].Expected);
        ref       = callValueCPU.Expected;
        sumDelta += delta;
        sumRef   += fabs(ref);
        printf("Exp : %f | %f\t", callValueCPU.Expected,   callValueGPU[i].Expected);
        printf("Conf: %f | %f\n", callValueCPU.Confidence,
callValueGPU[i].Confidence);
    }

    printf("L1 norm: %E\n", sumDelta / sumRef);
#endif

    printf("Shutting down...\n");

    for (int i=0; i<GPU_N; i++)
    {
        sdkStartTimer(&hTimer[i]);
        checkCudaErrors(cudaSetDevice(i));
        cudaDeviceReset();
    }

    delete[] optionSolver;
    delete[] callValueBS;
    delete[] callValueGPU;
    delete[] optionData;
    delete[] threadID;
    delete[] hTimer;

    printf("Test Summary...\n");
    printf("L1 norm        : %E\n", sumDelta / sumRef);
    printf("Average reserve: %f\n", sumReserve);
    printf(sumReserve > 1.0f ? "Test passed\n" : "Test failed!\n");
    exit(sumReserve > 1.0f ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

```
#ifndef MONTECARLO_REDUCTION_CUH
#define MONTECARLO_REDUCTION_CUH

template<class T, unsigned int blockSize>
__device__ void sumReduceSharedMem(volatile T *sum, volatile T *sum2, int tid)
{
    // do reduction in shared mem
    if (blockSize >= 512)
    {
        if (tid < 256)
        {
            sum[tid] += sum[tid + 256];
            sum2[tid] += sum2[tid + 256];
        }

        __syncthreads();
    }
    if (blockSize >= 256)
    {
        if (tid < 128)
        {
            sum[tid] += sum[tid + 128];
            sum2[tid] += sum2[tid + 128];
        }

        __syncthreads();
    }
    if (blockSize >= 128)
    {
        if (tid <  64)
        {
            sum[tid] += sum[tid +  64];
            sum2[tid] += sum2[tid +  64];
        }

        __syncthreads();
    }
    if (tid < 32)
    {
        if (blockSize >=  64)
        {
            sum[tid] += sum[tid + 32];
            sum2[tid] += sum2[tid + 32];
        }
        if (blockSize >=  32)
        {
            sum[tid] += sum[tid + 16];
            sum2[tid] += sum2[tid + 16];
```

```
        }
        if (blockSize >=  16)
        {
            sum[tid] += sum[tid +  8];
            sum2[tid] += sum2[tid +  8];
        }
        if (blockSize >=   8)
        {
            sum[tid] += sum[tid +  4];
            sum2[tid] += sum2[tid +  4];
        }
        if (blockSize >=   4)
        {
            sum[tid] += sum[tid +  2];
            sum2[tid] += sum2[tid +  2];
        }
        if (blockSize >=   2)
        {
            sum[tid] += sum[tid +  1];
            sum2[tid] += sum2[tid +  1];
        }
    }
}
////////////////////////////////////////////////////////////////////////////
// This function calculates total sum for each of the two input arrays.
// SUM_N must be power of two
// Unrolling provides a bit of a performance improvement for small
// to medium path counts.
////////////////////////////////////////////////////////////////////////////
#define UNROLL_REDUCTION

template<class T, int SUM_N, int blockSize>
__device__ void sumReduce(T *sum, T *sum2)
{
#ifdef UNROLL_REDUCTION
    for (int pos = threadIdx.x; pos < SUM_N; pos += blockSize)
    {
        __syncthreads();
        sumReduceSharedMem<T, blockSize>(sum, sum2, pos);
    }
#else
    for (int stride = SUM_N / 2; stride > 0; stride >>= 1)
    {
        __syncthreads();

        for (int pos = threadIdx.x; pos < stride; pos += blockSize)
        {
            sum[pos] += sum[pos + stride];
            sum2[pos] += sum2[pos + stride];
        }
    }
#endif
}
#endif
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#ifndef MULTITHREADING_H
#define MULTITHREADING_H


//Simple portable thread library.

#if _WIN32
//Windows threads.
#include <windows.h>

typedef HANDLE CUTThread;
typedef unsigned(WINAPI *CUT_THREADROUTINE)(void *);

#define CUT_THREADPROC unsigned WINAPI
#define  CUT_THREADEND return 0

#else
//POSIX threads.
#include <pthread.h>

typedef pthread_t CUTThread;
typedef void *(*CUT_THREADROUTINE)(void *);

#define CUT_THREADPROC void
#define  CUT_THREADEND
#endif

#ifdef __cplusplus
extern "C" {
#endif

    //Create thread.
    CUTThread cutStartThread(CUT_THREADROUTINE, void *data);

    //Wait for thread to finish.
    void cutEndThread(CUTThread thread);

    //Destroy thread.
    void cutDestroyThread(CUTThread thread);

    //Wait for multiple threads.
    void cutWaitForThreads(const CUTThread *threads, int num);

#ifdef __cplusplus
} //extern "C"
#endif

#endif //MULTITHREADING_H
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#include <multithreading.h>

#if _WIN32
//Create thread
CUTThread cutStartThread(CUT_THREADROUTINE func, void *data)
{
    return CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)func, data, 0, NULL);
}

//Wait for thread to finish
void cutEndThread(CUTThread thread)
{
    WaitForSingleObject(thread, INFINITE);
    CloseHandle(thread);
}

//Destroy thread
void cutDestroyThread(CUTThread thread)
{
    TerminateThread(thread, 0);
    CloseHandle(thread);
}

//Wait for multiple threads
void cutWaitForThreads(const CUTThread *threads, int num)
{
    WaitForMultipleObjects(num, threads, true, INFINITE);

    for (int i = 0; i < num; i++)
    {
        CloseHandle(threads[i]);
    }
}

#else
//Create thread
CUTThread cutStartThread(CUT_THREADROUTINE func, void *data)
{
    pthread_t thread;
    pthread_create(&thread, NULL, func, data);
    return thread;
}

//Wait for thread to finish
void cutEndThread(CUTThread thread)
{
```

```
    pthread_join(thread, NULL);
}

//Destroy thread
void cutDestroyThread(CUTThread thread)
{
    pthread_cancel(thread);
}

//Wait for multiple threads
void cutWaitForThreads(const CUTThread *threads, int num)
{
    for (int i = 0; i < num; i++)
    {
        cutEndThread(threads[i]);
    }
}

#endif
```

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#ifndef REALTYPE_H
#define REALTYPE_H



//Throw out an error for unsupported target
#if defined(DOUBLE_PRECISION) && defined(__CUDACC__) &&
defined(CUDA_NO_SM_13_DOUBLE_INTRINSICS)
#error -arch sm_13 nvcc flag is required to compile in double-precision mode
#endif


#ifndef DOUBLE_PRECISION
typedef float real;
#else
typedef double real;
#endif



#endif
```

A.8. Generador de números casi-aleatorios

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
GENCODE_SM13    := -gencode arch=compute_13,code=sm_13
endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
```

```
################################################################################

# Target rules
all: build

build: quasirandomGenerator

quasirandomGenerator_SM10.o: quasirandomGenerator_SM10.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_SM10) $(GENCODE_SM20) $
(GENCODE_SM30) -o $@ -c $<

quasirandomGenerator_SM13.o: quasirandomGenerator_SM13.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_SM13) $(GENCODE_SM20) $
(GENCODE_SM30) -o $@ -c $<

quasirandomGenerator_gold.o: quasirandomGenerator_gold.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

quasirandomGenerator.o: quasirandomGenerator.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

quasirandomGenerator: quasirandomGenerator.o quasirandomGenerator_gold.o
quasirandomGenerator_SM10.o quasirandomGenerator_SM13.o
	$(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./quasirandomGenerator

clean:
	rm -f quasirandomGenerator quasirandomGenerator.o
quasirandomGenerator_gold.o quasirandomGenerator_SM10.o
quasirandomGenerator_SM13.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/quasirandomGenerator

clobber: clean
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

// CUDA Runtime
#include <cuda_runtime.h>

// Utilities and system includes
#include <helper_functions.h>
#include <helper_cuda.h>

#include "quasirandomGenerator_common.h"

////////////////////////////////////////////////////////////////////////////////
// CPU code
////////////////////////////////////////////////////////////////////////////////
extern "C" void initQuasirandomGenerator(
    unsigned int table[QRNG_DIMENSIONS][QRNG_RESOLUTION]
);

extern "C" float getQuasirandomValue(
    unsigned int table[QRNG_DIMENSIONS][QRNG_RESOLUTION],
    int i,
    int dim
);

extern "C" double getQuasirandomValue63(INT64 i, int dim);
extern "C" double MoroInvCNDcpu(unsigned int p);

////////////////////////////////////////////////////////////////////////////////
// GPU code
////////////////////////////////////////////////////////////////////////////////
extern "C" void initTable_SM10(unsigned int tableCPU[QRNG_DIMENSIONS]
[QRNG_RESOLUTION]);
extern "C" void quasirandomGenerator_SM10(float *d_Output, unsigned int seed, unsigned
int N);
extern "C" void inverseCND_SM10(float *d_Output, unsigned int *d_Input, unsigned int
N);
extern "C" void initTable_SM13(unsigned int tableCPU[QRNG_DIMENSIONS]
[QRNG_RESOLUTION]);
extern "C" void quasirandomGenerator_SM13(float *d_Output, unsigned int seed, unsigned
int N);
extern "C" void inverseCND_SM13(float *d_Output, unsigned int *d_Input, unsigned int
N);

const int N = 1048576;

int main(int argc, char **argv)
{
    // Start logs
    printf("%s Starting...\n\n", argv[0]);
```

```
    unsigned int useDoublePrecision;

    char *precisionChoice;
    getCmdLineArgumentString(argc, (const char **)argv, "type", &precisionChoice);

    if (precisionChoice == NULL)
    {
        useDoublePrecision = 0;
    }
    else
    {
        if (!STRCASECMP(precisionChoice, "double"))
        {
            useDoublePrecision = 1;
        }
        else
        {
            useDoublePrecision = 0;
        }
    }

    unsigned int tableCPU[QRNG_DIMENSIONS][QRNG_RESOLUTION];

    float *h_OutputGPU, *d_Output;

    int dim, pos;
    double delta, ref, sumDelta, sumRef, L1norm, gpuTime;

    StopWatchInterface *hTimer = NULL;

    if (sizeof(INT64) != 8)
    {
        printf("sizeof(INT64) != 8\n");
        return 0;
    }

    // use command-line specified CUDA device, otherwise use device with highest
Gflops/s
    int dev = findCudaDevice(argc, (const char **)argv);

    sdkCreateTimer(&hTimer);

    int deviceIndex;
    checkCudaErrors(cudaGetDevice(&deviceIndex));
    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, deviceIndex));
    int version = deviceProp.major * 10 + deviceProp.minor;

    if (useDoublePrecision && version < 13)
    {
        printf("Double precision not supported.\n");
        cudaDeviceReset();
        return 0;
    }

    printf("Allocating GPU memory...\n");
    checkCudaErrors(cudaMalloc((void **)&d_Output, QRNG_DIMENSIONS * N *
sizeof(float)));

    printf("Allocating CPU memory...\n");
```

```
    h_OutputGPU = (float *)malloc(QRNG_DIMENSIONS * N * sizeof(float));

    printf("Initializing QRNG tables...\n\n");
    initQuasirandomGenerator(tableCPU);

    if (useDoublePrecision)
    {
        initTable_SM13(tableCPU);
    }
    else
    {
        initTable_SM10(tableCPU);
    }

    printf("Testing QRNG...\n\n");
    checkCudaErrors(cudaMemset(d_Output, 0, QRNG_DIMENSIONS * N * sizeof(float)));
    int numIterations = 20;

    for (int i = -1; i < numIterations; i++)
    {
        if (i == 0)
        {
            checkCudaErrors(cudaDeviceSynchronize());
            sdkResetTimer(&hTimer);
            sdkStartTimer(&hTimer);
        }

        if (useDoublePrecision)
        {
            quasirandomGenerator_SM13(d_Output, 0, N);
        }
        else
        {
            quasirandomGenerator_SM10(d_Output, 0, N);
        }
    }

    checkCudaErrors(cudaDeviceSynchronize());
    sdkStopTimer(&hTimer);
    gpuTime = sdkGetTimerValue(&hTimer)/(double)numIterations*1e-3;
    printf("quasirandomGenerator, Throughput = %.4f GNumbers/s, Time = %.5f s, Size =
%u Numbers, NumDevsUsed = %u, Workgroup = %u\n",
           (double)QRNG_DIMENSIONS * (double)N * 1.0E-9 / gpuTime, gpuTime,
QRNG_DIMENSIONS*N, 1, 128*QRNG_DIMENSIONS);

    printf("\nReading GPU results...\n");
    checkCudaErrors(cudaMemcpy(h_OutputGPU, d_Output, QRNG_DIMENSIONS * N *
sizeof(float), cudaMemcpyDeviceToHost));

    printf("Comparing to the CPU results...\n\n");
    sumDelta = 0;
    sumRef = 0;

    for (dim = 0; dim < QRNG_DIMENSIONS; dim++)
        for (pos = 0; pos < N; pos++)
        {
            ref       = getQuasirandomValue63(pos, dim);
            delta     = (double)h_OutputGPU[dim * N + pos] - ref;
            sumDelta += fabs(delta);
            sumRef   += fabs(ref);
```

```
        }

    printf("L1 norm: %E\n", sumDelta / sumRef);

    printf("\nTesting inverseCNDgpu()...\n\n");
    checkCudaErrors(cudaMemset(d_Output, 0, QRNG_DIMENSIONS * N * sizeof(float)));

    for (int i = -1; i < numIterations; i++)
    {
        if (i == 0)
        {
            checkCudaErrors(cudaDeviceSynchronize());
            sdkResetTimer(&hTimer);
            sdkStartTimer(&hTimer);
        }

        if (useDoublePrecision)
        {
            inverseCND_SM13(d_Output, NULL, QRNG_DIMENSIONS * N);
        }
        else
        {
            inverseCND_SM10(d_Output, NULL, QRNG_DIMENSIONS * N);
        }
    }

    checkCudaErrors(cudaDeviceSynchronize());
    sdkStopTimer(&hTimer);
    gpuTime = sdkGetTimerValue(&hTimer)/(double)numIterations*1e-3;
    printf("quasirandomGenerator-inverse, Throughput = %.4f GNumbers/s, Time = %.5f s,
Size = %u Numbers, NumDevsUsed = %u, Workgroup = %u\n",
           (double)QRNG_DIMENSIONS * (double)N * 1E-9 / gpuTime, gpuTime,
QRNG_DIMENSIONS*N, 1, 128);

    printf("Reading GPU results...\n");
    checkCudaErrors(cudaMemcpy(h_OutputGPU, d_Output, QRNG_DIMENSIONS * N *
sizeof(float), cudaMemcpyDeviceToHost));

    printf("\nComparing to the CPU results...\n");
    sumDelta = 0;
    sumRef = 0;
    unsigned int distance = ((unsigned int)-1) / (QRNG_DIMENSIONS * N + 1);

    for (pos = 0; pos < QRNG_DIMENSIONS * N; pos++)
    {
        unsigned int d = (pos + 1) * distance;
        ref       = MoroInvCNDcpu(d);
        delta     = (double)h_OutputGPU[pos] - ref;
        sumDelta += fabs(delta);
        sumRef   += fabs(ref);
    }

    printf("L1 norm: %E\n\n", L1norm = sumDelta / sumRef);
    printf("Shutting down...\n");
    sdkDeleteTimer(&hTimer);
    free(h_OutputGPU);
    checkCudaErrors(cudaFree(d_Output));
    cudaDeviceReset();
    exit(L1norm < 1e-6 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

```
#ifndef QUASIRANDOMGENERATOR_COMMON_H
#define QUASIRANDOMGENERATOR_COMMON_H



////////////////////////////////////////////////////////////////////////////////
// Global types and constants
////////////////////////////////////////////////////////////////////////////////
typedef long long int INT64;

#define QRNG_DIMENSIONS 3
#define QRNG_RESOLUTION 31
#define INT_SCALE (1.0f / (float)0x80000001U)



#endif
```

```
#include <stdio.h>
#include <math.h>

#include "quasirandomGenerator_common.h"



////////////////////////////////////////////////////////////////////////////////
// Table generation functions
////////////////////////////////////////////////////////////////////////////////
// Internal 64(63)-bit table
static INT64 cjn[63][QRNG_DIMENSIONS];

static int GeneratePolynomials(int buffer[QRNG_DIMENSIONS], bool primitive)
{
    int i, j, n, p1, p2, l;
    int e_p1, e_p2, e_b;

    //generate all polynomials to buffer
    for (n = 1, buffer[0] = 0x2, p2 = 0, l = 0; n < QRNG_DIMENSIONS; ++n)
    {
        //search for the next irreducable polynomial
        for (p1 = buffer[n - 1] + 1; ; ++p1)
        {
            //find degree of polynomial p1
            for (e_p1 = 30; (p1 & (1 << e_p1)) == 0; --e_p1) {}

            // try to divide p1 by all polynomials in buffer
            for (i = 0; i < n; ++i)
            {
                // find the degree of buffer[i]
                for (e_b = e_p1; (buffer[i] & (1 << e_b)) == 0; --e_b) {}

                // divide p2 by buffer[i] until the end
                for (p2 = (buffer[i] << ((e_p2 = e_p1) - e_b)) ^ p1; p2 >= buffer[i];
p2 = (buffer[i] << (e_p2 - e_b)) ^ p2)
                {
                    for (; (p2 & (1 << e_p2)) == 0; --e_p2) {}
                }// compute new degree of p2

                // division without remainder!!! p1 is not irreducable
                if (p2 == 0)
                {
                    break;
                }
            }
```

```
            //all divisions were with remainder - p1 is irreducable
            if (p2 != 0)
            {
                e_p2 = 0;

                if (primitive)
                {
                    //check that p1 has only one cycle (i.e. is monic, or primitive)
                    j = ~(0xffffffff << (e_p1 + 1));
                    e_b = (1 << e_p1) | 0x1;

                    for (p2 = e_b, e_p2 = (1 << e_p1) - 2; e_p2 > 0; --e_p2)
                    {
                        p2 <<= 1;
                        i = p2 & p1;
                        i = (i & 0x55555555) + ((i >> 1) & 0x55555555);
                        i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
                        i = (i & 0x07070707) + ((i >> 4) & 0x07070707);
                        p2 |= (i % 255) & 1;

                        if ((p2 & j) == e_b) break;
                    }
                }

                //it is monic - add it to the list of polynomials
                if (e_p2 == 0)
                {
                    buffer[n] = p1;
                    l += e_p1;
                    break;
                }
            }
        }
    }

    return l + 1;
}



////////////////////////////////////////////////////////////////////////////
//  @misc{Bratley92:LDS,
//    author = "B. Fox and P. Bratley and H. Niederreiter",
//    title = "Implementation and test of low discrepancy sequences",
//    text = "B. L. Fox, P. Bratley, and H. Niederreiter. Implementation and test of
//      low discrepancy sequences. ACM Trans. Model. Comput. Simul., 2(3):195--213,
//      July 1992.",
//    year = "1992" }
////////////////////////////////////////////////////////////////////////////
static void GenerateCJ()
{
    int buffer[QRNG_DIMENSIONS];
    int *polynomials;
    int n, p1, l, e_p1;

    // Niederreiter (in contrast to Sobol) allows to use not primitive, but just
irreducable polynomials
    l = GeneratePolynomials(buffer, false);
```

```
// convert all polynomials from buffer to polynomials table
polynomials = new int[l + 2 * QRNG_DIMENSIONS + 1];

for (n = 0, l = 0; n < QRNG_DIMENSIONS; ++n)
{
    //find degree of polynomial p1
    for (p1 = buffer[n], e_p1 = 30; (p1 & (1 << e_p1)) == 0; --e_p1) {}

    //fill polynomials table with values for this polynomial
    polynomials[l++] = 1;

    for (--e_p1; e_p1 >= 0; --e_p1)
    {
        polynomials[l++] = (p1 >> e_p1) & 1;
    }

    polynomials[l++] = -1;
}

polynomials[l] = -1;

// irreducable polynomial p
int *p = polynomials, e, d;
// polynomial b
int b_arr[1024], *b, m;
// v array
int v_arr[1024], *v;
// temporary polynomial, required to do multiplication of p and b
int t_arr[1024], *t;
// subsidiary variables
int i, j, u, m1, ip, it;

// cycle over monic irreducible polynomials
for (d = 0; p[0] != -1; p += e + 2)
{
    // allocate memory for cj array for dimention (ip + 1)
    for (i = 0; i < 63; ++i)
    {
        cjn[i][d] = 0;
    }

    // determine the power of irreducable polynomial
    for (e = 0; p[e + 1] != -1; ++e) {}

    // polynomial b in the beginning is just '1'
    (b = b_arr + 1023)[m = 0] = 1;
    // v array needs only (63 + e - 2) length
    v = v_arr + 1023 - (63 + e - 2);

    // cycle over all coefficients
    for (j = 63 - 1, u = e; j >= 0; --j, ++u)
    {
        if (u == e)
        {
            u = 0;

            // multiply b by p (polynomials multiplication)
            for (i = 0, t = t_arr + 1023 - (m1 = m); i <= m; ++i)
            {
                t[i] = b[i];
```

```
                }

                b = b_arr + 1023 - (m += e);

                for (i = 0; i <= m; ++i)
                {
                    b[i] = 0;

                    for (ip = e - (m - i), it = m1; ip <= e && it >= 0; ++ip, --it)
                    {
                        if (ip >= 0)
                        {
                            b[i] ^= p[ip] & t[it];
                        }
                    }
                }

                // multiplication of polynomials finished

                // calculate v
                for (i = 0; i < m1; ++i)
                {
                    v[i] = 0;
                }

                for (; i < m; ++i)
                {
                    v[i] = 1;
                }

                for (; i <= 63 + e - 2; ++i)
                {
                    v[i] = 0;

                    for (it = 1; it <= m; ++it)
                    {
                        v[i] ^= v[i - it] & b[it];
                    }
                }
            }

            // copy calculated v to cj
            for (i = 0; i < 63; i++)
            {
                cjn[i][d] |= (INT64)v[i + u] << j;
            }
        }

        ++d;
    }

    delete []polynomials;
}

//Generate 63-bit quasirandom number for given index and dimension and normalize
extern "C" double getQuasirandomValue63(INT64 i, int dim)
{
    const double INT63_SCALE = (1.0 / (double)0x8000000000000001ULL);
    INT64 result = 0;
```

```
    for (int bit = 0; bit < 63; bit++, i >>= 1)
        if (i & 1) result ^= cjn[bit][dim];

    return (double)(result + 1) * INT63_SCALE;
}



////////////////////////////////////////////////////////////////////////////////
// Initialization (table setup)
////////////////////////////////////////////////////////////////////////////////
extern "C" void initQuasirandomGenerator(
    unsigned int table[QRNG_DIMENSIONS][QRNG_RESOLUTION]
)
{
    GenerateCJ();

    for (int dim = 0; dim < QRNG_DIMENSIONS; dim++)
        for (int bit = 0; bit < QRNG_RESOLUTION; bit++)
            table[dim][bit] = (int)((cjn[bit][dim] >> 32) & 0x7FFFFFFF);
}



////////////////////////////////////////////////////////////////////////////////
// Generate 31-bit quasirandom number for given index and dimension
////////////////////////////////////////////////////////////////////////////////
extern "C" float getQuasirandomValue(
    unsigned int table[QRNG_DIMENSIONS][QRNG_RESOLUTION],
    int i,
    int dim
)
{
    int result = 0;

    for (int bit = 0; bit < QRNG_RESOLUTION; bit++, i >>= 1)
        if (i & 1) result ^= table[dim][bit];

    return (float)(result + 1) * INT_SCALE;
}



////////////////////////////////////////////////////////////////////////////////
// Moro's Inverse Cumulative Normal Distribution function approximation
////////////////////////////////////////////////////////////////////////////////
extern "C" double MoroInvCNDcpu(unsigned int x)
{
    const double a1 = 2.50662823884;
    const double a2 = -18.61500062529;
    const double a3 = 41.39119773534;
    const double a4 = -25.44106049637;
    const double b1 = -8.4735109309;
    const double b2 = 23.08336743743;
    const double b3 = -21.06224101826;
    const double b4 = 3.13082909833;
    const double c1 = 0.337475482272615;
    const double c2 = 0.976169019091719;
    const double c3 = 0.160797971491821;
    const double c4 = 2.76438810333863E-02;
```

```
    const double c5 = 3.8405729373609E-03;
    const double c6 = 3.951896511919E-04;
    const double c7 = 3.21767881768E-05;
    const double c8 = 2.888167364E-07;
    const double c9 = 3.960315187E-07;

    double z;

    bool negate = false;

    // Ensure the conversion to floating point will give a value in the
    // range (0,0.5] by restricting the input to the bottom half of the
    // input domain. We will later reflect the result if the input was
    // originally in the top half of the input domain
    if (x >= 0x80000000UL)
    {
        x = 0xffffffffUL - x;
        negate = true;
    }

    // x is now in the range [0,0x80000000) (i.e. [0,0x7fffffff])
    // Convert to floating point in (0,0.5]
    const double x1 = 1.0 / static_cast<double>(0xffffffffUL);
    const double x2 = x1 / 2.0;
    double p1 = x * x1 + x2;
    // Convert to floating point in (-0.5,0]
    double p2 = p1 - 0.5;

    // The input to the Moro inversion is p2 which is in the range
    // (-0.5,0]. This means that our output will be the negative side
    // of the bell curve (which we will reflect if "negate" is true).

    // Main body of the bell curve for |p| < 0.42
    if (p2 > -0.42)
    {
        z = p2 * p2;
        z = p2 * (((a4 * z + a3) * z + a2) * z + a1) / ((((b4 * z + b3) * z + b2) * z
+ b1) * z + 1.0);
    }
    // Special case (Chebychev) for tail
    else
    {
        z = log(-log(p1));
        z = - (c1 + z * (c2 + z * (c3 + z * (c4 + z * (c5 + z * (c6 + z * (c7 + z *
(c8 + z * c9)))))));
    }

    // If the original input (x) was in the top half of the range, reflect
    // to get the positive side of the bell curve
    return negate ? -z : z;
}
```

quasirandomGenerator_kernel.cuh

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */



#ifndef QUASIRANDOMGENERATOR_KERNEL_CUH
#define QUASIRANDOMGENERATOR_KERNEL_CUH


#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>
#include "realtype.h"
#include "quasirandomGenerator_common.h"



//Fast integer multiplication
#define MUL(a, b) __umul24(a, b)



////////////////////////////////////////////////////////////////////////////////
// Niederreiter quasirandom number generation kernel
////////////////////////////////////////////////////////////////////////////////
static __constant__ unsigned int c_Table[QRNG_DIMENSIONS][QRNG_RESOLUTION];

static __global__ void quasirandomGeneratorKernel(
    float *d_Output,
    unsigned int seed,
    unsigned int N
)
{
    unsigned int *dimBase = &c_Table[threadIdx.y][0];
    unsigned int      tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
    unsigned int  threadN = MUL(blockDim.x, gridDim.x);

    for (unsigned int pos = tid; pos < N; pos += threadN)
    {
        unsigned int result = 0;
        unsigned int data = seed + pos;

        for (int bit = 0; bit < QRNG_RESOLUTION; bit++, data >>= 1)
            if (data & 1)
            {
                result ^= dimBase[bit];
            }

        d_Output[MUL(threadIdx.y, N) + pos] = (float)(result + 1) * INT_SCALE;
```

```
    }
}

//Table initialization routine
static void initTableGPU(unsigned int tableCPU[QRNG_DIMENSIONS][QRNG_RESOLUTION])
{
    checkCudaErrors(cudaMemcpyToSymbol(
                        c_Table,
                        tableCPU,
                        QRNG_DIMENSIONS * QRNG_RESOLUTION * sizeof(unsigned int)
                    ));
}

//Host-side interface
static void quasirandomGeneratorGPU(float *d_Output, unsigned int seed, unsigned
int N)
{
    dim3 threads(128, QRNG_DIMENSIONS);
    quasirandomGeneratorKernel<<<128, threads>>>(d_Output, seed, N);
    getLastCudaError("quasirandomGeneratorKernel() execution failed.\n");
}



////////////////////////////////////////////////////////////////////////////////
// Moro's Inverse Cumulative Normal Distribution function approximation
////////////////////////////////////////////////////////////////////////////////
#ifndef DOUBLE_PRECISION
__device__ inline float MoroInvCNDgpu(unsigned int x)
{
    const float a1 = 2.50662823884f;
    const float a2 = -18.61500062529f;
    const float a3 = 41.39119773534f;
    const float a4 = -25.44106049637f;
    const float b1 = -8.4735109309f;
    const float b2 = 23.08336743743f;
    const float b3 = -21.06224101826f;
    const float b4 = 3.13082909833f;
    const float c1 = 0.337475482272615f;
    const float c2 = 0.976169019091719f;
    const float c3 = 0.160797971491821f;
    const float c4 = 2.76438810333863E-02f;
    const float c5 = 3.8405729373609E-03f;
    const float c6 = 3.951896511919E-04f;
    const float c7 = 3.21767881768E-05f;
    const float c8 = 2.888167364E-07f;
    const float c9 = 3.960315187E-07f;

    float z;

    bool negate = false;

    // Ensure the conversion to floating point will give a value in the
    // range (0,0.5] by restricting the input to the bottom half of the
    // input domain. We will later reflect the result if the input was
    // originally in the top half of the input domain
    if (x >= 0x80000000UL)
    {
```

```
        x = 0xffffffffUL - x;
        negate = true;
    }

    // x is now in the range [0,0x80000000) (i.e. [0,0x7fffffff])
    // Convert to floating point in (0,0.5]
    const float x1 = 1.0f / static_cast<float>(0xffffffffUL);
    const float x2 = x1 / 2.0f;
    float p1 = x * x1 + x2;
    // Convert to floating point in (-0.5,0]
    float p2 = p1 - 0.5f;

    // The input to the Moro inversion is p2 which is in the range
    // (-0.5,0]. This means that our output will be the negative side
    // of the bell curve (which we will reflect if "negate" is true).

    // Main body of the bell curve for |p| < 0.42
    if (p2 > -0.42f)
    {
        z = p2 * p2;
        z = p2 * (((a4 * z + a3) * z + a2) * z + a1) / ((((b4 * z + b3) * z + b2)
* z + b1) * z + 1.0f);
    }
    // Special case (Chebychev) for tail
    else
    {
        z = __logf(-__logf(p1));
        z = - (c1 + z * (c2 + z * (c3 + z * (c4 + z * (c5 + z * (c6 + z * (c7 + z
* (c8 + z * c9)))))));
    }

    // If the original input (x) was in the top half of the range, reflect
    // to get the positive side of the bell curve
    return negate ? -z : z;
}
#else
__device__ inline double MoroInvCNDgpu(unsigned int x)
{
    const double a1 = 2.50662823884;
    const double a2 = -18.61500062529;
    const double a3 = 41.39119773534;
    const double a4 = -25.44106049637;
    const double b1 = -8.4735109309;
    const double b2 = 23.08336743743;
    const double b3 = -21.06224101826;
    const double b4 = 3.13082909833;
    const double c1 = 0.337475482272615;
    const double c2 = 0.976169019091719;
    const double c3 = 0.160797971491821;
    const double c4 = 2.76438810333863E-02;
    const double c5 = 3.8405729373609E-03;
    const double c6 = 3.951896511919E-04;
    const double c7 = 3.21767881768E-05;
    const double c8 = 2.888167364E-07;
    const double c9 = 3.960315187E-07;

    double z;
```

```
    bool negate = false;

    // Ensure the conversion to floating point will give a value in the
    // range (0,0.5] by restricting the input to the bottom half of the
    // input domain. We will later reflect the result if the input was
    // originally in the top half of the input domain
    if (x >= 0x80000000UL)
    {
        x = 0xffffffffUL - x;
        negate = true;
    }

    // x is now in the range [0,0x80000000) (i.e. [0,0x7fffffff])
    // Convert to floating point in (0,0.5]
    const double x1 = 1.0 / static_cast<double>(0xffffffffUL);
    const double x2 = x1 / 2.0;
    double p1 = x * x1 + x2;
    // Convert to floating point in (-0.5,0]
    double p2 = p1 - 0.5;

    // The input to the Moro inversion is p2 which is in the range
    // (-0.5,0]. This means that our output will be the negative side
    // of the bell curve (which we will reflect if "negate" is true).

    // Main body of the bell curve for |p| < 0.42
    if (p2 > -0.42)
    {
        z = p2 * p2;
        z = p2 * (((a4 * z + a3) * z + a2) * z + a1) / ((((b4 * z + b3) * z + b2)
* z + b1) * z + 1.0);
    }
    // Special case (Chebychev) for tail
    else
    {
        z = log(-log(p1));
        z = - (c1 + z * (c2 + z * (c3 + z * (c4 + z * (c5 + z * (c6 + z * (c7 + z
* (c8 + z * c9)))))))));
    }

    // If the original input (x) was in the top half of the range, reflect
    // to get the positive side of the bell curve
    return negate ? -z : z;
}
#endif


////////////////////////////////////////////////////////////////////////////////
// Main kernel. Choose between transforming
// input sequence and uniform ascending (0, 1) sequence
////////////////////////////////////////////////////////////////////////////////
static __global__ void inverseCNDKernel(
    float *d_Output,
    unsigned int *d_Input,
    unsigned int pathN
)
{
    unsigned int distance = ((unsigned int)-1) / (pathN + 1);
    unsigned int     tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
```

```
    unsigned int threadN = MUL(blockDim.x, gridDim.x);

    //Transform input number sequence if it's supplied
    if (d_Input)
    {
        for (unsigned int pos = tid; pos < pathN; pos += threadN)
        {
            unsigned int d = d_Input[pos];
            d_Output[pos] = (float)MoroInvCNDgpu(d);
        }
    }
    //Else generate input uniformly placed samples on the fly
    //and write to destination
    else
    {
        for (unsigned int pos = tid; pos < pathN; pos += threadN)
        {
            unsigned int d = (pos + 1) * distance;
            d_Output[pos] = (float)MoroInvCNDgpu(d);
        }
    }
}

static void inverseCNDgpu(float *d_Output, unsigned int *d_Input, unsigned int N)
{
    inverseCNDKernel<<<128, 128>>>(d_Output, d_Input, N);
    getLastCudaError("inverseCNDKernel() execution failed.\n");
}



#endif
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */



#include "quasirandomGenerator_kernel.cuh"



extern "C" void initTable_SM10(unsigned int tableCPU[QRNG_DIMENSIONS]
[QRNG_RESOLUTION])
{
    initTableGPU(tableCPU);
}

extern "C" void quasirandomGenerator_SM10(float *d_Output, unsigned int seed,
unsigned int N)
{
    quasirandomGeneratorGPU(d_Output, seed, N);
}

extern "C" void inverseCND_SM10(float *d_Output, unsigned int *d_Input, unsigned
int N)
{
    inverseCNDgpu(d_Output, d_Input, N);
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */



#define DOUBLE_PRECISION
#include "quasirandomGenerator_kernel.cuh"



extern "C" void initTable_SM13(unsigned int tableCPU[QRNG_DIMENSIONS]
[QRNG_RESOLUTION])
{
    initTableGPU(tableCPU);
}

extern "C" void quasirandomGenerator_SM13(float *d_Output, unsigned int seed,
unsigned int N)
{
    quasirandomGeneratorGPU(d_Output, seed, N);
}

extern "C" void inverseCND_SM13(float *d_Output, unsigned int *d_Input, unsigned
int N)
{
    inverseCNDgpu(d_Output, d_Input, N);
}
```

```
#ifndef REALTYPE_H
#define REALTYPE_H



//Throw out an error for unsupported target
#if defined(DOUBLE_PRECISION) && defined(__CUDACC__) &&
defined(CUDA_NO_SM_13_DOUBLE_INTRINSICS)
#error -arch sm_13 nvcc flag is required to compile in double-precision mode
#endif


#ifndef DOUBLE_PRECISION
typedef float real;
#else
typedef double real;
#endif



#endif
```

A.9. Punteros de función

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findgllib.mk is used to find the necessary GL Libraries for specific distributions
#               this is supported on Mac OSX and Linux Platforms
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   # $(info DISTRO1 = $(DISTRO) $(DISTVER))
   ifeq ($(DISTRO),)
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null)
     # $(info DISTRO2 = $(DISTRO) $(DISTVER))
   endif
   ifeq ($(DISTRO),)
     # third, we can search in /etc/os-release or /etc/{distro}-release
```

```
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
     # $(info DISTRO3 = $(DISTRO) $(DISTVER))
   endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # $(info) >> findgllib.mk -> LINUX path <<<)
    # Each set of Linux Distros have different paths for where to find their OpenGL
libraries reside
    ifeq ("$(DISTRO)","ubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","debian")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","suse")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","suse linux")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
```

```
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
    GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?= /usr/X11R6/lib
    GLLINK    ?= -L/usr/X11R6/lib
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","red")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhatenterpriseworkstation")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","centos")
  ifeq ($(OS_SIZE),64)
```

```
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      GLPATH := /usr/arm-linux-gnueabihf/lib
      GLLINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        GLPATH += $(TARGET_FS)/usr/lib/nvidia-current $(TARGET_FS)/usr/lib/arm-linux-
gnueabihf
        GLLINK += -L$(TARGET_FS)/usr/lib/nvidia-current -L$(TARGET_FS)/usr/lib/arm-
linux-gnueabihf
      endif
    endif

  # find libGL, libGLU, libXi,
  GLLIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libGL.so  -print 2>/dev/null)
  GLULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libGLU.so -print 2>/dev/null)
  X11LIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libX11.so -print 2>/dev/null)
  XILIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libXi.so  -print 2>/dev/null)
  XMULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libXmu.so -print 2>/dev/null)

  ifeq ("$(GLLIB)",'')
      $(info >>> WARNING - libGL.so not found, refer to CUDA Samples release notes for
how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(GLULIB)",'')
      $(info >>> WARNING - libGLU.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(X11LIB)",'')
      $(info >>> WARNING - libX11.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XILIB)",'')
      $(info >>> WARNING - libXi.so not found, refer to CUDA Samples release notes for
how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XMULIB)",'')
      $(info >>> WARNING - libXmu.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
else
    # This would be the Mac OS X path if we had to do anything special
endif
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

// OpenGL Graphics includes
#include <GL/glew.h>
#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

// Includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <helper_functions.h> // helper functions for timing, string parsing

#include <cuda_runtime.h>     // CUDA Runtime
#include <cuda_gl_interop.h>  // CUDA OpenGL intero

#include <helper_cuda.h>      // includes for CUDA initialization and error
checking
#include <helper_cuda_gl.h>

#include "FunctionPointers_kernels.h"

#define EXIT_WAIVED 2
#define MIN_RUNTIME_VERSION 3010
#define MIN_COMPUTE_VERSION 0x20

//
// Cuda example code that implements the Sobel edge detection
// filter. This code works for 8-bit monochrome images.
//
// Use the '-' and '=' keys to change the scale factor.
//
// Other keys:
// I: display image
// T: display Sobel Edge Detection (computed solely with texture)
// S: display Sobel Edge Detection (computed with texture and shared memory)

void cleanup(void);
void initializeData(char *file);

#define MAX_EPSILON_ERROR 5.0f

static char *sSDKsample = "CUDA Function Pointers (SobelFilter)";

const char *filterMode[] =
```

```
{
    "No Filtering",
    "Sobel Texture",
    "Sobel SMEM+Texture",
    NULL
};

static int wWidth   = 512; // Window width
static int wHeight  = 512; // Window height
static int imWidth  = 0;   // Image width
static int imHeight = 0;   // Image height
static int blockOp = 0;
static int pointOp = 1;

// Code to handle Auto verification
const int frameCheckNumber = 4;
int fpsCount = 0;      // FPS count for averaging
int fpsLimit = 8;      // FPS limit for sampling
unsigned int frameCount = 0;
StopWatchInterface *timer = NULL;
unsigned int g_Bpp;

int g_TotalErrors = 0;

int *pArgc = NULL;
char **pArgv = NULL;

bool g_bQAReadback = false;

// Display Data
static GLuint pbo_buffer = 0;  // Front and back CA buffers
struct cudaGraphicsResource *cuda_pbo_resource; // CUDA Graphics Resource (to
transfer PBO)

static GLuint texid = 0;       // Texture for display
unsigned char *pixels = NULL;  // Image pixel data on the host
float imageScale = 1.f;        // Image exposure
enum SobelDisplayMode g_SobelDisplayMode;

#define OFFSET(i) ((char *)NULL + (i))
#define MAX(a,b) ((a > b) ? a : b)
#define REFRESH_DELAY     10 //ms

void computeFPS()
{
    frameCount++;
    fpsCount++;

    if (fpsCount == fpsLimit)
    {
        char fps[256];
        float ifps = 1.f / (sdkGetAverageTimerValue(&timer) / 1000.f);
        sprintf(fps, "FunctionPointers [CUDA Edge Detection] (%s): %3.1f fps",
                filterMode[g_SobelDisplayMode], ifps);

        glutSetWindowTitle(fps);
        fpsCount = 0;
```

```
        fpsLimit = (int)MAX(ifps, 1.f);
        sdkResetTimer(&timer);
    }
}

// This is the normal display path
void display(void)
{
    sdkStartTimer(&timer);

    // Sobel operation
    Pixel *data = NULL;

    // map PBO to get CUDA device pointer
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_pbo_resource, 0));
    size_t num_bytes;
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&data,
&num_bytes,
                                                         cuda_pbo_resource));
    //printf("CUDA mapped PBO: May access %ld bytes\n", num_bytes);

    sobelFilter(data, imWidth, imHeight, g_SobelDisplayMode, imageScale, blockOp,
pointOp);
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_pbo_resource, 0));

    glClear(GL_COLOR_BUFFER_BIT);

    glBindTexture(GL_TEXTURE_2D, texid);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, imWidth, imHeight,
                    GL_LUMINANCE, GL_UNSIGNED_BYTE, OFFSET(0));
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

    glDisable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glBegin(GL_QUADS);
    glVertex2f(0, 0);
    glTexCoord2f(0, 0);
    glVertex2f(0, 1);
    glTexCoord2f(1, 0);
    glVertex2f(1, 1);
    glTexCoord2f(1, 1);
    glVertex2f(1, 0);
    glTexCoord2f(0, 1);
    glEnd();
    glBindTexture(GL_TEXTURE_2D, 0);

    glutSwapBuffers();

    sdkStopTimer(&timer);
    computeFPS();
}
```

```
void timerEvent(int value)
{
    glutPostRedisplay();
    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
}

void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    char temp[256];

    switch (key)
    {
        case 27:
            exit(EXIT_SUCCESS);
            break;

        case '-':
            imageScale -= 0.1f;
            printf("brightness = %4.2f\n", imageScale);
            break;

        case '=':
            imageScale += 0.1f;
            printf("brightness = %4.2f\n", imageScale);
            break;

        case 'i':
        case 'I':
            g_SobelDisplayMode = SOBELDISPLAY_IMAGE;
            sprintf(temp, "Function Pointers [CUDA Edge Detection] (%s)",
filterMode[g_SobelDisplayMode]);
            glutSetWindowTitle(temp);
            break;

        case 's':
        case 'S':
            g_SobelDisplayMode = SOBELDISPLAY_SOBELSHARED;
            sprintf(temp, "Function Pointers [CUDA Edge Detection] (%s)",
filterMode[g_SobelDisplayMode]);
            glutSetWindowTitle(temp);
            break;

        case 't':
        case 'T':
            g_SobelDisplayMode = SOBELDISPLAY_SOBELTEX;
            sprintf(temp, "Function Pointers [CUDA Edge Detection] (%s)",
filterMode[g_SobelDisplayMode]);
            glutSetWindowTitle(temp);
            break;

        case 'b':
        case 'B':
            blockOp = (blockOp + 1)%LAST_BLOCK_FILTER;
            break;

        case 'p':
        case 'P':
            pointOp = (pointOp +1)%LAST_POINT_FILTER;
```

```
            break;

        default:
            break;

    }
}

void reshape(int x, int y)
{
    glViewport(0, 0, x, y);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 1, 0, 1, 0, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void cleanup(void)
{
    cudaGraphicsUnregisterResource(cuda_pbo_resource);

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
    glDeleteBuffers(1, &pbo_buffer);
    glDeleteTextures(1, &texid);
    deleteTexture();

    sdkDeleteTimer(&timer);
}

void initializeData(char *file)
{
    GLint bsize;
    unsigned int w, h;
    size_t file_length= strlen(file);

    if (!strcmp(&file[file_length-3], "pgm"))
    {
        if (sdkLoadPGM<unsigned char>(file, &pixels, &w, &h) != true)
        {
            printf("Failed to load PGM image file: %s\n", file);
            exit(EXIT_FAILURE);
        }

        g_Bpp = 1;
    }
    else if (!strcmp(&file[file_length-3], "ppm"))
    {
        if (sdkLoadPPM4(file, &pixels, &w, &h) != true)
        {
            printf("Failed to load PPM image file: %s\n", file);
            exit(EXIT_FAILURE);
        }

        g_Bpp = 4;
    }
    else
    {
```

```
        cudaDeviceReset();
        exit(EXIT_FAILURE);
    }

    imWidth = (int)w;
    imHeight = (int)h;
    setupTexture(imWidth, imHeight, pixels, g_Bpp);

    // copy function pointer tables to host side for later use
    setupFunctionTables();

    memset(pixels, 0x0, g_Bpp * sizeof(Pixel) * imWidth * imHeight);

    if (!g_bQAReadback)
    {
        // use OpenGL Path
        glGenBuffers(1, &pbo_buffer);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo_buffer);
        glBufferData(GL_PIXEL_UNPACK_BUFFER,
                     g_Bpp * sizeof(Pixel) * imWidth * imHeight,
                     pixels, GL_STREAM_DRAW);
        glGetBufferParameteriv(GL_PIXEL_UNPACK_BUFFER, GL_BUFFER_SIZE, &bsize);

        if ((GLuint)bsize != (g_Bpp * sizeof(Pixel) * imWidth * imHeight))
        {
            printf("Buffer object (%d) has incorrect size (%d).\n",
(unsigned)pbo_buffer, (unsigned)bsize);
            cudaDeviceReset();
            exit(EXIT_FAILURE);
        }

        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

        // register this buffer object with CUDA
        checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_pbo_resource,
pbo_buffer, cudaGraphicsMapFlagsWriteDiscard));

        glGenTextures(1, &texid);
        glBindTexture(GL_TEXTURE_2D, texid);
        glTexImage2D(GL_TEXTURE_2D, 0, ((g_Bpp==1) ? GL_LUMINANCE : GL_BGRA),
                     imWidth, imHeight,  0, GL_LUMINANCE, GL_UNSIGNED_BYTE, NULL);
        glBindTexture(GL_TEXTURE_2D, 0);

        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glPixelStorei(GL_PACK_ALIGNMENT, 1);
    }
}

void loadDefaultImage(char *loc_exec)
{

    printf("Reading image: lena.pgm\n");
    const char *image_filename = "lena.pgm";
    char *image_path = sdkFindFilePath(image_filename, loc_exec);

    if (image_path == NULL)
    {
        printf("Failed to read image file: <%s>\n", image_filename);
```

```
        exit(EXIT_FAILURE);
    }

    initializeData(image_path);
    free(image_path);
}

void initGL(int *argc, char **argv)
{
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowSize(wWidth, wHeight);
    glutCreateWindow("Function Pointers [CUDA Edge Detection]n");

    glewInit();

    if (!glewIsSupported("GL_VERSION_1_5 GL_ARB_vertex_buffer_object
GL_ARB_pixel_buffer_object"))
    {
        fprintf(stderr, "Error: failed to get minimal extensions for demo\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "  OpenGL version 1.5\n");
        fprintf(stderr, "  GL_ARB_vertex_buffer_object\n");
        fprintf(stderr, "  GL_ARB_pixel_buffer_object\n");
        cudaDeviceReset();
        exit(EXIT_WAIVED);
    }
}

bool checkCUDAProfile(int dev, int min_runtime, int min_compute)
{
    int runtimeVersion = 0;

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);

    fprintf(stderr,"\nDevice %d: \"%s\"\n", dev, deviceProp.name);
    cudaRuntimeGetVersion(&runtimeVersion);
    fprintf(stderr,"  CUDA Runtime Version     :\t%d.%d\n", runtimeVersion/1000,
(runtimeVersion%100)/10);
    fprintf(stderr,"  CUDA Compute Capability  :\t%d.%d\n", deviceProp.major,
deviceProp.minor);

    if (runtimeVersion >= min_runtime && ((deviceProp.major<<4) +
deviceProp.minor) >= min_compute)
    {
        return true;
    }
    else
    {
        return false;
    }
}

int findCapableDevice(int argc, char **argv)
{
    int dev;
    int bestDev = -1;
```

```
    int deviceCount = 0;
    cudaError_t error_id = cudaGetDeviceCount(&deviceCount);

    if (error_id != cudaSuccess)
    {
        printf("cudaGetDeviceCount returned %d\n-> %s\n", (int)error_id,
cudaGetErrorString(error_id));
        exit(EXIT_FAILURE);
    }

    if (deviceCount == 0)
    {
        fprintf(stderr,"There is no device supporting CUDA.\n");
    }
    else
    {
        fprintf(stderr,"Found %d CUDA Capable Device(s).\n", deviceCount);
    }

    for (dev = 0; dev < deviceCount; ++dev)
    {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

        if (checkCUDAProfile(dev, MIN_RUNTIME_VERSION, MIN_COMPUTE_VERSION))
        {
            fprintf(stderr,"\nFound CUDA Capable Device %d: \"%s\"\n", dev,
deviceProp.name);

            if (bestDev == -1)
            {
                bestDev = dev;
                fprintf(stderr, "Setting active device to %d\n", bestDev);
            }
        }
    }

    if (bestDev == -1)
    {
        fprintf(stderr, "\nNo configuration with available capabilities was found.
Test has been waived.\n");
        fprintf(stderr, "This SDK sample has minimum requirements:\n");
        fprintf(stderr, "\tCUDA Compute Capability >= %d.%d is required\n",
MIN_COMPUTE_VERSION/16, MIN_COMPUTE_VERSION%16);
        fprintf(stderr, "\tCUDA Runtime Version    >= %d.%d is required\n",
MIN_RUNTIME_VERSION/1000, (MIN_RUNTIME_VERSION%100)/10);
    }

    return bestDev;
}

void checkDeviceMeetComputeSpec(int argc, char **argv)
{
    int device = 0;
    cudaGetDevice(&device);

    if (checkCUDAProfile(device, MIN_RUNTIME_VERSION, MIN_COMPUTE_VERSION))
```

```
    {
        fprintf(stderr,"\nCUDA Capable Device %d, meets minimum required
specs.\n", device);
    }
    else
    {
        fprintf(stderr, "\nNo configuration with minimum compute capabilities
found.  Exiting...\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "\tCUDA Compute Capability >= %d.%d is required\n",
MIN_COMPUTE_VERSION/16, MIN_COMPUTE_VERSION%16);
        fprintf(stderr, "\tCUDA Runtime Version    >= %d.%d is required\n",
MIN_RUNTIME_VERSION/1000, (MIN_RUNTIME_VERSION%100)/10);
        cudaDeviceReset();
        exit(EXIT_WAIVED);
    }
}

void runAutoTest(int argc, char *argv[])
{
    printf("[%s] (automated testing w/ readback)\n", sSDKsample);
    int devID = findCudaDevice(argc, (const char **)argv);

    // Ensure that SM 2.0 or higher device is available before running
    checkDeviceMeetComputeSpec(argc, argv);

    loadDefaultImage(argv[0]);

    Pixel *d_result;
    checkCudaErrors(cudaMalloc((void **)&d_result,
imWidth*imHeight*sizeof(Pixel)));

    char *ref_file = NULL;
    char  dump_file[256];

    int mode = 0;
    mode = getCmdLineArgumentInt(argc, (const char **)argv, "mode");
    getCmdLineArgumentString(argc, (const char **)argv, "file", &ref_file);

    switch (mode)
    {
        case 0:
            g_SobelDisplayMode = SOBELDISPLAY_IMAGE;
            sprintf(dump_file, "lena_orig.pgm");
            break;

        case 1:
            g_SobelDisplayMode = SOBELDISPLAY_SOBELTEX;
            sprintf(dump_file, "lena_tex.pgm");
            break;

        case 2:
            g_SobelDisplayMode = SOBELDISPLAY_SOBELSHARED;
            sprintf(dump_file, "lena_shared.pgm");
            break;

        default:
            printf("Invalid Filter Mode File\n");
```

```
            exit(EXIT_FAILURE);
            break;
    }

    printf("AutoTest: %s <%s>\n", sSDKsample, filterMode[g_SobelDisplayMode]);
    sobelFilter(d_result, imWidth, imHeight, g_SobelDisplayMode, imageScale,
blockOp, pointOp);
    checkCudaErrors(cudaDeviceSynchronize());

    unsigned char *h_result = (unsigned char
*)malloc(imWidth*imHeight*sizeof(Pixel));
    checkCudaErrors(cudaMemcpy(h_result, d_result, imWidth*imHeight*sizeof(Pixel),
cudaMemcpyDeviceToHost));
    sdkSavePGM(dump_file, h_result, imWidth, imHeight);

    if (!sdkComparePGM(dump_file, sdkFindFilePath(ref_file, argv[0]),
MAX_EPSILON_ERROR, 0.15f, false))
    {
        g_TotalErrors++;
    }

    checkCudaErrors(cudaFree(d_result));
    free(h_result);

    if (g_TotalErrors != 0)
    {
        printf("Test failed!\n");
        exit(EXIT_FAILURE);
    }

    printf("Test passed!\n");
    exit(EXIT_SUCCESS);
}


int main(int argc, char **argv)
{
    pArgc = &argc;
    pArgv = argv;

    printf("%s Starting...\n\n", argv[0]);

    if (checkCmdLineFlag(argc, (const char **)argv, "help"))
    {
        printf("\nUsage: FunctionPointers (SobelFilter) <options>\n");
        printf("\t\t-mode=n (0=original, 1=texture, 2=smem + texture)\n");
        printf("\t\t-file=ref_orig.pgm (ref_tex.pgm, ref_shared.pgm)\n\n");

        exit(EXIT_WAIVED);
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "file"))
    {
        g_bQAReadback = true;
        runAutoTest(argc, argv);
    }

    // use command-line specified CUDA device, otherwise use device with highest
```

```
Gflops/s
    if (checkCmdLineFlag(argc, (const char **)argv, "device"))
    {
        printf("   This SDK does not explicitly support -device=n when running
with OpenGL.\n");
        printf("   When specifying -device=n (n=0,1,2,....) the sample must not
use OpenGL.\n");
        printf("   See details below to run without OpenGL:\n\n");
        printf(" > %s -device=n\n\n", argv[0]);
        printf("exiting...\n");
        exit(EXIT_WAIVED);
    }

    if (!g_bQAReadback)
    {
        // First initialize OpenGL context, so we can properly set the GL for
CUDA.
        // This is necessary in order to achieve optimal performance with
OpenGL/CUDA interop.
        initGL(&argc, argv);

        int dev = findCapableDevice(argc, argv);

            checkDeviceMeetComputeSpec(argc, argv);

            if (dev != -1)
        {
            cudaGLSetGLDevice(dev);
        }
        else
        {
            exit(EXIT_WAIVED);
        }

        sdkCreateTimer(&timer);
        sdkResetTimer(&timer);

        glutDisplayFunc(display);
        glutKeyboardFunc(keyboard);
        glutReshapeFunc(reshape);

        loadDefaultImage(argv[0]);

        // If code is not printing the USage, then we execute this path.
        printf("I: display Image (no filtering)\n");
        printf("T: display Sobel Edge Detection (Using Texture)\n");
        printf("S: display Sobel Edge Detection (Using SMEM+Texture)\n");
        printf("Use the '-' and '=' keys to change the brightness.\n");
        printf("b: switch block filter operation (Mean/Sobel)\n");
        printf("p: switch point filter operation (Threshold ON/OFF)\n");
        fflush(stdout);
        atexit(cleanup);
        glutTimerFunc(REFRESH_DELAY, timerEvent,0);
        glutMainLoop();
    }
    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#ifndef __SOBELFILTER_KERNELS_H_
#define __SOBELFILTER_KERNELS_H_

typedef unsigned char Pixel;

// global determines which filter to invoke
enum SobelDisplayMode
{
    SOBELDISPLAY_IMAGE = 0,
    SOBELDISPLAY_SOBELTEX,
    SOBELDISPLAY_SOBELSHARED
};

// Enums to set up the function table
// note: if you change these be sure to recompile those files
// that include this header or ensure the .h is in the
// dependencies for the related object files
enum POINT_ENUM
{
    SOBEL_FILTER=0,
    BOX_FILTER,
    LAST_POINT_FILTER
};

enum BLOCK_ENUM
{
    THRESHOLD_FILTER = 0,
    NULL_FILTER,
    LAST_BLOCK_FILTER
};

extern enum SobelDisplayMode g_SobelDisplayMode;

extern "C" void sobelFilter(Pixel *odata, int iw, int ih, enum SobelDisplayMode
mode, float fScale, int blockOperation, int pointOperation);
extern "C" void setupTexture(int iw, int ih, Pixel *data, int Bpp);
extern "C" void deleteTexture(void);
extern "C" void initFilter(void);
void setupFunctionTables();

#endif
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <helper_cuda.h>

#include "FunctionPointers_kernels.h"

// Texture reference for reading image
texture<unsigned char, 2> tex;
extern __shared__ unsigned char LocalBlock[];
static cudaArray *array = NULL;

#define RADIUS 1

// pixel value used for thresholding function, works well with sample image 'lena'
#define THRESHOLD 150.0f

#ifdef FIXED_BLOCKWIDTH
#define BlockWidth 80
#define SharedPitch 384
#endif

// A function pointer can be declared explicity like this line:
//__device__ unsigned char (*pointFunction)(unsigned char, float ) = NULL;
// or by using typedef's like below:

typedef unsigned char(*blockFunction_t)(
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    float);

typedef unsigned char(*pointFunction_t)(
    unsigned char, float);

__device__ blockFunction_t blockFunction;

__device__ unsigned char
ComputeSobel(unsigned char ul, // upper left
             unsigned char um, // upper middle
             unsigned char ur, // upper right
             unsigned char ml, // middle left
             unsigned char mm, // middle (unused)
             unsigned char mr, // middle right
             unsigned char ll, // lower left
             unsigned char lm, // lower middle
             unsigned char lr, // lower right
```

```
            float fScale)
{
    short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
    short Vert = ul + 2*um + ur - ll - 2*lm - lr;
    short Sum = (short)(fScale*(abs((int)Horz)+abs((int)Vert)));
    return (unsigned char)((Sum < 0) ? 0 : ((Sum > 255) ? 255 : Sum)) ;
}

// define a function pointer and initialize to NULL
__device__ unsigned char(*varFunction)(
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    unsigned char, unsigned char, unsigned char,
    float x
) = NULL;

__device__ unsigned char
ComputeBox(unsigned char ul,   // upper left
           unsigned char um, // upper middle
           unsigned char ur, // upper right
           unsigned char ml, // middle left
           unsigned char mm, // middle...middle
           unsigned char mr, // middle right
           unsigned char ll, // lower left
           unsigned char lm, // lower middle
           unsigned char lr, // lower right
           float fscale
          )
{

    short Sum = (short)(ul+um+ur + ml+mm+mr + ll+lm+lr)/9;
    Sum *= fscale;
    return (unsigned char)((Sum < 0) ? 0 : ((Sum > 255) ? 255 : Sum)) ;
}
__device__ unsigned char
Threshold(unsigned char in, float thresh)
{
    if (in > thresh)
    {
        return 0xFF;
    }
    else
    {
        return 0;
    }
}


// Declare function tables, one for the point function chosen, one for the
// block function chosen.  The number of entries is determined by the
// enum in FunctionPointers_kernels.h
__device__ blockFunction_t blockFunction_table[LAST_BLOCK_FILTER];
__device__ pointFunction_t pointFunction_table[LAST_POINT_FILTER];

// Declare device side function pointers.  We retrieve them later with
// cudaMemcpyFromSymbol to set our function tables above in some
// particular order specified at runtime.
__device__ blockFunction_t pComputeSobel = ComputeSobel;
```

```
__device__ blockFunction_t pComputeBox   = ComputeBox;
__device__ pointFunction_t pComputeThreshold = Threshold;

// Allocate host side tables to mirror the device side, and later, we
// fill these tables with the function pointers.  This lets us send
// the pointers to the kernel on invocation, as a method of choosing
// which function to run.
blockFunction_t h_blockFunction_table[2];
pointFunction_t h_pointFunction_table[2];


// Perform a filter operation on the data, using shared memory
// The actual operation performed is
// determined by the function pointer "blockFunction" and selected
// by the integer argument "blockOperation" and has access
// to an apron around the current pixel being processed.
// Following the block operation, a per-pixel operation,
// pointed to by pPointFunction is performed before the final
// pixel is produced.
__global__ void
SobelShared(uchar4 *pSobelOriginal, unsigned short SobelPitch,
#ifndef FIXED_BLOCKWIDTH
            short BlockWidth, short SharedPitch,
#endif
            short w, short h, float fScale,
            int blockOperation, pointFunction_t pPointFunction
           )
{
    short u = 4*blockIdx.x*BlockWidth;
    short v = blockIdx.y*blockDim.y + threadIdx.y;
    short ib;

    int SharedIdx = threadIdx.y * SharedPitch;

    for (ib = threadIdx.x; ib < BlockWidth+2*RADIUS; ib += blockDim.x)
    {
        LocalBlock[SharedIdx+4*ib+0] = tex2D(tex,
                                             (float)(u+4*ib-RADIUS+0), (float)(v-
RADIUS));
        LocalBlock[SharedIdx+4*ib+1] = tex2D(tex,
                                             (float)(u+4*ib-RADIUS+1), (float)(v-
RADIUS));
        LocalBlock[SharedIdx+4*ib+2] = tex2D(tex,
                                             (float)(u+4*ib-RADIUS+2), (float)(v-
RADIUS));
        LocalBlock[SharedIdx+4*ib+3] = tex2D(tex,
                                             (float)(u+4*ib-RADIUS+3), (float)(v-
RADIUS));
    }

    if (threadIdx.y < RADIUS*2)
    {
        //
        // copy trailing RADIUS*2 rows of pixels into shared
        //
        SharedIdx = (blockDim.y+threadIdx.y) * SharedPitch;

        for (ib = threadIdx.x; ib < BlockWidth+2*RADIUS; ib += blockDim.x)
```

```
        {
            LocalBlock[SharedIdx+4*ib+0] = tex2D(tex,
                                                 (float)(u+4*ib-RADIUS+0), (float)
(v+blockDim.y-RADIUS));
            LocalBlock[SharedIdx+4*ib+1] = tex2D(tex,
                                                 (float)(u+4*ib-RADIUS+1), (float)
(v+blockDim.y-RADIUS));
            LocalBlock[SharedIdx+4*ib+2] = tex2D(tex,
                                                 (float)(u+4*ib-RADIUS+2), (float)
(v+blockDim.y-RADIUS));
            LocalBlock[SharedIdx+4*ib+3] = tex2D(tex,
                                                 (float)(u+4*ib-RADIUS+3), (float)
(v+blockDim.y-RADIUS));
        }
    }

    __syncthreads();

    u >>= 2;    // index as uchar4 from here
    uchar4 *pSobel = (uchar4 *)(((char *) pSobelOriginal)+v*SobelPitch);
    SharedIdx = threadIdx.y * SharedPitch;

    blockFunction = blockFunction_table[blockOperation];

    for (ib = threadIdx.x; ib < BlockWidth; ib += blockDim.x)
    {

        uchar4 out;

        unsigned char pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+0];
        unsigned char pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+1];
        unsigned char pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+2];
        unsigned char pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+0];
        unsigned char pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+1];
        unsigned char pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+2];
        unsigned char pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+0];
        unsigned char pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+1];
        unsigned char pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+2];

        out.x = (*blockFunction)(pix00, pix01, pix02,
                                 pix10, pix11, pix12,
                                 pix20, pix21, pix22, fScale);

        pix00 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+3];
        pix10 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+3];
        pix20 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+3];
        out.y = (*blockFunction)(pix01, pix02, pix00,
                                 pix11, pix12, pix10,
                                 pix21, pix22, pix20, fScale);

        pix01 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+4];
        pix11 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+4];
        pix21 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+4];
        out.z = (*blockFunction)(pix02, pix00, pix01,
                                 pix12, pix10, pix11,
                                 pix22, pix20, pix21, fScale);

        pix02 = LocalBlock[SharedIdx+4*ib+0*SharedPitch+5];
```

```
        pix12 = LocalBlock[SharedIdx+4*ib+1*SharedPitch+5];
        pix22 = LocalBlock[SharedIdx+4*ib+2*SharedPitch+5];
        out.w = (*blockFunction)(pix00, pix01, pix02,
                                 pix10, pix11, pix12,
                                 pix20, pix21, pix22, fScale);

        if (pPointFunction != NULL)
        {
            out.x = (*pPointFunction)(out.x, THRESHOLD);
            out.y = (*pPointFunction)(out.y, THRESHOLD);
            out.z = (*pPointFunction)(out.z, THRESHOLD);
            out.w = (*pPointFunction)(out.w, THRESHOLD);
        }

        if (u+ib < w/4 && v < h)
        {
            pSobel[u+ib] = out;
        }

    }

    __syncthreads();
}

__global__ void
SobelCopyImage(Pixel *pSobelOriginal, unsigned int Pitch,
               int w, int h, float fscale)
{
    unsigned char *pSobel =
        (unsigned char *)(((char *) pSobelOriginal)+blockIdx.x*Pitch);

    for (int i = threadIdx.x; i < w; i += blockDim.x)
    {
        pSobel[i] = min(max((tex2D(tex, (float) i, (float) blockIdx.x) * fscale),
0.f), 255.f);
    }
}

// Perform block and pointer filtering using texture lookups.
// The block and point operations are determined by the
// input argument (see comment above for "SobelShared" function)
__global__ void
SobelTex(Pixel *pSobelOriginal, unsigned int Pitch,
         int w, int h, float fScale, int blockOperation, pointFunction_t
pPointOperation)
{
    unsigned char *pSobel =
        (unsigned char *)(((char *) pSobelOriginal)+blockIdx.x*Pitch);
    unsigned char tmp = 0;

    for (int i = threadIdx.x; i < w; i += blockDim.x)
    {
        unsigned char pix00 = tex2D(tex, (float) i-1, (float) blockIdx.x-1);
        unsigned char pix01 = tex2D(tex, (float) i+0, (float) blockIdx.x-1);
        unsigned char pix02 = tex2D(tex, (float) i+1, (float) blockIdx.x-1);
        unsigned char pix10 = tex2D(tex, (float) i-1, (float) blockIdx.x+0);
        unsigned char pix11 = tex2D(tex, (float) i+0, (float) blockIdx.x+0);
        unsigned char pix12 = tex2D(tex, (float) i+1, (float) blockIdx.x+0);
```

```
        unsigned char pix20 = tex2D(tex, (float) i-1, (float) blockIdx.x+1);
        unsigned char pix21 = tex2D(tex, (float) i+0, (float) blockIdx.x+1);
        unsigned char pix22 = tex2D(tex, (float) i+1, (float) blockIdx.x+1);
        tmp = (*(blockFunction_table[blockOperation]))(pix00, pix01, pix02,
                                                       pix10, pix11, pix12,
                                                       pix20, pix21, pix22,
fScale);

        if (pPointOperation != NULL)
        {
            tmp = (*pPointOperation)(tmp, 150.0);
        }

        pSobel[i] = tmp;
    }
}

extern "C" void setupTexture(int iw, int ih, Pixel *data, int Bpp)
{
    cudaChannelFormatDesc desc;

    if (Bpp == 1)
    {
        desc = cudaCreateChannelDesc<unsigned char>();
    }
    else
    {
        desc = cudaCreateChannelDesc<uchar4>();
    }

    checkCudaErrors(cudaMallocArray(&array, &desc, iw, ih));
    checkCudaErrors(cudaMemcpyToArray(array, 0, 0, data, Bpp*sizeof(Pixel)*iw*ih,
cudaMemcpyHostToDevice));
}

extern "C" void deleteTexture(void)
{
    checkCudaErrors(cudaFreeArray(array));
}

// Copy the pointers from the function tables to the host side
void setupFunctionTables()
{
    // Dynamically assign the function table.
    // Copy the function pointers to their appropriate locations according to the
enum
    checkCudaErrors(cudaMemcpyFromSymbol(&h_blockFunction_table[SOBEL_FILTER],
pComputeSobel, sizeof(blockFunction_t)));
    checkCudaErrors(cudaMemcpyFromSymbol(&h_blockFunction_table[BOX_FILTER],
pComputeBox, sizeof(blockFunction_t)));

    // do the same for the point function, where the 2nd function is NULL ("no-op"
filter, skipped in kernel code)
    checkCudaErrors(cudaMemcpyFromSymbol(&h_pointFunction_table[THRESHOLD_FILTER],
pComputeThreshold, sizeof(pointFunction_t)));
    h_pointFunction_table[NULL_FILTER] = NULL;

    // now copy the function tables back to the device, so if we wish we can use
```

```
an index into the table to choose them
    // We have now set the order in the function table according to our enum.
    checkCudaErrors(cudaMemcpyToSymbol(blockFunction_table, h_blockFunction_table,
sizeof(blockFunction_t)*LAST_BLOCK_FILTER));
    checkCudaErrors(cudaMemcpyToSymbol(pointFunction_table, h_pointFunction_table,
sizeof(pointFunction_t)*LAST_POINT_FILTER));
}


// Wrapper for the __global__ call that sets up the texture and threads
// Below two methods for selecting the image processing function to run are shown.
// BlockOperation is an integer kernel argument used as an index into the
blockFunction_table on the device side
// pPointOp is itself a function pointer passed as a kernel argument, retrieved
from a host side copy of the function table
extern "C" void sobelFilter(Pixel *odata, int iw, int ih, enum SobelDisplayMode
mode, float fScale, int blockOperation, int pointOperation)
{
    checkCudaErrors(cudaBindTextureToArray(tex, array));
    pointFunction_t pPointOp = h_pointFunction_table[pointOperation];

    switch (mode)
    {
        case SOBELDISPLAY_IMAGE:
            SobelCopyImage<<<ih, 384>>>(odata, iw, iw, ih, fScale);
            break;
        case SOBELDISPLAY_SOBELTEX:
            SobelTex<<<ih, 384>>>(odata, iw, iw, ih, fScale, blockOperation,
pPointOp);
            break;
        case SOBELDISPLAY_SOBELSHARED:
            {
                dim3 threads(16,4);
#ifndef FIXED_BLOCKWIDTH
                int BlockWidth = 80; // must be divisible by 16 for coalescing
#endif
                dim3 blocks = dim3(iw/(4*BlockWidth)+(0!=iw%(4*BlockWidth)),
                                   ih/threads.y+(0!=ih%threads.y));
                int SharedPitch = ~0x3f&(4*(BlockWidth+2*RADIUS)+0x3f);
                int sharedMem = SharedPitch*(threads.y+2*RADIUS);

                // for the shared kernel, width must be divisible by 4
                iw &= ~3;

                SobelShared<<<blocks, threads, sharedMem>>>((uchar4 *) odata,
                                                            iw,
#ifndef FIXED_BLOCKWIDTH
                                                            BlockWidth,
SharedPitch,
#endif
                                                            iw, ih, fScale,
blockOperation, pPointOp);
            }
            break;
    }

    checkCudaErrors(cudaUnbindTexture(tex));
}
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# Makefile include to help find GL Libraries
EXEC            ?=
include ./findgllib.mk

# OpenGL specific libraries
ifneq ($(DARWIN),)
  # Mac OSX specific libraries and paths to include
  LIBRARIES += -L/System/Library/Frameworks/OpenGL.framework/Libraries
```

```
  LIBRARIES += -lGL -lGLU ../../common/lib/darwin/libGLEW.a
  ALL_LDFLAGS += -Xlinker -framework -Xlinker GLUT
else
  LIBRARIES += -L../../common/lib/$(OSLOWER)/$(OS_ARCH) $(GLLINK)
  LIBRARIES += -lGL -lGLU -lX11 -lXi -lXmu -lglut -lGLEW
endif

# CUDA code generation flags
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM20) $(GENCODE_SM30)

##############################################################################

# Target rules
all: build

build: FunctionPointers

FunctionPointers_kernels.o: FunctionPointers_kernels.cu FunctionPointers_kernels.h
      $(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

FunctionPointers.o: FunctionPointers.cpp
      $(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

FunctionPointers: FunctionPointers.o FunctionPointers_kernels.o
      $(EXEC) $(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
      $(EXEC) mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))
      $(EXEC) cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
      $(EXEC) ./FunctionPointers

clean:
      $(EXEC) rm -f FunctionPointers FunctionPointers.o FunctionPointers_kernels.o
*.ppm
      $(EXEC) rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/FunctionPointers

clobber: clean
```

A.10. Merge Sort

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#include <helper_cuda.h>
#include <assert.h>
#include "mergeSort_common.h"



inline __device__ void Comparator(
    uint &keyA,
    uint &valA,
    uint &keyB,
    uint &valB,
    uint arrowDir
)
{
    uint t;

    if ((keyA > keyB) == arrowDir)
    {
        t = keyA;
        keyA = keyB;
        keyB = t;
        t = valA;
        valA = valB;
        valB = t;
    }
}

__global__ void bitonicSortSharedKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint arrayLength,
    uint sortDir
)
{
    //Shared memory storage for one or more short vectors
    __shared__ uint s_key[SHARED_SIZE_LIMIT];
    __shared__ uint s_val[SHARED_SIZE_LIMIT];

    //Offset to the beginning of subbatch and load data
    d_SrcKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_SrcVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    s_key[threadIdx.x +                       0] =
```

```
d_SrcKey[                       0];
    s_val[threadIdx.x +                         0] =
d_SrcVal[                       0];
    s_key[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] = d_SrcKey[(SHARED_SIZE_LIMIT /
2)];
    s_val[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] = d_SrcVal[(SHARED_SIZE_LIMIT /
2)];

    for (uint size = 2; size < arrayLength; size <<= 1)
    {
        //Bitonic merge
        uint dir = (threadIdx.x & (size / 2)) != 0;

        for (uint stride = size / 2; stride > 0; stride >>= 1)
        {
            __syncthreads();
            uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
            Comparator(
                s_key[pos +      0], s_val[pos +      0],
                s_key[pos + stride], s_val[pos + stride],
                dir
            );
        }
    }

    //ddd == sortDir for the last bitonic merge step
    {
        for (uint stride = arrayLength / 2; stride > 0; stride >>= 1)
        {
            __syncthreads();
            uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
            Comparator(
                s_key[pos +      0], s_val[pos +      0],
                s_key[pos + stride], s_val[pos + stride],
                sortDir
            );
        }
    }

    __syncthreads();
    d_DstKey[                       0] = s_key[threadIdx.x +
0];
    d_DstVal[                       0] = s_val[threadIdx.x +
0];
    d_DstKey[(SHARED_SIZE_LIMIT / 2)] = s_key[threadIdx.x + (SHARED_SIZE_LIMIT /
2)];
    d_DstVal[(SHARED_SIZE_LIMIT / 2)] = s_val[threadIdx.x + (SHARED_SIZE_LIMIT /
2)];
}

//Helper function (also used by odd-even merge sort)
extern "C" uint factorRadix2(uint *log2L, uint L)
{
    if (!L)
    {
        *log2L = 0;
        return 0;
    }
```

```
    else
    {
        for (*log2L = 0; (L & 1) == 0; L >>= 1, *log2L++);

        return L;
    }
}

extern "C" void bitonicSortShared(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint batchSize,
    uint arrayLength,
    uint sortDir
)
{
    //Nothing to sort
    if (arrayLength < 2)
    {
        return;
    }

    //Only power-of-two array lengths are supported by this implementation
    uint log2L;
    uint factorizationRemainder = factorRadix2(&log2L, arrayLength);
    assert(factorizationRemainder == 1);

    uint  blockCount = batchSize * arrayLength / SHARED_SIZE_LIMIT;
    uint threadCount = SHARED_SIZE_LIMIT / 2;

    assert(arrayLength <= SHARED_SIZE_LIMIT);
    assert((batchSize * arrayLength) % SHARED_SIZE_LIMIT == 0);

    bitonicSortSharedKernel<<<blockCount, threadCount>>>(d_DstKey, d_DstVal,
d_SrcKey, d_SrcVal, arrayLength, sortDir);
    getLastCudaError("bitonicSortSharedKernel<<<>>> failed!\n");
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 3: merge elementary intervals
////////////////////////////////////////////////////////////////////////////////
static inline __host__ __device__ uint iDivUp(uint a, uint b)
{
    return ((a % b) == 0) ? (a / b) : (a / b + 1);
}

static inline __host__ __device__ uint getSampleCount(uint dividend)
{
    return iDivUp(dividend, SAMPLE_STRIDE);
}

template<uint sortDir> static inline __device__ void ComparatorExtended(
    uint &keyA,
    uint &valA,
```

```
    uint &flagA,
    uint &keyB,
    uint &valB,
    uint &flagB,
    uint arrowDir
)
{
    uint t;

    if (
        (!(flagA || flagB) && ((keyA > keyB) == arrowDir)) ||
        ((arrowDir == sortDir) && (flagA == 1)) ||
        ((arrowDir != sortDir) && (flagB == 1))
    )
    {
        t = keyA;
        keyA = keyB;
        keyB = t;
        t = valA;
        valA = valB;
        valB = t;
        t = flagA;
        flagA = flagB;
        flagB = t;
    }
}

template<uint sortDir> __global__ void bitonicMergeElementaryIntervalsKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N
)
{
    __shared__ uint s_key[2 * SAMPLE_STRIDE];
    __shared__ uint s_val[2 * SAMPLE_STRIDE];
    __shared__ uint s_inf[2 * SAMPLE_STRIDE];

    const uint   intervalI = blockIdx.x & ((2 * stride) / SAMPLE_STRIDE - 1);
    const uint segmentBase = (blockIdx.x - intervalI) * SAMPLE_STRIDE;
    d_SrcKey += segmentBase;
    d_SrcVal += segmentBase;
    d_DstKey += segmentBase;
    d_DstVal += segmentBase;

    //Set up threadblock-wide parameters
    __shared__ uint startSrcA, lenSrcA, startSrcB, lenSrcB, startDst;

    if (threadIdx.x == 0)
    {
        uint segmentElementsA = stride;
        uint segmentElementsB = umin(stride, N - segmentBase - stride);
        uint  segmentSamplesA = stride / SAMPLE_STRIDE;
        uint  segmentSamplesB = getSampleCount(segmentElementsB);
```

```
        uint   segmentSamples = segmentSamplesA + segmentSamplesB;

        startSrcA    = d_LimitsA[blockIdx.x];
        startSrcB    = d_LimitsB[blockIdx.x];
        startDst     = startSrcA + startSrcB;

        uint endSrcA = (intervalI + 1 < segmentSamples) ? d_LimitsA[blockIdx.x +
1] : segmentElementsA;
        uint endSrcB = (intervalI + 1 < segmentSamples) ? d_LimitsB[blockIdx.x +
1] : segmentElementsB;
        lenSrcA      = endSrcA - startSrcA;
        lenSrcB      = endSrcB - startSrcB;
    }

    s_inf[threadIdx.x +             0] = 1;
    s_inf[threadIdx.x + SAMPLE_STRIDE] = 1;

    //Load input data
    __syncthreads();

    if (threadIdx.x < lenSrcA)
    {
        s_key[threadIdx.x] = d_SrcKey[0 + startSrcA + threadIdx.x];
        s_val[threadIdx.x] = d_SrcVal[0 + startSrcA + threadIdx.x];
        s_inf[threadIdx.x] = 0;
    }

    //Prepare for bitonic merge by inversing the ordering
    if (threadIdx.x < lenSrcB)
    {
        s_key[2 * SAMPLE_STRIDE - 1 - threadIdx.x] = d_SrcKey[stride + startSrcB +
threadIdx.x];
        s_val[2 * SAMPLE_STRIDE - 1 - threadIdx.x] = d_SrcVal[stride + startSrcB +
threadIdx.x];
        s_inf[2 * SAMPLE_STRIDE - 1 - threadIdx.x] = 0;
    }

    //"Extended" bitonic merge
    for (uint stride = SAMPLE_STRIDE; stride > 0; stride >>= 1)
    {
        __syncthreads();
        uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
        ComparatorExtended<sortDir>(
            s_key[pos +      0], s_val[pos +      0], s_inf[pos +      0],
            s_key[pos + stride], s_val[pos + stride], s_inf[pos + stride],
            sortDir
        );
    }

    //Store sorted data
    __syncthreads();
    d_DstKey += startDst;
    d_DstVal += startDst;

    if (threadIdx.x < lenSrcA)
    {
        d_DstKey[threadIdx.x] = s_key[threadIdx.x];
        d_DstVal[threadIdx.x] = s_val[threadIdx.x];
```

```
    }

    if (threadIdx.x < lenSrcB)
    {
        d_DstKey[lenSrcA + threadIdx.x] = s_key[lenSrcA + threadIdx.x];
        d_DstVal[lenSrcA + threadIdx.x] = s_val[lenSrcA + threadIdx.x];
    }
}

extern "C" void bitonicMergeElementaryIntervals(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);

    uint mergePairs =
        (lastSegmentElements > stride) ?
        getSampleCount(N) :
        (N - lastSegmentElements) / SAMPLE_STRIDE;

    if (sortDir)
    {
        bitonicMergeElementaryIntervalsKernel<1U><<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
            stride,
            N
        );
        getLastCudaError("mergeElementaryIntervalsKernel<1> failed\n");
    }
    else
    {
        bitonicMergeElementaryIntervalsKernel<0U><<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
            stride,
            N
        );
        getLastCudaError("mergeElementaryIntervalsKernel<0> failed\n");
    }
}
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <helper_functions.h>
#include <helper_cuda.h>
#include "mergeSort_common.h"


////////////////////////////////////////////////////////////////////////////////
// Test driver
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    uint *h_SrcKey, *h_SrcVal, *h_DstKey, *h_DstVal;
    uint *d_SrcKey, *d_SrcVal, *d_BufKey, *d_BufVal, *d_DstKey, *d_DstVal;
    StopWatchInterface *hTimer = NULL;

    const uint   N = 4 * 1048576;
    const uint DIR = 1;
    const uint numValues = 65536;

    printf("%s Starting...\n\n", argv[0]);

    int dev = findCudaDevice(argc, (const char **) argv);

    if (dev == -1)
    {
        return EXIT_FAILURE;
    }

    printf("Allocating and initializing host arrays...\n\n");
    sdkCreateTimer(&hTimer);
    h_SrcKey = (uint *)malloc(N * sizeof(uint));
    h_SrcVal = (uint *)malloc(N * sizeof(uint));
    h_DstKey = (uint *)malloc(N * sizeof(uint));
    h_DstVal = (uint *)malloc(N * sizeof(uint));

    srand(2009);

    for (uint i = 0; i < N; i++)
    {
        h_SrcKey[i] = rand() % numValues;
    }
```

```
    fillValues(h_SrcVal, N);

    printf("Allocating and initializing CUDA arrays...\n\n");
    checkCudaErrors(cudaMalloc((void **)&d_DstKey, N * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_DstVal, N * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_BufKey, N * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_BufVal, N * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_SrcKey, N * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_SrcVal, N * sizeof(uint)));
    checkCudaErrors(cudaMemcpy(d_SrcKey, h_SrcKey, N * sizeof(uint),
cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_SrcVal, h_SrcVal, N * sizeof(uint),
cudaMemcpyHostToDevice));

    printf("Initializing GPU merge sort...\n");
    initMergeSort();

    printf("Running GPU merge sort...\n");
    checkCudaErrors(cudaDeviceSynchronize());
    sdkResetTimer(&hTimer);
    sdkStartTimer(&hTimer);
    mergeSort(
        d_DstKey,
        d_DstVal,
        d_BufKey,
        d_BufVal,
        d_SrcKey,
        d_SrcVal,
        N,
        DIR
    );
    checkCudaErrors(cudaDeviceSynchronize());
    sdkStopTimer(&hTimer);
    printf("Time: %f ms\n", sdkGetTimerValue(&hTimer));

    printf("Reading back GPU merge sort results...\n");
    checkCudaErrors(cudaMemcpy(h_DstKey, d_DstKey, N * sizeof(uint),
cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(h_DstVal, d_DstVal, N * sizeof(uint),
cudaMemcpyDeviceToHost));

    printf("Inspecting the results...\n");
    uint keysFlag = validateSortedKeys(
                        h_DstKey,
                        h_SrcKey,
                        1,
                        N,
                        numValues,
                        DIR
                    );

    uint valuesFlag = validateSortedValues(
                          h_DstKey,
                          h_DstVal,
                          h_SrcKey,
                          1,
                          N
```

```
                    );

    printf("Shutting down...\n");
    closeMergeSort();
    sdkDeleteTimer(&hTimer);
    checkCudaErrors(cudaFree(d_SrcVal));
    checkCudaErrors(cudaFree(d_SrcKey));
    checkCudaErrors(cudaFree(d_BufVal));
    checkCudaErrors(cudaFree(d_BufKey));
    checkCudaErrors(cudaFree(d_DstVal));
    checkCudaErrors(cudaFree(d_DstKey));
    free(h_DstVal);
    free(h_DstKey);
    free(h_SrcVal);
    free(h_SrcKey);
    cudaDeviceReset();

    exit((keysFlag && valuesFlag) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(GENCODE_SM30)
```

```
################################################################################

# Target rules
all: build

build: mergeSort

mergeSort.o: mergeSort.cu mergeSort_common.h
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

bitonic.o: bitonic.cu mergeSort_common.h
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

mergeSort_host.o: mergeSort_host.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

mergeSort_validate.o: mergeSort_validate.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

main.o: main.cpp
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

mergeSort: main.o mergeSort_validate.o mergeSort_host.o bitonic.o mergeSort.o
	$(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./mergeSort

clean:
	rm -f mergeSort main.o mergeSort_validate.o mergeSort_host.o bitonic.o
mergeSort.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/mergeSort

clobber: clean
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */




/*
 * Based on "Designing efficient sorting algorithms for manycore GPUs"
 * by Nadathur Satish, Mark Harris, and Michael Garland
 * http://mgarland.org/files/papers/gpusort-ipdps09.pdf
 *
 * Victor Podlozhnyuk 09/24/2009
 */



#include <assert.h>
#include <helper_cuda.h>
#include "mergeSort_common.h"



////////////////////////////////////////////////////////////////////////////////
// Helper functions
////////////////////////////////////////////////////////////////////////////////
static inline __host__ __device__ uint iDivUp(uint a, uint b)
{
    return ((a % b) == 0) ? (a / b) : (a / b + 1);
}

static inline __host__ __device__ uint getSampleCount(uint dividend)
{
    return iDivUp(dividend, SAMPLE_STRIDE);
}

#define W (sizeof(uint) * 8)
static inline __device__ uint nextPowerOfTwo(uint x)
{
    /*
        --x;
        x |= x >> 1;
        x |= x >> 2;
        x |= x >> 4;
        x |= x >> 8;
        x |= x >> 16;
        return ++x;
    */
    return 1U << (W - __clz(x - 1));
}
```

```
template<uint sortDir> static inline __device__ uint binarySearchInclusive(uint
val, uint *data, uint L, uint stride)
{
    if (L == 0)
    {
        return 0;
    }

    uint pos = 0;

    for (; stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] <= val)) || (!sortDir && (data[newPos -
1] >= val)))
        {
            pos = newPos;
        }
    }

    return pos;
}

template<uint sortDir> static inline __device__ uint binarySearchExclusive(uint
val, uint *data, uint L, uint stride)
{
    if (L == 0)
    {
        return 0;
    }

    uint pos = 0;

    for (; stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] < val)) || (!sortDir && (data[newPos -
1] > val)))
        {
            pos = newPos;
        }
    }

    return pos;
}



////////////////////////////////////////////////////////////////////////////////
// Bottom-level merge sort (binary search-based)
////////////////////////////////////////////////////////////////////////////////
template<uint sortDir> __global__ void mergeSortSharedKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
```

```
    uint *d_SrcVal,
    uint arrayLength
)
{
    __shared__ uint s_key[SHARED_SIZE_LIMIT];
    __shared__ uint s_val[SHARED_SIZE_LIMIT];

    d_SrcKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_SrcVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstKey += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    d_DstVal += blockIdx.x * SHARED_SIZE_LIMIT + threadIdx.x;
    s_key[threadIdx.x +                       0] =
d_SrcKey[                       0];
    s_val[threadIdx.x +                       0] =
d_SrcVal[                       0];
    s_key[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] = d_SrcKey[(SHARED_SIZE_LIMIT /
2)];
    s_val[threadIdx.x + (SHARED_SIZE_LIMIT / 2)] = d_SrcVal[(SHARED_SIZE_LIMIT /
2)];

    for (uint stride = 1; stride < arrayLength; stride <<= 1)
    {
        uint     lPos = threadIdx.x & (stride - 1);
        uint *baseKey = s_key + 2 * (threadIdx.x - lPos);
        uint *baseVal = s_val + 2 * (threadIdx.x - lPos);

        __syncthreads();
        uint keyA = baseKey[lPos +      0];
        uint valA = baseVal[lPos +      0];
        uint keyB = baseKey[lPos + stride];
        uint valB = baseVal[lPos + stride];
        uint posA = binarySearchExclusive<sortDir>(keyA, baseKey + stride, stride,
stride) + lPos;
        uint posB = binarySearchInclusive<sortDir>(keyB, baseKey +      0, stride,
stride) + lPos;

        __syncthreads();
        baseKey[posA] = keyA;
        baseVal[posA] = valA;
        baseKey[posB] = keyB;
        baseVal[posB] = valB;
    }

    __syncthreads();
    d_DstKey[                       0] = s_key[threadIdx.x +
0];
    d_DstVal[                       0] = s_val[threadIdx.x +
0];
    d_DstKey[(SHARED_SIZE_LIMIT / 2)] = s_key[threadIdx.x + (SHARED_SIZE_LIMIT /
2)];
    d_DstVal[(SHARED_SIZE_LIMIT / 2)] = s_val[threadIdx.x + (SHARED_SIZE_LIMIT /
2)];
}

static void mergeSortShared(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
```

```
    uint *d_SrcVal,
    uint batchSize,
    uint arrayLength,
    uint sortDir
)
{
    if (arrayLength < 2)
    {
        return;
    }

    assert(SHARED_SIZE_LIMIT % arrayLength == 0);
    assert(((batchSize * arrayLength) % SHARED_SIZE_LIMIT) == 0);
    uint  blockCount = batchSize * arrayLength / SHARED_SIZE_LIMIT;
    uint threadCount = SHARED_SIZE_LIMIT / 2;

    if (sortDir)
    {
        mergeSortSharedKernel<1U><<<blockCount, threadCount>>>(d_DstKey, d_DstVal,
d_SrcKey, d_SrcVal, arrayLength);
        getLastCudaError("mergeSortShared<1><<<>>> failed\n");
    }
    else
    {
        mergeSortSharedKernel<0U><<<blockCount, threadCount>>>(d_DstKey, d_DstVal,
d_SrcKey, d_SrcVal, arrayLength);
        getLastCudaError("mergeSortShared<0><<<>>> failed\n");
    }
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 1: generate sample ranks
////////////////////////////////////////////////////////////////////////////////
template<uint sortDir> __global__ void generateSampleRanksKernel(
    uint *d_RanksA,
    uint *d_RanksB,
    uint *d_SrcKey,
    uint stride,
    uint N,
    uint threadCount
)
{
    uint pos = blockIdx.x * blockDim.x + threadIdx.x;

    if (pos >= threadCount)
    {
        return;
    }

    const uint           i = pos & ((stride / SAMPLE_STRIDE) - 1);
    const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);
    d_SrcKey += segmentBase;
    d_RanksA += segmentBase / SAMPLE_STRIDE;
    d_RanksB += segmentBase / SAMPLE_STRIDE;

    const uint segmentElementsA = stride;
```

```
    const uint segmentElementsB = umin(stride, N - segmentBase - stride);
    const uint  segmentSamplesA = getSampleCount(segmentElementsA);
    const uint  segmentSamplesB = getSampleCount(segmentElementsB);

    if (i < segmentSamplesA)
    {
        d_RanksA[i] = i * SAMPLE_STRIDE;
        d_RanksB[i] = binarySearchExclusive<sortDir>(
                          d_SrcKey[i * SAMPLE_STRIDE], d_SrcKey + stride,
                          segmentElementsB, nextPowerOfTwo(segmentElementsB)
                      );
    }

    if (i < segmentSamplesB)
    {
        d_RanksB[(stride / SAMPLE_STRIDE) + i] = i * SAMPLE_STRIDE;
        d_RanksA[(stride / SAMPLE_STRIDE) + i] = binarySearchInclusive<sortDir>(
                                                     d_SrcKey[stride + i *
SAMPLE_STRIDE], d_SrcKey + 0,
                                                     segmentElementsA,
nextPowerOfTwo(segmentElementsA)
                                                 );
    }
}

static void generateSampleRanks(
    uint *d_RanksA,
    uint *d_RanksB,
    uint *d_SrcKey,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint         threadCount = (lastSegmentElements > stride) ? (N + 2 * stride -
lastSegmentElements) / (2 * SAMPLE_STRIDE) : (N - lastSegmentElements) / (2 *
SAMPLE_STRIDE);

    if (sortDir)
    {
        generateSampleRanksKernel<1U><<<iDivUp(threadCount, 256), 256>>>(d_RanksA,
d_RanksB, d_SrcKey, stride, N, threadCount);
        getLastCudaError("generateSampleRanksKernel<1U><<<>>> failed\n");
    }
    else
    {
        generateSampleRanksKernel<0U><<<iDivUp(threadCount, 256), 256>>>(d_RanksA,
d_RanksB, d_SrcKey, stride, N, threadCount);
        getLastCudaError("generateSampleRanksKernel<0U><<<>>> failed\n");
    }
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 2: generate sample ranks and indices
////////////////////////////////////////////////////////////////////////////////
```

```
__global__ void mergeRanksAndIndicesKernel(
    uint *d_Limits,
    uint *d_Ranks,
    uint stride,
    uint N,
    uint threadCount
)
{
    uint pos = blockIdx.x * blockDim.x + threadIdx.x;

    if (pos >= threadCount)
    {
        return;
    }

    const uint           i = pos & ((stride / SAMPLE_STRIDE) - 1);
    const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);
    d_Ranks  += (pos - i) * 2;
    d_Limits += (pos - i) * 2;

    const uint segmentElementsA = stride;
    const uint segmentElementsB = umin(stride, N - segmentBase - stride);
    const uint  segmentSamplesA = getSampleCount(segmentElementsA);
    const uint  segmentSamplesB = getSampleCount(segmentElementsB);

    if (i < segmentSamplesA)
    {
        uint dstPos = binarySearchExclusive<1U>(d_Ranks[i], d_Ranks +
segmentSamplesA, segmentSamplesB, nextPowerOfTwo(segmentSamplesB)) + i;
        d_Limits[dstPos] = d_Ranks[i];
    }

    if (i < segmentSamplesB)
    {
        uint dstPos = binarySearchInclusive<1U>(d_Ranks[segmentSamplesA + i],
d_Ranks, segmentSamplesA, nextPowerOfTwo(segmentSamplesA)) + i;
        d_Limits[dstPos] = d_Ranks[segmentSamplesA + i];
    }
}

static void mergeRanksAndIndices(
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint *d_RanksA,
    uint *d_RanksB,
    uint stride,
    uint N
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint         threadCount = (lastSegmentElements > stride) ? (N + 2 * stride -
lastSegmentElements) / (2 * SAMPLE_STRIDE) : (N - lastSegmentElements) / (2 *
SAMPLE_STRIDE);

    mergeRanksAndIndicesKernel<<<iDivUp(threadCount, 256), 256>>>(
        d_LimitsA,
        d_RanksA,
        stride,
```

```
        N,
        threadCount
    );
    getLastCudaError("mergeRanksAndIndicesKernel(A)<<<>>> failed\n");

    mergeRanksAndIndicesKernel<<<iDivUp(threadCount, 256), 256>>>(
        d_LimitsB,
        d_RanksB,
        stride,
        N,
        threadCount
    );
    getLastCudaError("mergeRanksAndIndicesKernel(B)<<<>>> failed\n");
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 3: merge elementary intervals
////////////////////////////////////////////////////////////////////////////////
template<uint sortDir> inline __device__ void merge(
    uint *dstKey,
    uint *dstVal,
    uint *srcAKey,
    uint *srcAVal,
    uint *srcBKey,
    uint *srcBVal,
    uint lenA,
    uint nPowTwoLenA,
    uint lenB,
    uint nPowTwoLenB
)
{
    uint keyA, valA, keyB, valB, dstPosA, dstPosB;

    if (threadIdx.x < lenA)
    {
        keyA = srcAKey[threadIdx.x];
        valA = srcAVal[threadIdx.x];
        dstPosA = binarySearchExclusive<sortDir>(keyA, srcBKey, lenB, nPowTwoLenB)
+ threadIdx.x;
    }

    if (threadIdx.x < lenB)
    {
        keyB = srcBKey[threadIdx.x];
        valB = srcBVal[threadIdx.x];
        dstPosB = binarySearchInclusive<sortDir>(keyB, srcAKey, lenA, nPowTwoLenA)
+ threadIdx.x;
    }

    __syncthreads();

    if (threadIdx.x < lenA)
    {
        dstKey[dstPosA] = keyA;
        dstVal[dstPosA] = valA;
    }
```

```
    if (threadIdx.x < lenB)
    {
        dstKey[dstPosB] = keyB;
        dstVal[dstPosB] = valB;
    }
}

template<uint sortDir> __global__ void mergeElementaryIntervalsKernel(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N
)
{
    __shared__ uint s_key[2 * SAMPLE_STRIDE];
    __shared__ uint s_val[2 * SAMPLE_STRIDE];

    const uint   intervalI = blockIdx.x & ((2 * stride) / SAMPLE_STRIDE - 1);
    const uint segmentBase = (blockIdx.x - intervalI) * SAMPLE_STRIDE;
    d_SrcKey += segmentBase;
    d_SrcVal += segmentBase;
    d_DstKey += segmentBase;
    d_DstVal += segmentBase;

    //Set up threadblock-wide parameters
    __shared__ uint startSrcA, startSrcB, lenSrcA, lenSrcB, startDstA, startDstB;

    if (threadIdx.x == 0)
    {
        uint segmentElementsA = stride;
        uint segmentElementsB = umin(stride, N - segmentBase - stride);
        uint  segmentSamplesA = getSampleCount(segmentElementsA);
        uint  segmentSamplesB = getSampleCount(segmentElementsB);
        uint   segmentSamples = segmentSamplesA + segmentSamplesB;

        startSrcA    = d_LimitsA[blockIdx.x];
        startSrcB    = d_LimitsB[blockIdx.x];
        uint endSrcA = (intervalI + 1 < segmentSamples) ? d_LimitsA[blockIdx.x +
1] : segmentElementsA;
        uint endSrcB = (intervalI + 1 < segmentSamples) ? d_LimitsB[blockIdx.x +
1] : segmentElementsB;
        lenSrcA      = endSrcA - startSrcA;
        lenSrcB      = endSrcB - startSrcB;
        startDstA    = startSrcA + startSrcB;
        startDstB    = startDstA + lenSrcA;
    }

    //Load main input data
    __syncthreads();

    if (threadIdx.x < lenSrcA)
    {
        s_key[threadIdx.x +            0] = d_SrcKey[0 + startSrcA +
```

```
threadIdx.x];
        s_val[threadIdx.x +             0] = d_SrcVal[0 + startSrcA +
threadIdx.x];
    }

    if (threadIdx.x < lenSrcB)
    {
        s_key[threadIdx.x + SAMPLE_STRIDE] = d_SrcKey[stride + startSrcB +
threadIdx.x];
        s_val[threadIdx.x + SAMPLE_STRIDE] = d_SrcVal[stride + startSrcB +
threadIdx.x];
    }

    //Merge data in shared memory
    __syncthreads();
    merge<sortDir>(
        s_key,
        s_val,
        s_key + 0,
        s_val + 0,
        s_key + SAMPLE_STRIDE,
        s_val + SAMPLE_STRIDE,
        lenSrcA, SAMPLE_STRIDE,
        lenSrcB, SAMPLE_STRIDE
    );

    //Store merged data
    __syncthreads();

    if (threadIdx.x < lenSrcA)
    {
        d_DstKey[startDstA + threadIdx.x] = s_key[threadIdx.x];
        d_DstVal[startDstA + threadIdx.x] = s_val[threadIdx.x];
    }

    if (threadIdx.x < lenSrcB)
    {
        d_DstKey[startDstB + threadIdx.x] = s_key[lenSrcA + threadIdx.x];
        d_DstVal[startDstB + threadIdx.x] = s_val[lenSrcA + threadIdx.x];
    }
}

static void mergeElementaryIntervals(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint          mergePairs = (lastSegmentElements > stride) ?
getSampleCount(N) : (N - lastSegmentElements) / SAMPLE_STRIDE;
```

```
    if (sortDir)
    {
        mergeElementaryIntervalsKernel<1U><<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
            stride,
            N
        );
        getLastCudaError("mergeElementaryIntervalsKernel<1> failed\n");
    }
    else
    {
        mergeElementaryIntervalsKernel<0U><<<mergePairs, SAMPLE_STRIDE>>>(
            d_DstKey,
            d_DstVal,
            d_SrcKey,
            d_SrcVal,
            d_LimitsA,
            d_LimitsB,
            stride,
            N
        );
        getLastCudaError("mergeElementaryIntervalsKernel<0> failed\n");
    }
}



extern "C" void bitonicSortShared(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint batchSize,
    uint arrayLength,
    uint sortDir
);

extern "C" void bitonicMergeElementaryIntervals(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint *d_LimitsA,
    uint *d_LimitsB,
    uint stride,
    uint N,
    uint sortDir
);



static uint *d_RanksA, *d_RanksB, *d_LimitsA, *d_LimitsB;
static const uint MAX_SAMPLE_COUNT = 32768;
```

```
extern "C" void initMergeSort(void)
{
    checkCudaErrors(cudaMalloc((void **)&d_RanksA,  MAX_SAMPLE_COUNT *
sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_RanksB,  MAX_SAMPLE_COUNT *
sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_LimitsA, MAX_SAMPLE_COUNT *
sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_LimitsB, MAX_SAMPLE_COUNT *
sizeof(uint)));
}

extern "C" void closeMergeSort(void)
{
    checkCudaErrors(cudaFree(d_RanksA));
    checkCudaErrors(cudaFree(d_RanksB));
    checkCudaErrors(cudaFree(d_LimitsB));
    checkCudaErrors(cudaFree(d_LimitsA));
}

extern "C" void mergeSort(
    uint *d_DstKey,
    uint *d_DstVal,
    uint *d_BufKey,
    uint *d_BufVal,
    uint *d_SrcKey,
    uint *d_SrcVal,
    uint N,
    uint sortDir
)
{
    uint stageCount = 0;

    for (uint stride = SHARED_SIZE_LIMIT; stride < N; stride <<= 1, stageCount++);

    uint *ikey, *ival, *okey, *oval;

    if (stageCount & 1)
    {
        ikey = d_BufKey;
        ival = d_BufVal;
        okey = d_DstKey;
        oval = d_DstVal;
    }
    else
    {
        ikey = d_DstKey;
        ival = d_DstVal;
        okey = d_BufKey;
        oval = d_BufVal;
    }

    assert(N <= (SAMPLE_STRIDE * MAX_SAMPLE_COUNT));
    assert(N % SHARED_SIZE_LIMIT == 0);
    mergeSortShared(ikey, ival, d_SrcKey, d_SrcVal, N / SHARED_SIZE_LIMIT,
SHARED_SIZE_LIMIT, sortDir);
```

```
    for (uint stride = SHARED_SIZE_LIMIT; stride < N; stride <<= 1)
    {
        uint lastSegmentElements = N % (2 * stride);

        //Find sample ranks and prepare for limiters merge
        generateSampleRanks(d_RanksA, d_RanksB, ikey, stride, N, sortDir);

        //Merge ranks and indices
        mergeRanksAndIndices(d_LimitsA, d_LimitsB, d_RanksA, d_RanksB, stride, N);

        //Merge elementary intervals
        mergeElementaryIntervals(okey, oval, ikey, ival, d_LimitsA, d_LimitsB,
stride, N, sortDir);

        if (lastSegmentElements <= stride)
        {
            //Last merge segment consists of a single array which just needs to be
passed through
            checkCudaErrors(cudaMemcpy(okey + (N - lastSegmentElements), ikey + (N
- lastSegmentElements), lastSegmentElements * sizeof(uint),
cudaMemcpyDeviceToDevice));
            checkCudaErrors(cudaMemcpy(oval + (N - lastSegmentElements), ival + (N
- lastSegmentElements), lastSegmentElements * sizeof(uint),
cudaMemcpyDeviceToDevice));
        }

        uint *t;
        t = ikey;
        ikey = okey;
        okey = t;
        t = ival;
        ival = oval;
        oval = t;
    }
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */



////////////////////////////////////////////////////////////////////////////////
// Shortcut definitions
////////////////////////////////////////////////////////////////////////////////
typedef unsigned int uint;

#define SHARED_SIZE_LIMIT 1024U
#define     SAMPLE_STRIDE 128



////////////////////////////////////////////////////////////////////////////////
// Extensive sort validation routine
////////////////////////////////////////////////////////////////////////////////
extern "C" uint validateSortedKeys(
    uint *resKey,
    uint *srcKey,
    uint batchSize,
    uint arrayLength,
    uint numValues,
    uint sortDir
);

extern "C" void fillValues(
    uint *val,
    uint N
);

extern "C" int validateSortedValues(
    uint *resKey,
    uint *resVal,
    uint *srcKey,
    uint batchSize,
    uint arrayLength
);



////////////////////////////////////////////////////////////////////////////////
// CUDA merge sort
////////////////////////////////////////////////////////////////////////////////
extern "C" void initMergeSort(void);

extern "C" void closeMergeSort(void);
```

```
extern "C" void mergeSort(
    uint *dstKey,
    uint *dstVal,
    uint *bufKey,
    uint *bufVal,
    uint *srcKey,
    uint *srcVal,
    uint N,
    uint sortDir
);



////////////////////////////////////////////////////////////////////////////////
// CPU "emulation"
////////////////////////////////////////////////////////////////////////////////
extern "C" void mergeSortHost(
    uint *dstKey,
    uint *dstVal,
    uint *bufKey,
    uint *bufVal,
    uint *srcKey,
    uint *srcVal,
    uint N,
    uint sortDir
);
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */



#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mergeSort_common.h"



////////////////////////////////////////////////////////////////////////////////
// Helper functions
////////////////////////////////////////////////////////////////////////////////
static void checkOrder(uint *data, uint N, uint sortDir)
{
    if (N <= 1)
    {
        return;
    }

    for (uint i = 0; i < N - 1; i++)
        if ((sortDir && (data[i] > data[i + 1])) || (!sortDir && (data[i] < data[i
+ 1])))
        {
            fprintf(stderr, "checkOrder() failed!!!\n");
            exit(EXIT_FAILURE);
        }
}

static uint umin(uint a, uint b)
{
    return (a <= b) ? a : b;
}

static uint getSampleCount(uint dividend)
{
    return ((dividend % SAMPLE_STRIDE) != 0) ? (dividend / SAMPLE_STRIDE + 1) :
(dividend / SAMPLE_STRIDE);
}

static uint nextPowerOfTwo(uint x)
{
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
```

```
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

static uint binarySearchInclusive(uint val, uint *data, uint L, uint sortDir)
{
    if (L == 0)
    {
        return 0;
    }

    uint pos = 0;

    for (uint stride = nextPowerOfTwo(L); stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] <= val)) || (!sortDir && (data[newPos -
1] >= val)))
        {
            pos = newPos;
        }
    }

    return pos;
}

static uint binarySearchExclusive(uint val, uint *data, uint L, uint sortDir)
{
    if (L == 0)
    {
        return 0;
    }

    uint pos = 0;

    for (uint stride = nextPowerOfTwo(L); stride > 0; stride >>= 1)
    {
        uint newPos = umin(pos + stride, L);

        if ((sortDir && (data[newPos - 1] < val)) || (!sortDir && (data[newPos -
1] > val)))
        {
            pos = newPos;
        }
    }

    return pos;
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 1: find sample ranks in each segment
////////////////////////////////////////////////////////////////////////////////
static void generateSampleRanks(
    uint *ranksA,
```

```
    uint *ranksB,
    uint *srcKey,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint         sampleCount = (lastSegmentElements > stride) ? (N + 2 * stride -
lastSegmentElements) / (2 * SAMPLE_STRIDE) : (N - lastSegmentElements) / (2 *
SAMPLE_STRIDE);

    for (uint pos = 0; pos < sampleCount; pos++)
    {
        const uint           i = pos & ((stride / SAMPLE_STRIDE) - 1);
        const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);

        const uint lenA = stride;
        const uint lenB = umin(stride, N - segmentBase - stride);
        const uint   nA = stride / SAMPLE_STRIDE;
        const uint   nB = getSampleCount(lenB);

        if (i < nA)
        {
            ranksA[(segmentBase +      0) / SAMPLE_STRIDE + i] = i *
SAMPLE_STRIDE;
            ranksB[(segmentBase +      0) / SAMPLE_STRIDE + i] =
binarySearchExclusive(srcKey[segmentBase + i * SAMPLE_STRIDE], srcKey +
segmentBase + stride, lenB, sortDir);
        }

        if (i < nB)
        {
            ranksB[(segmentBase + stride) / SAMPLE_STRIDE + i] = i *
SAMPLE_STRIDE;
            ranksA[(segmentBase + stride) / SAMPLE_STRIDE + i] =
binarySearchInclusive(srcKey[segmentBase + stride + i * SAMPLE_STRIDE], srcKey +
segmentBase, lenA, sortDir);
        }
    }
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 2: merge ranks and indices to derive elementary intervals
////////////////////////////////////////////////////////////////////////////////
static void mergeRanksAndIndices(
    uint *limits,
    uint *ranks,
    uint stride,
    uint N
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint         sampleCount = (lastSegmentElements > stride) ? (N + 2 * stride -
lastSegmentElements) / (2 * SAMPLE_STRIDE) : (N - lastSegmentElements) / (2 *
SAMPLE_STRIDE);
```

```
    for (uint pos = 0; pos < sampleCount; pos++)
    {
        const uint           i = pos & ((stride / SAMPLE_STRIDE) - 1);
        const uint segmentBase = (pos - i) * (2 * SAMPLE_STRIDE);

        const uint lenA = stride;
        const uint lenB = umin(stride, N - segmentBase - stride);
        const uint   nA = stride / SAMPLE_STRIDE;
        const uint   nB = getSampleCount(lenB);

        if (i < nA)
        {
            uint dstPosA = binarySearchExclusive(ranks[(segmentBase + 0) /
SAMPLE_STRIDE + i], ranks + (segmentBase + stride) / SAMPLE_STRIDE, nB, 1) + i;
            assert(dstPosA < nA + nB);
            limits[(segmentBase / SAMPLE_STRIDE) + dstPosA] = ranks[(segmentBase +
0) / SAMPLE_STRIDE + i];
        }

        if (i < nB)
        {
            uint dstPosA = binarySearchInclusive(ranks[(segmentBase + stride) /
SAMPLE_STRIDE + i], ranks + (segmentBase + 0) / SAMPLE_STRIDE, nA, 1) + i;
            assert(dstPosA < nA + nB);
            limits[(segmentBase / SAMPLE_STRIDE) + dstPosA] = ranks[(segmentBase +
stride) / SAMPLE_STRIDE + i];
        }
    }
}



////////////////////////////////////////////////////////////////////////////////
// Merge step 3: merge elementary intervals (each interval is <= SAMPLE_STRIDE)
////////////////////////////////////////////////////////////////////////////////
static void merge(
    uint *dstKey,
    uint *dstVal,
    uint *srcAKey,
    uint *srcAVal,
    uint *srcBKey,
    uint *srcBVal,
    uint lenA,
    uint lenB,
    uint sortDir
)
{
    checkOrder(srcAKey, lenA, sortDir);
    checkOrder(srcBKey, lenB, sortDir);

    for (uint i = 0; i < lenA; i++)
    {
        uint dstPos = binarySearchExclusive(srcAKey[i], srcBKey, lenB, sortDir) +
i;
        assert(dstPos < lenA + lenB);
        dstKey[dstPos] = srcAKey[i];
        dstVal[dstPos] = srcAVal[i];
```

```
    }

    for (uint i = 0; i < lenB; i++)
    {
        uint dstPos = binarySearchInclusive(srcBKey[i], srcAKey, lenA, sortDir) +
i;
        assert(dstPos < lenA + lenB);
        dstKey[dstPos] = srcBKey[i];
        dstVal[dstPos] = srcBVal[i];
    }
}

static void mergeElementaryIntervals(
    uint *dstKey,
    uint *dstVal,
    uint *srcKey,
    uint *srcVal,
    uint *limitsA,
    uint *limitsB,
    uint stride,
    uint N,
    uint sortDir
)
{
    uint lastSegmentElements = N % (2 * stride);
    uint          mergePairs = (lastSegmentElements > stride) ?
getSampleCount(N) : (N - lastSegmentElements) / SAMPLE_STRIDE;

    for (uint pos = 0; pos < mergePairs; pos++)
    {
        uint           i = pos & ((2 * stride) / SAMPLE_STRIDE - 1);
        uint segmentBase = (pos - i) * SAMPLE_STRIDE;

        const uint lenA = stride;
        const uint lenB = umin(stride, N - segmentBase - stride);
        const uint   nA = stride / SAMPLE_STRIDE;
        const uint   nB = getSampleCount(lenB);
        const uint    n = nA + nB;

        const uint   startPosA = limitsA[pos];
        const uint     endPosA = (i + 1 < n) ? limitsA[pos + 1] : lenA;
        const uint   startPosB = limitsB[pos];
        const uint     endPosB = (i + 1 < n) ? limitsB[pos + 1] : lenB;
        const uint startPosDst = startPosA + startPosB;

        assert(startPosA <= endPosA && endPosA <= lenA);
        assert(startPosB <= endPosB && endPosB <= lenB);
        assert((endPosA - startPosA) <= SAMPLE_STRIDE);
        assert((endPosB - startPosB) <= SAMPLE_STRIDE);

        merge(
            dstKey  + segmentBase + startPosDst,
            dstVal  + segmentBase + startPosDst,
            (srcKey + segmentBase +      0) + startPosA,
            (srcVal + segmentBase +      0) + startPosA,
            (srcKey + segmentBase + stride) + startPosB,
            (srcVal + segmentBase + stride) + startPosB,
            endPosA - startPosA,
```

```
            endPosB - startPosB,
            sortDir
        );
    }
}



////////////////////////////////////////////////////////////////////////////////
// Retarded bubble sort
////////////////////////////////////////////////////////////////////////////////
static void bubbleSort(uint *key, uint *val, uint N, uint sortDir)
{
    if (N <= 1)
    {
        return;
    }

    for (uint bottom = 0; bottom < N - 1; bottom++)
    {
        uint savePos = bottom;
        uint saveKey = key[bottom];

        for (uint i = bottom + 1; i < N; i++)
            if (
                (sortDir && (key[i] < saveKey)) ||
                (!sortDir && (key[i] > saveKey))
            )
            {
                savePos = i;
                saveKey = key[i];
            }

        if (savePos != bottom)
        {
            uint t;
            t = key[savePos];
            key[savePos] = key[bottom];
            key[bottom] = t;
            t = val[savePos];
            val[savePos] = val[bottom];
            val[bottom] = t;
        }
    }
}



////////////////////////////////////////////////////////////////////////////////
// Interface function
////////////////////////////////////////////////////////////////////////////////
extern "C" void mergeSortHost(
    uint *dstKey,
    uint *dstVal,
    uint *bufKey,
    uint *bufVal,
    uint *srcKey,
    uint *srcVal,
```

```
    uint N,
    uint sortDir
)
{
    uint *ikey, *ival, *okey, *oval;
    uint stageCount = 0;

    for (uint stride = SHARED_SIZE_LIMIT; stride < N; stride <<= 1, stageCount++);

    if (stageCount & 1)
    {
        ikey = bufKey;
        ival = bufVal;
        okey = dstKey;
        oval = dstVal;
    }
    else
    {
        ikey = dstKey;
        ival = dstVal;
        okey = bufKey;
        oval = bufVal;
    }

    printf("Bottom-level sort...\n");
    memcpy(ikey, srcKey, N * sizeof(uint));
    memcpy(ival, srcVal, N * sizeof(uint));

    for (uint pos = 0; pos < N; pos += SHARED_SIZE_LIMIT)
    {
        bubbleSort(ikey + pos, ival + pos, umin(SHARED_SIZE_LIMIT, N - pos),
sortDir);
    }

    printf("Merge...\n");
    uint  *ranksA = (uint *)malloc(getSampleCount(N) * sizeof(uint));
    uint  *ranksB = (uint *)malloc(getSampleCount(N) * sizeof(uint));
    uint *limitsA = (uint *)malloc(getSampleCount(N) * sizeof(uint));
    uint *limitsB = (uint *)malloc(getSampleCount(N) * sizeof(uint));
    memset(ranksA,  0xFF, getSampleCount(N) * sizeof(uint));
    memset(ranksB,  0xFF, getSampleCount(N) * sizeof(uint));
    memset(limitsA, 0xFF, getSampleCount(N) * sizeof(uint));
    memset(limitsB, 0xFF, getSampleCount(N) * sizeof(uint));

    for (uint stride = SHARED_SIZE_LIMIT; stride < N; stride <<= 1)
    {
        uint lastSegmentElements = N % (2 * stride);

        //Find sample ranks and prepare for limiters merge
        generateSampleRanks(ranksA, ranksB, ikey, stride, N, sortDir);

        //Merge ranks and indices
        mergeRanksAndIndices(limitsA, ranksA, stride, N);
        mergeRanksAndIndices(limitsB, ranksB, stride, N);

        //Merge elementary intervals
        mergeElementaryIntervals(okey, oval, ikey, ival, limitsA, limitsB, stride,
N, sortDir);
```

```
        if (lastSegmentElements <= stride)
        {
            //Last merge segment consists of a single array which just needs to be
passed through
            memcpy(okey + (N - lastSegmentElements), ikey + (N -
lastSegmentElements), lastSegmentElements * sizeof(uint));
            memcpy(oval + (N - lastSegmentElements), ival + (N -
lastSegmentElements), lastSegmentElements * sizeof(uint));
        }

        uint *t;
        t = ikey;
        ikey = okey;
        okey = t;
        t = ival;
        ival = oval;
        oval = t;
    }

    free(limitsB);
    free(limitsA);
    free(ranksB);
    free(ranksA);
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */


#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mergeSort_common.h"



////////////////////////////////////////////////////////////////////////////////
// Validate sorted keys array (check for integrity and proper order)
////////////////////////////////////////////////////////////////////////////////
extern "C" uint validateSortedKeys(
    uint *resKey,
    uint *srcKey,
    uint batchSize,
    uint arrayLength,
    uint numValues,
    uint sortDir
)
{
    uint *srcHist;
    uint *resHist;

    if (arrayLength < 2)
    {
        printf("validateSortedKeys(): arrays too short, exiting...\n");
        return 1;
    }

    printf("...inspecting keys array: ");
    srcHist = (uint *)malloc(numValues * sizeof(uint));
    resHist = (uint *)malloc(numValues * sizeof(uint));

    int flag = 1;

    for (uint j = 0; j < batchSize; j++, srcKey += arrayLength, resKey +=
arrayLength)
    {
        //Build histograms for keys arrays
        memset(srcHist, 0, numValues * sizeof(uint));
        memset(resHist, 0, numValues * sizeof(uint));

        for (uint i = 0; i < arrayLength; i++)
        {
            if ((srcKey[i] < numValues) && (resKey[i] < numValues))
```

```
            {
                srcHist[srcKey[i]]++;
                resHist[resKey[i]]++;
            }
            else
            {
                fprintf(stderr, "***Set %u source/result key arrays are not
limited properly***\n", j);
                flag = 0;
                goto brk;
            }
        }

        //Compare the histograms
        for (uint i = 0; i < numValues; i++)
            if (srcHist[i] != resHist[i])
            {
                fprintf(stderr, "***Set %u source/result keys histograms do not
match***\n", j);
                flag = 0;
                goto brk;
            }

        //Finally check the ordering
        for (uint i = 0; i < arrayLength - 1; i++)
            if ((sortDir && (resKey[i] > resKey[i + 1])) || (!sortDir &&
(resKey[i] < resKey[i + 1])))
            {
                fprintf(stderr, "***Set %u result key array is not ordered
properly***\n", j);
                flag = 0;
                goto brk;
            }
    }

brk:
    free(resHist);
    free(srcHist);

    if (flag) printf("OK\n");

    return flag;
}



////////////////////////////////////////////////////////////////////////////////
// Value validation / stability check routines
////////////////////////////////////////////////////////////////////////////////
extern "C" void fillValues(
    uint *val,
    uint N
)
{
    for (uint i = 0; i < N; i++)
        val[i] = i;
}
```

```
extern "C" int validateSortedValues(
    uint *resKey,
    uint *resVal,
    uint *srcKey,
    uint batchSize,
    uint arrayLength
)
{
    int correctFlag = 1, stableFlag = 1;

    printf("...inspecting keys and values array: ");

    for (uint i = 0; i < batchSize; i++, resKey += arrayLength, resVal +=
arrayLength)
    {
        for (uint j = 0; j < arrayLength; j++)
        {
            if (resKey[j] != srcKey[resVal[j]])
                correctFlag = 0;

            if ((j < arrayLength - 1) && (resKey[j] == resKey[j + 1]) &&
(resVal[j] > resVal[j + 1]))
                stableFlag = 0;
        }
    }

    printf(correctFlag ? "OK\n" : "***corrupted!!!***\n");
    printf(stableFlag ? "...stability property: stable!\n" : "...stability
property: NOT stable\n");

    return correctFlag;
}
```

Apéndice B

Paralelismo dinámico

B.1. Quad Tree

```
/**
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

#include <thrust/random.h>
#include <thrust/device_vector.h>
#include <helper_cuda.h>

////////////////////////////////////////////////////////////////////////////////
// A structure of 2D points (structure of arrays).
////////////////////////////////////////////////////////////////////////////////
class Points
{
        float *m_x;
        float *m_y;

    public:
        // Constructor.
        __host__ __device__ Points() : m_x(NULL), m_y(NULL) {}

        // Constructor.
        __host__ __device__ Points(float *x, float *y) : m_x(x), m_y(y) {}

        // Get a point.
        __host__ __device__ __forceinline__ float2 get_point(int idx) const
        {
            return make_float2(m_x[idx], m_y[idx]);
        }

        // Set a point.
        __host__ __device__ __forceinline__ void set_point(int idx, const float2
&p)
        {
            m_x[idx] = p.x;
            m_y[idx] = p.y;
        }

        // Set the pointers.
        __host__ __device__ __forceinline__ void set(float *x, float *y)
        {
            m_x = x;
            m_y = y;
        }
};

////////////////////////////////////////////////////////////////////////////////
// A 2D bounding box
////////////////////////////////////////////////////////////////////////////////
class Bounding_box
{
```

```
        // Extreme points of the bounding box.
        float2 m_p_min;
        float2 m_p_max;

    public:
        // Constructor. Create a unit box.
        __host__ __device__ Bounding_box()
        {
            m_p_min = make_float2(0.0f, 0.0f);
            m_p_max = make_float2(1.0f, 1.0f);
        }

        // Compute the center of the bounding-box.
        __host__ __device__ void compute_center(float2 &center) const
        {
            center.x = 0.5f * (m_p_min.x + m_p_max.x);
            center.y = 0.5f * (m_p_min.y + m_p_max.y);
        }

        // The points of the box.
        __host__ __device__ __forceinline__ const float2 &get_max() const
        {
            return m_p_max;
        }

        __host__ __device__ __forceinline__ const float2 &get_min() const
        {
            return m_p_min;
        }

        // Does a box contain a point.
        __host__ __device__ bool contains(const float2 &p) const
        {
            return p.x >= m_p_min.x && p.y < m_p_max.x && p.y >= m_p_min.y && p.y
< m_p_max.y;
        }

        // Define the bounding box.
        __host__ __device__ void set(float min_x, float min_y, float max_x, float
max_y)
        {
            m_p_min.x = min_x;
            m_p_min.y = min_y;
            m_p_max.x = max_x;
            m_p_max.y = max_y;
        }
};

////////////////////////////////////////////////////////////////////////////////
// A node of a quadree.
////////////////////////////////////////////////////////////////////////////////
class Quadtree_node
{
        // The identifier of the node.
        int m_id;
        // The bounding box of the tree.
        Bounding_box m_bounding_box;
        // The range of points.
```

```
        int m_begin, m_end;


    public:
        // Constructor.
        __host__ __device__ Quadtree_node() : m_id(0), m_begin(0), m_end(0)
        {}

        // The ID of a node at its level.
        __host__ __device__ int id() const
        {
            return m_id;
        }

        // The ID of a node at its level.
        __host__ __device__ void set_id(int new_id)
        {
            m_id = new_id;
        }

        // The bounding box.
        __host__ __device__ __forceinline__ const Bounding_box &bounding_box()
const
        {
            return m_bounding_box;
        }

        // Set the bounding box.
        __host__ __device__ __forceinline__ void set_bounding_box(float min_x,
float min_y, float max_x, float max_y)
        {
            m_bounding_box.set(min_x, min_y, max_x, max_y);
        }

        // The number of points in the tree.
        __host__ __device__ __forceinline__ int num_points() const
        {
            return m_end - m_begin;
        }

        // The range of points in the tree.
        __host__ __device__ __forceinline__ int points_begin() const
        {
            return m_begin;
        }

        __host__ __device__ __forceinline__ int points_end() const
        {
            return m_end;
        }

        // Define the range for that node.
        __host__ __device__ __forceinline__ void set_range(int begin, int end)
        {
            m_begin = begin;
            m_end = end;
        }
};
```

```
////////////////////////////////////////////////////////////////////////////////
// Algorithm parameters.
////////////////////////////////////////////////////////////////////////////////
struct Parameters
{
    // Choose the right set of points to use as in/out.
    int point_selector;
    // The number of nodes at a given level (2^k for level k).
    int num_nodes_at_this_level;
    // The recursion depth.
    int depth;
    // The max value for depth.
    const int max_depth;
    // The minimum number of points in a node to stop recursion.
    const int min_points_per_node;

    // Constructor set to default values.
    __host__ __device__ Parameters(int max_depth, int min_points_per_node) :
        point_selector(0),
        num_nodes_at_this_level(1),
        depth(0),
        max_depth(max_depth),
        min_points_per_node(min_points_per_node)
    {}

    // Copy constructor. Changes the values for next iteration.
    __host__ __device__ Parameters(const Parameters &params, bool) :
        point_selector((params.point_selector+1) % 2),
        num_nodes_at_this_level(4*params.num_nodes_at_this_level),
        depth(params.depth+1),
        max_depth(params.max_depth),
        min_points_per_node(params.min_points_per_node)
    {}
};

////////////////////////////////////////////////////////////////////////////////
// Build a quadtree on the GPU. Use CUDA Dynamic Parallelism.
//
// The algorithm works as follows. The host (CPU) launches one block of
// NUM_THREADS_PER_BLOCK threads. That block will do the following steps:
//
// 1- Check the number of points and its depth.
//
// We impose a maximum depth to the tree and a minimum number of points per
// node. If the maximum depth is exceeded or the minimum number of points is
// reached. The threads in the block exit.
//
// Before exiting, they perform a buffer swap if it is needed. Indeed, the
// algorithm uses two buffers to permute the points and make sure they are
// properly distributed in the quadtree. By design we want all points to be
// in the first buffer of points at the end of the algorithm. It is the reason
// why we may have to swap the buffer before leavin (if the points are in the
// 2nd buffer).
//
// 2- Count the number of points in each child.
//
// If the depth is not too high and the number of points is sufficient, the
```

```
// block has to dispatch the points into four geometrical buckets: Its
// children. For that purpose, we compute the center of the bounding box and
// count the number of points in each quadrant.
//
// The set of points is divided into sections. Each section is given to a
// warp of threads (32 threads). Warps use __ballot and __popc intrinsics
// to count the points. See the Programming Guide for more information about
// those functions.
//
// 3- Scan the warps' results to know the "global" numbers.
//
// Warps work independently from each other. At the end, each warp knows the
// number of points in its section. To know the numbers for the block, the
// block has to run a scan/reduce at the block level. It's a traditional
// approach. The implementation in that sample is not as optimized as what
// could be found in fast radix sorts, for example, but it relies on the same
// idea.
//
// 4- Move points.
//
// Now that the block knows how many points go in each of its 4 children, it
// remains to dispatch the points. It is straightforward.
//
// 5- Launch new blocks.
//
// The block launches four new blocks: One per children. Each of the four blocks
// will apply the same algorithm.
////////////////////////////////////////////////////////////////////////////////
template< int NUM_THREADS_PER_BLOCK >
__global__
void build_quadtree_kernel(Quadtree_node *nodes, Points *points, Parameters
params)
{
    // The number of warps in a block.
    const int NUM_WARPS_PER_BLOCK = NUM_THREADS_PER_BLOCK / warpSize;

    // Shared memory to store the number of points.
    extern __shared__ int smem[];

    // s_num_pts[4][NUM_WARPS_PER_BLOCK];
    // Addresses of shared memory.
    volatile int *s_num_pts[4];

    for (int i = 0 ; i < 4 ; ++i)
        s_num_pts[i] = (volatile int *) &smem[i*NUM_WARPS_PER_BLOCK];

    // Compute the coordinates of the threads in the block.
    const int warp_id = threadIdx.x / warpSize;
    const int lane_id = threadIdx.x % warpSize;

    // Mask for compaction.
    int lane_mask_lt = (1 << lane_id) - 1; // Same as: asm( "mov.u32 %0, %
%lanemask_lt;" : "=r"(lane_mask_lt) );

    // The current node.
    Quadtree_node &node = nodes[blockIdx.x];
    node.set_id(node.id() + blockIdx.x);
```

```
    // The number of points in the node.
    int num_points = node.num_points();

    //
    // 1- Check the number of points and its depth.
    //

    // Stop the recursion here. Make sure points[0] contains all the points.
    if (params.depth >= params.max_depth || num_points <=
params.min_points_per_node)
    {
        if (params.point_selector == 1)
        {
            int it = node.points_begin(), end = node.points_end();

            for (it += threadIdx.x ; it < end ; it += NUM_THREADS_PER_BLOCK)
                if (it < end)
                    points[0].set_point(it, points[1].get_point(it));
        }

        return;
    }

    // Compute the center of the bounding box of the points.
    const Bounding_box &bbox = node.bounding_box();
    float2 center;
    bbox.compute_center(center);

    // Find how many points to give to each warp.
    int num_points_per_warp = max(warpSize, (num_points + NUM_WARPS_PER_BLOCK-1) /
NUM_WARPS_PER_BLOCK);

    // Each warp of threads will compute the number of points to move to each
quadrant.
    int range_begin = node.points_begin() + warp_id * num_points_per_warp;
    int range_end   = min(range_begin + num_points_per_warp, node.points_end());

    //
    // 2- Count the number of points in each child.
    //

    // Reset the counts of points per child.
    if (lane_id == 0)
    {
        s_num_pts[0][warp_id] = 0;
        s_num_pts[1][warp_id] = 0;
        s_num_pts[2][warp_id] = 0;
        s_num_pts[3][warp_id] = 0;
    }

    // Input points.
    const Points &in_points = points[params.point_selector];

    // Compute the number of points.
    for (int range_it = range_begin + lane_id ; __any(range_it < range_end) ;
range_it += warpSize)
    {
        // Is it still an active thread?
```

```
        bool is_active = range_it < range_end;

        // Load the coordinates of the point.
        float2 p = is_active ? in_points.get_point(range_it) : make_float2(0.0f,
0.0f);

        // Count top-left points.
        int num_pts = __popc(__ballot(is_active && p.x < center.x && p.y >=
center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[0][warp_id] += num_pts;

        // Count top-right points.
        num_pts = __popc(__ballot(is_active && p.x >= center.x && p.y >=
center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[1][warp_id] += num_pts;

        // Count bottom-left points.
        num_pts = __popc(__ballot(is_active && p.x < center.x && p.y < center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[2][warp_id] += num_pts;

        // Count bottom-right points.
        num_pts = __popc(__ballot(is_active && p.x >= center.x && p.y <
center.y));

        if (num_pts > 0 && lane_id == 0)
            s_num_pts[3][warp_id] += num_pts;
    }

    // Make sure warps have finished counting.
    __syncthreads();

    //
    // 3- Scan the warps' results to know the "global" numbers.
    //

    // First 4 warps scan the numbers of points per child (inclusive scan).
    if (warp_id < 4)
    {
        int num_pts = lane_id < NUM_WARPS_PER_BLOCK ? s_num_pts[warp_id]
[lane_id] : 0;
#pragma unroll

        for (int offset = 1 ; offset < NUM_WARPS_PER_BLOCK ; offset *= 2)
        {
            int n = __shfl_up(num_pts, offset, NUM_WARPS_PER_BLOCK);

            if (lane_id >= offset)
                num_pts += n;
        }

        if (lane_id < NUM_WARPS_PER_BLOCK)
            s_num_pts[warp_id][lane_id] = num_pts;
```

```
    }

    __syncthreads();

    // Compute global offsets.
    if (warp_id == 0)
    {
        int sum = s_num_pts[0][NUM_WARPS_PER_BLOCK-1];

        for (int row = 1 ; row < 4 ; ++row)
        {
            int tmp = s_num_pts[row][NUM_WARPS_PER_BLOCK-1];

            if (lane_id < NUM_WARPS_PER_BLOCK)
                s_num_pts[row][lane_id] += sum;

            sum += tmp;
        }
    }

    __syncthreads();

    // Make the scan exclusive.
    if (threadIdx.x < 4*NUM_WARPS_PER_BLOCK)
    {
        int val = threadIdx.x == 0 ? 0 : smem[threadIdx.x-1];
        val += node.points_begin();
        smem[threadIdx.x] = val;
    }

    __syncthreads();

    //
    // 4- Move points.
    //

    // Output points.
    Points &out_points = points[(params.point_selector+1) % 2];

    // Reorder points.
    for (int range_it = range_begin + lane_id ; __any(range_it < range_end) ;
range_it += warpSize)
    {
        // Is it still an active thread?
        bool is_active = range_it < range_end;

        // Load the coordinates of the point.
        float2 p = is_active ? in_points.get_point(range_it) : make_float2(0.0f,
0.0f);

        // Count top-left points.
        bool pred = is_active && p.x < center.x && p.y >= center.y;
        int vote = __ballot(pred);
        int dest = s_num_pts[0][warp_id] + __popc(vote & lane_mask_lt);

        if (pred)
            out_points.set_point(dest, p);
```

```
    if (lane_id == 0)
        s_num_pts[0][warp_id] += __popc(vote);

    // Count top-right points.
    pred = is_active && p.x >= center.x && p.y >= center.y;
    vote = __ballot(pred);
    dest = s_num_pts[1][warp_id] + __popc(vote & lane_mask_lt);

    if (pred)
        out_points.set_point(dest, p);

    if (lane_id == 0)
        s_num_pts[1][warp_id] += __popc(vote);

    // Count bottom-left points.
    pred = is_active && p.x < center.x && p.y < center.y;
    vote = __ballot(pred);
    dest = s_num_pts[2][warp_id] + __popc(vote & lane_mask_lt);

    if (pred)
        out_points.set_point(dest, p);

    if (lane_id == 0)
        s_num_pts[2][warp_id] += __popc(vote);

    // Count bottom-right points.
    pred = is_active && p.x >= center.x && p.y < center.y;
    vote = __ballot(pred);
    dest = s_num_pts[3][warp_id] + __popc(vote & lane_mask_lt);

    if (pred)
        out_points.set_point(dest, p);

    if (lane_id == 0)
        s_num_pts[3][warp_id] += __popc(vote);
}

__syncthreads();

//
// 5- Launch new blocks.
//

// The last thread launches new blocks.
if (threadIdx.x == NUM_THREADS_PER_BLOCK-1)
{
    // The children.
    Quadtree_node *children = &nodes[params.num_nodes_at_this_level];

    // The offsets of the children at their level.
    int child_offset = 4*node.id();

    // Set IDs.
    children[child_offset+0].set_id(4*node.id()+ 0);
    children[child_offset+1].set_id(4*node.id()+ 4);
    children[child_offset+2].set_id(4*node.id()+ 8);
    children[child_offset+3].set_id(4*node.id()+12);
```

```
        // Points of the bounding-box.
        const float2 &p_min = bbox.get_min();
        const float2 &p_max = bbox.get_max();

        // Set the bounding boxes of the children.
        children[child_offset+0].set_bounding_box(p_min.x , center.y, center.x,
p_max.y);    // Top-left.
        children[child_offset+1].set_bounding_box(center.x, center.y, p_max.x ,
p_max.y);    // Top-right.
        children[child_offset+2].set_bounding_box(p_min.x , p_min.y , center.x,
center.y);   // Bottom-left.
        children[child_offset+3].set_bounding_box(center.x, p_min.y , p_max.x ,
center.y);   // Bottom-right.

        // Set the ranges of the children.
        children[child_offset+0].set_range(node.points_begin(),   s_num_pts[0]
[warp_id]);
        children[child_offset+1].set_range(s_num_pts[0][warp_id], s_num_pts[1]
[warp_id]);
        children[child_offset+2].set_range(s_num_pts[1][warp_id], s_num_pts[2]
[warp_id]);
        children[child_offset+3].set_range(s_num_pts[2][warp_id], s_num_pts[3]
[warp_id]);

        // Launch 4 children.
        build_quadtree_kernel<NUM_THREADS_PER_BLOCK><<<4, NUM_THREADS_PER_BLOCK, 4
*NUM_WARPS_PER_BLOCK *sizeof(int)>>>(children, points, Parameters(params, true));
    }
}

////////////////////////////////////////////////////////////////////////////////
// Make sure a Quadtree is properly defined.
////////////////////////////////////////////////////////////////////////////////
bool check_quadtree(const Quadtree_node *nodes, int idx, int num_pts, Points *pts,
Parameters params)
{
    const Quadtree_node &node = nodes[idx];
    int num_points = node.num_points();

    if (params.depth == params.max_depth || num_points <=
params.min_points_per_node)
    {
        int num_points_in_children = 0;

        num_points_in_children += nodes[params.num_nodes_at_this_level +
4*idx+0].num_points();
        num_points_in_children += nodes[params.num_nodes_at_this_level +
4*idx+1].num_points();
        num_points_in_children += nodes[params.num_nodes_at_this_level +
4*idx+2].num_points();
        num_points_in_children += nodes[params.num_nodes_at_this_level +
4*idx+3].num_points();

        if (num_points_in_children != node.num_points())
            return false;

        return check_quadtree(&nodes[params.num_nodes_at_this_level], 4*idx+0,
num_pts, pts, Parameters(params, true)) &&
```

```
               check_quadtree(&nodes[params.num_nodes_at_this_level], 4*idx+1,
num_pts, pts, Parameters(params, true)) &&
               check_quadtree(&nodes[params.num_nodes_at_this_level], 4*idx+2,
num_pts, pts, Parameters(params, true)) &&
               check_quadtree(&nodes[params.num_nodes_at_this_level], 4*idx+3,
num_pts, pts, Parameters(params, true));
    }

    const Bounding_box &bbox = node.bounding_box();

    for (int it = node.points_begin() ; it < node.points_end() ; ++it)
    {
        if (it >= num_pts)
            return false;

        float2 p = pts->get_point(it);

        if (!bbox.contains(p))
            return false;
    }

    return true;
}

////////////////////////////////////////////////////////////////////////////////
// Parallel random number generator.
////////////////////////////////////////////////////////////////////////////////
struct Random_generator
{
    __host__ __device__ unsigned int hash(unsigned int a)
    {
        a = (a+0x7ed55d16) + (a<<12);
        a = (a^0xc761c23c) ^ (a>>19);
        a = (a+0x165667b1) + (a<<5);
        a = (a+0xd3a2646c) ^ (a<<9);
        a = (a+0xfd7046c5) + (a<<3);
        a = (a^0xb55a4f09) ^ (a>>16);
        return a;
    }

    __host__ __device__ __forceinline__ thrust::tuple<float, float> operator()()
    {
        unsigned seed = hash(blockIdx.x*blockDim.x + threadIdx.x);
        thrust::default_random_engine rng(seed);
        thrust::random::uniform_real_distribution<float> distrib;
        return thrust::make_tuple(distrib(rng), distrib(rng));
    }
};

////////////////////////////////////////////////////////////////////////////////
// Main entry point.
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    // Constants to control the algorithm.
    const int num_points = 1024;
    const int max_depth  = 8;
    const int min_points_per_node = 16;
```

```
    // Find/set the device.
    // The test requires an architecture SM35 or greater (CDP capable).
    int warp_size = 0;
    int cuda_device = findCudaDevice(argc, (const char **)argv);
    cudaDeviceProp deviceProps;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProps, cuda_device));
    int cdpCapable = (deviceProps.major == 3 && deviceProps.minor >= 5) ||
deviceProps.major >=4;

    printf("GPU device %s has compute capabilities (SM %d.%d)\n",
deviceProps.name, deviceProps.major, deviceProps.minor);

    if (!cdpCapable)
    {
        std::cerr << "cdpQuadTree requires SM 3.5 or higher to use CUDA Dynamic
Parallelism.  Exiting...\n" << std::endl;
        exit(EXIT_SUCCESS);
    }

    warp_size = deviceProps.warpSize;

    // Allocate memory for points.
    thrust::device_vector<float> x_d0(num_points);
    thrust::device_vector<float> x_d1(num_points);
    thrust::device_vector<float> y_d0(num_points);
    thrust::device_vector<float> y_d1(num_points);

    // Generate random points.
    Random_generator rnd;
    thrust::generate(
        thrust::make_zip_iterator(thrust::make_tuple(x_d0.begin(), y_d0.begin())),
        thrust::make_zip_iterator(thrust::make_tuple(x_d0.end(), y_d0.end())),
        rnd);

    // Host structures to analyze the device ones.
    Points points_init[2];
    points_init[0].set(thrust::raw_pointer_cast(&x_d0[0]),
thrust::raw_pointer_cast(&y_d0[0]));
    points_init[1].set(thrust::raw_pointer_cast(&x_d1[0]),
thrust::raw_pointer_cast(&y_d1[0]));

    // Allocate memory to store points.
    Points *points;
    checkCudaErrors(cudaMalloc((void **) &points, 2*sizeof(Points)));
    checkCudaErrors(cudaMemcpy(points, points_init, 2*sizeof(Points),
cudaMemcpyHostToDevice));

    // We could use a close form...
    int max_nodes = 0;

    for (int i = 0, num_nodes_at_level = 1 ; i < max_depth ; ++i,
num_nodes_at_level *= 4)
        max_nodes += num_nodes_at_level;

    // Allocate memory to store the tree.
    Quadtree_node root;
    root.set_range(0, num_points);
```

```
    Quadtree_node *nodes;
    checkCudaErrors(cudaMalloc((void **) &nodes,
max_nodes*sizeof(Quadtree_node)));
    checkCudaErrors(cudaMemcpy(nodes, &root, sizeof(Quadtree_node),
cudaMemcpyHostToDevice));

    // We set the recursion limit for CDP to max_depth.
    cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, max_depth);

    // Build the quadtree.
    Parameters params(max_depth, min_points_per_node);
    std::cout << "Launching CDP kernel to build the quadtree" << std::endl;
    const int NUM_THREADS_PER_BLOCK = 128; // Do not use less than 128 threads.
    const int NUM_WARPS_PER_BLOCK = NUM_THREADS_PER_BLOCK / warp_size;
    const size_t smem_size = 4*NUM_WARPS_PER_BLOCK*sizeof(int);
    build_quadtree_kernel<NUM_THREADS_PER_BLOCK><<<1, NUM_THREADS_PER_BLOCK,
smem_size>>>(nodes, points, params);
    checkCudaErrors(cudaGetLastError());

    // Copy points to CPU.
    thrust::host_vector<float> x_h(x_d0);
    thrust::host_vector<float> y_h(y_d0);
    Points host_points;
    host_points.set(thrust::raw_pointer_cast(&x_h[0]),
thrust::raw_pointer_cast(&y_h[0]));

    // Copy nodes to CPU.
    Quadtree_node *host_nodes = new Quadtree_node[max_nodes];
    checkCudaErrors(cudaMemcpy(host_nodes, nodes, max_nodes
*sizeof(Quadtree_node), cudaMemcpyDeviceToHost));

    // Validate the results.
    bool ok = check_quadtree(host_nodes, 0, num_points, &host_points, params);
    std::cout << "Results: " << (ok ? "OK" : "FAILED") << std::endl;

    // Free CPU memory.
    delete[] host_nodes;

    // Free memory.
    checkCudaErrors(cudaFree(nodes));
    checkCudaErrors(cudaFree(points));

    cudaDeviceReset();

    exit(ok ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
GENCODE_SM35    := -gencode arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   :=  $(GENCODE_SM35)
#$(GENCODE_SM30)

ALL_CCFLAGS += -dc

LIBRARIES += -lcudadevrt
```

```
################################################################################

# Target rules
all: build

build: cdpQuadtree

cdpQuadtree.o: cdpQuadtree.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

cdpQuadtree: cdpQuadtree.o
	$(NVCC) $(ALL_LDFLAGS) $(GENCODE_FLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./cdpQuadtree

clean:
	rm -f cdpQuadtree cdpQuadtree.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/cdpQuadtree

clobber: clean
```

B.2. QuickSort avanzado

```
/**
* Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
*
* Please refer to the NVIDIA end user license agreement (EULA) associated
* with this source code for terms and conditions that govern your use of
* this software. Any use, reproduction, disclosure, or distribution of
* this software and related documentation outside the terms of the EULA
* is strictly prohibited.
*
*/

////////////////////////////////////////////////////////////////////////
//
//  QUICKSORT.CU
//
//  Implementation of a parallel quicksort in CUDA. It comes in
//  several parts:
//
//  1. A small-set insertion sort. We do this on any set with <=32 elements
//  2. A partitioning kernel, which - given a pivot - separates an input
//     array into elements <=pivot, and >pivot. Two quicksorts will then
//     be launched to resolve each of these.
//  3. A quicksort co-ordinator, which figures out what kernels to launch
//     and when.
//
////////////////////////////////////////////////////////////////////////
#include <thrust/random.h>
#include <thrust/device_vector.h>
#include <helper_cuda.h>
#include <helper_string.h>
#include "cdpQuicksort.h"

////////////////////////////////////////////////////////////////////////
// Inline PTX call to return index of highest non-zero bit in a word
////////////////////////////////////////////////////////////////////////
static __device__ __forceinline__ unsigned int __qsflo(unsigned int word)
{
    unsigned int ret;
    asm volatile("bfind.u32 %0, %1;" : "=r"(ret) : "r"(word));
    return ret;
}

////////////////////////////////////////////////////////////////////////
//
//  ringbufAlloc
//
//  Allocates from a ringbuffer. Allows for not failing when we run out
//  of stack for tracking the offset counts for each sort subsection.
//
//  We use the atomicMax trick to allow out-of-order retirement. If we
//  hit the size limit on the ringbuffer, then we spin-wait for people
//  to complete.
//
////////////////////////////////////////////////////////////////////////
template< typename T >
static __device__ T *ringbufAlloc(qsortRingbuf *ringbuf)
{
```

```
    // Wait for there to be space in the ring buffer. We'll retry only a fixed
    // number of times and then fail, to avoid an out-of-memory deadlock.
    unsigned int loop = 10000;

    while (((ringbuf->head - ringbuf->tail) >= ringbuf->stacksize) && (loop-- >
0));

    if (loop == 0)
        return NULL;

    // Note that the element includes a little index book-keeping, for freeing
later.
    unsigned int index = atomicAdd((unsigned int *) &ringbuf->head, 1);
    T *ret = (T *)(ringbuf->stackbase) + (index & (ringbuf->stacksize-1));
    ret->index = index;

    return ret;
}

////////////////////////////////////////////////////////////////////////////////
//
//  ringBufFree
//
//  Releases an element from the ring buffer. If every element is released
//  up to and including this one, we can advance the tail to indicate that
//  space is now available.
//
////////////////////////////////////////////////////////////////////////////////
template< typename T >
static __device__ void ringbufFree(qsortRingbuf *ringbuf, T *data)
{
    unsigned int index = data->index;       // Non-wrapped index to free
    unsigned int count = atomicAdd((unsigned int *)&(ringbuf->count), 1) + 1;
    unsigned int max = atomicMax((unsigned int *)&(ringbuf->max), index + 1);

    // Update the tail if need be. Note we update "max" to be the new value in
ringbuf->max
    if (max < (index+1)) max = index+1;

    if (max == count)
        atomicMax((unsigned int *)&(ringbuf->tail), count);
}

////////////////////////////////////////////////////////////////////////////////
//
//  qsort_warp
//
//  Simplest possible implementation, does a per-warp quicksort with no inter-warp
//  communication. This has a high atomic issue rate, but the rest should actually
//  be fairly quick because of low work per thread.
//
//  A warp finds its section of the data, then writes all data <pivot to one
//  buffer and all data >pivot to the other. Atomics are used to get a unique
//  section of the buffer.
//
//  Obvious optimisation: do multiple chunks per warp, to increase in-flight loads
//  and cover the instruction overhead.
//
```

```
////////////////////////////////////////////////////////////////////////////
__global__ void qsort_warp(unsigned *indata,
                           unsigned *outdata,
                           unsigned int offset,
                           unsigned int len,
                           qsortAtomicData *atomicData,
                           qsortRingbuf *atomicDataStack,
                           unsigned int source_is_indata,
                           unsigned int depth)
{
    // Find my data offset, based on warp ID
    unsigned int thread_id = threadIdx.x + (blockIdx.x << QSORT_BLOCKSIZE_SHIFT);
    //unsigned int warp_id = threadIdx.x >> 5;   // Used for debug only
    unsigned int lane_id = threadIdx.x & (warpSize-1);

    // Exit if I'm outside the range of sort to be done
    if (thread_id >= len)
        return;

    //
    // First part of the algorithm. Each warp counts the number of elements that
are
    // greater/less than the pivot.
    //
    // When a warp knows its count, it updates an atomic counter.
    //

    // Read in the data and the pivot. Arbitrary pivot selection for now.
    unsigned pivot = indata[offset + len/2];
    unsigned data  = indata[offset + thread_id];

    // Count how many are <= and how many are > pivot.
    // If all are <= pivot then we adjust the comparison
    // because otherwise the sort will move nothing and
    // we'll iterate forever.
    unsigned int greater = (data > pivot);
    unsigned int gt_mask = __ballot(greater);

    if (gt_mask == 0)
    {
        greater = (data >= pivot);
        gt_mask = __ballot(greater);    // Must re-ballot for adjusted comparator
    }

    unsigned int lt_mask = __ballot(!greater);
    unsigned int gt_count = __popc(gt_mask);
    unsigned int lt_count = __popc(lt_mask);

    // Atomically adjust the lt_ and gt_offsets by this amount. Only one thread
need do this. Share the result using shfl
    unsigned int lt_offset, gt_offset;

    if (lane_id == 0)
    {
        if (lt_count > 0)
            lt_offset = atomicAdd((unsigned int *) &atomicData->lt_offset,
lt_count);
```

```
        if (gt_count > 0)
            gt_offset = len - (atomicAdd((unsigned int *) &atomicData->gt_offset,
gt_count) + gt_count);
    }

    lt_offset = __shfl((int)lt_offset, 0);   // Everyone pulls the offsets from
lane 0
    gt_offset = __shfl((int)gt_offset, 0);

    __syncthreads();

    // Now compute my own personal offset within this. I need to know how many
    // threads with a lane ID less than mine are going to write to the same buffer
    // as me. We can use popc to implement a single-operation warp scan in this
case.
    unsigned lane_mask_lt;
    asm("mov.u32 %0, %%lanemask_lt;" : "=r"(lane_mask_lt));
    unsigned int my_mask = greater ? gt_mask : lt_mask;
    unsigned int my_offset = __popc(my_mask & lane_mask_lt);

    // Move data.
    my_offset += greater ? gt_offset : lt_offset;
    outdata[offset + my_offset] = data;


    // Count up if we're the last warp in. If so, then Kepler will launch the next
    // set of sorts directly from here.
    if (lane_id == 0)
    {
        // Count "elements written". If I wrote the last one, then trigger the
next qsorts
        unsigned int mycount = lt_count + gt_count;

        if (atomicAdd((unsigned int *) &atomicData->sorted_count, mycount) +
mycount == len)
        {
            // We're the last warp to do any sorting. Therefore it's up to us to
launch the next stage.
            unsigned int lt_len = atomicData->lt_offset;
            unsigned int gt_len = atomicData->gt_offset;

            cudaStream_t lstream, rstream;
            cudaStreamCreateWithFlags(&lstream, cudaStreamNonBlocking);
            cudaStreamCreateWithFlags(&rstream, cudaStreamNonBlocking);

            // Begin by freeing our atomicData storage. It's better for the
ringbuffer algorithm
            // if we free when we're done, rather than re-using (makes for less
fragmentation).
            ringbufFree<qsortAtomicData>(atomicDataStack, atomicData);

            // Exceptional case: if "lt_len" is zero, then all values in the batch
            // are equal. We are then done (may need to copy into correct buffer,
though)
            if (lt_len == 0)
            {
                if (source_is_indata)
                    cudaMemcpyAsync(indata+offset, outdata+offset,
```

```
gt_len*sizeof(unsigned), cudaMemcpyDeviceToDevice, lstream);

                return;
            }

            // Start with lower half first
            if (lt_len > BITONICSORT_LEN)
            {
                // If we've exceeded maximum depth, fall through to backup
big_bitonicsort
                if (depth >= QSORT_MAXDEPTH)
                {
                    // The final bitonic stage sorts in-place in "outdata". We
therefore
                    // re-use "indata" as the out-of-range tracking buffer. For
(2^n)+1
                    // elements we need (2^(n+1)) bytes of oor buffer. The backup
qsort
                    // buffer is at least this large when sizeof(QTYPE) >= 2.
                    big_bitonicsort<<< 1, BITONICSORT_LEN, 0, lstream >>>(outdata,
source_is_indata ? indata : outdata, indata, offset, lt_len);
                }
                else
                {
                    // Launch another quicksort. We need to allocate more storage
for the atomic data.
                    if ((atomicData =
ringbufAlloc<qsortAtomicData>(atomicDataStack)) == NULL)
                        printf("Stack-allocation error. Failing left child
launch.\n");
                    else
                    {
                        atomicData->lt_offset = atomicData->gt_offset =
atomicData->sorted_count = 0;
                        unsigned int numblocks = (unsigned int)(lt_len+
(QSORT_BLOCKSIZE-1)) >> QSORT_BLOCKSIZE_SHIFT;
                        qsort_warp<<< numblocks, QSORT_BLOCKSIZE, 0, lstream
>>>(outdata, indata, offset, lt_len, atomicData, atomicDataStack, !
source_is_indata, depth+1);
                    }
                }
            }
            else if (lt_len > 1)
            {
                // Final stage uses a bitonic sort instead. It's important to
                // make sure the final stage ends up in the correct (original)
buffer.
                // We launch the smallest power-of-2 number of threads that we
can.
                unsigned int bitonic_len = 1 << (__qsflo(lt_len-1U)+1);
                bitonicsort<<< 1, bitonic_len, 0, lstream >>>(outdata,
source_is_indata ? indata : outdata, offset, lt_len);
            }
            // Finally, if we sorted just one single element, we must still make
            // sure that it winds up in the correct place.
            else if (source_is_indata && (lt_len == 1))
                indata[offset] = outdata[offset];
```

```
            if (cudaPeekAtLastError() != cudaSuccess)
                printf("Left-side launch fail: %s\n",
cudaGetErrorString(cudaGetLastError()));


            // Now the upper half.
            if (gt_len > BITONICSORT_LEN)
            {
                // If we've exceeded maximum depth, fall through to backup
big_bitonicsort
                if (depth >= QSORT_MAXDEPTH)
                    big_bitonicsort<<< 1, BITONICSORT_LEN, 0, rstream >>>(outdata,
source_is_indata ? indata : outdata, indata, offset+lt_len, gt_len);
                else
                {
                    // Allocate new atomic storage for this launch
                    if ((atomicData =
ringbufAlloc<qsortAtomicData>(atomicDataStack)) == NULL)
                        printf("Stack allocation error! Failing right-side
launch.\n");
                    else
                    {
                        atomicData->lt_offset = atomicData->gt_offset =
atomicData->sorted_count = 0;
                        unsigned int numblocks = (unsigned int)(gt_len+
(QSORT_BLOCKSIZE-1)) >> QSORT_BLOCKSIZE_SHIFT;
                        qsort_warp<<< numblocks, QSORT_BLOCKSIZE, 0, rstream
>>>(outdata, indata, offset+lt_len, gt_len, atomicData, atomicDataStack, !
source_is_indata, depth+1);
                    }
                }
            }
            else if (gt_len > 1)
            {
                unsigned int bitonic_len = 1 << (__qsflo(gt_len-1U)+1);
                bitonicsort<<< 1, bitonic_len, 0, rstream >>>(outdata,
source_is_indata ? indata : outdata, offset+lt_len, gt_len);
            }
            else if (source_is_indata && (gt_len == 1))
                indata[offset+lt_len] = outdata[offset+lt_len];

            if (cudaPeekAtLastError() != cudaSuccess)
                printf("Right-side launch fail: %s\n",
cudaGetErrorString(cudaGetLastError()));
        }
    }
}

////////////////////////////////////////////////////////////////////////////
//
//  run_quicksort
//
//  Host-side code to run the Kepler version of quicksort. It's pretty
//  simple, because all launch control is handled on the device via CDP.
//
//  All parallel quicksorts require an equal-sized scratch buffer. This
//  must be passed in ahead of time.
//
```

```
//  Returns the time elapsed for the sort.
//
////////////////////////////////////////////////////////////////////////////
float run_quicksort_cdp(unsigned *gpudata, unsigned *scratchdata, unsigned int
count, cudaStream_t stream)
{
    unsigned int stacksize = QSORT_STACK_ELEMS;

    // This is the stack, for atomic tracking of each sort's status
    qsortAtomicData *gpustack;
    checkCudaErrors(cudaMalloc((void **)&gpustack, stacksize *
sizeof(qsortAtomicData)));
    checkCudaErrors(cudaMemset(gpustack, 0, sizeof(qsortAtomicData)));     // Only
need set first entry to 0

    // Create the memory ringbuffer used for handling the stack.
    // Initialise everything to where it needs to be.
    qsortRingbuf buf;
    qsortRingbuf *ringbuf;
    checkCudaErrors(cudaMalloc((void **)&ringbuf, sizeof(qsortRingbuf)));
    buf.head = 1;           // We start with one allocation
    buf.tail = 0;
    buf.count = 0;
    buf.max = 0;
    buf.stacksize = stacksize;
    buf.stackbase = gpustack;
    checkCudaErrors(cudaMemcpy(ringbuf, &buf, sizeof(buf),
cudaMemcpyHostToDevice));


    // Timing events...
    cudaEvent_t ev1, ev2;
    checkCudaErrors(cudaEventCreate(&ev1));
    checkCudaErrors(cudaEventCreate(&ev2));
    checkCudaErrors(cudaEventRecord(ev1));

    // Now we trivially launch the qsort kernel
    if (count > BITONICSORT_LEN)
    {
        unsigned int numblocks = (unsigned int)(count+(QSORT_BLOCKSIZE-1)) >>
QSORT_BLOCKSIZE_SHIFT;
        qsort_warp<<< numblocks, QSORT_BLOCKSIZE, 0, stream >>>(gpudata,
scratchdata, 0U, count, gpustack, ringbuf, true, 0);
    }
    else
    {
        bitonicsort<<< 1, BITONICSORT_LEN >>>(gpudata, gpudata, 0, count);
    }

    checkCudaErrors(cudaGetLastError());
    checkCudaErrors(cudaEventRecord(ev2));
    checkCudaErrors(cudaDeviceSynchronize());

    float elapse=0.0f;

    if (cudaPeekAtLastError() != cudaSuccess)
        printf("Launch failure: %s\n", cudaGetErrorString(cudaGetLastError()));
    else
```

```
        checkCudaErrors(cudaEventElapsedTime(&elapse, ev1, ev2));

    // Sanity check that the stack allocator is doing the right thing
    checkCudaErrors(cudaMemcpy(&buf, ringbuf, sizeof(*ringbuf),
cudaMemcpyDeviceToHost));

    if (count > BITONICSORT_LEN && buf.head != buf.tail)
    {
        printf("Stack allocation error!\nRingbuf:\n");
        printf("\t head = %u\n", buf.head);
        printf("\t tail = %u\n", buf.tail);
        printf("\tcount = %u\n", buf.count);
        printf("\t  max = %u\n", buf.max);
    }

    // Release our stack data once we're done
    checkCudaErrors(cudaFree(ringbuf));
    checkCudaErrors(cudaFree(gpustack));

    return elapse;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
int run_qsort(unsigned int size, int seed, int debug, int loop, int verbose)
{
    if (seed > 0)
        srand(seed);

    // Create and set up our test
    unsigned *gpudata, *scratchdata;
    checkCudaErrors(cudaMalloc((void **)&gpudata, size*sizeof(unsigned)));
    checkCudaErrors(cudaMalloc((void **)&scratchdata, size*sizeof(unsigned)));

    // Create CPU data.
    unsigned *data = new unsigned[size];
    unsigned int min = loop ? loop : size;
    unsigned int max = size;
    loop = (loop == 0) ? 1 : loop;

    for (size=min; size<=max; size+=loop)
    {
        if (verbose)
            printf(" Input: ");

        for (unsigned int i=0; i<size; i++)
        {
            // Build data 8 bits at a time
            data[i] = 0;
            char *ptr = (char *)&(data[i]);

            for (unsigned j=0; j<sizeof(unsigned); j++)
            {
                // Easy-to-read data in debug mode
                if (debug)
                {
                    *ptr++ = (char)(rand() % 10);
                    break;
```

```
            }

            *ptr++ = (char)(rand() & 255);
        }

        if (verbose)
        {
            if (i && !(i%32))
                printf("\n        ");

            printf("%u ", data[i]);
        }
    }

    if (verbose)
        printf("\n");

    checkCudaErrors(cudaMemcpy(gpudata, data, size*sizeof(unsigned),
cudaMemcpyHostToDevice));

    // So we're now populated and ready to go! We size our launch as
    // blocks of up to BLOCKSIZE threads, and appropriate grid size.
    // One thread is launched per element.
    float elapse;
    elapse = run_quicksort_cdp(gpudata, scratchdata, size, NULL);

    //run_bitonicsort<SORTTYPE>(gpudata, scratchdata, size, verbose);
    checkCudaErrors(cudaDeviceSynchronize());

    // Copy back the data and verify correct sort
    checkCudaErrors(cudaMemcpy(data, gpudata, size*sizeof(unsigned),
cudaMemcpyDeviceToHost));

    if (verbose)
    {
        printf("Output: ");

        for (unsigned int i=0; i<size; i++)
        {
            if (i && !(i%32)) printf("\n        ");

            printf("%u ", data[i]);
        }

        printf("\n");
    }

    unsigned int check;

    for (check=1; check<size; check++)
    {
        if (data[check] < data[check-1])
        {
            printf("FAILED at element: %d\n", check);
            break;
        }
    }
```

```
        if (check != size)
        {
            printf("    cdp_quicksort FAILED\n");
            exit(EXIT_FAILURE);
        }
        else
            printf("    cdp_quicksort PASSED\n");

        // Display the time between event recordings
        printf("Sorted %u elems in %.3f ms (%.3f Melems/sec)\n", size, elapse,
(float)size/(elapse*1000.0f));
        fflush(stdout);
    }

    // Release everything and we're done
    checkCudaErrors(cudaFree(scratchdata));
    checkCudaErrors(cudaFree(gpudata));
    delete(data);
    return 0;
}

static void usage()
{
    printf("Syntax: qsort [-size=<num>] [-seed=<num>] [-debug] [-loop-step=<num>]
[-verbose]\n");
    printf("If loop_step is non-zero, will run from 1->array_len in steps of
loop_step\n");
}


// Host side entry
int main(int argc, char *argv[])
{
    int size = 5000;     // TODO: make this 1e6
    unsigned int seed = 100;    // TODO: make this 0
    int debug = 0;
    int loop = 0;
    int verbose = 0;

    if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
        checkCmdLineFlag(argc, (const char **)argv, "h"))
    {
        usage();
        printf("&&&& cdp_quicksort WAIVED\n");
        exit(EXIT_WAIVED);
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "size"))
    {
        size = getCmdLineArgumentInt(argc, (const char **)argv, "size");
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "seed"))
    {
        seed = getCmdLineArgumentInt(argc, (const char **)argv, "seed");
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "loop-step"))
```

```
    {
        loop = getCmdLineArgumentInt(argc, (const char **)argv, "loop-step");
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "loop-step"))
    {
        debug = 1;
    }

    if (checkCmdLineFlag(argc, (const char **)argv, "verbose"))
    {
        verbose = 1;
    }

    // Get device properties
    int cuda_device = findCudaDevice(argc, (const char **)argv);
    cudaDeviceProp properties;
    checkCudaErrors(cudaGetDeviceProperties(&properties, cuda_device));
    int cdpCapable = (properties.major == 3 && properties.minor >= 5) ||
properties.major >=4;

    printf("GPU device %s has compute capabilities (SM %d.%d)\n", properties.name,
properties.major, properties.minor);

    if (!cdpCapable)
    {
        printf("cdpLUDecomposition requires SM 3.5 or higher to use CUDA Dynamic
Parallelism.  Exiting...\n");
        exit(EXIT_SUCCESS);
    }

    printf("Running qsort on %d elements with seed %d, on %s\n", size, seed,
properties.name);

    run_qsort(size, seed, debug, loop, verbose);
    checkCudaErrors(cudaDeviceReset());
    exit(EXIT_SUCCESS);
}
```

```
// This is a basic, recursive bitonic sort taken from
// http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm
//
// The parallel code is based on:
// http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm
//
// The multithread code is from me.

#include <stdio.h>
#include "cdpQuicksort.h"

// Inline PTX call to return index of highest non-zero bit in a word
static __device__ __forceinline__ unsigned int __btflo(unsigned int word)
{
    unsigned int ret;
    asm volatile("bfind.u32 %0, %1;" : "=r"(ret) : "r"(word));
    return ret;
}


////////////////////////////////////////////////////////////////////////////
//
//  qcompare
//
//  Comparison function. Note difference from libc standard in
//  that we take by reference, not by pointer. I can't figure
//  out how to get a template-to-pointer specialisation working.
//  Perhaps it requires a class?
//
////////////////////////////////////////////////////////////////////////////
__device__ __forceinline__ int qcompare(unsigned &val1, unsigned &val2)
{
    return (val1 > val2) ? 1 : (val1 == val2) ? 0 : -1;
}

////////////////////////////////////////////////////////////////////////////
//
//  Basic any-N bitonic sort. We sort "len" elements of "indata", starting
//  from the "offset" elements into the input data array. Note that "outdata"
//  can safely be the same as "indata" for an in-place sort (we stage through
//  shared memory).
//
//  We handle non-power-of-2 sizes by padding out to the next largest power of 2.
//  This is the fully-generic version, for sorting arbitrary data which does not
//  have a clear "maximum" value. We track "invalid" entries in a separate array
//  to make sure that they always sorts as "max value" elements. A template
//  parameter "OOR" allows specialisation to optimise away the invalid tracking.
//
//  We can do a more specialised version for int/longlong/flat/double, in which
//  we pad out the array with max-value-of-type elements. That's another function.
//
//  The last step copies from indata -> outdata... the rest is done in-place.
//  We use shared memory as temporary storage, which puts an upper limit on
//  how much data we can sort per block.
//
////////////////////////////////////////////////////////////////////////////
static __device__ __forceinline__ void bitonicsort_kernel(unsigned *indata,
```

```
unsigned *outdata, unsigned int offset, unsigned int len)
{
    __shared__ unsigned sortbuf[1024];     // Max of 1024 elements - TODO: make
this dynamic

    // First copy data into shared memory.
    unsigned int inside = (threadIdx.x < len);
    sortbuf[threadIdx.x] = inside ? indata[threadIdx.x + offset] : 0xffffffffu;
    __syncthreads();

    // Now the sort loops
    // Here, "k" is the sort level (remember bitonic does a multi-level butterfly
style sort)
    // and "j" is the partner element in the butterfly.
    // Two threads each work on one butterfly, because the read/write needs to
happen
    // simultaneously
    for (unsigned int k=2; k<=blockDim.x; k*=2) // Butterfly stride increments in
powers of 2
    {
        for (unsigned int j=k>>1; j>0; j>>=1) // Strides also in powers of to, up
to <k
        {
            unsigned int swap_idx = threadIdx.x ^ j; // Index of element we're
compare-and-swapping with
            unsigned my_elem = sortbuf[threadIdx.x];
            unsigned swap_elem = sortbuf[swap_idx];

            __syncthreads();

            // The k'th bit of my threadid (and hence my sort item ID)
            // determines if we sort ascending or descending.
            // However, since threads are reading from the top AND the bottom of
            // the butterfly, if my ID is > swap_idx, then ascending means
mine<swap.
            // Finally, if either my_elem or swap_elem is out of range, then it
            // ALWAYS acts like it's the largest number.
            // Confusing? It saves us two writes though.
            unsigned int ascend = k * (swap_idx < threadIdx.x);
            unsigned int descend = k * (swap_idx > threadIdx.x);
            bool swap = false;

            if ((threadIdx.x & k) == ascend)
            {
                if (my_elem > swap_elem)
                    swap = true;
            }

            if ((threadIdx.x & k) == descend)
            {
                if (my_elem < swap_elem)
                    swap = true;
            }

            // If we had to swap, then write my data to the other element's
position.
            // Don't forget to track out-of-range status too!
            if (swap)
```

```
            {
                sortbuf[swap_idx] = my_elem;
            }

            __syncthreads();
        }
    }

    // Copy the sorted data from shared memory back to the output buffer
    if (threadIdx.x < len)
        outdata[threadIdx.x + offset] = sortbuf[threadIdx.x];
}

////////////////////////////////////////////////////////////////////////////////
//  This is an emergency-CTA sort, which sorts an arbitrary sized chunk
//  using a single block. Useful for if qsort runs out of nesting depth.
//
//  Note that bitonic sort needs enough storage to pad up to the nearest
//  power of 2. This means that the double-buffer is always large enough
//  (when combined with the main buffer), but we do not get enough space
//  to keep OOR information.
//
//  This in turn means that this sort does not work with a generic data
//  type. It must be a directly-comparable (i.e. with max value) type.
//
////////////////////////////////////////////////////////////////////////////////
static __device__ __forceinline__ void big_bitonicsort_kernel(unsigned *indata,
unsigned *outdata, unsigned *backbuf, unsigned int offset, unsigned int len)
{
    unsigned int len2 = 1 << (__btflo(len-1U)+1);   // Round up len to nearest
power-of-2

    if (threadIdx.x >= len2) return;                // Early out for case where
more threads launched than there is data

    // First, set up our unused values to be the max data type.
    for (unsigned int i=len; i<len2; i+=blockDim.x)
    {
        unsigned int index = i + threadIdx.x;

        if (index < len2)
        {
            // Must split our index between two buffers
            if (index < len)
                indata[index+offset] = 0xffffffffu;
            else
                backbuf[index+offset-len] = 0xffffffffu;
        }
    }

    __syncthreads();

    // Now the sort loops
    // Here, "k" is the sort level (remember bitonic does a multi-level butterfly
style sort)
    // and "j" is the partner element in the butterfly.
    // Two threads each work on one butterfly, because the read/write needs to
happen
```

```
    // simultaneously
    for (unsigned int k=2; k<=len2; k*=2) // Butterfly stride increments in powers
of 2
    {
        for (unsigned int j=k>>1; j>0; j>>=1)   // Strides also in powers of to,
up to <k
        {
            for (unsigned int i=0; i<len2; i+=blockDim.x)
            {
                unsigned int index = threadIdx.x + i;
                unsigned int swap_idx = index ^ j; // Index of element we're
compare-and-swapping with

                // Only do the swap for index<swap_idx (avoids collision between
other threads)
                if (swap_idx > index)
                {
                    unsigned my_elem, swap_elem;

                    if (index < len)
                        my_elem = indata[index+offset];
                    else
                        my_elem = backbuf[index+offset-len];

                    if (swap_idx < len)
                        swap_elem = indata[swap_idx+offset];
                    else
                        swap_elem = backbuf[swap_idx+offset-len];

                    // The k'th bit of my index (and hence my sort item ID)
                    // determines if we sort ascending or descending.
                    // Also, if either my_elem or swap_elem is out of range, then
it
                    // ALWAYS acts like it's the largest number.
                    bool swap = false;

                    if ((index & k) == 0)
                    {
                        if (my_elem > swap_elem)
                            swap = true;
                    }

                    if ((index & k) == k)
                    {
                        if (my_elem < swap_elem)
                            swap = true;
                    }

                    // If we had to swap, then write my data to the other
element's position.
                    if (swap)
                    {
                        if (swap_idx < len)
                            indata[swap_idx+offset] = my_elem;
                        else
                            backbuf[swap_idx+offset-len] = my_elem;

                        if (index < len)
```

```
                            indata[index+offset] = swap_elem;
                        else
                            backbuf[index+offset-len] = swap_elem;
                    }
                }
            }

            __syncthreads();    // Only need to sync for each "j" pass
        }
    }

    // Copy the sorted data from the input to the output buffer, because we sort
in-place
    if (outdata != indata)
    {
        for (unsigned int i=0; i<len; i+=blockDim.x)
        {
            unsigned int index = i + threadIdx.x;

            if (index < len)
                outdata[index+offset] = indata[index+offset];
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
// KERNELS
////////////////////////////////////////////////////////////////////////////////

__global__ void bitonicsort(unsigned *indata, unsigned *outdata, unsigned int
offset, unsigned int len)
{
    bitonicsort_kernel(indata, outdata, offset, len);
}

__global__ void big_bitonicsort(unsigned *indata, unsigned *outdata, unsigned
*backbuf, unsigned int offset, unsigned int len)
{
    big_bitonicsort_kernel(indata, outdata, backbuf, offset, len);
}

////////////////////////////////////////////////////////////////////////////////
```

cdpQuicksort.h

```
// Definitions for GPU quicksort

#ifndef QUICKSORT_H
#define QUICKSORT_H

#define QSORT_BLOCKSIZE_SHIFT   9
#define QSORT_BLOCKSIZE         (1 << QSORT_BLOCKSIZE_SHIFT)
#define BITONICSORT_LEN         1024            // Must be power of 2!
#define QSORT_MAXDEPTH          16              // Will force final bitonic stage
at depth QSORT_MAXDEPTH+1

////////////////////////////////////////////////////////////////////////////
// The algorithm uses several variables updated by using atomic operations.
////////////////////////////////////////////////////////////////////////////
typedef struct __align__(128) qsortAtomicData_t
{
    volatile unsigned int lt_offset;    // Current output offset for <pivot
    volatile unsigned int gt_offset;    // Current output offset for >pivot
    volatile unsigned int sorted_count; // Total count sorted, for deciding when
to launch next wave
    volatile unsigned int index;        // Ringbuf tracking index. Can be ignored
if not using ringbuf.
} qsortAtomicData;

////////////////////////////////////////////////////////////////////////////
// A ring-buffer for rapid stack allocation
////////////////////////////////////////////////////////////////////////////
typedef struct qsortRingbuf_t
{
    volatile unsigned int head;         // Head pointer - we allocate from here
    volatile unsigned int tail;         // Tail pointer - indicates last still-in-
use element
    volatile unsigned int count;        // Total count allocated
    volatile unsigned int max;          // Max index allocated
    unsigned int stacksize;             // Wrap-around size of buffer (must be
power of 2)
    volatile void *stackbase;           // Pointer to the stack we're allocating
from
} qsortRingbuf;

// Stack elem count must be power of 2!
#define QSORT_STACK_ELEMS   1*1024*1024 // One million stack elements is a HUGE
number.

__global__ void qsort_warp(unsigned *indata, unsigned *outdata, unsigned int len,
qsortAtomicData *atomicData, qsortRingbuf *ringbuf, unsigned int source_is_indata,
unsigned int depth);
__global__ void bitonicsort(unsigned *indata, unsigned *outdata, unsigned int
offset, unsigned int len);
__global__ void big_bitonicsort(unsigned *indata, unsigned *outdata, unsigned
*backbuf, unsigned int offset, unsigned int len);

#endif // QUICKSORT_H
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
GENCODE_SM35    := -gencode arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   :=  $(GENCODE_SM35)

ALL_CCFLAGS += -dc

LIBRARIES += -lcudadevrt
```

```
################################################################################

# Target rules
all: build

build: cdpAdvancedQuicksort

cdpAdvancedQuicksort.o: cdpAdvancedQuicksort.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

cdpBitonicSort.o: cdpBitonicSort.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

cdpAdvancedQuicksort: cdpAdvancedQuicksort.o cdpBitonicSort.o
	$(NVCC) $(ALL_LDFLAGS) $(GENCODE_FLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./cdpAdvancedQuicksort

clean:
	rm -f cdpAdvancedQuicksort cdpAdvancedQuicksort.o cdpBitonicSort.o
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/cdpAdvancedQuicksort

clobber: clean
```

Apéndice C

Imágenes

C.1. Convolución

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string.h>
#include <math.h>
//#include <cutil_inline.h>
#include <helper_cuda.h>
//#include <helper_timer.h>
#include <stopwatch.h>
#include <cmath>
#include "convolution_kernel.cu"

using namespace std;

////////////////////////////////////////////////////////////////////////////
// Copy a 3D texture from a host (float*) array to a device cudaArray
// The extent should be specified with all dimensions in units of *elements*
void prepareCudaTexture(float* h_src,
                        cudaArray *d_dst,
                        cudaExtent const texExtent);


// Assume target memory has already been allocated, nPixels is odd
void createGaussian1D(float* targPtr,
                      int    nPixels,
                      float  sigma,
                      float  ctr=0.0f);

// Assume target memory has already been allocate, nPixels is odd
void createGaussian2D(float* targPtr,
                      int    nPixelsCol,
                      int    nPixelsRow,
                      float  sigmaCol,
                      float  sigmaRow,
                      float  ctrCol=0.0f,
                      float  ctrRow=0.0f);

// Assume diameter^2 target memory has already been allocated
void createBinaryCircle(float* targPtr,
                        int&   seNonZero,
                        int    diameter);

// Assume diameter^2 target memory has already been allocated
void createBinaryCircle(int* targPtr,
                        int& seNonZero,
                        int  diameter);

// Assume diameter^2 target memory has already been allocated
// This filter is used for edge detection.  Convolve with the
// kernel created by this function, and then look for the
// zero-crossings
// As always, we expect an odd diameter
// For LoG kernels, we always assume square and symmetric,
// which is why there are no options for different dimensions
void createLaplacianOfGaussianKernel(float* targPtr,
                                     int    diameter);
```

```
////////////////////////////////////////////////////////////////////////////
//
// Program main
//
// TODO:  Remove the CUTIL calls so libcutil is not required to compile/run
//
////////////////////////////////////////////////////////////////////////////
int main( int argc, char** argv)
{
   cout << endl << "Executing GPU-accelerated convolution..." << endl;

   ////////////////////////////////////////////////////////////////////////////
   ////////////////////////////////////////////////////////////////////////////
   // Query the devices on the system and select the fastest
   int deviceCount = 0;
      if (cudaGetDeviceCount(&deviceCount) != cudaSuccess)
   {
         cout << "cudaGetDeviceCount() FAILED." << endl;
      cout << "CUDA Driver and Runtime version may be mismatched.\n";
      return -1;
      }

   // Check to make sure we have at least on CUDA-capable device
   if( deviceCount == 0)
   {
      cout << "No CUDA devices available.  Exiting." << endl;
      return -1;
      }

   // Fastest device automatically selected.  Can override below
   //int fastestDeviceID = cutGetMaxGflopsDeviceId() ;
   int fastestDeviceID = 0;
   cudaSetDevice(fastestDeviceID);

   cudaDeviceProp gpuProp;
   cout << "CUDA-enabled devices on this system:  " << deviceCount <<  endl;
   for(int dev=0; dev<deviceCount; dev++)
   {
      cudaGetDeviceProperties(&gpuProp, dev);
      char* devName = gpuProp.name;
      int mjr = gpuProp.major;
      int mnr = gpuProp.minor;
      if( dev==fastestDeviceID )
         cout << "\t* ";
      else
         cout << "\t  ";

      printf("(%d) %20s \tCUDA Capability %d.%d\n", dev, devName, mjr, mnr);
   }
   // End of CUDA device query & selection
   ////////////////////////////////////////////////////////////////////////////
   ////////////////////////////////////////////////////////////////////////////


   // I've conceded, we better just do everything in COL-MAJOR
```

```
   // Use 17x17 because it's bigger than the block size, so we stress
   // the COPY_LIN_ARRAY_TO_SHMEM macro looping
   unsigned int imgW  = 512;
   unsigned int imgH  = 512;
   unsigned int psfW  = 3;
   unsigned int psfH  = 3;
   unsigned int nPix  = imgH*imgW;
   unsigned int nPsf  = psfW*psfH;
   // I would've expected 32x8 to reduce bank conflicts, but 8x32 is
   // about 30% faster, and both are much faster than 16x16
   unsigned int blockDimX = 8;              // X ~ COL
   unsigned int blockDimY = 32;              // Y ~ ROW
   unsigned int gridDimX = imgW/blockDimX;   // X ~ COL
   unsigned int gridDimY = imgH/blockDimY;   // Y ~ ROW

   unsigned int imgBytes = nPix*FLOAT_SZ;
   unsigned int psfBytes = nPsf*FLOAT_SZ;

   cout << endl;
   printf("Executing convolution on %dx%d image with %dx%d PSF\n",
imgW,imgH,psfW,psfH);

   // Allocate host-side memory
   float* imgIn  = (float*)malloc(imgBytes);
   float* imgOut = (float*)malloc(imgBytes);
   float* imgPsf = (float*)malloc(psfBytes);


   // Read the [very large] image
   // Data is stored in row-major, so reverse the loop to read in correctly
   // col-major, so we reverse the order of loops
   ifstream fpIn( "salt512.txt", ios::in);
   cout << "Reading image from file..." << endl;
   for(int r=0; r<imgH; r++)
      for(int c=0; c<imgH; c++)
          fpIn >> imgIn[c*imgH+r];
   fpIn.close();


   // Read in the PSF from a file
   // Here we use a highly asymmetric PSF so we can verify coordinate systems
   // PSF is read in as row-major data, yet we do all our processing in
   // col-major, so we reverse the order of loops

   //createGaussian2D(imgPsf, psfW, psfH, (float)psfW/5.5, (float)psfH/5.5);
   int seNonZero;
   createBinaryCircle(imgPsf, seNonZero, psfW);
   cout << " SE Non Zero Elements: " << seNonZero << endl;

   // Write out the PSF so can be checked later
   ofstream psfout("psf.txt", ios::out);
   cout << endl << "Point Spread Function:" << endl;
   for(int r=0; r<psfH; r++)
   {
      cout << "\t";
      for(int c=0; c<psfW; c++)
      {
         printf("%0.3f ", imgPsf[c*psfH+r]);
```

```
      psfout << imgPsf[c*psfH+r] << " ";
   }
   cout << endl;
   psfout << endl;
}
cout << endl;

// Allocate device memory and copy data to it
float* devIn;
float* devOut;
float* devPsf;
cudaMalloc((void**)&devIn,  imgBytes);
cudaMalloc((void**)&devOut, imgBytes);
cudaMalloc((void**)&devPsf, psfBytes);

dim3  GRID(  gridDimX,  gridDimY,   1);
dim3  BLOCK( blockDimX, blockDimY,  1);
printf("Grid  Dimensions: (%d, %d)\n", gridDimX,  gridDimY);
printf("Block Dimensions: (%d, %d)\n\n", blockDimX, blockDimY);

// GPU Timer Functions
//unsigned int timer = 0;
//cutilCheckError( cutCreateTimer( &timer));
//cutilCheckError( cutStartTimer( timer));

cudaMemcpy(devIn,  imgIn,   imgBytes, cudaMemcpyHostToDevice);
cudaMemcpy(devPsf, imgPsf,  psfBytes, cudaMemcpyHostToDevice);

// Set up kernel execution geometry
// ************************************************************************
// The data is on the HOST, do a full round-trip calculation w/ mem copies
cout << "Data loaded into HOST mem, executing kernel..." << endl;
kernelDilate<<<GRID, BLOCK>>>(devIn, devOut, devPsf,
                              imgW, imgH, psfW/2, psfH/2, seNonZero);
//cutilCheckMsg("Kernel execution failed");  // Check if kernel exec failed
cudaThreadSynchronize();

// Copy result from device to host
cudaMemcpy(imgOut, devOut, nPix*sizeof(float), cudaMemcpyDeviceToHost);

//cutilCheckError( cutStopTimer( timer));
//float gpuTime_w_copy = cutGetTimerValue(timer);
//cutilCheckError( cutDeleteTimer( timer));
// ************************************************************************


// ************************************************************************
// With data already on the device, time just the computations over 100 runs
int NITER = 20;
printf("Data already on DEVICE, running %d times...\n", NITER);
//cutilCheckError( cutCreateTimer( &timer));
//cutilCheckError( cutStartTimer( timer));
for(int i=0; i<NITER; i++)
{
   kernelDilate<<<GRID, BLOCK>>>(devIn, devOut, devPsf,
                                 imgW, imgH, psfW/2, psfH/2, seNonZero);
   //cutilCheckMsg("Kernel execution failed");  // Check if kernel exec failed
   cudaThreadSynchronize();
```

```
   }
   //cutilCheckError( cutStopTimer( timer));
   //float gpuTime_compute_only = cutGetTimerValue(timer)/(float)NITER;
   //cutilCheckError( cutDeleteTimer( timer));
   // ************************************************************************

   //float memCopyTime = gpuTime_w_copy - gpuTime_compute_only;
   //float cpySpeed = 2.0 * (float)imgBytes / (float)(memCopyTime/1000. * 1024 *
1024);
   cout << "Final Timing Results:" << endl << endl;
   //printf("\tMemory copies (Host-Dev) :  %.3f ms (%0.1f MB/s)\n", memCopyTime,
cpySpeed);
   //printf("\tRaw computation time     :  %.3f ms\n", gpuTime_compute_only);
   printf("\t-------------------------------------------------------\n");
   //printf("\tTotal time w/ mem copies :  %.3f ms\n\n", gpuTime_w_copy);


   // Writing I/O data to file, so I can check it with MATLAB
   // Again, data is stored col-major, but files are r/w in row-major
   // so we reverse the loops
   cout << "Writing before/after image to file..." << endl;
   ofstream fpOrig("origImage.txt", ios::out);
   ofstream fpOut("gpu_solution.txt", ios::out);
   // Write fpOut
   for(int r=0; r<imgH; r++)
   {
      for(int c=0; c<imgW; c++)
      {
         fpOrig << imgIn[c*imgW+r] << " ";
         fpOut << imgOut[c*imgW+r] << " ";
      }
      fpOrig << endl;
      fpOut << endl;
   }
   fpOrig.close();
   fpOut.close();

   // cleanup memory
   free(imgIn);
   free(imgOut);
   free(imgPsf);
   //free(imgPsfIntens);
   cudaFree(devIn);
   cudaFree(devOut);
   cudaFree(devPsf);

   cudaThreadExit();

   //cutilExit(argc, argv);
}

// Assume target memory has already been allocated, nPixels is odd
void createGaussian1D(float* targPtr,
                      int    nPixels,
                      float  sigma,
                      float  ctr)
{
   if(nPixels%2 != 1)
```

```
   {
      cout << "***Warning: createGaussian(...) only defined for odd pixel"  <<
endl;
      cout << "            dimensions.  Undefined behavior for even sizes." <<
endl;
   }

   float pxCtr = (float)(nPixels/2 + ctr);
   float sigmaSq = sigma*sigma;
   float denom = sqrt(2*M_PI*sigmaSq);
   float dist;
   for(int i=0; i<nPixels; i++)
   {
      dist = (float)i - pxCtr;
      targPtr[i] = exp(-0.5 * dist * dist / sigmaSq) / denom;
   }
}

// Assume target memory has already been allocate, nPixels is odd
// Use col-row (D00_UL_ES)
void createGaussian2D(float* targPtr,
                      int    nPixelsCol,
                      int    nPixelsRow,
                      float  sigmaCol,
                      float  sigmaRow,
                      float  ctrCol,
                      float  ctrRow)
{
   if(nPixelsCol%2 != 1 || nPixelsRow != 1)
   {
      cout << "***Warning: createGaussian(...) only defined for odd pixel"  <<
endl;
      cout << "            dimensions.  Undefined behavior for even sizes." <<
endl;
   }

   float pxCtrCol = (float)(nPixelsCol/2 + ctrCol);
   float pxCtrRow = (float)(nPixelsRow/2 + ctrRow);
   float distCol, distRow, distColSqNorm, distRowSqNorm;
   float denom = 2*M_PI*sigmaCol*sigmaRow;
   for(int c=0; c<nPixelsCol; c++)
   {
      distCol = (float)c - pxCtrCol;
      distColSqNorm = distCol*distCol / (sigmaCol*sigmaCol);
      for(int r=0; r<nPixelsRow; r++)
      {
         distRow = (float)r - pxCtrRow;
         distRowSqNorm = distRow*distRow / (sigmaRow*sigmaRow);

         targPtr[c*nPixelsRow+r] = exp(-0.5*(distColSqNorm + distRowSqNorm)) /
denom;
      }
   }
}

// Assume diameter^2 target memory has already been allocated
void createBinaryCircle(float* targPtr,
                        int&   seNonZero,
```

```
                        int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float rad;
   seNonZero = 0;
   for(int c=0; c<diameter; c++)
   {
      for(int r=0; r<diameter; r++)
      {
         rad = sqrt((c-pxCtr)*(c-pxCtr) + (r-pxCtr)*(r-pxCtr));
         if(rad <= pxCtr+0.5)
         {
            targPtr[c*diameter+r] = 1.0f;
            seNonZero++;
         }
         else
         {
            targPtr[c*diameter+r] = 0.0f;
         }
      }
   }
}

// Assume diameter^2 target memory has already been allocated
void createBinaryCircle(int*   targPtr,
                        int&   seNonZero,
                        int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float rad;
   for(int c=0; c<diameter; c++)
   {
      for(int r=0; r<diameter; r++)
      {
         rad = sqrt((c-pxCtr)*(c-pxCtr) + (r-pxCtr)*(r-pxCtr));
         if(rad <= pxCtr+0.5)
            targPtr[c*diameter+r] = 1;
         else
            targPtr[c*diameter+r] = 0;
      }
   }
}

// Assume diameter^2 target memory has already been allocated
// This filter is used for edge detection.  Convolve with the
// kernel created by this function, and then look for the
// zero-crossings
// As always, we expect an odd diameter
// For LoG kernels, we always assume square and symmetric,
// which is why there are no options for different dimensions
void createLaplacianOfGaussianKernel(float* targPtr,
                                     int    diameter)
{
   float pxCtr = (float)(diameter-1) / 2.0f;
   float dc, dr, dcSq, drSq;
   float sigma = diameter/10.0f;
   float sigmaSq = sigma*sigma;
   for(int c=0; c<diameter; c++)
```

```
   {
      dc = (float)c - pxCtr;
      dcSq = dc*dc;
      for(int r=0; r<diameter; r++)
      {
         dr = (float)r - pxCtr;
         drSq = dr*dr;

         float firstTerm  = (dcSq + drSq - 2*sigmaSq) / (sigmaSq * sigmaSq);
         float secondTerm = exp(-0.5 * (dcSq + drSq) / sigmaSq);
         targPtr[c*diameter+r] = firstTerm * secondTerm;
      }
   }
}

void prepareCudaTexture(float* h_src,
                        cudaArray *d_dst,
                        cudaExtent const texExtent)
{
   cudaMemcpy3DParms copyParams = {0};
   cudaPitchedPtr cppImgPsf = make_cudaPitchedPtr( (void*)h_src,
                                                   texExtent.width*FLOAT_SZ,
                                                   texExtent.width,
                                                   texExtent.height);
   copyParams.srcPtr   = cppImgPsf;
   copyParams.dstArray = d_dst;
   copyParams.extent   = texExtent;
   copyParams.kind     = cudaMemcpyHostToDevice;
   checkCudaErrors( cudaMemcpy3D(&copyParams) );
}
```

```
                                convolution_kernel.cu
#ifndef _CONVOLUTION_KERNEL_CU
#define _CONVOLUTION_KERNEL_CU

using namespace std;

#include <stdio.h>

#define IDX_1D(col, row, stride) ((col * stride) + row)
#define COL_2D(index, stride) (index / stride)
#define ROW_2D(index, stride) (index % stride)
#define ROUNDUP32(integer) ( ((integer-1)/32 + 1) * 32 )

#define SHMEM 8192
#define FLOAT_SZ sizeof(float)

texture<float, 3, cudaReadModeElementType> texPsf;
textureReference* texPsfRef;
// Use tex3D(texPsf, x, y, z) to access texture data


////////////////////////////////////////////////////////////////////////////
//
// This macros is defined because EVERY convolution-like function has the same
// variables.  Mainly, the pixel identifiers for this thread based on block
// size, and the size of the padded rectangle that each block will work with
//
////////////////////////////////////////////////////////////////////////////
#define CREATE_CONVOLUTION_VARIABLES(psfColRad, psfRowRad) \
\
   const int cornerCol = blockDim.x*blockIdx.x;   \
   const int cornerRow = blockDim.y*blockIdx.y;   \
   const int globalCol = cornerCol + threadIdx.x;   \
   const int globalRow = cornerRow + threadIdx.y;   \
   const int globalIdx = IDX_1D(globalCol, globalRow, imgRows);   \
\
   const int localCol    = threadIdx.x;   \
   const int localRow    = threadIdx.y;   \
   const int localIdx    = IDX_1D(localCol, localRow, blockDim.y);   \
   const int localPixels = blockDim.x*blockDim.y;   \
\
   const int padRectStride = blockDim.y + 2*psfRowRad;   \
   const int padRectCol    = localCol + psfColRad;   \
   const int padRectRow    = localRow + psfRowRad;   \
   /*const int padRectIdx    = IDX_1D(padRectCol, padRectRow, padRectStride); */ \
   const int padRectPixels = padRectStride * (blockDim.x + 2*psfColRad);   \
\
   __shared__ char sharedMem[SHMEM];   \
   float* shmPadRect  = (float*)sharedMem;   \
   float* shmOutput   = (float*)&shmPadRect[ROUNDUP32(padRectPixels)];   \
   int nLoop;



////////////////////////////////////////////////////////////////////////////
//
// Every block will need a buffered copy of the input data in its shared memory,
// so it doesn't do multiple global memory reads to find the energy contributing
```

```
// to it's own value.
//
// This copy is very much like COPY_LIN_ARRAY_TO_SHMEM except that this isn't
// a linear array, and this needs to accommodate pixels that fall out of bounds
// from the image.  Threads are temporarily reassigned to execute this copy in
// parallel.
//
////////////////////////////////////////////////////////////////////////////
#define PREPARE_PADDED_RECTANGLE(psfColRad, psfRowRad) \
\
   nLoop = (padRectPixels/localPixels)+1;   \
   for(int loopIdx=0; loopIdx<nLoop; loopIdx++)   \
   {   \
      int prIndex = loopIdx*localPixels + localIdx;   \
      if(prIndex < padRectPixels)   \
      {   \
         int prCol = COL_2D(prIndex, padRectStride);   \
         int prRow = ROW_2D(prIndex, padRectStride);   \
         int glCol = cornerCol + prCol - psfColRad;   \
         int glRow = cornerRow + prRow - psfRowRad;   \
         int glIdx = IDX_1D(glCol, glRow, imgRows);   \
         if(glRow >= 0        &&  \
            glRow <  imgRows  &&  \
            glCol >= 0        &&  \
            glCol <  imgCols)   \
            shmPadRect[prIndex] = imgInPtr[glIdx];   \
         else   \
            shmPadRect[prIndex] = 0.0f;   \
      }   \
   }   \

////////////////////////////////////////////////////////////////////////////
//
// Same as above, except for binary images, using -1 as "OFF" and +1 as "ON"
// The user is not expected to do this him/herself, and it's easy enough to
// manipulate the data on the way in and out (just don't forget to convert back
// before copying out the result
//
////////////////////////////////////////////////////////////////////////////
#define PREPARE_PADDED_RECTANGLE_BINARY(psfColRad, psfRowRad) \
\
   nLoop = (padRectPixels/localPixels)+1;   \
   for(int loopIdx=0; loopIdx<nLoop; loopIdx++)   \
   {   \
      int prIndex = loopIdx*localPixels + localIdx;   \
      if(prIndex < padRectPixels)   \
      {   \
         int prCol = COL_2D(prIndex, padRectStride);   \
         int prRow = ROW_2D(prIndex, padRectStride);   \
         int glCol = cornerCol + prCol - psfColRad;   \
         int glRow = cornerRow + prRow - psfRowRad;   \
         int glIdx = IDX_1D(glCol, glRow, imgRows);   \
         if(glRow >= 0        &&  \
            glRow <  imgRows  &&  \
            glCol >= 0        &&  \
            glCol <  imgCols)   \
            shmPadRect[prIndex] = imgInPtr[glIdx]*2 - 1;   \
         else   \
```

```
            shmPadRect[prIndex] = -1; \
      }   \
   }   \

////////////////////////////////////////////////////////////////////////////
//
// Frequently, we want to pull some linear arrays into shared memory (usually
// PSFs) which will be queried often, and we want them close to the threads.
//
// This macro temporarily reassigns all the threads to do the memory copy from
// global memory to shared memory in parallel.  Since the array may be bigger
// than the blocksize, some threads may be doing multiple mem copies
//
////////////////////////////////////////////////////////////////////////////
#define COPY_LIN_ARRAY_TO_SHMEM(srcPtr, dstPtr, nValues) \
   nLoop = (nValues/localPixels)+1;   \
   for(int loopIdx=0; loopIdx<nLoop; loopIdx++)   \
   {   \
      int prIndex = loopIdx*localPixels + localIdx;   \
      if(prIndex < nValues)   \
      {   \
         dstPtr[prIndex] = srcPtr[prIndex]; \
      } \
   }




__global__ void   convolveBasic(
               float* imgInPtr,
               float* imgOutPtr,
               float* imgPsfPtr,
               int    imgCols,
               int    imgRows,
               int    psfColRad,
               int    psfRowRad)
{

   CREATE_CONVOLUTION_VARIABLES(psfColRad, psfRowRad);
   shmOutput[localIdx] = 0.0f;

   const int psfStride = psfRowRad*2+1;
   const int psfPixels = psfStride*(psfColRad*2+1);
   float* shmPsf = (float*)&shmOutput[ROUNDUP32(localPixels)];

   COPY_LIN_ARRAY_TO_SHMEM(imgPsfPtr, shmPsf, psfPixels);

   PREPARE_PADDED_RECTANGLE(psfColRad, psfRowRad);


   __syncthreads();


   float accumFloat = 0.0f;
   for(int coff=-psfColRad; coff<=psfColRad; coff++)
   {
      for(int roff=-psfRowRad; roff<=psfRowRad; roff++)
      {
```

```
         int psfCol = psfColRad - coff;
         int psfRow = psfRowRad - roff;
         int psfIdx = IDX_1D(psfCol, psfRow, psfStride);
         float psfVal = shmPsf[psfIdx];

         int shmPRCol = padRectCol + coff;
         int shmPRRow = padRectRow + roff;
         int shmPRIdx = IDX_1D(shmPRCol, shmPRRow, padRectStride);
         accumFloat += psfVal * shmPadRect[shmPRIdx];
      }
   }
   shmOutput[localIdx] = accumFloat;
   __syncthreads();

   imgOutPtr[globalIdx] = shmOutput[localIdx];
}


/*
// TODO: Still need to debug this function
__global__ void   convolveBilateral(
               float* imgInPtr,
               float* imgOutPtr,
               float* imgPsfPtr,
               float* intPsfPtr,
               int    imgCols,
               int    imgRows,
               int    psfColRad,
               int    psfRowRad,
               int    intPsfRad)
{

   CREATE_CONVOLUTION_VARIABLES(psfColRad, psfRowRad);
   shmOutput[localIdx] = 0.0f;

   const int psfStride = psfRowRad*2+1;
   const int psfPixels = psfStride*(psfColRad*2+1);
   float* shmPsf  = (float*)&shmOutput[ROUNDUP32(localPixels)];
   float* shmPsfI = (float*)&shmPsf[ROUNDUP32(psfPixels)];

   COPY_LIN_ARRAY_TO_SHMEM(imgPsfPtr, shmPsf,  psfPixels);
   COPY_LIN_ARRAY_TO_SHMEM(intPsfPtr, shmPsfI, 2*intPsfRad+1);

   PREPARE_PADDED_RECTANGLE(psfColRad, psfRowRad);


   __syncthreads();


   float accumFloat = 0.0f;
   float myVal = shmPadRect[padRectIdx];
   for(int coff=-psfColRad; coff<=psfColRad; coff++)
   {
      for(int roff=-psfRowRad; roff<=psfRowRad; roff++)
      {
         int psfCol = psfColRad - coff;
         int psfRow = psfRowRad - roff;
         int psfIdx = IDX_1D(psfCol, psfRow, psfStride);
```

```
         float psfVal = shmPsf[psfIdx];

         int shmPRCol = padRectCol + coff;
         int shmPRRow = padRectRow + roff;
         int shmPRIdx = IDX_1D(shmPRCol, shmPRRow, padRectStride);
         float thatVal = shmPadRect[shmPRIdx];

         float intVal = shmPsfI[(int)(thatVal-myVal+intPsfRad)];

         accumFloat += psfVal * intVal *shmPadRect[shmPRIdx];
      }
   }
   shmOutput[localIdx] = accumFloat;
   __syncthreads();

   imgOutPtr[globalIdx] = shmOutput[localIdx];
}
*/


////////////////////////////////////////////////////////////////////////////
//
// For binary morphology (erode/dilate), we use {-1, 0, +1} for two reasons:
//
//    1)  {-1, 0, +1} ~ {OFF, DONTCARE, ON} for more advanced morph operations
//    2)  Computationally, we can multiply SE components to img components and
//        get faster results than using if-statements in the inner loop
//
////////////////////////////////////////////////////////////////////////////
__global__ void  kernelDilate(
               float* imgInPtr,
               float* imgOutPtr,
               float* imgSEPtr,
               int    imgCols,
               int    imgRows,
               int    psfColRad,
               int    psfRowRad,
               int    seNonZero)
{

   CREATE_CONVOLUTION_VARIABLES(psfColRad, psfRowRad);

   const int psfStride = psfRowRad*2+1;
   const int psfPixels = psfStride*(psfColRad*2+1);
   float* shmPsf = (float*)&shmOutput[ROUNDUP32(localPixels)];

   COPY_LIN_ARRAY_TO_SHMEM(imgSEPtr, shmPsf, psfPixels);

   PREPARE_PADDED_RECTANGLE_BINARY(psfColRad, psfRowRad);

   shmOutput[localIdx] = -1.0f;

   __syncthreads();

   float accumFloat = 0.0f;
   for(int coff=-psfColRad; coff<=psfColRad; coff++)
   {
      for(int roff=-psfRowRad; roff<=psfRowRad; roff++)
```

```
      {
         int psfCol = psfColRad + coff;
         int psfRow = psfRowRad + roff;
         int psfIdx = IDX_1D(psfCol, psfRow, psfStride);
         float psfVal = shmPsf[psfIdx];

         int shmPRCol = padRectCol + coff;
         int shmPRRow = padRectRow + roff;
         int shmPRIdx = IDX_1D(shmPRCol, shmPRRow, padRectStride);
         accumFloat += psfVal * shmPadRect[shmPRIdx];
      }
   }
   // If not a single on pixel overlapped with the SE, accumFloat==-seNonZero
   // Since we're using floats, don't use == operator
   if(-accumFloat < seNonZero-0.5)
      shmOutput[localIdx] = 1.0f;

   __syncthreads();

   imgOutPtr[globalIdx] = (shmOutput[localIdx]+1)/2.0f;
}

////////////////////////////////////////////////////////////////////////////
//
// For binary morphology, we use {-1, 0, +1} for two reasons:
//
//    1)  {-1, 0, +1} ~ {OFF, DONTCARE, ON} for more advanced morph operations
//    2)  Computationally, we can multiply SE components to img components and
//        get faster results than using if-statements in the inner loop
//
////////////////////////////////////////////////////////////////////////////
__global__ void  kernelErode(
               float* imgInPtr,
               float* imgOutPtr,
               float* imgSEPtr,
               int    imgCols,
               int    imgRows,
               int    psfColRad,
               int    psfRowRad,
               int    seNonZero)
{

   CREATE_CONVOLUTION_VARIABLES(psfColRad, psfRowRad);

   const int psfStride = psfRowRad*2+1;
   const int psfPixels = psfStride*(psfColRad*2+1);
   float* shmPsf = (float*)&shmOutput[ROUNDUP32(localPixels)];

   COPY_LIN_ARRAY_TO_SHMEM(imgSEPtr, shmPsf, psfPixels);

   PREPARE_PADDED_RECTANGLE_BINARY(psfColRad, psfRowRad);

   shmOutput[localIdx] = -1.0f;

   __syncthreads();

   float accumFloat = 0.0f;
   for(int coff=-psfColRad; coff<=psfColRad; coff++)
```

```
   {
      for(int roff=-psfRowRad; roff<=psfRowRad; roff++)
      {
         int psfCol = psfColRad + coff;
         int psfRow = psfRowRad + roff;
         int psfIdx = IDX_1D(psfCol, psfRow, psfStride);
         float psfVal = shmPsf[psfIdx];

         int shmPRCol = padRectCol + coff;
         int shmPRRow = padRectRow + roff;
         int shmPRIdx = IDX_1D(shmPRCol, shmPRRow, padRectStride);
         accumFloat += psfVal * shmPadRect[shmPRIdx];
      }
   }
   // If every pixel was identical as expected, accumFloat==seNonZero
   // Since we're using floats, don't use == operator
   if(accumFloat > seNonZero-0.5)
      shmOutput[localIdx] = 1.0f;

   __syncthreads();

   imgOutPtr[globalIdx] = (shmOutput[localIdx]+1)/2.0f;
}

#endif
```

Makefile

```
################################################################################
#
# Copyright 1993-2006 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Build script for project
#
################################################################################

# Add source files here
EXECUTABLE  := convolution
# CUDA source files (compiled with cudacc)
CUFILES           := convolution.cu
# CUDA dependency files
CU_DEPS           := convolution_kernel.cu
# C/C++ source files (compiled with gcc / c++)
#CCPFILES         :=


################################################################################
# Rules and targets

include common/common.mk
```

```
################################################################################
#
# Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
#
# NVIDIA Corporation and its licensors retain all intellectual property and
# proprietary rights in and to this software and related documentation.
# Any use, reproduction, disclosure, or distribution of this software
# and related documentation without an express license agreement from
# NVIDIA Corporation is strictly prohibited.
#
# Please refer to the applicable NVIDIA end user license agreement (EULA)
# associated with this source code for terms and conditions that govern
# your use of this NVIDIA software.
#
################################################################################
#
# Common build script for CUDA source projects for Linux and Mac platforms
#
################################################################################

.SUFFIXES : .cu .cu_dbg.o .c_dbg.o .cpp_dbg.o .cu_rel.o .c_rel.o .cpp_rel.o .cubin
.ptx

# Add new SM Versions here as devices with new Compute Capability are released
SM_VERSIONS   := 10 11 12 13 20

CUDA_INSTALL_PATH ?= /usr/local/cuda

ifdef cuda-install
      CUDA_INSTALL_PATH := $(cuda-install)
endif

# detect OS
OSUPPER = $(shell uname -s 2>/dev/null | tr [:lower:] [:upper:])
OSLOWER = $(shell uname -s 2>/dev/null | tr [:upper:] [:lower:])

# 'linux' is output for Linux system, 'darwin' for OS X
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# detect 32-bit or 64-bit platform
HP_64 = $(shell uname -m | grep 64)
OSARCH= $(shell uname -m)

# Basic directory setup for SDK
# (override directories only if they are not already defined)
SRCDIR     ?=
ROOTDIR    ?= .
ROOTBINDIR ?= $(ROOTDIR)/bin
BINDIR     ?= $(ROOTBINDIR)/$(OSLOWER)
ROOTOBJDIR ?= obj
LIBDIR     := $(ROOTDIR)/lib
COMMONDIR  := $(ROOTDIR)/common
SHAREDDIR  := $(ROOTDIR)/common/shared/
```

```
# Compilers
NVCC       := $(CUDA_INSTALL_PATH)/bin/nvcc
CXX        := g++
CC         := gcc
LINK       := g++ -fPIC

# Includes
INCLUDES  += -I. -I$(CUDA_INSTALL_PATH)/include -I$(COMMONDIR)/inc -I$
(SHAREDDIR)/inc

# Warning flags
CXXWARN_FLAGS := \
      -W -Wall \
      -Wimplicit \
      -Wswitch \
      -Wformat \
      -Wchar-subscripts \
      -Wparentheses \
      -Wmultichar \
      -Wtrigraphs \
      -Wpointer-arith \
      -Wcast-align \
      -Wreturn-type \
      -Wno-unused-function \
      $(SPACE)

CWARN_FLAGS := $(CXXWARN_FLAGS) \
      -Wstrict-prototypes \
      -Wmissing-prototypes \
      -Wmissing-declarations \
      -Wnested-externs \
      -Wmain \

# architecture flag for nvcc and gcc compilers build
CUBIN_ARCH_FLAG :=
CXX_ARCH_FLAGS  :=
NVCCFLAGS       :=
LIB_ARCH        := $(OSARCH)

# Determining the necessary Cross-Compilation Flags
# 32-bit OS, but we target 64-bit cross compilation
ifeq ($(x86_64),1)
    NVCCFLAGS       += -m64
    LIB_ARCH         = x86_64
    CUDPPLIB_SUFFIX  = x86_64
    ifneq ($(DARWIN),)
         CXX_ARCH_FLAGS += -arch x86_64
    else
         CXX_ARCH_FLAGS += -m64
    endif
else
# 64-bit OS, and we target 32-bit cross compilation
    ifeq ($(i386),1)
        NVCCFLAGS       += -m32
        LIB_ARCH         = i386
        CUDPPLIB_SUFFIX  = i386
        ifneq ($(DARWIN),)
```

```
             CXX_ARCH_FLAGS += -arch i386
        else
             CXX_ARCH_FLAGS += -m32
        endif
    else
        ifeq "$(strip $(HP_64))" ""
            LIB_ARCH        = i386
            CUDPPLIB_SUFFIX = i386
            NVCCFLAGS      += -m32
            ifneq ($(DARWIN),)
               CXX_ARCH_FLAGS += -arch i386
            else
               CXX_ARCH_FLAGS += -m32
            endif
        else
            LIB_ARCH        = x86_64
            CUDPPLIB_SUFFIX = x86_64
            NVCCFLAGS      += -m64
            ifneq ($(DARWIN),)
               CXX_ARCH_FLAGS += -arch x86_64
            else
               CXX_ARCH_FLAGS += -m64
            endif
        endif
    endif
endif


ifeq ($(noinline),1)
   NVCCFLAGS   += -Xopencc -noinline
   # Compiler-specific flags, when using noinline, we don't build for SM1x
   GENCODE_SM10 :=
   GENCODE_SM20 := -gencode=arch=compute_20,code=\"sm_20,compute_20\"
else
   # Compiler-specific flags (by default, we always use sm_10 and sm_20), unless
we use the SMVERSION template
   #GENCODE_SM10 := -gencode=arch=compute_10,code=\"sm_10,compute_10\"
   GENCODE_SM20 := -gencode=arch=compute_20,code=\"sm_20,compute_20\"
endif

CXXFLAGS  += $(CXXWARN_FLAGS) $(CXX_ARCH_FLAGS)
CFLAGS    += $(CWARN_FLAGS) $(CXX_ARCH_FLAGS)
LINKFLAGS +=
LINK      += $(LINKFLAGS) $(CXX_ARCH_FLAGS)

# This option for Mac allows CUDA applications to work without requiring to set
DYLD_LIBRARY_PATH
ifneq ($(DARWIN),)
   LINK += -Xlinker -rpath $(CUDA_INSTALL_PATH)/lib
endif

# Common flags
COMMONFLAGS += $(INCLUDES) -DUNIX

# Debug/release configuration
ifeq ($(dbg),1)
      COMMONFLAGS += -g
      NVCCFLAGS   += -D_DEBUG -G0 --maxrregcount 20
```

```
      CXXFLAGS    += -D_DEBUG
      CFLAGS      += -D_DEBUG
      BINSUBDIR   := debug
      LIBSUFFIX   := D
else
      COMMONFLAGS += -O2
      BINSUBDIR   := release
      LIBSUFFIX   :=
      NVCCFLAGS   += --compiler-options -fno-strict-aliasing
      CXXFLAGS    += -fno-strict-aliasing
      CFLAGS      += -fno-strict-aliasing
endif

# architecture flag for cubin build
CUBIN_ARCH_FLAG :=

# OpenGL is used or not (if it is used, then it is necessary to include GLEW)
ifeq ($(USEGLLIB),1)
    ifneq ($(DARWIN),)
        OPENGLLIB := -L/System/Library/Frameworks/OpenGL.framework/Libraries
        OPENGLLIB += -lGL -lGLU $(COMMONDIR)/lib/$(OSLOWER)/libGLEW.a
    else
# this case for linux platforms
      OPENGLLIB := -lGL -lGLU -lX11 -lXi -lXmu
# check if x86_64 flag has been set, otherwise, check HP_64 is i386/x86_64
        ifeq ($(x86_64),1)
             OPENGLLIB += -lGLEW_x86_64 -L/usr/X11R6/lib64
        else
             ifeq ($(i386),)
                 ifeq "$(strip $(HP_64))" ""
                   OPENGLLIB += -lGLEW -L/usr/X11R6/lib
                 else
                   OPENGLLIB += -lGLEW_x86_64 -L/usr/X11R6/lib64
                 endif
             endif
        endif
# check if i386 flag has been set, otehrwise check HP_64 is i386/x86_64
        ifeq ($(i386),1)
             OPENGLLIB += -lGLEW -L/usr/X11R6/lib
        else
             ifeq ($(x86_64),)
                 ifeq "$(strip $(HP_64))" ""
                   OPENGLLIB += -lGLEW -L/usr/X11R6/lib
                 else
                   OPENGLLIB += -lGLEW_x86_64 -L/usr/X11R6/lib64
                 endif
             endif
        endif
    endif
endif

ifeq ($(USEGLUT),1)
    ifneq ($(DARWIN),)
      OPENGLLIB += -framework GLUT
    else
        ifeq ($(x86_64),1)
           OPENGLLIB += -lglut -L/usr/lib64
        endif
```

```
        ifeq ($(i386),1)
           OPENGLLIB += -lglut -L/usr/lib
        endif

        ifeq ($(x86_64),)
            ifeq ($(i386),)
              OPENGLLIB += -lglut
            endif
        endif
    endif
endif

ifeq ($(USEPARAMGL),1)
      PARAMGLLIB := -lparamgl_$(LIB_ARCH)$(LIBSUFFIX)
endif

ifeq ($(USERENDERCHECKGL),1)
      RENDERCHECKGLLIB := -lrendercheckgl_$(LIB_ARCH)$(LIBSUFFIX)
endif

ifeq ($(USECUDPP), 1)
    CUDPPLIB := -lcudpp_$(CUDPPLIB_SUFFIX)$(LIBSUFFIX)

    ifeq ($(emu), 1)
        CUDPPLIB := $(CUDPPLIB)_emu
    endif
endif

ifeq ($(USENVCUVID), 1)
     ifneq ($(DARWIN),)
         NVCUVIDLIB := -Lcommon/lib/darwin -lnvcuvid
     endif
endif

# Libs
ifneq ($(DARWIN),)
    LIB       := -L$(CUDA_INSTALL_PATH)/lib -L$(LIBDIR) -L$(COMMONDIR)/lib/$
(OSLOWER) -L$(SHAREDDIR)/lib $(NVCUVIDLIB)
else
  ifeq "$(strip $(HP_64))" ""
    ifeq ($(x86_64),1)
       LIB       := -L$(CUDA_INSTALL_PATH)/lib64 -L$(LIBDIR) -L$(COMMONDIR)/lib/$
(OSLOWER) -L$(SHAREDDIR)/lib
    else
       LIB       := -L$(CUDA_INSTALL_PATH)/lib -L$(LIBDIR) -L$(COMMONDIR)/lib/$
(OSLOWER) -L$(SHAREDDIR)/lib
    endif
  else
    ifeq ($(i386),1)
       LIB       := -L$(CUDA_INSTALL_PATH)/lib -L$(LIBDIR) -L$(COMMONDIR)/lib/$
(OSLOWER) -L$(SHAREDDIR)/lib
    else
       LIB       := -L$(CUDA_INSTALL_PATH)/lib64 -L$(LIBDIR) -L$(COMMONDIR)/lib/$
(OSLOWER) -L$(SHAREDDIR)/lib
    endif
  endif
endif
```

```
# If dynamically linking to CUDA and CUDART, we exclude the libraries from the LIB
ifeq ($(USECUDADYNLIB),1)
     LIB += ${OPENGLLIB} $(PARAMGLLIB) $(RENDERCHECKGLLIB) $(CUDPPLIB) ${LIB} -ldl
-rdynamic
else
# static linking, we will statically link against CUDA and CUDART
  ifeq ($(USEDRVAPI),1)
     LIB += -lcuda   ${OPENGLLIB} $(PARAMGLLIB) $(RENDERCHECKGLLIB) $(CUDPPLIB) $
{LIB}
  else
     ifeq ($(emu),1)
         LIB += -lcudartemu
     else
         LIB += -lcudart
     endif
     LIB += ${OPENGLLIB} $(PARAMGLLIB) $(RENDERCHECKGLLIB) $(CUDPPLIB) ${LIB}
  endif
endif

ifeq ($(USECUFFT),1)
  ifeq ($(emu),1)
    LIB += -lcufftemu
  else
    LIB += -lcufft
  endif
endif

ifeq ($(USECUBLAS),1)
  ifeq ($(emu),1)
    LIB += -lcublasemu
  else
    LIB += -lcublas
  endif
endif

# Lib/exe configuration
ifneq ($(STATIC_LIB),)
      TARGETDIR := $(LIBDIR)
      TARGET   := $(subst .a,_$(LIB_ARCH)$(LIBSUFFIX).a,$(LIBDIR)/$(STATIC_LIB))
      LINKLINE  = ar rucv $(TARGET) $(OBJS)
else
      ifneq ($(OMIT_CUTIL_LIB),1)
            LIB += -lcutil_$(LIB_ARCH)$(LIBSUFFIX) -lshrutil_$(LIB_ARCH)$
(LIBSUFFIX)
      endif
      # Device emulation configuration
      ifeq ($(emu), 1)
            NVCCFLAGS   += -deviceemu
            CUDACCFLAGS +=
            BINSUBDIR   := emu$(BINSUBDIR)
            # consistency, makes developing easier
            CXXFLAGS          += -D__DEVICE_EMULATION__
            CFLAGS                  += -D__DEVICE_EMULATION__
      endif
      TARGETDIR := $(BINDIR)/$(BINSUBDIR)
      TARGET    := $(TARGETDIR)/$(EXECUTABLE)
      LINKLINE  = $(LINK) -o $(TARGET) $(OBJS) $(LIB)
endif
```

```
# check if verbose
ifeq ($(verbose), 1)
      VERBOSE :=
else
      VERBOSE := @
endif

################################################################################
# Check for input flags and set compiler flags appropriately
################################################################################
ifeq ($(fastmath), 1)
      NVCCFLAGS += -use_fast_math
endif

ifeq ($(nvccverbose), 1)
        NVCCFLAGS += -v
endif

ifeq ($(keep), 1)
      NVCCFLAGS += -keep
      NVCC_KEEP_CLEAN := *.i* *.cubin *.cu.c *.cudafe* *.fatbin.c *.ptx
endif

ifdef maxregisters
      NVCCFLAGS += -maxrregcount $(maxregisters)
endif

# Add cudacc flags
NVCCFLAGS += $(CUDACCFLAGS)

# Add common flags
NVCCFLAGS += $(COMMONFLAGS)
CXXFLAGS  += $(COMMONFLAGS)
CFLAGS    += $(COMMONFLAGS)

ifeq ($(nvcc_warn_verbose),1)
      NVCCFLAGS += $(addprefix --compiler-options ,$(CXXWARN_FLAGS))
      NVCCFLAGS += --compiler-options -fno-strict-aliasing
endif

################################################################################
# Set up object files
################################################################################
OBJDIR := $(ROOTOBJDIR)/$(LIB_ARCH)/$(BINSUBDIR)
OBJS +=  $(patsubst %.cpp,$(OBJDIR)/%.cpp.o,$(notdir $(CCPFILES)))
OBJS +=  $(patsubst %.c,$(OBJDIR)/%.c.o,$(notdir $(CFILES)))
OBJS +=  $(patsubst %.cu,$(OBJDIR)/%.cu.o,$(notdir $(CUFILES)))

################################################################################
# Set up cubin output files
################################################################################
CUBINDIR := $(SRCDIR)data
CUBINS +=  $(patsubst %.cu,$(CUBINDIR)/%.cubin,$(notdir $(CUBINFILES)))

################################################################################
# Set up PTX output files
################################################################################
```

```
PTXDIR := $(SRCDIR)data
PTXBINS +=  $(patsubst %.cu,$(PTXDIR)/%.ptx,$(notdir $(PTXFILES)))

################################################################################
# Rules
################################################################################
$(OBJDIR)/%.c.o : $(SRCDIR)%.c $(C_DEPS)
      echo $(VERBOSE)$(CC) $(CFLAGS) -o $@ -c $<
      $(VERBOSE)$(CC) $(CFLAGS) -o $@ -c $<

$(OBJDIR)/%.cpp.o : $(SRCDIR)%.cpp $(C_DEPS)
      echo $(VERBOSE)$(CXX) $(CXXFLAGS) -o $@ -c $<
      $(VERBOSE)$(CXX) $(CXXFLAGS) -o $@ -c $<

# Default arch includes gencode for sm_10, sm_20, and other archs from
GENCODE_ARCH declared in the makefile
$(OBJDIR)/%.cu.o : $(SRCDIR)%.cu $(CU_DEPS)
      echo $(VERBOSE)$(NVCC) $(GENCODE_SM10) $(GENCODE_ARCH) $(GENCODE_SM20) $
(NVCCFLAGS) $(SMVERSIONFLAGS) -o $@ -c $<
      $(VERBOSE)$(NVCC) $(GENCODE_SM10) $(GENCODE_ARCH) $(GENCODE_SM20) $
(NVCCFLAGS) $(SMVERSIONFLAGS) -o $@ -c $<

# Default arch includes gencode for sm_10, sm_20, and other archs from
GENCODE_ARCH declared in the makefile
$(CUBINDIR)/%.cubin : $(SRCDIR)%.cu cubindirectory
      echo $(VERBOSE)$(NVCC) $(GENCODE_SM10) $(GENCODE_ARCH) $(GENCODE_SM20) $
(CUBIN_ARCH_FLAG) $(NVCCFLAGS) $(SMVERSIONFLAGS) -o $@ -cubin $<
      $(VERBOSE)$(NVCC) $(GENCODE_SM10) $(GENCODE_ARCH) $(GENCODE_SM20) $
(CUBIN_ARCH_FLAG) $(NVCCFLAGS) $(SMVERSIONFLAGS) -o $@ -cubin $<

$(PTXDIR)/%.ptx : $(SRCDIR)%.cu ptxdirectory
      echo $(VERBOSE)$(NVCC) $(CUBIN_ARCH_FLAG) $(NVCCFLAGS) $(SMVERSIONFLAGS) -o
$@ -ptx $<
      $(VERBOSE)$(NVCC) $(CUBIN_ARCH_FLAG) $(NVCCFLAGS) $(SMVERSIONFLAGS) -o $@
-ptx $<

#
# The following definition is a template that gets instantiated for each SM
# version (sm_10, sm_13, etc.) stored in SMVERSIONS.  It does 2 things:
# 1. It adds to OBJS a .cu_sm_XX.o for each .cu file it finds in CUFILES_sm_XX.
# 2. It generates a rule for building .cu_sm_XX.o files from the corresponding
#    .cu file.
#
# The intended use for this is to allow Makefiles that use common.mk to compile
# files to different Compute Capability targets (aka SM arch version).  To do
# so, in the Makefile, list files for each SM arch separately, like so:
# This will be used over the default rule abov
#
# CUFILES_sm_10 := mycudakernel_sm10.cu app.cu
# CUFILES_sm_12 := anothercudakernel_sm12.cu
#
define SMVERSION_template
#OBJS += $(patsubst %.cu,$(OBJDIR)/%.cu_$(1).o,$(notdir $(CUFILES_$(1))))
OBJS += $(patsubst %.cu,$(OBJDIR)/%.cu_$(1).o,$(notdir $(CUFILES_sm_$(1))))
$(OBJDIR)/%.cu_$(1).o : $(SRCDIR)%.cu $(CU_DEPS)
#     $(VERBOSE)$(NVCC) -o $$@ -c $$< $(NVCCFLAGS)  $(1)
   # if we have noinline enabled, we only turn this enable this for SM 2.x
architectures
```

```
   ifeq ($(noinline),1)
      echo $(VERBOSE)$(NVCC) $(GENCODE_SM20) -o $$@ -c $$< $(NVCCFLAGS)
      $(VERBOSE)$(NVCC) $(GENCODE_SM20) -o $$@ -c $$< $(NVCCFLAGS)
   else
      echo $(VERBOSE)$(NVCC)
-gencode=arch=compute_$(1),code=\"sm_$(1),compute_$(1)\" $(GENCODE_SM20) -o $$@ -c
$$< $(NVCCFLAGS)
      $(VERBOSE)$(NVCC) -gencode=arch=compute_$(1),code=\"sm_$(1),compute_$(1)\" $
(GENCODE_SM20) -o $$@ -c $$< $(NVCCFLAGS)
   endif
endef

# This line invokes the above template for each arch version stored in
# SM_VERSIONS.  The call funtion invokes the template, and the eval
# function interprets it as make commands.
$(foreach smver,$(SM_VERSIONS),$(eval $(call SMVERSION_template,$(smver))))

$(TARGET): makedirectories $(OBJS) $(CUBINS) $(PTXBINS) Makefile
      $(VERBOSE)$(LINKLINE)

cubindirectory:
      $(VERBOSE)mkdir -p $(CUBINDIR)

ptxdirectory:
      $(VERBOSE)mkdir -p $(PTXDIR)

makedirectories:
      $(VERBOSE)mkdir -p $(LIBDIR)
      $(VERBOSE)mkdir -p $(OBJDIR)
      $(VERBOSE)mkdir -p $(TARGETDIR)


tidy :
      $(VERBOSE)find . | egrep "#" | xargs rm -f
      $(VERBOSE)find . | egrep "\~" | xargs rm -f

clean : tidy
      $(VERBOSE)rm -f $(OBJS)
      $(VERBOSE)rm -f $(CUBINS)
      $(VERBOSE)rm -f $(PTXBINS)
      $(VERBOSE)rm -f $(TARGET)
      $(VERBOSE)rm -f $(NVCC_KEEP_CLEAN)
      $(VERBOSE)rm -f $(ROOTBINDIR)/$(OSLOWER)/$(BINSUBDIR)/*.ppm
      $(VERBOSE)rm -f $(ROOTBINDIR)/$(OSLOWER)/$(BINSUBDIR)/*.pgm
      $(VERBOSE)rm -f $(ROOTBINDIR)/$(OSLOWER)/$(BINSUBDIR)/*.bin
      $(VERBOSE)rm -f $(ROOTBINDIR)/$(OSLOWER)/$(BINSUBDIR)/*.bmp

clobber : clean
      $(VERBOSE)rm -rf $(ROOTOBJDIR)
```

C.2. Disparidad estero(Vision estereo)

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# CUDA code generation flags
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM20) $(GENCODE_SM30)

################################################################################
```

```
# Target rules
all: build

build: stereoDisparity

stereoDisparity.o: stereoDisparity.cu
	$(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

stereoDisparity: stereoDisparity.o
	$(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	./stereoDisparity

clean:
	rm -f stereoDisparity stereoDisparity.o *.bin
	rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/stereoDisparity

clobber: clean
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/* A CUDA program that demonstrates how to compute a stereo disparity map using
 *   SIMD SAD (Sum of Absolute Difference) intrinsics
 */

// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, kernels
#include <cuda_runtime.h>
#include "stereoDisparity_kernel.cuh"


// includes, project
#include <helper_functions.h>  // helper for shared that are common to CUDA SDK
samples
#include <helper_cuda.h>       // helper for checking cuda initialization and
error checking
#include <helper_string.h>     // helper functions for string parsing


static char *sSDKsample = "[stereoDisparity]\0";

int iDivUp(int a, int b)
{
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}


////////////////////////////////////////////////////////////////////////////////
// declaration, forward
void runTest(int argc, char **argv);

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int
main(int argc, char **argv)
{
    runTest(argc, argv);
}

////////////////////////////////////////////////////////////////////////////////
//! CUDA Sample for calculating depth maps
```

```
////////////////////////////////////////////////////////////////////////////
void
runTest(int argc, char **argv)
{
    cudaDeviceProp deviceProp;
    deviceProp.major = 0;
    deviceProp.minor = 0;
    int dev = 0;

    // This will pick the best possible CUDA capable device
    dev = findCudaDevice(argc, (const char **)argv);

    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, dev));

    // Statistics about the GPU device
    printf("> GPU device has %d Multi-Processors, SM %d.%d compute
capabilities\n\n",
           deviceProp.multiProcessorCount, deviceProp.major, deviceProp.minor);

    int version = (deviceProp.major * 0x10 + deviceProp.minor);

    if (version < 0x20)
    {
        printf("%s: requires a minimum CUDA compute 2.0 capability\n",
sSDKsample);
        exit(EXIT_SUCCESS);
    }

    StopWatchInterface *timer;
    sdkCreateTimer(&timer);

    // Search paramters
    int minDisp = -16;
    int maxDisp = 0;

    // Load image data
    //allocate mem for the images on host side
    //initialize pointers to NULL to request lib call to allocate as needed
    // PPM images are loaded into 4 byte/pixel memory (RGBX)
    unsigned char *h_img0 = NULL;
    unsigned char *h_img1 = NULL;
    unsigned int w, h;
    char *fname0 = sdkFindFilePath("stereo.im0.640x533.ppm", argv[0]);
    char *fname1 = sdkFindFilePath("stereo.im1.640x533.ppm", argv[0]);

    printf("Loaded <%s> as image 0\n", fname0);

    if (!sdkLoadPPM4ub(fname0, &h_img0, &w, &h))
    {
        fprintf(stderr, "Failed to load <%s>\n", fname0);
    }

    printf("Loaded <%s> as image 1\n", fname1);

    if (!sdkLoadPPM4ub(fname1, &h_img1, &w, &h))
    {
        fprintf(stderr, "Failed to load <%s>\n", fname1);
    }
```

```
    dim3 numThreads = dim3(blockSize_x, blockSize_y, 1);
    dim3 numBlocks = dim3(iDivUp(w, numThreads.x), iDivUp(h, numThreads.y));
    unsigned int numData = w*h;
    unsigned int memSize = sizeof(int) * numData;

    //allocate mem for the result on host side
    unsigned int *h_odata = (unsigned int *)malloc(memSize);

    //initalize the memory
    for (unsigned int i = 0; i < numData; i++)
        h_odata[i] = 0;

    // allocate device memory for result
    unsigned int *d_odata, *d_img0, *d_img1;

    checkCudaErrors(cudaMalloc((void **) &d_odata, memSize));
    checkCudaErrors(cudaMalloc((void **) &d_img0, memSize));
    checkCudaErrors(cudaMalloc((void **) &d_img1, memSize));

    // copy host memory to device to initialize to zeros
    checkCudaErrors(cudaMemcpy(d_img0,  h_img0, memSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_img1,  h_img1, memSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(d_odata, h_odata, memSize,
cudaMemcpyHostToDevice));

    size_t offset = 0;
    cudaChannelFormatDesc ca_desc0 = cudaCreateChannelDesc<unsigned int>();
    cudaChannelFormatDesc ca_desc1 = cudaCreateChannelDesc<unsigned int>();

    tex2Dleft.addressMode[0] = cudaAddressModeClamp;
    tex2Dleft.addressMode[1] = cudaAddressModeClamp;
    tex2Dleft.filterMode     = cudaFilterModePoint;
    tex2Dleft.normalized     = false;
    tex2Dright.addressMode[0] = cudaAddressModeClamp;
    tex2Dright.addressMode[1] = cudaAddressModeClamp;
    tex2Dright.filterMode     = cudaFilterModePoint;
    tex2Dright.normalized     = false;
    checkCudaErrors(cudaBindTexture2D(&offset, tex2Dleft,  d_img0, ca_desc0, w, h,
w*4));
    assert(offset == 0);

    checkCudaErrors(cudaBindTexture2D(&offset, tex2Dright, d_img1, ca_desc1, w, h,
w*4));
    assert(offset == 0);

    // First run the warmup kernel (which we'll use to get the GPU in the correct
max power state
    stereoDisparityKernel<<<numBlocks, numThreads>>>(d_img0, d_img1, d_odata, w,
h, minDisp, maxDisp);
    cudaDeviceSynchronize();

    // Allocate CUDA events that we'll use for timing
    cudaEvent_t start, stop;
    checkCudaErrors(cudaEventCreate(&start));
    checkCudaErrors(cudaEventCreate(&stop));

    printf("Launching CUDA stereoDisparityKernel()\n");
```

```
    // Record the start event
    checkCudaErrors(cudaEventRecord(start, NULL));

    // launch the stereoDisparity kernel
    stereoDisparityKernel<<<numBlocks, numThreads>>>(d_img0, d_img1, d_odata, w,
h, minDisp, maxDisp);

    // Record the stop event
    checkCudaErrors(cudaEventRecord(stop, NULL));

    // Wait for the stop event to complete
    checkCudaErrors(cudaEventSynchronize(stop));

    // Check to make sure the kernel didn't fail
    getLastCudaError("Kernel execution failed");

    float msecTotal = 0.0f;
    checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));

    //Copy result from device to host for verification
    checkCudaErrors(cudaMemcpy(h_odata, d_odata, memSize,
cudaMemcpyDeviceToHost));

    printf("Input Size  [%dx%d], ", w, h);
    printf("Kernel size [%dx%d], ", (2*RAD+1), (2*RAD+1));
    printf("Disparities [%d:%d]\n", minDisp, maxDisp);

    printf("GPU processing time : %.4f (ms)\n", msecTotal);
    printf("Pixel throughput    : %.3f Mpixels/sec\n", ((float)(w
*h*1000.f)/msecTotal)/1000000);

    // calculate sum of resultant GPU image
    unsigned int checkSum = 0;

    for (unsigned int i=0 ; i<w *h ; i++)
    {
        checkSum += h_odata[i];
    }
    printf("GPU Checksum = %u, ", checkSum);

    // write out the resulting disparity image.
    unsigned char *dispOut = (unsigned char *)malloc(numData);
    int mult = 20;
    char *fnameOut = "output_GPU.pgm";

    for (unsigned int i=0; i<numData; i++)
    {
        dispOut[i] = (int)h_odata[i]*mult;
    }
    printf("GPU image: <%s>\n", fnameOut);
    sdkSavePGM(fnameOut, dispOut, w, h);

    //compute reference solution
    printf("Computing CPU reference...\n");
    cpu_gold_stereo((unsigned int *)h_img0, (unsigned int *)h_img1, (unsigned int
*)h_odata, w, h, minDisp, maxDisp);
    unsigned int cpuCheckSum = 0;
```

```
    for (unsigned int i=0 ; i<w *h ; i++)
    {
        cpuCheckSum += h_odata[i];
    }
    printf("CPU Checksum = %u, ", cpuCheckSum);
    char *cpuFnameOut = "output_CPU.pgm";

    for (unsigned int i=0; i<numData; i++)
    {
        dispOut[i] = (int)h_odata[i]*mult;
    }
    printf("CPU image: <%s>\n", cpuFnameOut);
    sdkSavePGM(cpuFnameOut, dispOut, w, h);

    // cleanup memory
    checkCudaErrors(cudaFree(d_odata));
    checkCudaErrors(cudaFree(d_img0));
    checkCudaErrors(cudaFree(d_img1));

    if (h_odata != NULL) free(h_odata);
    if (h_img0 != NULL) free(h_img0);
    if (h_img1 != NULL) free(h_img1);
    if (dispOut != NULL) free(dispOut);
    sdkDeleteTimer(&timer);

    cudaDeviceReset();

    exit((checkSum == cpuCheckSum) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/* Simple kernel computes a Stereo Disparity using CUDA SIMD SAD intrinsics. */

#ifndef _STEREODISPARITY_KERNEL_H_
#define _STEREODISPARITY_KERNEL_H_

#define blockSize_x 32
#define blockSize_y 8

// RAD is the radius of the region of support for the search
#define RAD 8
// STEPS is the number of loads we must perform to initialize the shared memory
area
// (see convolution SDK sample for example)
#define STEPS 3

texture<unsigned int, cudaTextureType2D, cudaReadModeElementType> tex2Dleft;
texture<unsigned int, cudaTextureType2D, cudaReadModeElementType> tex2Dright;


////////////////////////////////////////////////////////////////////////////
// This function applies the video instrinsic operations to compute a
// sum of absolute differences.  The absolute differences are computed
// and the optional .add instruction is used to sum the lanes.
//
// For more information, see also the documents:
//  "Using_Inline_PTX_Assembly_In_CUDA.pdf"
// and also the PTX ISA documentation for the architecture in question, e.g.:
//  "ptx_isa_3.0K.pdf"
// included in the NVIDIA GPU Computing Toolkit
////////////////////////////////////////////////////////////////////////////
__device__ unsigned int __usad4(unsigned int A, unsigned int B, unsigned int C=0)
{
    unsigned int result;
#if (__CUDA_ARCH__ >= 300) // Kepler (SM 3.x) supports a 4 vector SAD SIMD
    asm("vabsdiff4.u32.u32.u32.add" " %0, %1, %2, %3;": "=r"(result):"r"(A),
"r"(B), "r"(C));
#else // SM 2.0            // Fermi  (SM 2.x) supports only 1 SAD SIMD, so there
are 4 instructions
    asm("vabsdiff.u32.u32.u32.add" " %0, %1.b0, %2.b0, %3;": "=r"(result):"r"(A),
"r"(B), "r"(C));
    asm("vabsdiff.u32.u32.u32.add" " %0, %1.b1, %2.b1, %3;": "=r"(result):"r"(A),
"r"(B), "r"(result));
    asm("vabsdiff.u32.u32.u32.add" " %0, %1.b2, %2.b2, %3;": "=r"(result):"r"(A),
"r"(B), "r"(result));
    asm("vabsdiff.u32.u32.u32.add" " %0, %1.b3, %2.b3, %3;": "=r"(result):"r"(A),
"r"(B), "r"(result));
```

```
#endif
    return result;
}

////////////////////////////////////////////////////////////////////////////
//! Simple stereo disparity kernel to test atomic instructions
//! Algorithm Explanation:
//! For stereo disparity this performs a basic block matching scheme.
//! The sum of abs. diffs between and area of the candidate pixel in the left
images
//! is computed against different horizontal shifts of areas from the right.
//! Ths shift at which the difference is minimum is taken as how far that pixel
//! moved between left/right image pairs.   The recovered motion is the disparity
map
//! More motion indicates more parallax indicates a closer object.
//! @param g_img1  image 1 in global memory, RGBA, 4 bytes/pixel
//! @param g_img2  image 2 in global memory
//! @param g_odata disparity map output in global memory,  unsigned int
output/pixel
//! @param w image width in pixels
//! @param h image height in pixels
//! @param minDisparity leftmost search range
//! @param maxDisparity rightmost search range
////////////////////////////////////////////////////////////////////////////
__global__ void
stereoDisparityKernel(unsigned int *g_img0, unsigned int *g_img1,
                      unsigned int *g_odata,
                      int w, int h,
                      int minDisparity, int maxDisparity)
{
    // access thread id
    const int tidx = blockDim.x * blockIdx.x + threadIdx.x;
    const int tidy = blockDim.y * blockIdx.y + threadIdx.y;
    const unsigned int sidx = threadIdx.x+RAD;
    const unsigned int sidy = threadIdx.y+RAD;

    unsigned int imLeft;
    unsigned int imRight;
    unsigned int cost;
    unsigned int bestCost = 9999999;
    unsigned int bestDisparity = 0;
    __shared__ unsigned int diff[blockSize_y+2*RAD][blockSize_x+2*RAD];

    // store needed values for left image into registers (constant indexed local
vars)
    unsigned int imLeftA[STEPS];
    unsigned int imLeftB[STEPS];

    for (int i=0; i<STEPS; i++)
    {
        int offset = -RAD + i*RAD;
        imLeftA[i] = tex2D(tex2Dleft, tidx-RAD, tidy+offset);
        imLeftB[i] = tex2D(tex2Dleft, tidx-RAD+blockSize_x, tidy+offset);
    }

    // for a fixed camera system this could be hardcoded and loop unrolled
    for (int d=minDisparity; d<=maxDisparity; d++)
    {
```

```
        //LEFT
#pragma unroll
        for (int i=0; i<STEPS; i++)
        {
            int offset = -RAD + i*RAD;
            //imLeft = tex2D( tex2Dleft, tidx-RAD, tidy+offset );
            imLeft = imLeftA[i];
            imRight = tex2D(tex2Dright, tidx-RAD+d, tidy+offset);
            cost = __usad4(imLeft, imRight);
            diff[sidy+offset][sidx-RAD] = cost;
        }

        //RIGHT
#pragma unroll

        for (int i=0; i<STEPS; i++)
        {
            int offset = -RAD + i*RAD;

            if (threadIdx.x < 2*RAD)
            {
                //imLeft = tex2D( tex2Dleft, tidx-RAD+blockSize_x, tidy+offset );
                imLeft = imLeftB[i];
                imRight = tex2D(tex2Dright, tidx-RAD+blockSize_x+d, tidy+offset);
                cost = __usad4(imLeft, imRight);
                diff[sidy+offset][sidx-RAD+blockSize_x] = cost;
            }
        }

        __syncthreads();

        // sum cost horizontally
#pragma unroll

        for (int j=0; j<STEPS; j++)
        {
            int offset = -RAD + j*RAD;
            cost = 0;
#pragma unroll

            for (int i=-RAD; i<=RAD ; i++)
            {
                cost += diff[sidy+offset][sidx+i];
            }

            __syncthreads();
            diff[sidy+offset][sidx] = cost;
            __syncthreads();

        }

        // sum cost vertically
        cost = 0;
#pragma unroll

        for (int i=-RAD; i<=RAD ; i++)
        {
            cost += diff[sidy+i][sidx];
```

```
        }

        // see if it is better or not
        if (cost < bestCost)
        {
            bestCost = cost;
            bestDisparity = d+8;
        }

        __syncthreads();

    }

    if (tidy < h && tidx < w)
    {
        g_odata[tidy*w + tidx] = bestDisparity;
    }
}

void cpu_gold_stereo(unsigned int *img0, unsigned int *img1, unsigned int *odata,
                     int w, int h, int minDisparity, int maxDisparity)
{
    for (int y = 0 ; y< h ; y++)
    {
        for (int x = 0 ; x< w ; x++)
        {
            unsigned int bestCost = 9999999;
            unsigned int bestDisparity = 0;

            for (int d=minDisparity; d<=maxDisparity; d++)
            {
                unsigned int cost = 0;

                for (int i=-RAD; i<=RAD; i++)
                {
                    for (int j=-RAD; j<=RAD; j++)
                    {
                        //border clamping
                        int yy,xx,xxd;
                        yy = y+i;

                        if (yy < 0) yy = 0;

                        if (yy >= h) yy = h-1;

                        xx = x+j;

                        if (xx < 0) xx = 0;

                        if (xx >= w) xx = w-1;

                        xxd = x+j+d;

                        if (xxd < 0) xxd = 0;

                        if (xxd >= w) xxd = w-1;

                        // sum abs diff across components
```

```
                        unsigned char *A = (unsigned char *)&img0[yy*w + xx];
                        unsigned char *B = (unsigned char *)&img1[yy*w + xxd];
                        unsigned int absdiff = 0;

                        for (int k=0; k<4; k++)
                        {
                            absdiff += abs((int)(A[k] - B[k]));
                        }

                        cost += absdiff;
                    }
                }

                if (cost < bestCost)
                {
                    bestCost = cost;
                    bestDisparity = d+8;
                }

            }// end for disparities

            // store to best disparity
            odata[y*w + x ] = bestDisparity;
        }
    }
}
#endif // #ifndef _STEREODISPARITY_KERNEL_H_
```

C.3. CUDA FFT Simulaión de oceano

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)",'')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)",'')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
```

```
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to
include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially
when using the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
```

```
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","debian")
    CUDAPATH  ?= /usr/lib/nvidia-current
    CUDALINK  ?= -L/usr/lib/nvidia-current
    DFLT_PATH  = /usr/lib
endif
ifeq ("$(DISTRO)","suse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","suse linux")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?= /usr/lib64/nvidia
        CUDALINK  ?= -L/usr/lib64/nvidia
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print
2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-
install the NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
#  findgllib.mk is used to find the necessary GL Libraries for specific distributions
#               this is supported on Mac OSX and Linux Platforms
#
################################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   # $(info DISTRO1 = $(DISTRO) $(DISTVER))
   ifeq ($(DISTRO),)
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$
$/d" 2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null)
     # $(info DISTRO2 = $(DISTRO) $(DISTVER))
   endif
   ifeq ($(DISTRO),)
     # third, we can search in /etc/os-release or /etc/{distro}-release
```

```
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" |
grep -v "ID" | grep -v "DISTRIB")
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed
's/DISTRIB_RELEASE=//' | grep -v "DISTRIB_RELEASE")
     # $(info DISTRO3 = $(DISTRO) $(DISTVER))
   endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
      OS_SIZE = 32
      OS_ARCH = i686
endif
ifeq ($(x86_64),1)
      OS_SIZE = 64
      OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
      OS_SIZE = 32
      OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # $(info) >> findgllib.mk -> LINUX path <<<)
    # Each set of Linux Distros have different paths for where to find their OpenGL
libraries reside
    ifeq ("$(DISTRO)","ubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","debian")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","suse")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","suse linux")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
```

```
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
    GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?= /usr/X11R6/lib
    GLLINK    ?= -L/usr/X11R6/lib
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","red")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhatenterpriseworkstation")
  ifeq ($(OS_SIZE),64)
    GLPATH    ?= /usr/lib64/nvidia
    GLLINK    ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    GLPATH    ?=
    GLLINK    ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","centos")
  ifeq ($(OS_SIZE),64)
```

```
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif

    ifeq ($(ARMv7),1)
      GLPATH := /usr/arm-linux-gnueabihf/lib
      GLLINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        GLPATH += $(TARGET_FS)/usr/lib/nvidia-current $(TARGET_FS)/usr/lib/arm-linux-
gnueabihf
        GLLINK += -L$(TARGET_FS)/usr/lib/nvidia-current -L$(TARGET_FS)/usr/lib/arm-
linux-gnueabihf
      endif
    endif

  # find libGL, libGLU, libXi,
  GLLIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libGL.so  -print 2>/dev/null)
  GLULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libGLU.so -print 2>/dev/null)
  X11LIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libX11.so -print 2>/dev/null)
  XILIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libXi.so  -print 2>/dev/null)
  XMULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libXmu.so -print 2>/dev/null)

  ifeq ("$(GLLIB)",'')
      $(info >>> WARNING - libGL.so not found, refer to CUDA Samples release notes for
how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(GLULIB)",'')
      $(info >>> WARNING - libGLU.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(X11LIB)",'')
      $(info >>> WARNING - libX11.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XILIB)",'')
      $(info >>> WARNING - libXi.so not found, refer to CUDA Samples release notes for
how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XMULIB)",'')
      $(info >>> WARNING - libXmu.so not found, refer to CUDA Samples release notes
for how to find and install them. <<<)
      EXEC=@echo "[@]"
  endif
else
    # This would be the Mac OS X path if we had to do anything special
endif
```

```
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH ?= "/usr/local/cuda-5.5"

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
```

```
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# Makefile include to help find GL Libraries
EXEC            ?=
include ./findgllib.mk

# OpenGL specific libraries
ifneq ($(DARWIN),)
  # Mac OSX specific libraries and paths to include
  LIBRARIES += -L/System/Library/Frameworks/OpenGL.framework/Libraries
```

```
  LIBRARIES += -lGL -lGLU ../../common/lib/darwin/libGLEW.a
  ALL_LDFLAGS += -Xlinker -framework -Xlinker GLUT
else
  LIBRARIES += -L../../common/lib/$(OSLOWER)/$(OS_ARCH) $(GLLINK)
  LIBRARIES += -lGL -lGLU -lX11 -lXi -lXmu -lglut -lGLEW
endif

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(GENCODE_SM30)

LIBRARIES += -lcufft

################################################################################

# Target rules
all: build

build: oceanFFT

oceanFFT_kernel.o: oceanFFT_kernel.cu
	$(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

oceanFFT.o: oceanFFT.cpp
	$(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

oceanFFT: oceanFFT.o oceanFFT_kernel.o
	$(EXEC) $(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	$(EXEC) mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))
	$(EXEC) cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	$(EXEC) ./oceanFFT

clean:
	$(EXEC) rm -f oceanFFT oceanFFT.o oceanFFT_kernel.o
	$(EXEC) rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$
(abi))/oceanFFT

clobber: clean
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/*
    FFT-based Ocean simulation
    based on original code by Yury Uralsky and Calvin Lin

    This sample demonstrates how to use CUFFT to synthesize and
    render an ocean surface in real-time.

    See Jerry Tessendorf's Siggraph course notes for more details:
    http://tessendorf.org/reports.html

    It also serves as an example of how to generate multiple vertex
    buffer streams from CUDA and render them using GLSL shaders.
*/

#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  define NOMINMAX
#  include <windows.h>
#endif

// includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <cufft.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

#include <rendercheck_gl.h>

const char *sSDKsample = "CUDA FFT Ocean Simulation";
```

```
#define MAX_EPSILON 0.10f
#define THRESHOLD   0.15f
#define REFRESH_DELAY     10 //ms

////////////////////////////////////////////////////////////////////////////////
// constants
unsigned int windowW = 512, windowH = 512;

const unsigned int meshSize = 256;
const unsigned int spectrumW = meshSize + 4;
const unsigned int spectrumH = meshSize + 1;

const int frameCompare = 4;

// OpenGL vertex buffers
GLuint posVertexBuffer;
GLuint heightVertexBuffer, slopeVertexBuffer;
struct cudaGraphicsResource *cuda_posVB_resource, *cuda_heightVB_resource,
*cuda_slopeVB_resource; // handles OpenGL-CUDA exchange

GLuint indexBuffer;
GLuint shaderProg;
char *vertShaderPath = 0, *fragShaderPath = 0;

// mouse controls
int mouseOldX, mouseOldY;
int mouseButtons = 0;
float rotateX = 20.0f, rotateY = 0.0f;
float translateX = 0.0f, translateY = 0.0f, translateZ = -2.0f;

bool animate = true;
bool drawPoints = false;
bool wireFrame = false;
bool g_hasDouble = false;

// FFT data
cufftHandle fftPlan;
float2 *d_h0 = 0;   // heightfield at time 0
float2 *h_h0 = 0;
float2 *d_ht = 0;   // heightfield at time t
float2 *d_slope = 0;

// pointers to device object
float *g_hptr = NULL;
float2 *g_sptr = NULL;

// simulation parameters
const float g = 9.81f;              // gravitational constant
const float A = 1e-7f;              // wave scale factor
const float patchSize = 100;        // patch size
float windSpeed = 100.0f;
float windDir = CUDART_PI_F/3.0f;
float dirDepend = 0.07f;

StopWatchInterface *timer = NULL;
float animTime = 0.0f;
float prevTime = 0.0f;
```

```
float animationRate = -0.001f;

// Auto-Verification Code
const int frameCheckNumber = 4;
int fpsCount = 0;        // FPS count for averaging
int fpsLimit = 1;        // FPS limit for sampling
unsigned int frameCount = 0;
unsigned int g_TotalErrors = 0;

////////////////////////////////////////////////////////////////////////////////
// kernels
//#include <oceanFFT_kernel.cu>

extern "C"
void cudaGenerateSpectrumKernel(float2 *d_h0,
                                float2 *d_ht,
                                unsigned int in_width,
                                unsigned int out_width,
                                unsigned int out_height,
                                float animTime,
                                float patchSize);

extern "C"
void cudaUpdateHeightmapKernel(float  *d_heightMap,
                               float2 *d_ht,
                               unsigned int width,
                               unsigned int height);

extern "C"
void cudaCalculateSlopeKernel(float *h, float2 *slopeOut,
                              unsigned int width, unsigned int height);

////////////////////////////////////////////////////////////////////////////////
// forward declarations
void runAutoTest(int argc, char **argv);
void runGraphicsTest(int argc, char **argv);

// GL functionality
bool initGL(int *argc, char **argv);
void createVBO(GLuint *vbo, int size);
void deleteVBO(GLuint *vbo);
void createMeshIndexBuffer(GLuint *id, int w, int h);
void createMeshPositionVBO(GLuint *id, int w, int h);
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName);

// rendering callbacks
void display();
void keyboard(unsigned char key, int x, int y);
void mouse(int button, int state, int x, int y);
void motion(int x, int y);
void reshape(int w, int h);
void timerEvent(int value);

// Cuda functionality
void runCuda();
void runCudaTest(bool bHasDouble, char *exec_path);
void generate_h0(float2 *h0);
void generateFftInput();
```

```
////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    // check for command line arguments
    if (checkCmdLineFlag(argc, (const char **)argv, "qatest"))
    {
        animate       = false;
        fpsLimit = frameCheckNumber;
        runAutoTest(argc, argv);
    }
    else
    {
        printf("[%s]\n\n"
               "Left mouse button          - rotate\n"
               "Middle mouse button        - pan\n"
               "Right mouse button         - zoom\n"
               "'w' key                    - toggle wireframe\n", sSDKsample);

        runGraphicsTest(argc, argv);
    }

    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}

////////////////////////////////////////////////////////////////////////////
//! Run test
////////////////////////////////////////////////////////////////////////////
void runAutoTest(int argc, char **argv)
{
    printf("%s Starting...\n\n", argv[0]);

    // Cuda init
    int dev = findCudaDevice(argc, (const char **)argv);

    cudaDeviceProp deviceProp;
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Compute capability %d.%d\n", deviceProp.major, deviceProp.minor);
    int version = deviceProp.major*10 + deviceProp.minor;
    g_hasDouble = (version >= 13);

    // create FFT plan
    checkCudaErrors(cufftPlan2d(&fftPlan, meshSize, meshSize, CUFFT_C2C));

    // allocate memory
    int spectrumSize = spectrumW*spectrumH*sizeof(float2);
    checkCudaErrors(cudaMalloc((void **)&d_h0, spectrumSize));
    h_h0 = (float2 *) malloc(spectrumSize);
    generate_h0(h_h0);
    checkCudaErrors(cudaMemcpy(d_h0, h_h0, spectrumSize, cudaMemcpyHostToDevice));

    int outputSize =  meshSize*meshSize*sizeof(float2);
    checkCudaErrors(cudaMalloc((void **)&d_ht, outputSize));
    checkCudaErrors(cudaMalloc((void **)&d_slope, outputSize));
```

```
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);
    prevTime = sdkGetTimerValue(&timer);

    runCudaTest(g_hasDouble, argv[0]);

    checkCudaErrors(cudaFree(d_ht));
    checkCudaErrors(cudaFree(d_slope));
    checkCudaErrors(cudaFree(d_h0));
    checkCudaErrors(cufftDestroy(fftPlan));
    free(h_h0);

    cudaDeviceReset();

    exit(g_TotalErrors==0 ? EXIT_SUCCESS : EXIT_FAILURE);
}

////////////////////////////////////////////////////////////////////////////////
//! Run test
////////////////////////////////////////////////////////////////////////////////
void runGraphicsTest(int argc, char **argv)
{
    printf("[%s] ", sSDKsample);
    printf("\n");

    if (checkCmdLineFlag(argc, (const char **)argv, "device"))
    {
        printf("[%s]\n", argv[0]);
        printf("   Does not explicitly support -device=n in OpenGL mode\n");
        printf("   To use -device=n, the sample must be running w/o OpenGL\n\n");
        printf(" > %s -device=n -qatest\n", argv[0]);
        printf("exiting...\n");
        exit(EXIT_SUCCESS);
    }

    // First initialize OpenGL context, so we can properly set the GL for CUDA.
    // This is necessary in order to achieve optimal performance with OpenGL/CUDA
interop.
    if (false == initGL(&argc, argv))
    {
        cudaDeviceReset();
        return;
    }

    findCudaGLDevice(argc, (const char **)argv);

    // create FFT plan
    checkCudaErrors(cufftPlan2d(&fftPlan, meshSize, meshSize, CUFFT_C2C));

    // allocate memory
    int spectrumSize = spectrumW*spectrumH*sizeof(float2);
    checkCudaErrors(cudaMalloc((void **)&d_h0, spectrumSize));
    h_h0 = (float2 *) malloc(spectrumSize);
    generate_h0(h_h0);
    checkCudaErrors(cudaMemcpy(d_h0, h_h0, spectrumSize, cudaMemcpyHostToDevice));

    int outputSize =  meshSize*meshSize*sizeof(float2);
    checkCudaErrors(cudaMalloc((void **)&d_ht, outputSize));
```

```
    checkCudaErrors(cudaMalloc((void **)&d_slope, outputSize));

    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);
    prevTime = sdkGetTimerValue(&timer);

    // create vertex buffers and register with CUDA
    createVBO(&heightVertexBuffer, meshSize*meshSize*sizeof(float));
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_heightVB_resource,
heightVertexBuffer, cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&slopeVertexBuffer, outputSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_slopeVB_resource,
slopeVertexBuffer, cudaGraphicsMapFlagsWriteDiscard));

    // create vertex and index buffer for mesh
    createMeshPositionVBO(&posVertexBuffer, meshSize, meshSize);
    createMeshIndexBuffer(&indexBuffer, meshSize, meshSize);

    runCuda();

    // register callbacks
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);

    // start rendering mainloop
    glutMainLoop();
    cudaDeviceReset();
}

float urand()
{
    return rand() / (float)RAND_MAX;
}

// Generates Gaussian random number with mean 0 and standard deviation 1.
float gauss()
{
    float u1 = urand();
    float u2 = urand();

    if (u1 < 1e-6f)
    {
        u1 = 1e-6f;
    }

    return sqrtf(-2 * logf(u1)) * cosf(2*CUDART_PI_F * u2);
}

// Phillips spectrum
// (Kx, Ky) - normalized wave vector
// Vdir - wind angle in radians
// V - wind speed
// A - constant
```

```
float phillips(float Kx, float Ky, float Vdir, float V, float A, float dir_depend)
{
    float k_squared = Kx * Kx + Ky * Ky;

    if (k_squared == 0.0f)
    {
        return 0.0f;
    }

    // largest possible wave from constant wind of velocity v
    float L = V * V / g;

    float k_x = Kx / sqrtf(k_squared);
    float k_y = Ky / sqrtf(k_squared);
    float w_dot_k = k_x * cosf(Vdir) + k_y * sinf(Vdir);

    float phillips = A * expf(-1.0f / (k_squared * L * L)) / (k_squared *
k_squared) * w_dot_k * w_dot_k;

    // filter out waves moving opposite to wind
    if (w_dot_k < 0.0f)
    {
        phillips *= dir_depend;
    }

    // damp out waves with very small length w << l
    float w = L / 10000;
    //phillips *= expf(-k_squared * w * w);

    return phillips;
}

// Generate base heightfield in frequency space
void generate_h0(float2 *h0)
{
    for (unsigned int y = 0; y<=meshSize; y++)
    {
        for (unsigned int x = 0; x<=meshSize; x++)
        {
            float kx = (-(int)meshSize / 2.0f + x) * (2.0f * CUDART_PI_F /
patchSize);
            float ky = (-(int)meshSize / 2.0f + y) * (2.0f * CUDART_PI_F /
patchSize);

            float P = sqrtf(phillips(kx, ky, windDir, windSpeed, A, dirDepend));

            if (kx == 0.0f && ky == 0.0f)
            {
                P = 0.0f;
            }

            //float Er = urand()*2.0f-1.0f;
            //float Ei = urand()*2.0f-1.0f;
            float Er = gauss();
            float Ei = gauss();

            float h0_re = Er * P * CUDART_SQRT_HALF_F;
            float h0_im = Ei * P * CUDART_SQRT_HALF_F;
```

```
            int i = y*spectrumW+x;
            h0[i].x = h0_re;
            h0[i].y = h0_im;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Run the Cuda kernels
////////////////////////////////////////////////////////////////////////////////
void runCuda()
{
    size_t num_bytes;

    // generate wave spectrum in frequency domain
    cudaGenerateSpectrumKernel(d_h0, d_ht, spectrumW, meshSize, meshSize,
animTime, patchSize);

    // execute inverse FFT to convert to spatial domain
    checkCudaErrors(cufftExecC2C(fftPlan, d_ht, d_ht, CUFFT_INVERSE));

    // update heightmap values in vertex buffer
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_heightVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_hptr,
&num_bytes, cuda_heightVB_resource));

    cudaUpdateHeightmapKernel(g_hptr, d_ht, meshSize, meshSize);

    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_heightVB_resource, 0));

    // calculate slope for shading
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_slopeVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_sptr,
&num_bytes, cuda_slopeVB_resource));

    cudaCalculateSlopeKernel(g_hptr, g_sptr, meshSize, meshSize);

    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_slopeVB_resource, 0));
}

const char *sSpatialDomain[] =
{
    "ref_spatialDomain.bin",
    "ref_spatialDomain_sm13.bin",
    NULL
};

const char *sSlopeShading[] =
{
    "ref_slopeShading.bin",
    "ref_slopeShading_sm13.bin",
    NULL
};

void runCudaTest(bool bHasDouble, char *exec_path)
{
    checkCudaErrors(cudaMalloc((void **)&g_hptr,
```

```
meshSize*meshSize*sizeof(float)));
    checkCudaErrors(cudaMalloc((void **)&g_sptr,
meshSize*meshSize*sizeof(float2)));

    // generate wave spectrum in frequency domain
    cudaGenerateSpectrumKernel(d_h0, d_ht, spectrumW, meshSize, meshSize,
animTime, patchSize);

    // execute inverse FFT to convert to spatial domain
    checkCudaErrors(cufftExecC2C(fftPlan, d_ht, d_ht, CUFFT_INVERSE));

    // update heightmap values
    cudaUpdateHeightmapKernel(g_hptr, d_ht, meshSize, meshSize);

    {
        float *hptr = (float *)malloc(meshSize*meshSize*sizeof(float));
        cudaMemcpy((void *)hptr, (void *)g_hptr, meshSize*meshSize*sizeof(float),
cudaMemcpyDeviceToHost);
        sdkDumpBin((void *)hptr, meshSize*meshSize*sizeof(float),
"spatialDomain.bin");

        if (!sdkCompareBin2BinFloat("spatialDomain.bin",
sSpatialDomain[bHasDouble], meshSize*meshSize*sizeof(float),
                                    MAX_EPSILON, THRESHOLD, exec_path))
        {
            g_TotalErrors++;
        }

        free(hptr);
    }

    // calculate slope for shading
    cudaCalculateSlopeKernel(g_hptr, g_sptr, meshSize, meshSize);

    {
        float2 *sptr = (float2 *)malloc(meshSize*meshSize*sizeof(float2));
        cudaMemcpy((void *)sptr, (void *)g_sptr, meshSize*meshSize*sizeof(float2),
cudaMemcpyDeviceToHost);
        sdkDumpBin(sptr, meshSize*meshSize*sizeof(float2), "slopeShading.bin");

        if (!sdkCompareBin2BinFloat("slopeShading.bin", sSlopeShading[bHasDouble],
meshSize*meshSize*sizeof(float2),
                                    MAX_EPSILON, THRESHOLD, exec_path))
        {
            g_TotalErrors++;
        }

        free(sptr);
    }

    checkCudaErrors(cudaFree(g_hptr));
    checkCudaErrors(cudaFree(g_sptr));
}


//void computeFPS()
//{
//    frameCount++;
```

```
//    fpsCount++;
//
//    if (fpsCount == fpsLimit) {
//        fpsCount = 0;
//    }
//}

////////////////////////////////////////////////////////////////////////////////
//! Display callback
////////////////////////////////////////////////////////////////////////////////
void display()
{
    // run CUDA kernel to generate vertex positions
    if (animate)
    {
        runCuda();
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(translateX, translateY, translateZ);
    glRotatef(rotateX, 1.0, 0.0, 0.0);
    glRotatef(rotateY, 0.0, 1.0, 0.0);

    // render from the vbo
    glBindBuffer(GL_ARRAY_BUFFER, posVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, heightVertexBuffer);
    glClientActiveTexture(GL_TEXTURE0);
    glTexCoordPointer(1, GL_FLOAT, 0, 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glBindBuffer(GL_ARRAY_BUFFER, slopeVertexBuffer);
    glClientActiveTexture(GL_TEXTURE1);
    glTexCoordPointer(2, GL_FLOAT, 0, 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(shaderProg);

    // Set default uniform variables parameters for the vertex shader
    GLuint uniHeightScale, uniChopiness, uniSize;

    uniHeightScale = glGetUniformLocation(shaderProg, "heightScale");
    glUniform1f(uniHeightScale, 0.5f);

    uniChopiness   = glGetUniformLocation(shaderProg, "chopiness");
    glUniform1f(uniChopiness, 1.0f);

    uniSize        = glGetUniformLocation(shaderProg, "size");
    glUniform2f(uniSize, (float) meshSize, (float) meshSize);

    // Set default uniform variables parameters for the pixel shader
    GLuint uniDeepColor, uniShallowColor, uniSkyColor, uniLightDir;
```

```
    uniDeepColor = glGetUniformLocation(shaderProg, "deepColor");
    glUniform4f(uniDeepColor, 0.0f, 0.1f, 0.4f, 1.0f);

    uniShallowColor = glGetUniformLocation(shaderProg, "shallowColor");
    glUniform4f(uniShallowColor, 0.1f, 0.3f, 0.3f, 1.0f);

    uniSkyColor = glGetUniformLocation(shaderProg, "skyColor");
    glUniform4f(uniSkyColor, 1.0f, 1.0f, 1.0f, 1.0f);

    uniLightDir = glGetUniformLocation(shaderProg, "lightDir");
    glUniform3f(uniLightDir, 0.0f, 1.0f, 0.0f);
    // end of uniform settings

    glColor3f(1.0, 1.0, 1.0);

    if (drawPoints)
    {
        glDrawArrays(GL_POINTS, 0, meshSize * meshSize);
    }
    else
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);

        glPolygonMode(GL_FRONT_AND_BACK, wireFrame ? GL_LINE : GL_FILL);
        glDrawElements(GL_TRIANGLE_STRIP, ((meshSize*2)+2)*(meshSize-1),
GL_UNSIGNED_INT, 0);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

    glDisableClientState(GL_VERTEX_ARRAY);
    glClientActiveTexture(GL_TEXTURE0);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glClientActiveTexture(GL_TEXTURE1);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(0);

    glutSwapBuffers();

    //computeFPS();
}

void timerEvent(int value)
{
    float time = sdkGetTimerValue(&timer);

    if (animate)
    {
        animTime += (time - prevTime) * animationRate;
    }

    glutPostRedisplay();
    prevTime = time;

    glutTimerFunc(REFRESH_DELAY, timerEvent, 0);
```

```
}

void cleanup()
{
    sdkDeleteTimer(&timer);
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_heightVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_slopeVB_resource));

    deleteVBO(&posVertexBuffer);
    deleteVBO(&heightVertexBuffer);
    deleteVBO(&slopeVertexBuffer);

    checkCudaErrors(cudaFree(d_h0));
    checkCudaErrors(cudaFree(d_slope));
    free(h_h0);
    cufftDestroy(fftPlan);
}

////////////////////////////////////////////////////////////////////////////////
//! Keyboard events handler
////////////////////////////////////////////////////////////////////////////////
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    switch (key)
    {
        case (27) :
            cleanup();
            exit(EXIT_SUCCESS);

        case 'w':
            wireFrame = !wireFrame;
            break;

        case 'p':
            drawPoints = !drawPoints;
            break;

        case ' ':
            animate = !animate;
            break;
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Mouse event handlers
////////////////////////////////////////////////////////////////////////////////
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
    {
        mouseButtons |= 1<<button;
    }
    else if (state == GLUT_UP)
    {
        mouseButtons = 0;
    }

    mouseOldX = x;
```

```
    mouseOldY = y;
    glutPostRedisplay();
}

void motion(int x, int y)
{
    float dx, dy;
    dx = (float)(x - mouseOldX);
    dy = (float)(y - mouseOldY);

    if (mouseButtons == 1)
    {
        rotateX += dy * 0.2f;
        rotateY += dx * 0.2f;
    }
    else if (mouseButtons == 2)
    {
        translateX += dx * 0.01f;
        translateY -= dy * 0.01f;
    }
    else if (mouseButtons == 4)
    {
        translateZ += dy * 0.01f;
    }

    mouseOldX = x;
    mouseOldY = y;
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double) w / (double) h, 0.1, 10.0);

    windowW = w;
    windowH = h;
}

////////////////////////////////////////////////////////////////////////////////
//! Initialize GL
////////////////////////////////////////////////////////////////////////////////
bool initGL(int *argc, char **argv)
{
    // Create GL context
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(windowW, windowH);
    glutCreateWindow("CUDA FFT Ocean Simulation");

    vertShaderPath = sdkFindFilePath("ocean.vert", argv[0]);
    fragShaderPath = sdkFindFilePath("ocean.frag", argv[0]);

    if (vertShaderPath == NULL || fragShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL vertex and fragment
```

```
shaders!\n");
        exit(EXIT_FAILURE);
    }

    // initialize necessary OpenGL extensions
    glewInit();

    if (! glewIsSupported("GL_VERSION_2_0 "
                          ))
    {
        fprintf(stderr, "ERROR: Support for necessary OpenGL extensions
missing.");
        fflush(stderr);
        return false;
    }

    if (!glewIsSupported("GL_VERSION_1_5 GL_ARB_vertex_buffer_object
GL_ARB_pixel_buffer_object"))
    {
        fprintf(stderr, "Error: failed to get minimal extensions for demo\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "  OpenGL version 1.5\n");
        fprintf(stderr, "  GL_ARB_vertex_buffer_object\n");
        fprintf(stderr, "  GL_ARB_pixel_buffer_object\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    // default initialization
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);

    // load shader
    shaderProg = loadGLSLProgram(vertShaderPath, fragShaderPath);

    SDK_CHECK_ERROR_GL();
    return true;
}

////////////////////////////////////////////////////////////////////////////////
//! Create VBO
////////////////////////////////////////////////////////////////////////////////
void createVBO(GLuint *vbo, int size)
{
    // create buffer object
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    SDK_CHECK_ERROR_GL();
}

////////////////////////////////////////////////////////////////////////////////
//! Delete VBO
////////////////////////////////////////////////////////////////////////////////
void deleteVBO(GLuint *vbo)
{
```

```
    glDeleteBuffers(1, vbo);
    *vbo = 0;
}

// create index buffer for rendering quad mesh
void createMeshIndexBuffer(GLuint *id, int w, int h)
{
    int size = ((w*2)+2)*(h-1)*sizeof(GLuint);

    // create index buffer
    glGenBuffersARB(1, id);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *id);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);

    // fill with indices for rendering mesh as triangle strips
    GLuint *indices = (GLuint *) glMapBuffer(GL_ELEMENT_ARRAY_BUFFER,
GL_WRITE_ONLY);

    if (!indices)
    {
        return;
    }

    for (int y=0; y<h-1; y++)
    {
        for (int x=0; x<w; x++)
        {
            *indices++ = y*w+x;
            *indices++ = (y+1)*w+x;
        }

        // start new strip with degenerate triangle
        *indices++ = (y+1)*w+(w-1);
        *indices++ = (y+1)*w;
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

// create fixed vertex buffer to store mesh vertices
void createMeshPositionVBO(GLuint *id, int w, int h)
{
    createVBO(id, w*h*4*sizeof(float));

    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }

    for (int y=0; y<h; y++)
    {
        for (int x=0; x<w; x++)
        {
            float u = x / (float)(w-1);
```

```
            float v = y / (float)(h-1);
            *pos++ = u*2.0f-1.0f;
            *pos++ = 0.0f;
            *pos++ = v*2.0f-1.0f;
            *pos++ = 1.0f;
        }
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// Attach shader to a program
int attachShader(GLuint prg, GLenum type, const char *name)
{
    GLuint shader;
    FILE *fp;
    int size, compiled;
    char *src;

    fp = fopen(name, "rb");

    if (!fp)
    {
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    src = (char *)malloc(size);

    fseek(fp, 0, SEEK_SET);
    fread(src, sizeof(char), size, fp);
    fclose(fp);

    shader = glCreateShader(type);
    glShaderSource(shader, 1, (const char **)&src, (const GLint *)&size);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS, (GLint *)&compiled);

    if (!compiled)
    {
        char log[2048];
        int len;

        glGetShaderInfoLog(shader, 2048, (GLsizei *)&len, log);
        printf("Info log: %s\n", log);
        glDeleteShader(shader);
        return 0;
    }

    free(src);

    glAttachShader(prg, shader);
    glDeleteShader(shader);

    return 1;
}
```

```
// Create shader program from vertex shader and fragment shader files
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName)
{
    GLint linked;
    GLuint program;

    program = glCreateProgram();

    if (!attachShader(program, GL_VERTEX_SHADER, vertFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach vertex shader from file %s\n",
vertFileName);
        return 0;
    }

    if (!attachShader(program, GL_FRAGMENT_SHADER, fragFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach fragment shader from file %s\n",
fragFileName);
        return 0;
    }

    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);

    if (!linked)
    {
        glDeleteProgram(program);
        char temp[256];
        glGetProgramInfoLog(program, 256, 0, temp);
        fprintf(stderr, "Failed to link program: %s\n", temp);
        return 0;
    }

    return program;
}
```

```
/*
 * Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

////////////////////////////////////////////////////////////////////////////////
#include <cufft.h>
#include <math_constants.h>

//Round a / b to nearest higher integer value
int cuda_iDivUp(int a, int b)
{
    return (a + (b - 1)) / b;
}


// complex math functions
__device__
float2 conjugate(float2 arg)
{
    return make_float2(arg.x, -arg.y);
}

__device__
float2 complex_exp(float arg)
{
    return make_float2(cosf(arg), sinf(arg));
}

__device__
float2 complex_add(float2 a, float2 b)
{
    return make_float2(a.x + b.x, a.y + b.y);
}

__device__
float2 complex_mult(float2 ab, float2 cd)
{
    return make_float2(ab.x * cd.x - ab.y * cd.y, ab.x * cd.y + ab.y * cd.x);
}

// generate wave heightfield at time t based on initial heightfield and dispersion
relationship
__global__ void generateSpectrumKernel(float2 *h0,
                                       float2 *ht,
                                       unsigned int in_width,
                                       unsigned int out_width,
                                       unsigned int out_height,
                                       float t,
                                       float patchSize)
{
```

```
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int in_index = y*in_width+x;
    unsigned int in_mindex = (out_height - y)*in_width + (out_width - x); //
mirrored
    unsigned int out_index = y*out_width+x;

    // calculate wave vector
    float2 k;
    k.x = (-(int)out_width / 2.0f + x) * (2.0f * CUDART_PI_F / patchSize);
    k.y = (-(int)out_width / 2.0f + y) * (2.0f * CUDART_PI_F / patchSize);

    // calculate dispersion w(k)
    float k_len = sqrtf(k.x*k.x + k.y*k.y);
    float w = sqrtf(9.81f * k_len);

    if ((x < out_width) && (y < out_height))
    {
        float2 h0_k = h0[in_index];
        float2 h0_mk = h0[in_mindex];

        // output frequency-space complex values
        ht[out_index] = complex_add(complex_mult(h0_k, complex_exp(w * t)),
complex_mult(conjugate(h0_mk), complex_exp(-w * t)));
        //ht[out_index] = h0_k;
    }
}

// update height map values based on output of FFT
__global__ void updateHeightmapKernel(float  *heightMap,
                                      float2 *ht,
                                      unsigned int width)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int i = y*width+x;

    // cos(pi * (m1 + m2))
    float sign_correction = ((x + y) & 0x01) ? -1.0f : 1.0f;

    heightMap[i] = ht[i].x * sign_correction;
}

// generate slope by partial differences in spatial domain
__global__ void calculateSlopeKernel(float *h, float2 *slopeOut, unsigned int
width, unsigned int height)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int i = y*width+x;

    float2 slope = make_float2(0.0f, 0.0f);

    if ((x > 0) && (y > 0) && (x < width-1) && (y < height-1))
    {
        slope.x = h[i+1] - h[i-1];
        slope.y = h[i+width] - h[i-width];
    }
```

```
    slopeOut[i] = slope;
}

// wrapper functions
extern "C"
void cudaGenerateSpectrumKernel(float2 *d_h0,
                                float2 *d_ht,
                                unsigned int in_width,
                                unsigned int out_width,
                                unsigned int out_height,
                                float animTime,
                                float patchSize)
{
    dim3 block(8, 8, 1);
    dim3 grid(cuda_iDivUp(out_width, block.x), cuda_iDivUp(out_height, block.y),
1);
    generateSpectrumKernel<<<grid, block>>>(d_h0, d_ht, in_width, out_width,
out_height, animTime, patchSize);
}

extern "C"
void cudaUpdateHeightmapKernel(float  *d_heightMap,
                               float2 *d_ht,
                               unsigned int width,
                               unsigned int height)
{
    dim3 block(8, 8, 1);
    dim3 grid(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    updateHeightmapKernel<<<grid, block>>>(d_heightMap, d_ht, width);
}

extern "C"
void cudaCalculateSlopeKernel(float *hptr, float2 *slopeOut,
                              unsigned int width, unsigned int height)
{
    dim3 block(8, 8, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateSlopeKernel<<<grid2, block>>>(hptr, slopeOut, width, height);
}
```

Índice alfabético

Bibliografía

[1] NVIDIA Obtenido de: `https://developer.nvidia.com/`, Consultado el: 8 de julio de 2014.

[2] Victor Podlozhnyuk y Mark Harris,*Monte Carlo Option Pricing*, Julio 2012, Obtenido de: NVIDIA_CUDA-5.5_Samples.

[3] Victor Podlozhnyuk, *Image Convolution with CUDA*, Julio 2012, Obtenido de: `http://developer.download.nvidia.com/assets/cuda/files/convolutionSeparable.pdf`, Consultado el: 8 de julio de 2014.

[4] Alan Reiner, *CUDA Image Convolution*, Obtenido de: `https://github.com/etotheipi/Basic-CUDA-Convolution`, Consultado el: 8 de julio de 2014.