



ugr

Universidad
de Granada

ARQUITECTURA Y COMPUTACIÓN DE ALTAS PRESTACIONES
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 4

SUMA DE VECTORES CON CUDA

Autor

Vladislav Nikolov Vasilev

Rama

Ingeniería de Computadores



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2019-2020

Índice

1. Introducción	2
2. Especificaciones de la GPU	2
3. Versión secuencial	4
4. Versión paralela con CUDA	4
5. Experimentación y resultados	6
6. Conclusiones	10
Referencias	11

1. Introducción

En esta práctica se pide implementar dos versiones de un mismo programa: una primera versión secuencial y una versión en **CUDA**. El objetivo del programa es, dados dos vectores **A** y **B**, ambos de tamaño n , obtener un vector de salida **C** del mismo tamaño en el que $\forall \mathbf{C}_i \in \mathbf{C}$, su valor venga determinado por la expresión:

$$\mathbf{C}_i = \left(\left(\frac{\log(5 \cdot \mathbf{A}_i \cdot 100 \cdot \mathbf{B}_i) + 7 \cdot \mathbf{A}_i}{0.33} \right)^3 \right)^7 \quad (1)$$

Además, se ha especificado que los vectores de entrada **A** y **B** se tienen que replicar un número de veces, de forma que su longitud se vea incrementada. En este caso, se ha decidido **que se van a tener 5 copias de los vectores originales**, es decir, que los vectores van a tener una longitud de $5n$, donde n es el tamaño original especificado en los archivos proporcionados.

Para cada uno de los nueve problemas proporcionados se van a tomar medidas de tiempo de lo que se tarda en hacer todas las operaciones con los dos programas. Con ellas se representará la evolución del tiempo de ejecución en cada caso en función del tamaño del problema y se hará un estudio de la ganancia.

Adicionalmente, se pide que se obtengan las especificaciones de la GPU que se está utilizando, de manera que se tenga más información sobre esta.

2. Especificaciones de la GPU

Primeramente vamos a ver cuáles son las especificaciones de la GPU que vamos a utilizar. Para ello se ha creado un programa que puede encontrarse en el archivo `deviceProperties.cu`, el cuál muestra la siguiente información:

- Nombre del dispositivo.
- Número de SMs (procesadores).
- Número de SPs por SM (cores por procesador).
- Número total de SPs.
- Memoria total de la gráfica en MB.
- Memoria compartida por SM en KB.

- Memoria compartida por bloque en KB.
- Número máximo de hebras por bloque.
- Número máximo de hebras por cada dimensión.

El código que muestra esta información es el siguiente:

```
1 int n_devices;
2
3 cudaGetDeviceCount(&n_devices);
4
5 int i;
6
7 for (i = 0; i < n_devices; i++)
8 {
9     cudaDeviceProp prop;
10    cudaGetDeviceProperties(&prop, i);
11    int numSPs = getSPcores(prop);
12
13    printf("Device number: %d\n", i);
14    printf(" Device name: %s\n", prop.name);
15    printf(" Number of SMs: %d\n", prop.multiProcessorCount);
16    printf(" Number of SPs per SM: %d\n", numSPs);
17    printf(" Total Number of SPs: %d\n",
18           numSPs * prop.multiProcessorCount);
19    printf(" Total Available Global Memory Size (MB): %zu\n",
20           prop.totalGlobalMem / (1<<20));
21    printf(" Shared Memory Per SM (KB): %zu\n",
22           prop.sharedMemPerMultiprocessor / (1<<10));
23    printf(" Shared Memory Per Block (KB): %zu\n",
24           prop.sharedMemPerBlock / (1<<10));
25    printf(" Max. Threads Per Block: %d\n", prop.maxThreadsPerBlock);
26    ;
27    printf(" Max. Threads Per Dimension: x: %d, y: %d, z: %d\n",
28           prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.
29           maxThreadsDim[2]);
30 }
```

La función `getSPcores` se ha extraído de una respuesta a una pregunta en *StackOverflow* [1]. En caso de tener más de una tarjeta gráfica (que no es el caso), se mostraría la información para cada una de ellas. En este caso se ha obtenido la siguiente información:

```
Device number: 0
Device name: GeForce GTX 1050
Number of SMs: 5
Number of SPs per SM: 128
Total Number of SPs: 640
Total Available Global Memory Size MB: 4040
```

Shared Memory Per SM KB: 96
Shared Memory Per Block KB: 48
Max. Threads Per Block: 1024
Max. Threads Per Dimension: x: 1024, y: 1024, z: 64

3. Versión secuencial

Una vez que hemos visto las especificaciones de nuestra gráfica, vamos a ver cómo sería una primera versión secuencial del programa.

La versión secuencial carga los ficheros de datos que se han especificado como parámetros, carga los valores, los copia un número de veces que se determina como parámetro y realiza la operación, obteniendo el vector **C** de salida. La función que realiza el cálculo que se puede ver en la ecuación (1) es la siguiente:

```
1 void add_vectors(float *A, float *B, float *C, int N)
2 {
3     int i;
4
5     for (i = 0; i < N; i++)
6     {
7         C[i] = pow(pow(log(5*A[i]*100*B[i] + 7*A[i]) / 0.33, 3), 7);
8     }
9 }
```

La función recibe como parámetro los dos vectores de entrada, el de salida y el tamaño de los vectores, y con un bucle recorre los elementos de **A** y **B** y calcula el correspondiente elemento de **C**.

El resto del código se encuentra en el archivo `vectorAdd.c` y se puede consultar de ser necesario. Como en general son operaciones sencillas, no se van a explicar ni a incluir aquí, ya que no se considera que estén relacionadas con la asignatura (lectura de ficheros, reserva de memoria, etc.).

4. Versión paralela con CUDA

Una vez implementada y probada la versión secuencial se ha procedido a desarrollar la versión paralela. El código de esta versión puede encontrarse en el archivo `vectorAdd.cu`, pero vamos a comentar aquí algunas de las partes más importantes.

Lo primero es la función que aplica la operación de la ecuación (1) sobre los elementos de los vectores **A** y **B**. En este caso se ha declarado dicha función como un *kernel* de CUDA, y su implementación puede verse a continuación:

```
1 __global__ void addVectorsKernel(float *d_A, float *d_B, float *d_C,
2     int N)
3 {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5
6     if (i < N)
7     {
8         d_C[i] = pow(pow(log(5*d_A[i]*100*d_B[i] + 7*d_A[i]) / 0.33,
9             3), 7);
10    }
11 }
```

Cuando se especifica el *kernel*, `__global__` significa que el *kernel* se llama desde el anfitrión o *host* y se ejecuta en el dispositivo o *device* (es decir, en la tarjeta gráfica). La posición del elemento i -ésimo que se va a calcular viene determinada por el índice del bloque, el tamaño del bloque y el índice de la hebra dentro del bloque. Una vez que se ha determinado la posición es necesario comprobar que dicho valor no excede el tamaño de los vectores (podría darse en el caso en el que en un bloque haya más hebras que elementos restantes del vector, por ejemplo).

Dentro del programa principal también han habido algunos cambios. En el siguiente fragmento de código se puede ver lo más destacable de las modificaciones llevadas a cabo:

```
1 // Allocate CUDA arrays and copy data
2 float *d_A, *d_B, *d_C;
3
4 cudaMalloc(&d_A, N * sizeof(float));
5 cudaMalloc(&d_B, N * sizeof(float));
6 cudaMalloc(&d_C, N * sizeof(float));
7
8 cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
9 cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);
10
11
12 // Set number of threads and blocks
13
14 // Use 1024 threads per block since the device supports it
15 int DIM_BLOCK = 1 << 10;
16 int DIM_GRID = ((N - 1) / DIM_BLOCK) + 1;
17
18
19 // Add vectors and retrieve information from device
20 t1 = omp_get_wtime();
21
22 addVectorsKernel<<<DIM_GRID, DIM_BLOCK>>>(d_A, d_B, d_C, N);
23
24 cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);
25
26 t_time = omp_get_wtime() - t1;
```

Vemos que lo primero que se hace es declarar los arrays que van a estar en el dispositivo y se reserva memoria para ellos. Una vez hecho esto, se copian los vectores `h_A` y `h_B` en las correspondientes variables. Es importante destacar que a la hora de copiar se especifica el parámetro `cudaMemcpyHostToDevice`, el cuál indica que se copie del *host* al dispositivo (la tarjeta gráfica).

Una vez hecho esto, se ha procedido a determinar el tamaño del bloque (`DIM_BLOCK`) y el número de bloques o tamaño del *grid* (`DIM_GRID`). Se ha especificado que el tamaño del bloque sea de 1024 hebras, que es el valor máximo que puede tener, tal y como hemos visto en las especificaciones del dispositivo. Se ha elegido este número porque se ha considerado que es un número suficiente de hebras, además de porque es un múltiplo de 32, que es el tamaño del *warp* o el número de hebras que ejecuta cada `CUDA` core. Por otra parte, el número de bloques se ha escogido según la siguiente expresión:

$$\text{DIM_GRID} = \frac{N - 1}{\text{DIM_BLOCK}} + 1 \quad (2)$$

donde N es el número de elementos del vector y la fracción es una división entera. Al escoger el número de bloques de esta forma nos aseguramos de que siempre hay el número exacto de bloques, aunque en uno de ellos no se usen todas las hebras para hacer cálculos. Si calculásemos el número de bloques como $\frac{N}{\text{DIM_BLOCK}}$ el resultado solo sería correcto si N fuese múltiplo del tamaño del bloque. En el caso de que por ejemplo N fuese menor a dicho tamaño, el resultado sería 0, lo cuál no debería ser posible, ya que siempre tiene que haber al menos un bloque. Si N no fuese un múltiplo entonces al hacer la división se estaría subestimando el número de bloques necesarios, ya que el resultado de la división entera sería menor al número real de bloques necesarios.

Después de eso se ejecuta el *kernel* de la forma que se puede ver en la línea 22. Finalmente se copian los datos del vector que está en el dispositivo al vector que se encuentra en el *host* (línea 24). Es importante destacar que en esta ocasión para transferir los datos del dispositivo al *host* se ha especificado la opción `cudaMemcpyDeviceToHost`.

5. Experimentación y resultados

Una vez que hemos visto cómo se han implementado las dos versiones vamos a hacer la experimentación. Tal y como se ha dicho anteriormente, se van a tener 5 copias de los dos vectores de entrada, de forma que el tamaño original se verá multiplicado por 5. A la hora de compilar los programas no se ha utilizado

optimización, de forma que la comparación sea justa.

Se han medido los tiempos para cada problema y se han ordenado según el tamaño de los vectores de entrada. Hay algunos problemas que tienen el mismo tamaño, de forma que aparecerá el mismo tamaño con distintos tiempos. Se han creado una tabla y una gráfica que muestran los resultados. Una vez hecho esto, se ha calculado la ganancia para un tamaño determinado con la siguiente expresión:

$$S_{tam} = \frac{T_s^{tam}}{T_p^{tam}} \quad (3)$$

A continuación se puede ver la tabla con los tiempos de ejecución para cada caso, además de la correspondiente gráfica:

Tamaño de los vectores	Tiempo secuencial (s)	Tiempo CUDA (s)
500	0.000386	0.000039
500	0.000394	0.000041
5125	0.000961	0.000067
10240	0.001891	0.00011
20480	0.003607	0.000206
25000	0.004517	0.000279
30000	0.005243	0.000301
45000	0.007347	0.000432
45000	0.007437	0.000435
163840	0.026749	0.001523

Cuadro 1: Tiempo de ejecución secuencial y paralelo con CUDA para cada problema.

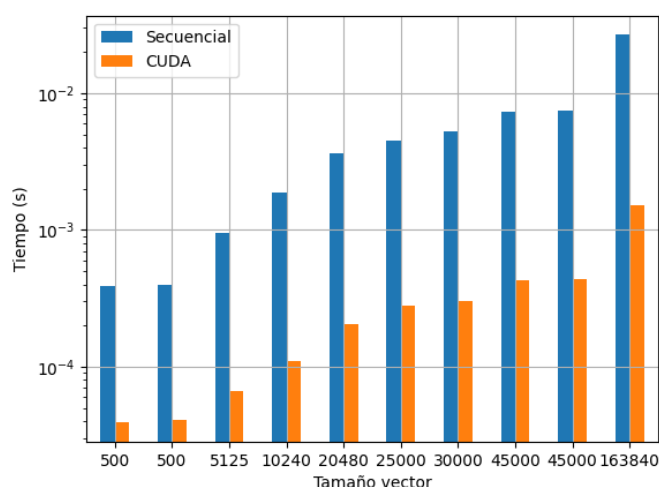


Figura 1: Evolución del tiempo de ejecución del programa secuencial y el programa en CUDA en función del tamaño del vector con escala logarítmica en el eje Y.

Vemos, como es obvio, que los tiempos de la versión secuencial son mucho mayores que los de la versión paralela en CUDA. Esto se debe a que las instrucciones del procesador de la CPU están optimizadas para que tarden muy pocos ciclos, mientras que las de la tarjeta no lo están, pero al ser procesadores de tipo SIMT (*Single Instruction Multiple Threads*), pueden ejecutar una misma instrucción sobre múltiples hebras a la vez, lo cuál permite reducir drásticamente los tiempos de ejecución.

Vemos también que en ambos casos los tiempos de ejecución van aumentando de manera parecida a medida que va aumentando el tamaño del problema, lo cuál es esperable. El ratio al que crecen es más o menos parecido excepto al principio, es decir, cuando se pasa de 500 a 10240 elementos. En este caso, el tiempo secuencial experimenta un crecimiento algo más pronunciado que el paralelo, pero a partir de ahí parece que crecen en más o menos la misma medida, estando, obviamente, siempre el tiempo paralelo varios órdenes de magnitud por debajo del secuencial.

Una vez que hemos analizado los tiempos, vamos a hacer un pequeño estudio de la ganancia. A continuación se ofrece una gráfica con los valores obtenidos según el tamaño del problema de forma ordenada (como en el caso anterior), además de una gráfica que muestra la evolución de la ganancia:

Tamaño de los vectores	Ganancia
500	9.8974359
500	9.6097561
5125	14.34328358
10240	17.19090909
20480	17.50970874
25000	16.18996416
30000	17.41860465
45000	17.00694444
45000	17.09655172
163840	17.56336179

Cuadro 2: Ganancia para cada problema.

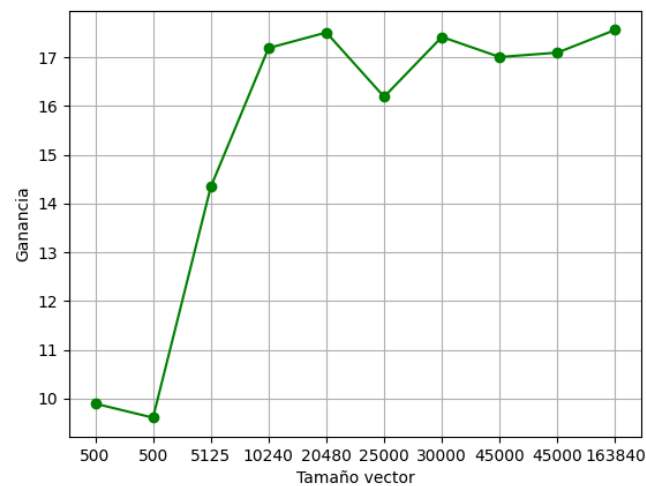


Figura 2: Ganancia obtenida en función del tamaño del vector.

Tal y como podemos ver en la gráfica y en la tabla, la ganancia obtenida es bastante grande y va aumentando a medida que aumenta el tamaño de los vectores de entrada. Empieza con un valor próximo a 10 y va creciendo hasta llegar a 17, donde se mantiene más o menos hasta el final. Solo se produce una bajada cuando los vectores tienen 25000 elementos, pero si analizamos los resultados de la tabla de tiempos 1 vemos que esto se debe a que el incremento de tiempo al pasar de 20480 a 25000 elementos es del 25 % en el caso secuencial, mientras que en el paralelo es de aproximadamente un 35 %. Por tanto, como es lógico, la ganancia obtenida será menor, ya que el tiempo paralelo ha aumentado en mayor mayor medida.

Podemos ver que la ganancia crece mucho entre los 500 y los 10240 elementos. Esto se debe, tal y como comentamos anteriormente, a que los tiempos secuenciales crecen mucho más rápido que los paralelos en estos casos. A partir de ahí, al crecer los dos tiempos a un ritmo más o menos parecido la ganancia se mantiene más o menos igual, excepto en el caso que hemos comentado en el párrafo anterior. Parece que, en general, para tamaños relativamente grandes la ganancia tiene un valor en torno a 17. Si tuviésemos problemas más grandes veríamos que la ganancia se sigue manteniendo en torno a dicho valor, fluctuando ligeramente. Sin embargo, una ganancia de 17 es algo bastante considerable, ya que eso significa que para problemas muy grandes tardaríamos 17 veces menos con el programa paralelo que con el secuencial. Además de eso, si dispusiéramos de una GPU más potente, la ganancia podría ser mucho mayor que la que tenemos aquí.

6. Conclusiones

El uso de GPUs ha permitido acelerar muchísimo los cálculos de grandes cantidades de datos. Esto es gracias a que utilizan procesadores de tipo SIMT, los cuáles permiten ejecutar las mismas instrucciones sobre un gran número de hebras a la vez en vez de ejecutar las instrucciones una detrás de otra de forma secuencial (en el caso de programas secuenciales, que es con lo que hemos estado comparando el rendimiento de nuestra tarjeta gráfica).

Las GPUs permiten obtener una ganancia extremadamente alta y son muy útiles en programas que manejen grandes cantidades de datos, como cálculo muy intensivo, gráficos, *machine learning* y, sobre todo, en *deep learning*, donde juegan un papel esencial en el entrenamiento de las redes profundas.

No obstante, es de menester recordar que las tarjetas gráficas no son la solución a todo. En problemas que requieren mucha sincronización o dependencias entre los datos no es la mejor idea utilizarlas, ya que los tiempos de comunicación son un grandísimo problema, sobre todo en arquitecturas de este tipo. Además, los tiempos de comunicación entre el *host* y el dispositivo pueden llegar a ser bastante elevados, con lo cuál si no se dispone de una cantidad de datos suficiente, la mayor parte del tiempo se va a estar enviando y recibiendo la información, lo cuál negaría completamente cualquier ganancia que permita obtener la tarjeta si una versión secuencial es capaz de hacerlo en el mismo tiempo. Por tanto, hace falta analizar muy bien el problema para determinar qué tipo de paradigma y tecnología son los más adecuados de utilizar.

Referencias

- [1] StackOverflow. *How can I get number of Cores in cuda device?*
[https://stackoverflow.com/questions/32530604/
how-can-i-get-number-of-cores-in-cuda-device](https://stackoverflow.com/questions/32530604/how-can-i-get-number-of-cores-in-cuda-device)