Metodología de la Programación Paralela

Silvia Perez Ruiz
E-mail: silvia.perezr@um.es, DNI: 49215974E
Raúl Hernández Martínez
E-mail: raul.hernandezm@um.es, DNI: 24423648V
Eduardo Gallego Nicolás
E-mail: eduardo.gallegon@um.es, DNI: 48844607J

Práctica 4. Introducción a la Programación de GPUs con CUDA

Grado en Ingeniería Informática
Metodología de la Programación Paralela
Profesor: Jose Matias Cutillas Lozano (josematias.cutillas@um.es)

UNIVERSIDAD DE
MURCIA

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

Índice

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

1 Cuestión 1.

Enunciado: Ejecuta el programa deviceQuery y obtén la siguiente información para la GPU.

1. Número de Streaming Multiprocessors (SMP): 3
2. Número de Streaming Processors (SP): 128
3. Total de CUDA Cores: 384
4. Número Máximo de Threads por SMP: 2048
5. Número Máximo de Threads por Bloque: 1024
6. Cantidad de Memoria Global: 2000 MBytes (2097020928 bytes)
7. Registros Disponibles por Bloque: 65536
8. Cantidad de Memoria Compartida por Bloque: 49152 bytes
9. Dimensiones Máximas del Grid: (2147483647, 65535, 65535) (x, y, z)
10. Dimensiones Máximas del Bloque de Threads: (1024, 1024, 64) (x, y, z)

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

2 Cuestión 2.

Enunciado: Utilizando el programa bandwidthTest:

2.1 Apartado a.

Enunciado: Representa en una gráfica el ancho de banda de las transferencias entre el host y el dispositivo, dentro del dispositivo, y entre el dispositivo y el host, con bloques de memoria desde 1 KB hasta 64 MB (en incrementos de 512 KB) cuando se usa "pageable memory".

Para poder ejecutar el programa, tuvimos que cambiar una cosa del makefile para que nos funcionara. Ahora el makefile se quedaría así:

```
GCC  = g++
NVCC = nvcc

CUDA_DIR = /usr/local/cuda
CUDA_LIB = -L$(CUDA_DIR)/lib64
CUDA_INC = -I$(CUDA_DIR)/include
CUDA_INC_COMMON = -I../Common
```

Listing 1: Makefile.

Como se puede ver solo cambiamos la variable de `CUDA_DIR`. Ejecutando eso nos salen unos datos que hemos representado en esta gráfica:

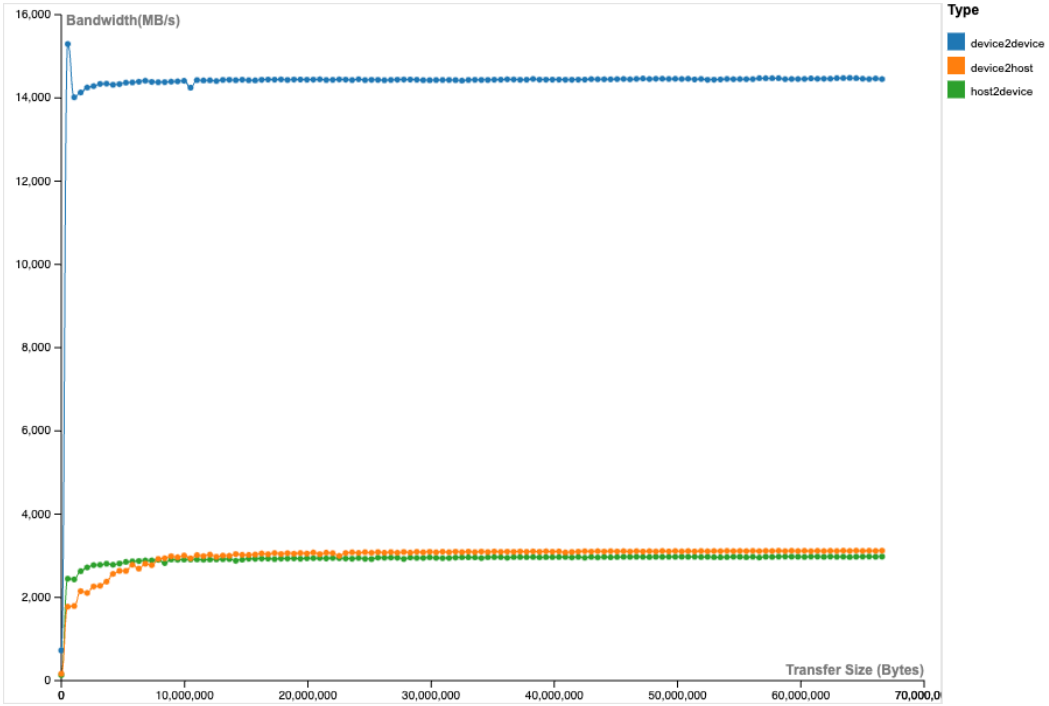

Figura 1: Pageable memory.

2.2 Apartado b.

Enunciado: Repite el experimento haciendo uso de "pinned memory" y explica las diferencias obtenidas.

Para poder hacer uso de "pinned memory"hay que modificar el Makefile dejandolo así:

```
./$(FILE) --device=0 --memory=pinned --mode=range --start=1024 --end=67108864 --increment
    =524288 --htod
./$(FILE) --device=0 --memory=pinned --mode=range --start=1024 --end=67108864 --increment
    =524288 --dtod
./$(FILE) --device=0 --memory=pinned --mode=range --start=1024 --end=67108864 --increment
    =524288 --dtoh
```

Listing 2: Makefile.

Ejecutando esto nos salen unos datos que hemos representado en esta gráfica:

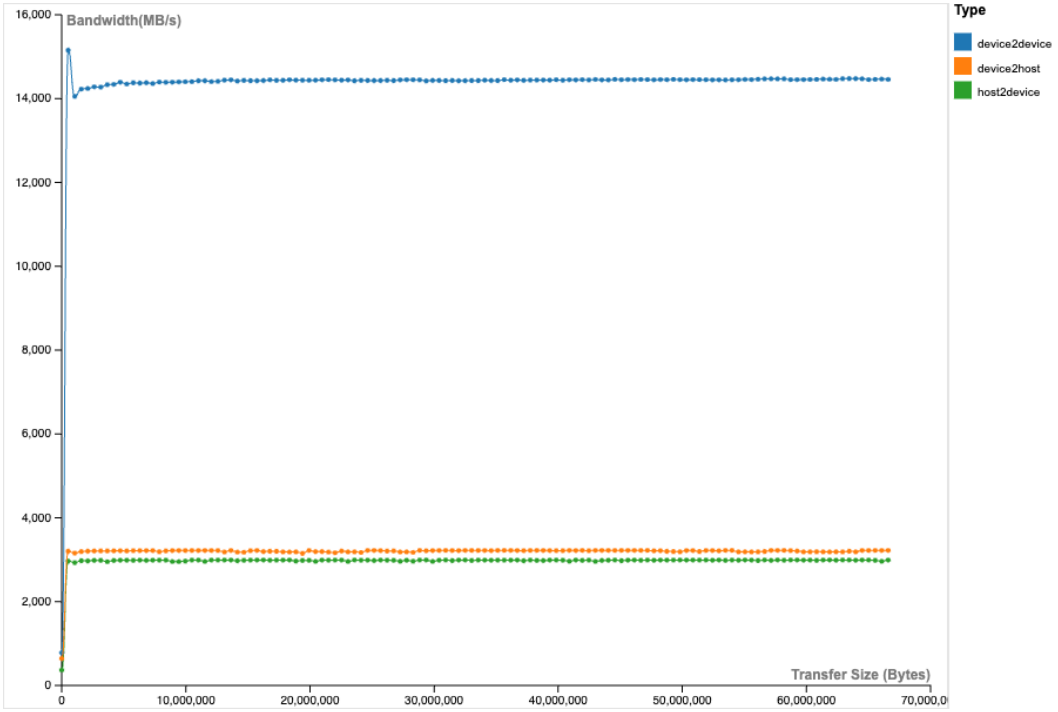

Figura 2: Pinned memory.

La memoria fija o pinned memory, es aquella que no puede ser paginada y la memoria paginada como bien dice su nombre es aquella que si puede ser paginada.

Como podemos ver en las dos gráficas, el ancho de banda en el device to device es mucho mayor que los otros modos. En la primera gráfica la diferencia entre los dos modos restantes es mínima, sin embargo en la memoria fija si que se puede apreciar un mayor ancho de banda en el device to host mode. Esto se debe a que en memoria fija la cantidad de información que se envía del host al device es menor que la cantidad que se envía del device al host, esto tiene sentido ya que al tener la información fija habrá menos fallos de página.

3 Cuestión 3.

Enunciado: Ejecuta el programa cudaTemplate con diferentes tamaños de grid y bloques de threads.

Tamaño grid(x,y)	Tamaño bloque threads (x,y)	Tiempo ejecucion
(4,4)	(2,2)	0.0395 ms
(8,8)	(2,2)	0.0405 ms
(8,8)	(4,4)	0.0415 ms
(16,16)	(2,2)	0.0441 ms
(16,16)	(4,4)	0.0492 ms
(16,16)	(8,8)	0.0715 ms
(32,32)	(2,2)	0.0586 ms
(32,32)	(4,4)	0.1067 ms
(32,32)	(8,8)	0.1643 ms
(32,32)	(16,16)	0.6223 ms

3.1 Apartado a.

Enunciado: ¿Qué hace el código del kernel?

El programa comienza incluyendo varios encabezados necesarios para la programación de CUDA y define algunas constantes, como el tamaño de la memoria compartida (SH_MEM_SIZE) y el tamaño de un array constante (CT_MEM_SIZE). Además, hace referencia a un archivo externo llamado çudaTemplate_kernel.cu"que contiene el kernel CUDA.

En la función principal (main), se procesan los argumentos de la línea de comandos para obtener las dimensiones de la cuadrícula y de los bloques. Luego, se reserva memoria en el host y en el dispositivo para almacenar datos (gid_h y gid_d, respectivamente). Después de configurar el dispositivo GPU a utilizar, se realiza la transferencia de datos desde el host al dispositivo y se inicializa la memoria del dispositivo. También se copia un array constante (const_h) a la memoria constante del dispositivo. A continuación, se crea un evento CUDA para medir el tiempo de ejecución del kernel. Se configuran los parámetros de ejecución del kernel, como las dimensiones de la cuadrícula y los bloques. Se imprime información sobre la configuración de ejecución.

El kernel denominado "foo"se ejecuta en la GPU con la configuración establecida. Se espera a que todos los hilos del kernel se completen y se realiza la transferencia de resultados desde el dispositivo al host.Se utiliza un evento para medir el tiempo de procesamiento, y se imprime el tiempo transcurrido. Luego, se verifica que los resultados obtenidos coincidan con un patrón esperado basado en el array constante.

Finalmente, se liberan los recursos, incluyendo la memoria en el dispositivo y en el host, y se imprime un mensaje indicando que el programa ha pasado la verificación.

En resumen, el código realiza una serie de operaciones en la GPU mediante CUDA, mide el tiempo de ejecución, verifica los resultados y libera los recursos utilizados.

3.2 Apartado b.

Enunciado: Analiza el uso que se hace de la memoria compartida en el código del kernel y verifica si el tamaño indicado es correcto. Si no lo fuese, explica cómo hay que modificar el código.

El tamaño de la memoria compartida (shared_mem_size) se calcula en función de las dimensiones del bloque (block.x y block.y).

Podemos pensar que el tamaño adecuado de memoria compartida debería ser igual al número de elementos del grid, ya que los datos se comparten exclusivamente entre subprocesos dentro del mismo bloque y no son accesibles desde diferentes bloques. En este caso, cada hilo sólo necesita una matriz de memoria compartida cuyo tamaño coincida con el bloque en el que se encuentra, ya que no intercambia información con hilos de otros bloques.

3.3 Apartado c.

Enunciado: ¿Podría eliminarse la primera llamada a __syncthreads() en el kernel? ¿Y la segunda?

Las llamadas a __syncthreads() desempeñan un papel muy importante en la sincronización de hilos dentro de un bloque. Estas llamadas aseguran que todos los hilos hayan alcanzado un punto de sincronización antes de continuar con la ejecución del kernel.

Para la primera llamada a __syncthreads(), shared_mem[tidb] parece redundante, ya que su contenido no se utiliza despues de la llamada a __syncthreads(). Por lo tanto, podríamos considerar la posibilidad de omitir la primera llamada a __syncthreads() y optimizar el código.

Para la segunda llamada a __syncthreads(), hay una escritura en la memoria compartida seguida de una la llamada __syncthreads(). Dado que cada hilo tiene un índice único y modifica una ubicación de memoria diferente, se considera que la sincronización entre los hilos podría no ser necesaria en este caso específico. Por lo tanto, se puede optar por eliminar la segunda llamada a __syncthreads().

3.4 Apartado d.

Enunciado: Modifica el código para que el kernel utilice bloques de threads tridimensionales, es decir, para que tenga en cuenta la coordenada z.

Para conseguir que los bloques threads sean tridimensionales tenemos que irnos al código de `cudatemplate.cu` para que obtenga la coordenada z del bloque thread.

Para conseguir esto, habría que hacer este cambio:

```
// global thread ID in thread block
//int tidb = (blockDim.x * threadIdx.y + threadIdx.x);

//               APARTADO D
```

```
5 int tidb = ((blockDim.x  * blockDim.y) * threadIdx.z + blockDim.x * threadIdx.y + threadIdx.
      x);
```

Listing 3: Cambio para hacer bloques threads tridimensionales.

Comentamos como estaba hecho antes la variable `tidb`. Haciendo este cambio, ya estamos teniendo en cuenta la tridimensionalidad del bloque. La variable `tidg` no hace falta modificarlo ya que el grid no puede tener tres dimensiones.

Si volvemos a ejecutar el programa `cudaTemplate` con los nuevos cambios se quedaría asi:

Tamaño grid(x,y)	Tamaño bloque threads (x,y)	Tiempo ejecucion
(4,4)	(2,2)	0.0408 ms
(8,8)	(2,2)	0.0403 ms
(8,8)	(4,4)	0.0412 ms
(16,16)	(2,2)	0.0436 ms
(16,16)	(4,4)	0.0508 ms
(16,16)	(8,8)	0.0717 ms
(32,32)	(2,2)	0.0607 ms
(32,32)	(4,4)	0.1237 ms
(32,32)	(8,8)	0.1631 ms
(32,32)	(16,16)	0.6174 ms

4 Cuestión 4.

Enunciado: Ejecuta el programa vectorAdd con diferentes tamaños de vector y threads por bloque.

4.1 Apartado a.

Enunciado: Anota los tiempos de ejecución obtenidos y analiza la influencia que tiene variar el número de threads por bloque para cada tamaño de problema.

Tamaño vector	Threads por bloque	Tiempo ejecucion
10000	16	0.1124 ms
20000	16	0.1822 ms
20000	32	0.1769 ms
30000	16	0.2631 ms
30000	32	0.2542 ms
30000	64	0.2540 ms
40000	16	0.3464 ms
40000	32	0.3303 ms
40000	64	0.3282 ms
40000	128	0.3178 ms
50000	16	0.4459 ms
50000	32	0.3997 ms
50000	64	0.4095 ms
50000	128	0.4082 ms
50000	256	0.4218 ms

Como podemos ver en la tabla, cuando aumentamos los threads por bloque el tiempo se ve reducido. No obstante, en la parte donde el tamaño del vector es 50000 hay un punto en el que cuando se sube mucho el número de hilos el tiempo no se reduce más si no que aumenta, a excepción de cuando los threads por bloque son 32 que vemos que si baja de 0.4459 pero aun así no reduce el tiempo que tenia cuando el tamaño vector es 40000 y los threads son 128 donde el tiempo es 0.3178.

Cuanto más grande es el tamaño más hilos tarda en llegar a ese punto. Más hilos aumentan el tiempo de ejecución en vez de reducirlo.

4.2 Apartado b.

Enunciado: Analiza el código del kernel y explica cómo se produce la suma de los vectores.

Vamos a revisar cada sección del código en vectorAdd_kernel.cu:

```
\_global\_ void vectorAdd(const float *X, const float *Y, float *V, int numElements)
```

Listing 4: Anotacion Global.

La anotación global indica que se trata de un kernel en CUDA. El kernel debe tener un tipo de retorno void.

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;
```

Listing 5: Identificador de hilo global.

Calculamos el identificador global del hilo utilizando las estructuras proporcionadas por CUDA. `blockIdx` contiene la posición del bloque en la rejilla desde 0 hasta `gridDim-1`. `threadIdx` es el índice del hilo dentro de su bloque asociado, desde 0 hasta `blockDim-1`.

```
if (tid < numElements)
```

Listing 6: Verificación del identificador global.

Para evitar acceder a una región fuera de nuestro conjunto de datos, verificamos que nuestro identificador global de hilo sea menor que la longitud de nuestro conjunto de datos.

```
V[tid] = X[tid] + Y[tid];
```

Listing 7: Verificación del identificador global.

El tid se utiliza para indexar las matrices en la memoria global del dispositivo. Cada hilo carga un valor de X y Y, y escribe la suma en V.

4.3 Apartado c.

Enunciado: Analiza el código que calcula el número de bloques del grid. ¿Qué sucede si el número total de elementos del vector no es múltiplo del tamaño del bloque de threads?

El fragmento de código se encarga de calcular el número de bloques en el grid en un entorno CUDA. La fórmula utilizada es:

$$blocksPerGrid = \frac{numElements + threadsPerBlock - 1}{threadsPerBlock}$$

$$blocksPerGrid = \frac{numElements}{threadsPerBlock} + \frac{threadsPerBlock}{threadsPerBlock} - \frac{1}{threadsPerBlock}$$

$$blocksPerGrid = \frac{numElements}{threadsPerBlock} + 1 - \frac{1}{threadsPerBlock}$$

Esta ecuación se descompone en partes para facilitar la comprensión. La variable `numElementos` representa el total de elementos en el vector, `hilosPorBloque` es el tamaño del bloque en términos de hilos, y `bloquesPorGrid` es el número de bloques en el grid.

Es esencial tener en cuenta que el tamaño del bloque se expresa en hilos, por lo que se denomina `hilosPorBloque`.

4.4 Apartado d.

Enunciado: Explica cómo puede afectar al rendimiento que el bloque de threads no sea potencia de 2. Para ello, realiza una breve búsqueda bibliográfica sobre el concepto

de warp, la unidad de planificación de hilos de CUDA.

Según los resultados de la ejecución, se observa que se logra un mejor rendimiento con un número de hilos que no sea una potencia de dos. Sin embargo, teóricamente, el rendimiento debería mejorar cuando los hilos son múltiplos de 2, pero hay un factor adicional que debemos considerar.

La razón por la que se prefieren los múltiplos de 2 se debe a una limitación conocida como la ”Granularidad de Asignación de 4 Warps”. Esta limitación afecta la asignación de recursos de hardware de la siguiente manera:

En SM1.x-2.x, los recursos se asignan a 2 warps simultáneamente.
En SM3.x-5.x, los recursos se asignan a 4 warps simultáneamente.

Cuando la configuración del núcleo es N, el hardware asigna recursos para N redondeado al múltiplo de la Granularidad de Asignación de Warps. Esta restricción reduce la complejidad del control y el tamaño de la tabla de asignación, disminuyendo así el área y el consumo de energía.

En CUDA, los registros se asignan en páginas de registro por grupos de hilos, ya sea en bloques para dispositivos sm_1x o en warps para dispositivos sm_2x y sm_3x. Los registros se asignan en múltiplos del tamaño de la página de registro, también conocido como ”granularidad de asignación de registros”según la calculadora de ocupación de CUDA.

Debido a esta granularidad del warp, se recomienda elegir un tamaño que sea múltiplo de 32. Aunque las opciones de tamaño de bloque de hilo de potencias de 2 son comunes, no son estrictamente necesarias. Esto explica por qué en ejecuciones anteriores, el rendimiento fue más rápido con 64 y 128 hilos (múltiplos de 32) en comparación con 65 y 129 hilos (que no son múltiplos de 32).

La idea principal es hacer que el número de hilos por bloque sea un múltiplo de 32, que es el tamaño del warp en las GPU utilizadas. El warp es la unidad de planificación de ejecuciones de hilos en la GPU. El planificador de warps envía a ejecutar cada warp de 32 hilos (del bloque correspondiente) al SM asociado. Por lo tanto, si el bloque tiene, por ejemplo, 70 hilos, habría 2 warps activos a pleno rendimiento y otro warp activo con solo 6 hilos realizando trabajo útil.