Arquitecturas Paralelas

Memoria prácticas CUDA

Juan Álvaro Caravaca Seliva

Ejercicio 1:

Producto de matrices

El popular algoritmo de la multiplicación de matrices cuadradas y densas nos servirá para ilustrar una serie de optimizaciones relacionadas con la descomposición del código en bloques de hilos y la ubicación de los datos en la jerarquía de memoria de la GPU.

El código C de partida es el siguiente:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)  {
    c[i][j] = 0;
    for (k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
  }
```

Conceptos que pondremos en práctica

El producto de matrices suele utilizarse para ilustrar muchas de las posibilidades de los lenguajes de programación paralela y arquitecturas paralelas. Completando fragmentos de código incompleto para el producto de matrices, en este ejercicio se verá:

- Cómo alojar y liberar memoria en el dispositivo.
- Cómo copiar datos desde CPU a GPU.
- Cómo copiar datos desde GPU a CPU.
- Cómo medir tiempos para accesos a memoria y tiempos de ejecución.
- Cómo invocar kernels sobre la GPU.
- Cómo escribir un programa para calcular el producto de dos matrices en GPU.

Cómo acceder a los ficheros fuente

Una vez más, se proporcionan los ficheros del código incompletos, debiendo rellenar el alumno algunas de sus sentencias. Se plantean dos versiones de dificultad creciente, ubicadas en los directorios `lab1.1-matrixmul.1` y `lab1.1-matrixmul.2`. La primera, para alumnos con poca o ninguna experiencia en programación con CUDA, y la segunda para programadores con alguna experiencia. Escoge aquella que consideres más afín a tu nivel de destreza. El código se divide en tres ficheros con la siguiente estructura:

- matrixmul.cu: Contiene el programa main() que aloja la memoria y llama al kernel CUDA.
- matrixmul_kernel.cu: Contiene el kernel CUDA que se lanza desde el programa anterior.
- matrixmul_gold.cpp: Es un programa C que computa secuencialmente el resultado en la CPU. Se utiliza para validar los cómputos realizados en paralelo en GPU.

Implementación

En esta versión inicial del producto de matrices declararemos una matriz de NxN hilos CUDA, en la que el (i,j)-ésimo hilo lee la fila `i` de la matriz A y la columna `j` de la matriz B para calcular el dato `C[i][j]` de la matriz resultante.

ATENCIÓN: Antes de ejecutar cualquier código relacionado con el producto de matrices, debes asegurarte de que el directorio desde el que vayas a lanzar las ejecuciones contiene los ficheros con las matrices de entrada al código. Estos ficheros se llaman matrix_8.bin, matrix_8.gold, matrix_128.bin, matrix_128.gold, matrix_512.bin y matrix_512.gold. Si no los tienes, puedes copiarlos del directorio matrix_data que cuelga del directorio padre donde se encuentran los archivos con tus programas fuente. Algunos alumnos prefieren lanzar las ejecuciones desde el mismo directorio en el que compilan, en cuyo caso al escribir en el shell el nombre del archivo ejecutable, éste debe ir precedido de la ruta donde lo ha generado el Makefile; otros alumnos prefieren cambiar al directorio donde está el ejecutable antes de lanzar la ejecución, en cuyo caso sólo se tiene que escribir en el shell el nombre del archivo precedido de "./" para lanzar las ejecuciones. En el primer caso, los archivos del directorio matrix_data deben colocarse en el directorio donde está el archivo Makefile, y en el segundo, en el directorio donde está el archivo ejecutable.

Modificaciones a realizar sobre el código CUDA

Paso 1:

Edita la función runtTest(. . .) en matrixmul.cu para completar la funcionalidad del producto de matrices en el host. Sigue los comentarios en el código fuente para llevar a cabo esta tarea.

- Primera parte del código:

1. Reserva memoria para matrices de entrada.
2. Copia la matriz de entrada desde la memoria RAM hasta la memoria del dispositivo.
3. Fíjate en el empleo de un temporizador para medir el tiempo necesario para llevar a cabo estas copias. ¿Podrías calcular el ancho de banda efectivo conseguido?

- Segunda parte del código:

1. Configura la ejecución del kernel e invócalo.
2. Fíjate en el temporizador que se utiliza para medir el tiempo de computación. Piensa en el carácter asíncrono de las invocaciones de los kernels CUDA para descubrir por qué se usa la función `cudaThreadSynchronize()`.

- Tercera parte del código:

1. Copia la matriz resultado desde la memoria del dispositivo hasta memoria principal de la CPU.
2. Se utiliza otro temporizador para volver a medir los tiempos de transferencia. ¿Qué ancho de banda efectivo has conseguido esta vez? ¿Es comparable con el ancho de banda calculado anteriormente?

- Cuarta parte del código: Libera la memoria del dispositivo.

Paso 2:

Edita la función `matrixMul(. . . )` en el fichero matrixmul_kernel.cu para completar la funcionalidad del producto de matrices en el dispositivo. Los comentarios en el código te ayudarán en esta tarea.

- Quinta parte del código:

1. Define el índice de salida donde cada hilo debería escribir sus datos.
2. Itera sobre los elementos de los vectores (filas y columnas) para llevar a cabo el producto escalar en cada hilo.
3. Multiplica y acumula elementos sobre la matriz resultado.

Paso 3:

Compila utilizando el comando `make`.

Ejecuta el programa para matrices de dimensiones 8x8, 128x128 y 512x512, cuyas matrices de entrada se encuentran en el directorio `matrix_data` en ficheros identificados por los sufijos 8, 128 y 512, respectivamente. Una vez copies estos ficheros a tu directorio local, sólo tienes que cambiar el parámetro "tamaño" de la línea de ejecución siguiente, sustituyéndolo por 8, 128 o 512, según corresponda):

$> ./matrixmul tamaño

Haciendo 8x8

```
[practicas02@tijola lab1.1-matrixmul.1]$ ./matrixmul 8
Input matrix file name: matrix_8.bin
Setup host side environment and launch kernel:
  Allocate host memory for matrices M and N.
    M: 8 x 8
    N: 8 x 8
  Allocate memory for the result on host side.
  Initialize the input matrices.
Opening the file matrix_8.bin ... succeeded.
Read contents of matrix:
1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11
5 6 7 8 9 10 11 12
6 7 8 9 10 11 12 13
7 8 9 10 11 12 13 14
8 9 10 11 12 13 14 15
  Allocate device memory.
  Copy host memory data to device.
  Allocate device memory for results and clean it.
  Setup kernel execution parameters.
  # of threads in a block: 2 x 2 (4)
  # of blocks in a grid  : 4 x 4 (16)
  Executing the kernel...
  Copy result from device to host.
Transfer time from CPU to GPU:  0.1970 ms.
Transfer CPU to GPU bandwidthInMBs:  2.4791 in MB/s.

GPU computation time:           0.0502 ms.
Transfer time from GPU to CPU:  0.0343 ms.

Transfer GPU to CPU bandwidthInMBs:  7.1103 in MB/s.

Total GPU processing time:      0.2815 ms.

Check results with those computed by CPU.
  Computing reference solution.
  CPU Processing time   : 0.0107 ms.

  CPU checksum: 35456
  GPU checksum: 35456
Opening the file lab1.1-matrixmul.bin ... succeeded.
Read contents of matrix:
204 240 276 312 348 384 420 456
240 284 328 372 416 460 504 548
276 328 380 432 484 536 588 640
312 372 432 492 552 612 672 732
348 416 484 552 620 688 756 824
384 460 536 612 688 764 840 916
420 504 588 672 756 840 924 1008
456 548 640 732 824 916 1008 1100
Opening the file lab1.1-matrixmul.gold ... succeeded.
Read contents of matrix:
204 240 276 312 348 384 420 456
240 284 328 372 416 460 504 548
276 328 380 432 484 536 588 640
312 372 432 492 552 612 672 732
348 416 484 552 620 688 756 824
384 460 536 612 688 764 840 916
420 504 588 672 756 840 924 1008
456 548 640 732 824 916 1008 1100
  Comparing file lab1.1-matrixmul.bin with lab1.1-matrixmul.gold ...
    Check ok? Passed.
[practicas02@tijola lab1.1-matrixmul.1]$
```

Haciendo 128x128

```
[practicas02@tijola lab1.1-matrixmul.1]$ ./matrixmul 128
Input matrix file name: matrix_128.bin
Setup host side environment and launch kernel:
  Allocate host memory for matrices M and N.
    M: 128 x 128
    N: 128 x 128
  Allocate memory for the result on host side.
  Initialize the input matrices.
  Allocate device memory.
  Copy host memory data to device.
  Allocate device memory for results and clean it.
  Setup kernel execution parameters.
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 8 x 8 (64)
  Executing the kernel...
  Copy result from device to host.
Transfer time from CPU to GPU:  0.2450 ms.
Transfer CPU to GPU bandwidthInMBs:  510.2874 in MB/s.

GPU computation time:           0.1311 ms.
Transfer time from GPU to CPU:  0.1044 ms.

Transfer GPU to CPU bandwidthInMBs:  598.5673 in MB/s.

Total GPU processing time:      0.4805 ms.

Check results with those computed by CPU.
  Computing reference solution.
  CPU Processing time   : 20.2019 ms.

  CPU checksum: 3.72229e+10
  GPU checksum: 3.72229e+10
  Comparing file lab1.1-matrixmul.bin with lab1.1-matrixmul.gold ...
    Check ok? Passed.
[practicas02@tijola lab1.1-matrixmul.1]$
```

Haciendo 512x512

```
[practicas02@tijola lab1.1-matrixmul.1]$ ./matrixmul 512
Input matrix file name: matrix_512.bin
Setup host side environment and launch kernel:
  Allocate host memory for matrices M and N.
    M: 512 x 512
    N: 512 x 512
  Allocate memory for the result on host side.
  Initialize the input matrices.
  Allocate device memory.
  Copy host memory data to device.
  Allocate device memory for results and clean it.
  Setup kernel execution parameters.
  # of threads in a block: 16 x 16 (256)
  # of blocks in a grid  : 32 x 32 (1024)
  Executing the kernel...
  Copy result from device to host.
Transfer time from CPU to GPU:  1.1246 ms.
Transfer CPU to GPU bandwidthInMBs:  1778.3973 in MB/s.

GPU computation time:           4.7160 ms.
Transfer time from GPU to CPU:  1.0963 ms.

Transfer GPU to CPU bandwidthInMBs:  912.1424 in MB/s.

Total GPU processing time:      6.9369 ms.

Check results with those computed by CPU.
  Computing reference solution.
  CPU Processing time   : 1343.5641 ms.

  CPU checksum: 3.81165e+13
  GPU checksum: 3.81165e+13
  Comparing file lab1.1-matrixmul.bin with lab1.1-matrixmul.gold ...
    Check ok? Passed.
[practicas02@tijola lab1.1-matrixmul.1]$
```

Apunta los datos relativos a tiempo de ejecución y transferencia de datos en una tabla como la que sigue:

Tamaño de la matriz	CPU->GPU	GPU->CPU	Ejecución	Ratio comparado con 128x128
8x8	0.1970	0.0343	0.2815	0.6170
128x128	0.2450	0.1044	0.4805	1
512x512	1.1246	1.0963	6.9369	11.0447

Conclusión: Como podemos observar, según aumentamos el tamaño de la matriz, se aumentan tanto como los tiempos de trasferencia como los de ejecución exponencialmente. Esto es debido a que hay que transportar más datos. Además, en la matriz de tamaño mayor (512x512), el tiempo de GPU a CPU es mayor que el de CPU a GPU, a la inversa que en las dos matrices anteriores, que ese tiempo es menor.

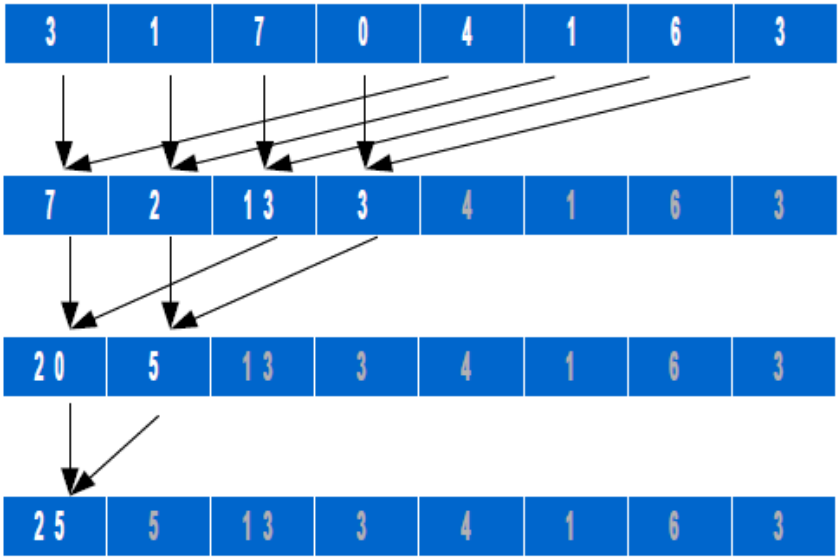
Ejercicio 2:

Suma por reducción

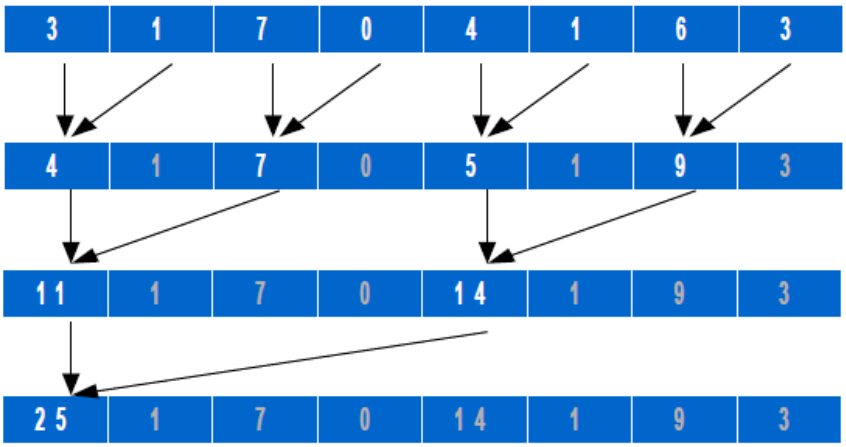
Este código realiza la suma de un vector de N elementos mediante un operador binario de reducción, es decir, en log2(N) pasos. Suponemos un N suficientemente grande como para que requiera la comunicación intra-bloque e inter-bloque entre los hilos CUDA implicados en la computación.

El objetivo de este ejercicio es familiarizarse con un tipo de operaciones muy común en computación científica: las reducciones. Una reducción es una combinación de todos los elementos de un vector en un valor único, utilizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular operaciones en paralelo, calculando el resultado en O(logN) pasos sin incrementar el número de operaciones realizadas. En las siguientes figuras tenemos los esquemas de las operaciones que se han seguido para la elaboración de los códigos:

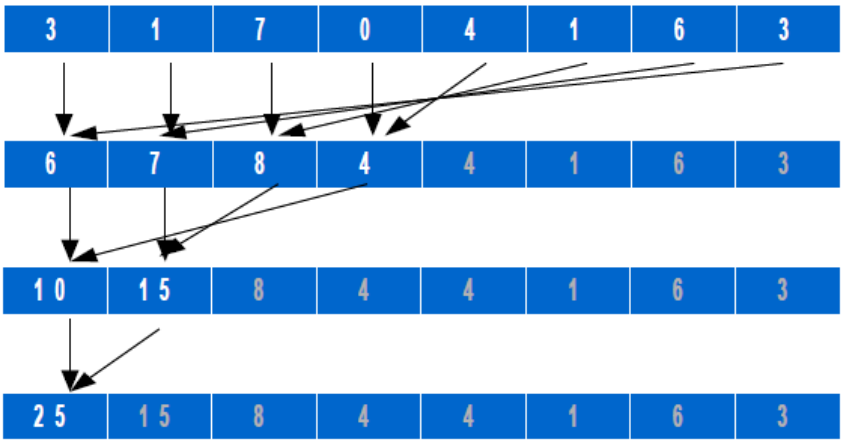
Esquema 1:



Esquema 2



Esquema 3



A partir el Esquema 1, se pueden idear diferentes patrones de acceso a los datos para ir asociando por pares los operandos de cada operación. Esto afecta al rendimiento de la memoria como enseguida veremos, y también a la complejidad de programación, ya que las expresiones que hay que crear para que los hilos generen los índices de acceso a sus datos en cada paso difieren en dificultad.

En el Esquema 2, cada hilo es responsable de la suma de dos nodos adyacentes en una distancia potencia de 2 para cada nivel de la operación. En el Esquema 3, cada hilo es responsable de la suma de dos nodos en distancias que van desde n-1 hasta 1.
Por ejemplo, en el primer nivel de la operación, el hilo número 1 suma los elementos 1 y 8, siendo la distancia entre los dos nodos 8-1 = 7. El hilo número 2 suma los elementos 2 y (8-1). La distancia correspondiente es pues 7-2 = 5. Y las distancias para los nodos sumados por los hilos número 3 y 4 son 6-3 = 2 y 5-4 = 1, respectivamente.

Hay que tener en cuenta el patrón de acceso regular en los dos últimos esquemas. La mayoría de aplicaciones utilizan patrones de este tipo para eliminar divergencias innecesarias o conflictos en accesos a bancos de memoria.

Ahora toca completar el código y ejecutarlo para obtener las soluciones y responder a las cuestiones. Se proponen dos alternativas de dificultad creciente, he escogido completar la segunda alternativa que tiene más líneas incompletas y, está colocada dentro del directorio lab1.2-reduction.2.

Ahora, los pasos a seguir para completar el código y responder las cuestiones:

1. En vector reduction.cu hay que reservar memoria en el dispositivo, copiar los datos de entrada desde la memoria del host hasta la memoria del dispositivo, completar los parámetros de invocación al kernel y copiar los resultados desde la memoria de dispositivo de vuelta a la memoria del host.
2. En vector_reduction_kernel.cu hay que cargar los datos desde la memoria global a la memoria compartida, realizar la reducción sobre los datos en la memoria compartida y almacenar de vuelta los datos en la memoria compartida. En la función reduction(...), están implementados los 3 esquemas anteriores.
3. Desde la consola, compilamos el código con el comando make y lo ejecutamos con ./vector_reduction. La versión sobre GPU se ejecutará siempre primero, seguida de la versión sobre CPU, que sirve como referencia para comprobar el resultado.
   Si el resultado es correcto, se imprime por pantalla el mensaje CORRECTO. En caso contrario, se imprime la frase INCORRECTO. El código se ejecuta sin argumentos, tomando como entrada un vector de datos de 512 elementos cuyos valores son aleatorios y la ejecuciónn se realiza para 256 hilos.

4. Los ejercicios no proporcionan información sobre tiempos de ejecución, as´ı que hay que añadirla extrayéndola directamente del CUDA profiler. Para activarlo, hay que usar una secuencia de comandos en el shell.

```
$> CUDA_PROFILE=1
$> CUDA_PROFILE_CONFIG=./profile_config
$> export CUDA_PROFILE
$> export CUDA_PROFILE_CONFIG
$> ./vector_reduction
```

Por defecto, el fichero de salida para el CUDA profiler es cuda profile.log. Los parámetros más interesantes del CUDA profiler son:

⋄ gputime: muestra los microsegundos que necesita cada kernel CUDA para ser ejecutado.
⋄ l1_shared_bank_conflict: es el número de conflictos en el acceso a los bancos de la memoria compartida en la que ubicamos el vector scratch.

⋄ shared_load y shared_store: se refieren al número de accesos en modo lectura y escritura en memoria compartida, respectivamente.

Compara los valores de l1_shared_bank_conflict y gputime para cada uno de los tres esquemas de reducción implementados. ¿Cuál de ellos es el más rápido?

```
Test CORRECTO: Coinciden los resultados de la CPU y la GPU
device: 260795.000000  host: 260795.000000
```

Esquema 1:

```
[practicas02@tijola lab1.2-reduction.1]$ cat cuda_profile_0.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff54602c1a190
method,gputime,cputime,occupancy,l1_shared_bank_conflict,shared_load,shared_store
method=[ memcpyHtoD ] gputime=[ 0.896 ] cputime=[ 7.000 ]
method=[ _Z9reductionPfi ] gputime=[ 5.920 ] cputime=[ 23.000 ] occupancy=[ 0.167 ] l1_shared_bank_conflict=[ 0 ] shared_load=[ 41 ] shared_store=[ 36 ]
method=[ memcpyDtoH ] gputime=[ 1.408 ] cputime=[ 11.000 ]
[practicas02@tijola lab1.2-reduction.1]$
```

Esquema 2:

```
[practicas02@tijola lab1.2-reduction.2]$ cat cuda_profile_0_reduction2.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff545ae078478
method,gputime,cputime,occupancy,l1_shared_bank_conflict,shared_load,shared_store
method=[ memcpyHtoD ] gputime=[ 0.864 ] cputime=[ 6.000 ]
method=[ _Z9reductionPfi ] gputime=[ 7.712 ] cputime=[ 22.000 ] occupancy=[ 0.167 ] l1_shared_bank_conflict=[ 120 ] shared_load=[ 111 ] shared_store=[ 71 ]
method=[ memcpyDtoH ] gputime=[ 1.408 ] cputime=[ 11.000 ]
[practicas02@tijola lab1.2-reduction.2]$
```

Esquema 3:

```
practicas02@tijola lab1.2-reduction.3]$ cat cuda_profile_0_reduction3.log
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff545dc46ccf0
method,gputime,cputime,occupancy,l1_shared_bank_conflict,shared_load,shared_store
method=[ memcpyHtoD ] gputime=[ 0.864 ] cputime=[ 6.000 ]
method=[ _Z9reductionPfi ] gputime=[ 5.792 ] cputime=[ 24.000 ] occupancy=[ 0.167 ] l1_shared_bank_conflict=[ 0 ] shared_load=[ 47 ] shared_store=[ 39 ]
method=[ memcpyDtoH ] gputime=[ 1.408 ] cputime=[ 11.000 ]
[practicas02@tijola lab1.2-reduction.3]$
```

Esquema de reducción	Conflictos en los bancos de memoria compartida (l1_shared_bank_conflict)	Tiempo de ejecución (gputime)
1	0	5.920

2	120	7.712
3	0	5.792

Sobre la pregunta de cuál es la más rápida, se puede responder con la tabla. El tercer esquema es el más rápido y seguido de cerca por el primero y ambos con 9 conflictos. En cambio, el segundo esquema es el más lento debido a la cantidad de conflictos que tiene.