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

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 $\mathbf{A}$ y $\mathbf{B}$, ambos de tamaño n, obtener un vector de salida $\mathbf{C}$ del mismo tamaño en el que $\forall \mathbf{C}_i \in \mathbf{C}$, su valor venga determinado por la expresión:

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

Además, se ha especificado que los vectores de entrada $\mathbf{A}$ y $\mathbf{B}$ 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.

De los diez problemas proporcionados se han descartado los problemas 3 y 9 debido a que tienen los mismos tamaños que otros problemas, más en concreto tienen los mismos tamaños que los problemas 0 y 4, respectivamente. Para cada uno de los ocho problemas restantes 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 infomración 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:

```
int n_devices;

cudaGetDeviceCount(&n_devices);

int i;

for (i = 0; i < n_devices; i++)
{
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);
    int numSPs = getSPcores(prop);

    printf("Device number: %d\n", i);
    printf(" Device name: %s\n", prop.name);
    printf(" Number of SMs: %d\n", prop.multiProcessorCount);
    printf(" Number of SPs per SM: %d\n", numSPs);
    printf(" Total Number of SPs: %d\n",
            numSPs * prop.multiProcessorCount);
    printf(" Total Available Global Memory Size (MB): %zu\n",
            prop.totalGlobalMem / (1<<20));
    printf(" Shared Memory Per SM (KB): %zu\n",
            prop.sharedMemPerMultiprocessor / (1<<10));
    printf(" Shared Memory Per Block (KB): %zu\n",
            prop.sharedMemPerBlock / (1<<10));
    printf(" Max. Threads Per Block: %d\n", prop.maxThreadsPerBlock)
    ;
    printf(" Max. Threads Per Dimension: x: %d, y: %d, z: %d\n",
            prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.
    maxThreadsDim[2]);
}
```

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:

```
__global__ void addVectorsKernel(float *d_A, float *d_B, float *d_C,
     int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
    {
        d_C[i] = pow(pow(log(5*d_A[i]*100*d_B[i] + 7*d_A[i]) / 0.33,
     3), 7);
    }
}
```

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:

```
// Allocate CUDA arrays
float *d_A, *d_B, *d_C;

cudaMalloc(&d_A, N * sizeof(float));
cudaMalloc(&d_B, N * sizeof(float));
cudaMalloc(&d_C, N * sizeof(float));

// Set number of threads and blocks
int DIM_BLOCK = 128;
int DIM_GRID = ((N - 1) / DIM_BLOCK) + 1;

// Add vectors and retrieve information from device
t1 = omp_get_wtime();

// Copy values
cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

addVectorsKernel<<<DIM_GRID, DIM_BLOCK>>>(d_A, d_B, d_C, N);

cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

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 reservada la memoria se procede 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 128 hebras debido a que es un múltiplo de 32, que es el tamaño del *warp* de hebras que es planificado y enviado a ejecutar, además de que es el mismo número que el de SPs por SM, lo cuál permitiría ejecutar un bloque entero a la vez en un SM. Aparte, se han hecho algunas pruebas con distintos tamaños de bloque y este ha dado unos resultados bastante buenos, con lo cuál se considera que, en general, puede funcionar bien para distintos tamaños. Por otra parte, el número de bloques se ha escogido según la siguiente expresión:

$$\texttt{DIM\_GRID} = \frac{N-1}{\texttt{DIM\_BLOCK}} + 1 \tag{2}$$

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 sea el número exacto de bloques, aunque en uno de ellos no se usen todas las hebras para hacer cómputos. Si calculásemos el número de bloques como $\frac{N}{\texttt{DIM\_BLOCK}}$ 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 necearios.

Habiendo hecho esto llega la sección en la que se miden los tiempos, donde se ejecuta el *kernel* declarado anteriormente. Primero 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 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) y se mide la diferencia de tiempos. 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 te-

ner 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 probado y se han ordenado dichas medidas según el tamaño del problema (el tamaño de los vectores de entrada). Se han creado tanto una tabla como una gráfica que muestran los resultados. Una vez hecho esto, se ha calculado la ganancia para un tamaño determinado con la siguiente expresión:

$$S_{tam} = \frac{T_s^{tam}}{T_p^{tam}} \tag{3}$$

Una vez dicho esto, vamos a ver los resultados obtenidos para la versión secuencial y la paralela de forma conjunta. A continuación se puede ver la tabla de tiempos, acompañada de su correspondiente gráfica:

Tamaño de los vectores	**Tiempo secuencial (s)**	**Tiempo CUDA (s)**
500	0.000357	0.000049
5125	0.000952	0.000086
10240	0.001887	0.000139
20480	0.003761	0.00025
25000	0.004338	0.000298
30000	0.005056	0.000361
45000	0.007283	0.000512
163840	0.02738	0.001758

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

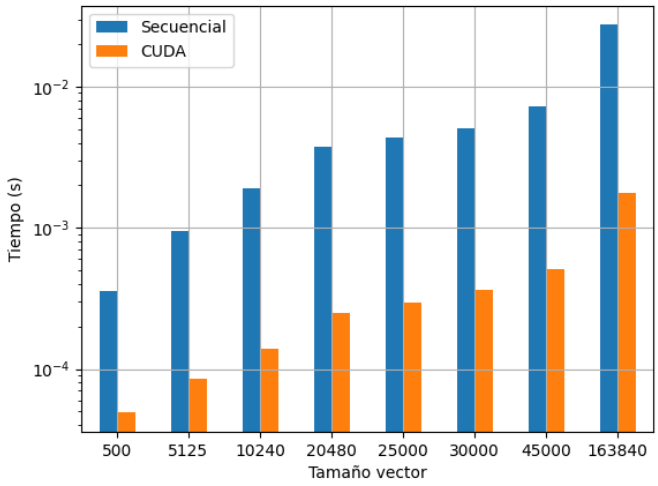

Figura 1: Evolución del tiempo de ejecución del programa secuencial y el programa en `CUDA` en función del tamaño del vector con escala logarítmica en el eje Y.

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 al procesar una mayor cantidad de información a la vez.

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 20480 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.

Al observar los tiempos podemos sacar también una serie de conclusiones. Una de ellas es que, al estar el tiempo de ejecución en la `GPU` por debajo del secuencial para todos los tamaños, podemos afirmar que en todos los casos ha merecido la pena paralelizar, ya que siempre hemos tardado menos tiempo con el programa paralelo. Para problemas más grandes, las diferencias de tiempo también serán bastante notables, ganando siempre la `GPU`.

Por la forma en la que crecen los tiempos podemos concluir también que, para

problemas más pequeños que el más pequeño que aparece reflejado en la gráfica, el tiempo de ejecución del programa secuencial podría ser menor al del paralelo, debido a que no habría transferencia de datos entre la memoria principal y la memoria global del dispositivo, lo cuál es un gran cuello de botella y donde se pasa una gran cantidad de tiempo. Por tanto, eso nos indicaría que para problemas realmente pequeños utiliar la tarjeta gráfica sería como matar moscas a cañonazos.

Finalmente, el hecho de que el tiempo de ejecución del programa paralelo vaya creciendo nos indica que hay muchos bloques que están a la espera de que el planificador los seleccione. Esto se debe a que disponemos de una gráfica con unos pocos SM, los cuáles están ocupados la mayoría del tiempo, y por tanto, la mayoría de bloques estarán a la espera durante bastante tiempo. Si tuviésemos más SMs, quizás los tiempos podrían ser algo menores.

Una vez que hemos analizado los tiempos, vamos a hacer un pequeño estudio de la ganancia. A continuación se ofrece una tabla 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	7.28571429
5125	11.06976744
10240	13.57553957
20480	15.044
25000	14.55704698
30000	14.00554017
45000	14.22460938
163840	15.5745165

Cuadro 2: Ganancia para cada problema.

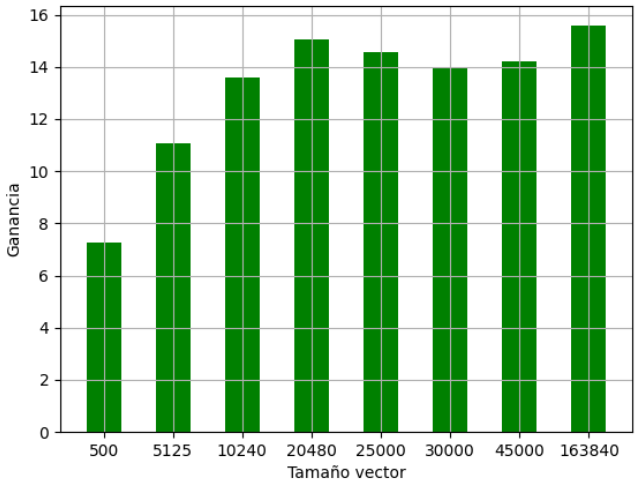

Figura 2: Ganancia obtenida en función del tamaño del vector.

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 algo más grande que 7 y va creciendo hasta llegar a un valor próximo a 15, y a partir de ahí va subiendo y bajando pero manteniéndose próxima a este valor. El motivo por el que se producen bajadas se debe a que en algunos casos la tasa de crecimiento del tiempo secuencial es algo inferior a la del tiempo paralelo, haciendo por tanto que la ganancia sea menor.

Podemos ver que la ganancia crece mucho entre los 500 y los 20480 elementos. Esto se debe a que, tal y como comentamos anteriormente, los tiempos secuenciales crecen mucho más rápido que los paralelos en estos casos. A partir de ahí, la ganancia oscial alrededor de 15, bajando en algunos casos por los motivos comentados anteriormente, y subiendo de nuevo hacia el final. Posiblemente, si probásemos con un problema mucho más grande, la ganancia obtenida podría estar en torno a 15 o aproximarse ligeramente a 16 debido a que los tiempos secuencial y paralelo crecen de forma más o menos pareja para problemas grandes.

Parece que, en general, la ganancia máxima que se puede conseguir con esta tarjeta es de aproximadamente 15, tal y como hemos comentado. Si tuviésemos una tarjeta gráfica más potente y con más SMs, los tiempos posiblemente serían mucho más pequeños, lo cuál a su vez supondría una mayor ganancia. No obstante, tener una ganancia cercana a 15 para problemas grandes no es ninguna tontería. Por ejemplo, un problema muy grande que tardase unos 60 minutos en la versión secuencial podría ser resuelto por la versión paralela en tan solo 4 minutos, lo cuál nos muestra la abismal diferencia que hay entre la versión paralela y la secuencial.

Por tanto, a pesar de que la `GPU` no sea todo lo potente que quisiéramos, nos permite resolver resolver problemas grandes en mucho menos tiempo, y por tanto, merece la pena paralelizarlos mediante `CUDA`.

6. Conclusiones

El uso de `GPUs` ha permitido acelerar muchísimo los cómputos 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 fundamental en el entrenamiento de las redes profundas.

No obstante, es de menester recordar que las tarjetas gráficas no son la solución a todos los problemas. Por ejemplo, si estamos ante un problema que requiera mucha sincronización y/o comunicación, posiblemente lo mejor sería escoger otro paradigma/lenguaje, ya que las tarjetas gráficas están especializadas sobre todo en problemas del tipo SPMD, mientras que para otro tipo de problemas como por ejemplo maestro/esclavo no son tan buenas. 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. Otra cosa que hay que tener en cuenta es que, debido a que todas las hebras ejecutan el mismo código, introducir bloques condicionales en los *kernels* es contraproducente, ya que solo se relantizaría el tiempo de ejecución, haciendo que la ejecución sea casi secuencial.

Por tanto, debemos, antes de nada, entender el tipo de problema al que nos estamos enfrentando, determinar cuál es la mejor metodología a seguir y resolverlo con las herramientas más adecuadas.

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`