PRÁCTICA AC – FASE IV

Programación GPGPU con CUDA

Realizada por **Pavel Razgovorov** (pr18@alu.ua.es) con N.I.E. Y0888160-Y del **grupo 1** de prácticas (lunes 13:00-15:00)

Índice

Introducción

Nos referimos a la programación GPGPU como un modelo de programación paralela de altas prestaciones mediante el uso de tarjetas gráficas de propósito general.

Historia

Los aceleradores gráficos siempre se habían concebido como un componente para el ordenador destinado al tratamiento de los gráficos 2D y 3D, pero no fue hasta 2002 que, gracias a las mejoras tecnológicas introducidas en aquella época, el Doctor en Informática Mark Harris acuñó el término GPGPU y lanzó una web [Vía] como recurso de información sobre todo aquello relacionado con este ámbito.

Uno de los primeros éxitos en conseguir paralelizar un algoritmo hasta tal punto de superar a un CPU fue el de la factorización LU de una matriz en el año 2005 [Vía].

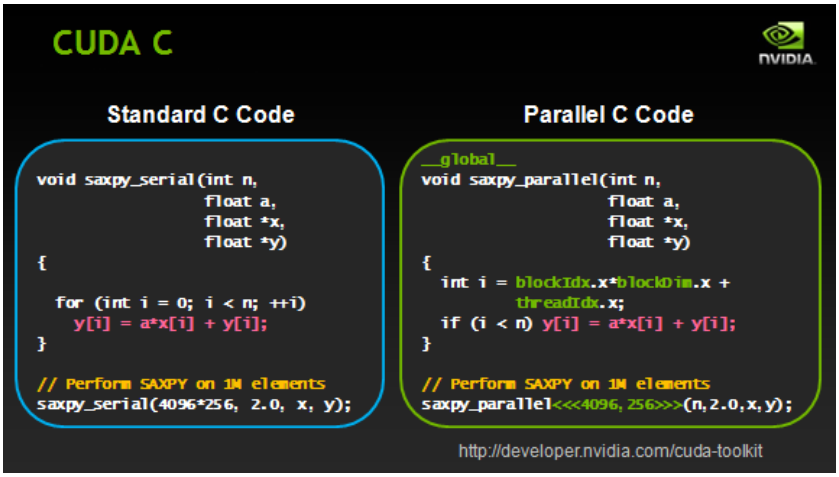
Características e implementaciones

Para conseguir sacar provecho al procesamiento paralelo mediante GPU, debes de sacarle provecho a la arquitectura que presenta cualquier típica tarjeta gráfica: muchos núcleos de procesamiento muy sencillos. Esto es, procesos que hagan un uso intensivo de cálculo aritmético, pero también unas estructuras de datos muy sencillas con el menor acceso posible a memoria aleatoria (los vectores, matrices y mallas 3D son los más favorecidos en estos aspectos).

Podemos encontrar interesantes API's para este tipo de programación, como puede ser DirectX de Microsoft, presente en casi el 100% de los juegos para PC; OpenGL, un equivalente a la primera, pero de código abierto; Metal, para los sistemas de Apple; Vulkan, antecesor de OpenGL que recientemente ha sido publicada y que estará disponible incluso en dispositivos Android de forma nativa [Vía]; STREAM para las GPU's de AMD [Vía] o CUDA de NVIDIA, que es la que utilizaremos en esta práctica.

Cómo funciona CUDA

La ventaja de las GPU es su facilidad para crear y destruir hilos de ejecución. Por tanto, el procedimiento a seguir es lanzar una misma rutina a ejecutar, pero cientos de miles de veces (un montón de hilos de ejecución separados en un buen par de bloques que operan sobre un mismo conjunto de datos). Para evitar problemas de concurrencia, usualmente se identifica cada proceso para que así cada uno haga una cosa diferente sin interferir a los demás.



Suma de vectores

Se nos propone analizar un código en CUDA de ejemplo cuya función es sumar dos vectores y responder a un par de cuestiones:

Analiza el kernel que se va a ejecutar en la GPU

```
__global__ void vecadd(float* C, const float* A, const float* B)
{
    const int i = (threadIdx.x + blockIdx.x * blockDim.x) *
COMPUTE_N_ELEMENTS_PER_THREAD;
      if( i < VECTOR_ELEMENTS )
      {
            for(int j=0; j < COMPUTE_N_ELEMENTS_PER_THREAD; j++)
            {
                  C[i+j] = A[i+j] + B[i+j];
            }
      }
}
```

Es una simple función que recibe tres arrays de flotante, dos fuentes y uno destino, con los cuales se realiza una suma. La cuestión está en que, a diferencia de cómo se haría usualmente, en esta función no estamos utilizando un bucle for para recorrer los arrays (sí utilizamos uno para sumar más de un dato por hilo, pero eso lo explicaremos en los siguientes apartados); aquí lo que se propone es calcular el índice del array como el identificador del hilo (y del bloque al que pertenece el mismo) para que así cada uno de los que entre tenga un índice del array distinto y puedan operar por separado.

Observa en el código la estructura de funciones que son invocadas hasta completar una llamada a una función CUDA

a. Reserva de memoria

Para reservar memoria en el programa se utiliza la función malloc de siempre; sin embargo, para trabajar con la GPU, también se debe de reservar la misma memoria, pero en el propio device utilizando la función cudaMalloc, muy parecida a la original de C (el único cambio apreciable es que lo que la primera devolvía como resultado ahora lo recibe como argumento por referencia)

```
cudaMalloc((void **) &C_d, VECTOR_ELEMENTS * sizeof(float));
```

*El cast (void **) que se le hace al vector C_d no es necesario*

b. Transferencia de datos CPU → GPU

También se hace una copia de datos entre un vector y otro a través de la función cudaMemcpy, similiar a la memcpy de C (en esta nueva se le añade un parámetro que funciona como FLAG para establecer su *modus operandi*)

```
cudaMemcpy(A_d, A, VECTOR_ELEMENTS * sizeof(float), cudaMemcpyHostToDevice);
```

```
cudaMemcpy(B_d, B, VECTOR_ELEMENTS * sizeof(float), cudaMemcpyHostToDevice);
```

c. Invocación kernel

Antes de invocar al kernel, se debe de establecer cuál será el tamaño de la cuadrícula (espacio de hilos que se utilizarán para la ejecución) y el tamaño de cada bloque.

```
dim3 block(512);
dim3 grid((VECTOR_ELEMENTS+(block.x*COMPUTE_N_ELEMENTS_PER_THREAD-
1))/(block.x*COMPUTE_N_ELEMENTS_PER_THREAD));
```

Podemos encontrar información útil (en inglés) en este documento [Vía]

Una vez declarados los recursos que se van a utilizar para la ejecución del procedimiento, se realiza una llamada a éste. La única diferencia entre llamar una función normal y una CUDA son los parámetros grid y block auxiliares que recibe:

```
vecadd<<<grid,block>>>(C_d,A_d,B_d);
```

Después de la llamada al kernel, es conveniente llamar a la función cudaThreadSyncronize() (actualmente obsoleta y sustituida por cudaDeviceSyncronize()) para esperar a que todos los hilos de ejecución finalicen, ya que es muy posible que alguno termine antes que otro; sin embargo, debemos de esperar a que termine el último para que el algoritmo se dé por finalizado (puede que esto no siempre sea necesario y podamos aprovechar el tiempo y no ponerlo).

d. Transferencia de datos GPU → CPU

La función que se utiliza es la misma que en el apartado b, pero ahora el FLAG utilizado es el "cudaMemcpyDeviceToHost" que quiere decir que la copia se realiza desde el dispositivo (la GPU) hasta el anfitrión (la CPU)

```
cudaMemcpy(C, C_d, VECTOR_ELEMENTS * sizeof(float), cudaMemcpyDeviceToHost);
```

¿Cuántas veces se ejecuta la función kernel?

Podemos saber cuántos hilos de ejecución se lanzan al kernel multiplicando las tres componentes (x, y, z) de block y grid. En este caso, block tiene (512, 1, 1) y grid ((VECTOR_ELEMENTS+(block.x*COMPUTE_N_ELEMENTS_PER_THREAD-1))/ (block.x*COMPUTE_N_ELEMENTS_PER_THREAD), 1, 1) que, sustituyendo valores, queda en 512 * 58594 = 30000128 hilos.

Teniendo en cuenta que, con los valores por defecto, que son 30.000.000 elementos y se calcula 1 elemento por hilo, nos está creando 128 hilos que sobran (aunque, ante estas dimensiones, el error resulta imperceptible).

Observa las constantes VECTOR_ELEMENTS y COMPUTE_N_ELEMENTS_PER_THREAD. ¿Qué función tienen en el código? Prueba a modificar sus valores y ejecutar el código proporcionado observando que ocurre. Razona correctamente los casos en los que la ejecución falle por algún motivo.

Son dos constantes en el programa que funcionan como parámetro del tamaño del problema a resolver y carga de trabajo por hilo, ya que la primera constante sirve para especificar el tamaño del array a sumar y la segunda para saber cuántos elementos se suma cada vez que se llama al kernel.

Si probamos a cambiar sus valores, podemos observar que, a cuanto más pequeño sea el problema y a cuantas más sumas por hilo, menor es la ganancia obtenida respecto al tiempo de ejecución CPU. En otras palabras, utilizar CUDA en algoritmos de este tipo sólo es útil si el tamaño del problema es muy considerable y si la carga de cada hilo es mínima.

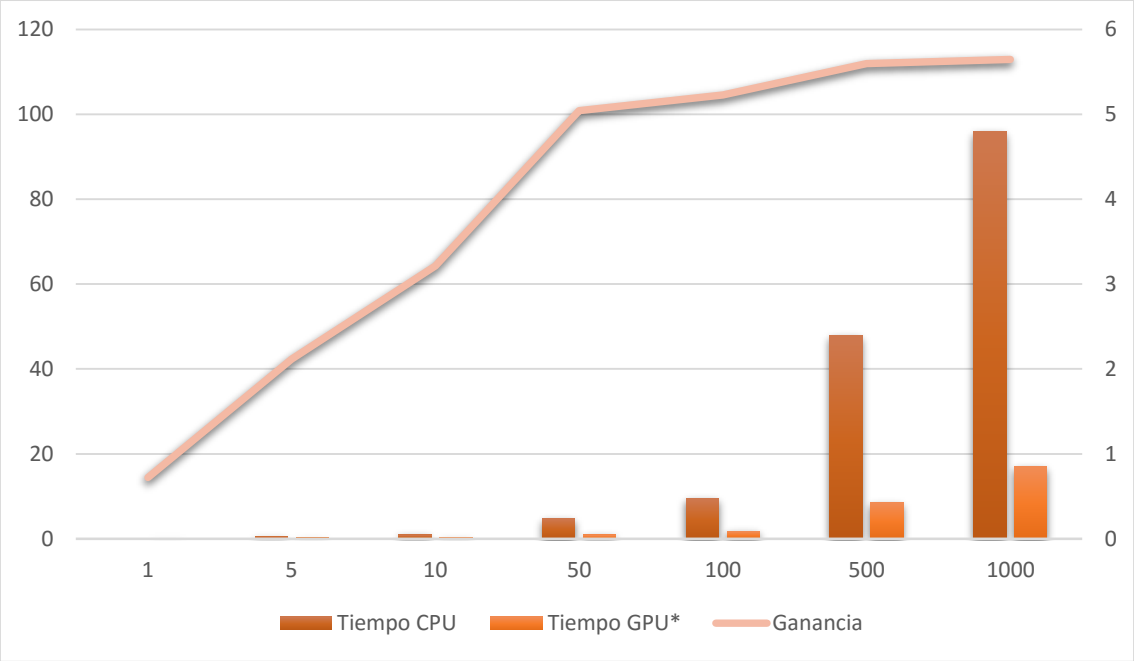
Si el tamaño del array supera los 33553920 números, parece ser que deja de ser posible acceder a los elementos de la memoria ya que la suma no es efectuada (el resultado que arroja para cada valor del array es 0, distinto del 3 esperado). Sin embargo, el kernel de CUDA no detecta ningún tipo de fallo (la función ERROR_CHECK no muestra ningún mensaje). Además, si el tamaño del array es incluso más grande, no permite reservar tanta memoria directamente. La cantidad de sumas por hilo a realizar, sin embargo, no afecta a la ejecución (a no ser que le des un valor mayor que el del tamaño del array, claro).

Tras ejecutar el código habrás observado que la GPU obtiene un tiempo de ejecución mayor que la CPU cuando las transferencias de datos se incluyen en el cómputo. Esto se debe a que las transferencias son el principal cuello de botella a la hora de portar un código a la GPU. La implementación paralela de la suma de 2 vectores no acarrea suficiente cómputo para ocultar las latencias producidas por las transferencias de memoria. Si la operación suma de vectores se realizase N veces, ¿se ocultaría entonces la latencia producida por las transferencias de datos entre GPU y CPU?

Para poner a prueba lo que presuponen en el enunciado de la pregunta, realizaremos un par de pruebas ejecutando el kernel repetidamente: declararemos una constante EXECUTIONS que trabajará como límite de un bucle for que envuelve la llamada al kernel, le iremos dando una serie de valores y apuntaremos sus resultados (teniendo en cuenta las transferencias).

EXECUTIONS	Tiempo CPU	Tiempo GPU*	Ganancia
1	0,097245	0,134409	0,72350
5	0,496155	0,234504	2,11577
10	0,974010	0,303221	3,21221
50	4,831894	0,958229	5,04253
100	9,527544	1,822464	5,22784
500	47,849100	8,549768	5,59654
1000	95,816624	16,969085	5,64654

El resultado obtenido es el siguiente:



Como podemos observar, a partir de las 50 ejecuciones (50 * 30M = 1500M de sumas) la ganancia comienza a ser considerable. Lo interesante es que, al tratarse de un problema cuadrático (porque sumamos n veces un array de tamaño m), el incremento de tamaño del problema no afecta tanto a la GPU como le puede afectar a la CPU; la subida no es tan acusada, y podemos observarlo en la última de las pruebas, en la que hay una diferencia de unos 80 segundos.

En definitiva, utilizar la GPU como herramienta de cómputo sólo es útil si la cantidad de datos a procesar es sumamente grande (no como en SSE, que con cualquier tamaño de problema siempre iba a ser mucho más rápido), pero la ganancia que se consigue es muy sustancial.

Calificaciones de la práctica grupal

Esta vez no hemos alcanzado el objetivo de la práctica. Todos (menos César y Sánchez, que entre que no tenían NVIDIA y/o tampoco son buenos programadores… Pero han ayudado en la memoria) hemos hecho todo lo posible, pero, aunque el código sea capaz de copiar la memoria a la GPU y la llamada en el kernell se produce sin errores aparentes, las operaciones realizadas no coinciden con el resultado esperado. Sin embargo, creo que todos han hecho lo posible y se han esforzado bastante, en especial Eddie, que ha hecho trabajado algo más que yo (suele ser al revés).