



Super Computing and Distributed Computing Camp

Manizales, Colombia Ago. - 2014



Super Computación y
Cálculo Científico UIS



Introducción

Este documento constituye el componente práctico con manos de la Aprenda CUDA en un tutorial de la tarde disponible aquí: <http://sc3.uis.edu.co/>

Se supone que usted tiene acceso a una computadora con una GPU NVIDIA CUDA y sistema operativo Linux. Para sistemas Windows, sólo tendrán que adaptarse configuraciones o el uso de clientes terminales para trabajar sobre una maquina remota que si emplee Linux. etc (por favor, consulte la documentación de NVIDIA).

En el primer ejercicio buscamos ayudarle a empezar con su primer código **CUDA**. El segundo ejercicio se utiliza, como punto de partida una aplicación **CUDA** que “funciona mal” y que va a utilizar varias técnicas para optimizar el código y mejorar el rendimiento de la **GPU**.

Para obtener los archivos de la plantilla:

```
wget https://dl.dropboxusercontent.com/u/49956056/SC3\_2014-IntroCuda.zip  
wget
```

(o descargar mediante el navegador web de esta dirección)

```
cd SC3_2014-IntroCuda
```

Para cada uno de los ejercicios existe una subcarpeta **src**, que contiene las plantillas de ejercicio.



CENTRO DE BIOINFORMÁTICA
Y BIOLOGÍA COMPUTACIONAL

Glosario de Apoyo

- host (CPU).
- Device o dispositivo (GPU)
- Array (Vector)
- Kernel (nucleo o función CUDA)
- NUM_BLOCKS
- THREADS_PER_BLOCK
- SMs (Streaming Multiprocessors)
- CUDA C (lenguaje nativo para programación paralela sobre GPUs NVIDIA)

Primeros pasos con CUDA

Introducción

En el siguiente ejercicio presentaremos el ejemplo de cálculo del número pi por el método Montecarlo, los archivos necesarios se encuentran **SC-Camp/CUDA/pi**.

Antes de adentrar en código haremos un recorrido rápido acerca del método Montecarlo para el cálculo de Pi, a fin de contextualizar.

Apéndice A: El método Montecarlo.

La base del método Montecarlo es el uso de números aleatorios para el cálculo de pi. se parte del hecho de tener el área de un círculo de radio 1 y centrado en el origen, adicionalmente este círculo está encerrado por un cuadrado como se puede verse en la figura.

Aquí se presenta la expresión que nos permite obtener la relación entre el área del cuadrado que circunscribe el círculo de radio 1. Nuestro interés en esta relación es obtener un valor proporcional al número pi dividido por 4.

Ahora para obtener esta relación por medio del método Montecarlo, utilizaremos números aleatorios normalmente distribuido en el intervalo (0 -1), para verificar cuantos números caen dentro de la circunferencia y cuantos fuera de ella.

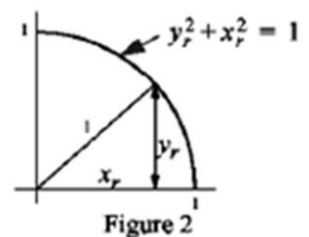
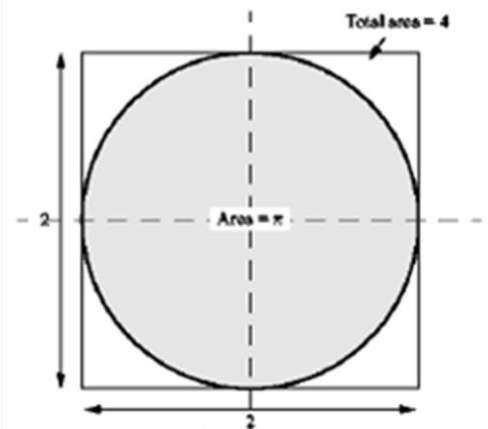
ANOTACIÓN: teniendo en cuenta, que al emplear números en el intervalo (0-1) nos hace referencia tan solo a puntos ubicados sobre el primer cuadrante.

Por tanto verificaremos que los puntos esparcidos cumplan la condición

$$\sqrt{x_p^2 + y_p^2} \leq 1 \quad \therefore y_p \leq \sqrt{1 - x_p^2}$$

Adicionalmente, el método Montecarlo predica que si se emplean mayor cantidad de puntos aleatorios, para la ejecución del método, la relación y por ende **precisión** del cálculo del número pi **augmenta**.

$$\frac{\text{Area Círculo}}{\text{Area Cuadrado}} = \frac{\pi(1)^2}{2 * 2}$$



Apéndice B: El código Fuente.

En este apartado haremos un repaso del código fuente para comprender mejor el manejo de CUDA para la solución de un ejemplo numérico aplicado y conocer las sentencias de la *librería Curand*, para la generación de números aleatorios sobre la GPU.

Kernel `gpu_monte_carlo`

La siguiente porción de código es un kernel, lo más similar a una función de programa hecho en lenguaje C convencional.

A diferencia de las funciones declaradas en lenguaje C, los kernels son ejecutados de forma paralela sobre los núcleos de la GPU, conforme al entorno de ejecución que se defina al momento de invocar al kernel en el programa principal `main()`. Esto se verá más adelante en el apartado *Lanzamiento de un kernel CUDA (ejecución de trabajo paralelo sobre la GPU)*.

```
// Creando el kernel de Montecarlo para el lenguaje CUDA
__global__ void gpu_monte_carlo(float *estimate, curandState *states) {
    unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int points_in_circle = 0;
    float x, y;
    curand_init(1234, tid, 0, &states[tid]); // Inicializacion de la libreria CURAND

    for(int i = 0; i < TRIALS_PER_THREAD; i++) {
        x = curand_uniform (&states[tid]);
        y = curand_uniform (&states[tid]);
        points_in_circle += (x*x + y*y <= 1.0f); // verificar si los puntos estan dentro de r=1
    }
    estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD; //aproximando pi
}
```

Por otra parte, los argumentos presentados en este kernel corresponden:

- **Estimate:** es un vector de float que recoge el acumulado de cuantos puntos han caído dentro del círculo.
- **States:** hace referencia a un tipo de variable `CurandState` propia de la librería `Curand`, que identifica en que hilo de la tarjeta se está efectuando la generación de un numero aleatorio, para definir una semilla que hará que el numero aleatorio en cada hilo no se repita para ningún otro. (Esto es muy importante para los métodos matemáticos basados en probabilidad)
- **Curand_uniform:** es una función implementada dentro de la librería `curand` encargada de generar números aleatoriamente distribuidos entre 0 y 1.

funcion `monte_carlo` en lenguaje C

La siguiente porción de código presenta la función que realiza el cálculo de PI de forma secuencial.

```
// funcion de Montecarlo que corre sobre CPU, en lenguaje C
float host_monte_carlo(long trials) {
    float x, y;
    long points_in_circle;
    for(long i = 0; i < trials; i++) {
        x = rand() / (float) RAND_MAX;
        y = rand() / (float) RAND_MAX;
        points_in_circle += (x*x + y*y <= 1.0f);
    }
    return 4.0f * points_in_circle / trials;
}
```

funcion main (Programa Principal)

```
// Variables necesarias para el programa
clock_t start, stop; // Variables para Cronometros en C
float host[BLOCKS * THREADS]; // un Vector de float en el Host o CPU
float *dev; // el vector de float para la GPU
curandState *devStates; // una variable de estados para la librería Curand

start = clock(); // Inicia captura de tiempo en C
```

Reserva de espacio de memoria sobre la GPU

```
// reservando memoria en GPU
cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
cudaMalloc (void **)&devStates, THREADS * BLOCKS * sizeof(curandState) );
```

Lanzamiento de un kernel CUDA (ejecución de trabajo paralelo sobre la GPU)

En este caso, tenemos un kernel el cual recibe tanto argumentos como parámetros de entorno para su ejecución; precisamente estos últimos son los que los diferencian de las funciones típicas del lenguaje C.

```
// lanzamiento de trabajo sobre GPU (invocacion de kernel CUDA)
gpu_monte_carlo<<<BLOCKS, THREADS>>>(dev, devStates);
```

De la porción de código anterior, tenemos los valores pasados entre los símbolos <<<BLOCKS, THREADS>>>, estos corchetes angulares reciben los parámetros para la configuración del entorno sobre el Hardware, de forma más clara; permiten definir cuantos grupos de hilos se van a emplear para la ejecución de un kernel, de forma que el API CUDA C mediante su compilador pueda decidir cuantos elementos de hardware he decidido utilizar para ejecutar esta porción sobre la GPU.

En nuestro caso el kernel `gpu_monte_carlo`, estamos lanzando una cantidad `BLOCKS` de grupos de tamaño `THREADS`; de forma que podremos tener un total de `BLOCKS*THREADS` unidades de cálculo (núcleos de procesamiento) para ejecutar nuestro kernel.

ANOTACIÓN: es importante mencionar que todo lo anterior es posible bajo 2 condiciones.

- **La independencia de datos** en nuestro programa o aplicación (principio de la programación paralela).
- **Las limitaciones físicas del hardware**, contar con la suficiente memoria y núcleos para efectuar mi calculo, de forma que estos límites no se excedan para obtener, los resultados esperados.

Transferencias de memoria en la API CUDA

Esta sentencia `cudaMemcpy`, es la que permite realizar copiado de variables o datos desde la GPU a CPU o viceversa. Para el caso específico de nuestro ejemplo recordemos que, la generación de los números aleatorios y la ponderación sobre cada hilo son realizados sobre la GPU, por tanto requerimos de traer dichos resultados numéricos a la CPU para poder imprimirlos en pantalla, realizar algún tipo de operación o tomar decisiones con base en estos datos.

```
// copiado hacia host de los resultados "Parciales" obtenidos en GPU
cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float), cudaMemcpyDeviceToHost);
```

Reducción (suma acumulada)

Esta suma acumulada es requerida en nuestro caso, y ha sido implementada en forma serial para simplicidad del ejemplo.

Es necesario realizar la suma de los resultados a lo largo de todos los nucleos de la GPU, en los cuales cada uno de ellos realizo un numero de iteraciones del método Montecarlo, por tanto necesitamos acumular los resultados de cada uno de ellos y luego “promediar”, para obtener el valor del numero pi.

```
/* Efectuando una operacion de reduccion para la obtencion del numero pi */
float pi_gpu;
for(int i = 0; i < BLOCKS * THREADS; i++) {
    pi_gpu += host[i];
}
pi_gpu /= (BLOCKS * THREADS); // realizando una división flotante
// para obtener el numero pi
stop = clock(); // Capturando tiempo desde lenguaje C
```

Apéndice C: A COMPILAR Y CORRER CUDA =)

Para compilar, ejecute el comando:

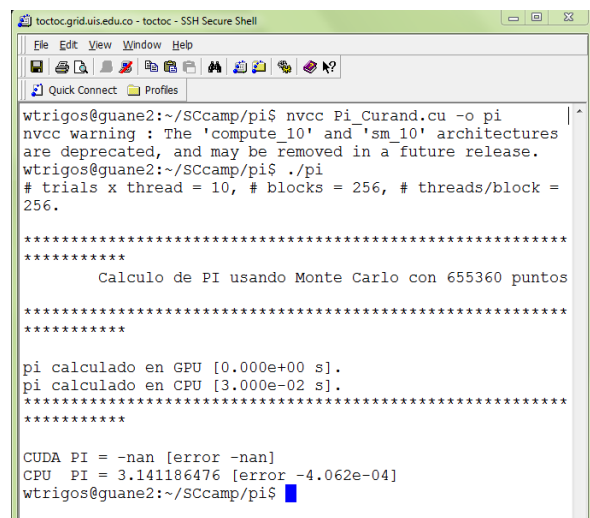
```
nvcc Pi_Curand.cu -o pi
```

Para ejecutar:

```
./pi
```

Elaborado: William J. Trigos G.

Revisado: Carlos J. Barrios, SC3



```
toctoc.grid.uis.edu.co - toctoc - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
wtrigos@guane2:~/SCcamp/pi$ nvcc Pi_Curand.cu -o pi
nvcc warning : The 'compute_10' and 'sm_10' architectures
are deprecated, and may be removed in a future release.
wtrigos@guane2:~/SCcamp/pi$ ./pi
# trials x thread = 10, # blocks = 256, # threads/block =
256.
*****
          Calculo de PI usando Monte Carlo con 655360 puntos
*****
pi calculado en GPU [0.000e+00 s].
pi calculado en CPU [3.000e-02 s].
*****
CUDA PI = -nan [error -nan]
CPU PI = 3.141186476 [error -4.062e-04]
wtrigos@guane2:~/SCcamp/pi$
```

Como pueden apreciar, la ejecución satisfactoria presenta el tiempo empleado por la GPU y CPU, para el cálculo de Pi; presentando el tiempo empleado, el valor de Pi y el nivel de error.

Apéndice D: Mejorando mi kernel y/o emplear librerías (relación Costo beneficio)

Por motivos de extensión, el código de este numeral no será explicado en este documento. Sin embargo, si ud desea ojear el código para interpretar lo que se hace en este puede hacerlo libremente; de lo contrario, por favor afiance conocimientos mientras el tutor llega a esta parte de la introducción y realiza la explicación del código en cada archivo.

El ejemplo mejorado del cálculo del número pi por el método Montecarlo, está compuesto por los archivos encuentran en la carpeta **SC-Camp/CUDA/pi/pi_mejorado**, en esa carpeta podrá encontrar 2 archivos, 1 de ellos contiene la implementación de los kernels para efectuar el cálculo sobre GPU.

El archivo **kernel.cu**, contiene el programa principal encargado de la ejecución de código en host y la invocación de los kernels CUDA para la ejecución sobre GPU.

El archivo **pi.cu**, contiene las implementación de los kernels CUDA para la ejecución sobre GPU. Cabe resaltar que una de las implementaciones utiliza una librería de alto nivel para programación en CUDA llamada **Thrust**, y adicionalmente se presentaran conceptos más avanzados en cuda.

COMO APRENDER MAS!!

Si considera necesario, pida instrucciones o asesoría o puede consultar la Guía de programación **CUDA C** y los documentos de referencia manuales disponibles

<http://developer.nvidia.com/nvidia-gpu-computing-documentation>

" hoy tienes el privilegio de aprender un poco, tu meta podría ser aprender un poco más; tu limitante solo es tu voluntad y dedicación"