Ejercicios de laboratorio

11 de mayo de 2011

Resumen

El objetivo de este boletín de ejercicios es asistir y afianzar algunos de los conceptos teóricos vistos en clase. El manual incluye un ejercicio introductorio para la familiarización del estudiante con el entorno, seguido de varios problemas ampliamente conocidos e ilustrativos, como el producto de matrices, reducciones y MRI. El apéndice final incluye un resumen de las técnicas utilizadas, como implementaciones por bloques, desenrollado de bucles, data prefetching, ...

En total se presentan 6 ejercicios que cubren conceptos básicos hasta optimización del rendimiento. Muchos de ellos presentan distintos niveles de dificultad. Es conveniente escoger primero el nivel básico, para después pasar a los niveles más avanzados.

1. Hola, mundo

1.1. Objetivo

El objetivo de este ejercicio se centra en construir un entorno de trabajo sobre el cluster de GPUs sobre el que trabajaremos, así como compilar y ejecutar nuestro primer programa CUDA.

1.2. Familiarización con el entorno

1.2.1. Paso 1

Utiliza un programa de acceso remoto (ssh) para acceder a la máquina `calendula`. Copia el paquete con los ejercicios y utilidades básicas para el resto de las prácticas desde el directorio ../COMUNES a tu home, y descomprímelo allí.

1.2.2. Paso 2: familiarización con el sistema de colas

Tenemos acceso a un cluster con 6 nodos. Cada uno de ellos está equipado con dos GPUs relativamente modernas. Aunque podríamos utilizar el sistema de colas para lanzar nuestros programas, nos resultará más cómodo acceder interactivamente a los nodos con GPU. Para ello, ejecuta:

```
qlogin -q gpu
```

No importa en qué nodo te encuentres, ya que todos son idénticos.

1.2.3. Paso 3

Accede al directorio lab0 y asegúrate de que contiene los ficheros necesarios para el desarrollo del ejercicio. Debes ver al menos dos ficheros: helloworld.cu y helloworld_kernel.cu.

1.3. Compilación del primer programa CUDA

A continuación se muestran las órdenes necesarias para construir el programa y la correspondiente salida que se debería ver tras la ejecución.

Atención: como verás, los printf sólo funcionan en modo emulación. Por lo tanto, cualquier printf y código de entrada/salida debe ser eliminado del kernel antes de ser ejecutado sobre la GPU (es decir, al compilarlo en modo no-emulación).

```
$> cd CUDA_WORKSHOP/projects/lab0-hello_world
$> ./make_emu.csh
$> ../../bin/linux/emudebug/lab0-hello_world
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 0}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 0}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 1}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 1}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 2}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 2}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 0, 3}
Hello World! I am a thread with BlockID: {0, 0}, ThreadID: {0, 1, 3}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 0}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 0}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 1}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 1}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 2}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 2}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 0, 3}
Hello World! I am a thread with BlockID: {1, 0}, ThreadID: {0, 1, 3}
```

Intenta compilar el programa manualmente, utilizando nvcc. Compílalo en modo emulación y en modo release. Observa dónde se almacenan los ejecutables.

Intenta descubrir las opciones que ofrece el compilador nvcc. ¿Podrías compilar este ejemplo básico con gcc/g++? ¿Por qué?

Modifica los valores tanto para número de bloques como para número de threads por bloque. Utiliza valores muy altos, e intenta descubrir el límite de la arquitectura para ellos.

¿Tiene la arquitectura algún otro límite de recursos? Busca en la Guía de Programación CUDA cómo reservar memoria compartida dentro del kernel. Juega con los tamaños reservados, y con el número de threads por bloque. ¿Hay alguna restricción?

2. Producto de matrices

2.1. Objetivo

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 reservar y liberar memoria en GPU.
- 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 sobre la GPU.

En su estado inicial, el código no compila correctamente. El primer paso, pues, es completar los espacios para conseguir un programa que compile.

Nivel de dificultad 1 : Para estudiantes con poca o ninguna experiencia con CUDA, utilizad `projects/lab1.1-matrixm`

Nivel de dificultad 2 : Para estudiantes con más experiencia con CUDA, utilizad `projects/lab1.1-matrixmul.2`.

2.2. Modificación de los programas originales

2.2.1. 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:

- Reserva memoria para matrices de entrada.
- Copia la matriz de entrada desde la memoria RAM hasta la memoria del dispositivo.
- Define un timer para medir el tiempo necesario para llevar a cabo estas copias. ¿Podrías calcular el ancho de banda efectivo conseguido?

Segunda parte del código:

- Configura la ejecución del kernel e invócalo.
- Define un timer para medir el tiempo de computación. Piensa en el carácter asíncrono de las invocaciones de los kernels CUDA.

Tercera parte del código:

- Copia la matriz resultado desde la memoria del dispositivo hasta memoria RAM.
- Define un timer para volver a medir los tiempos de transferencia. ¿Qué ancho de banda efectivo has conseguido? ¿Es comparable con el ancho de banda calculado anteriormente? (Si sientes curiosidad, prueba a compilar y ejecutar el programa bandwidthTest del SDK de Nvidia).

Cuarta parte del código:

- Libera la memoria del dispositivo.

2.2.2. 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:

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

2.2.3. Paso 3

Compila usando los Makefiles correspondientes. Usa make.csh para modo release y make_emu.csh para modo emulación.

Prueba tus códigos con distintos tamaños de matriz. Por ejemplo, dimensiones 8, 128, 512, 3072 o 4096.

En modo release:

```
$> ../../bin/linux/release/lab1.1-matrixmul <8, 128, 512, 3072, 4096>
```

Run the executable for emulation debug mode:

```
$> ../../bin/linux/emudebug/lab1.1-matrixmul <8, 128, 512, 3072, 4096>
```

Deberías ver una salida de este tipo:

```
Input matrix file name:
Setup host side environment and launch kernel:
Allocate host memory for matrices M and N.
M:
N:
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.
Setup kernel execution parameters.
# of threads in a block:
# of blocks in a grid :
Executing the kernel...
Copy result from device to host.
GPU memory access time:
GPU computation time :
GPU processing time :
Check results with those computed by CPU.
Computing reference solution.
CPU Processing time :
CPU checksum:
GPU checksum:
Comparing file lab1.1-matrixmul.bin with lab1.1-matrixmul.gold ...
Check ok? Passed.
```

Asegúrate, observando la última línea, de que el test es correcto. Apunta los datos relativos a tiempo de ejecución y transferencia de datos:

Tamaño de matriz	Transferencia a GPU (ms)	Transferencia de GPU (ms)	Ejecución (ms)	Ratio comparado con 128×128
8×8				
128×128				1
512×512				
3072×3072				
4096×4096				

¿Qué conclusiones puedes sacar a partir de estos números?

Como ejercicio adicional, prueba a migrar los códigos a aritmética en doble precisión. ¿En qué medida se ven degradadas las prestaciones?

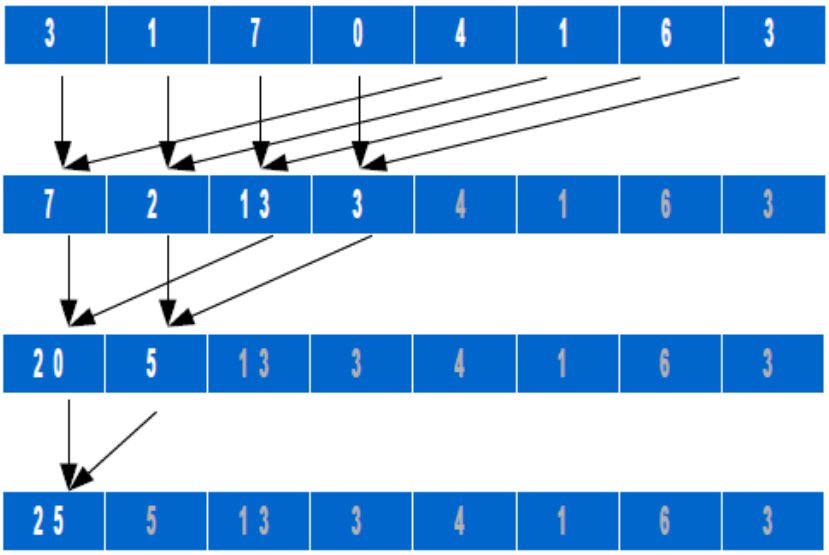

Figura 1: Primer esquema de reducción

3. Reducciones

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, utizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular varias operaciones en paralelo, calculando el resultado en O(logN) pasos sin incrementar el número de operaciones realizadas. Un ejemplo de este tipo de operación se muestra en la Figura 1.

Los objetivos del ejercicio son:

- Reservar y liberar memoria en la GPU.
- Copiar datos entre la CPU y la GPU.
- Copiar datos entre la GPU y la CPU.
- Escribir un kernel que implemente la reducción suma.
- Medir el tiempo de ejecución y los conflictos en el acceso a los bancos de memoria compartida usando CUDA Profiler.
- Observar cómo diferentes esquemas de reducción afectan al número de conflictos en accesos a bancos de memoria compartida.

Nivel de dificultad 1 : Para estudiantes con poca o ninguna experiencia con CUDA, utilizad `projects/lab1.2-reducti`
Este código no compila hasta que no se rellenen los blancos correspondientes.

Nivel de dificultad 2 : Para estudiantes con más experiencia con CUDA, utilizad `projects/lab1.2-reduction.2`.
Este código compila con warnings.

3.1. Modificaciones necesarias sobre los programas originales

3.1.1. Paso 1

Modifica la función computeOnDevice(...) definida en vector_reduction.cu.
Primera parte del código:

- Reserva memoria en el dispositivo.
- Copia los datos de entrada desde la memoria del host hasta la memoria del dispositivo.
- Copia los resultados desde la memoria de dispositivo de vuelta a la memoria del host.

3.1.2. Paso 2

En el fichero vector_reduction_kernel.cu, modifica la función para implementar el esquema de reducción mostrado en la Figura 1. Tu implementación debería usar memoria compartida para incrementar la eficiencia.

Segunda parte del código:

- Carga datos desde memoria global a memoria compartida.
- Realiza la reducción sobre los datos en memoria compartida.
- Almacena de vuelta los datos en memoria compartida.

3.1.3. Paso 3

Una vez finalizado el código, simplemente utiliza la orden make.csh (para modo release) en el directorio del proyecto para compilar el código fuente, o make_emu.csh para compilar en modo emulación.

Para el modo release, el ejecutable estará en:

```
../../bin/linux/release/vector_reduction
```

Para el modo emulación, el ejecutable estará en:

```
../../bin/linux/emudebug/vector_reduction
```

Esta aplicación tiene dos modos de operación distintos. La única diferencia es la entrada que se le proporciona tanto a las implementaciones en GPU y en CPU del código de reducción. En cualquier caso, la versión sobre GPU siempre es ejecutada primero, seguida de la versión sobre CPU, que sirve como referencia para comprobar el resultado. Si el resultado es correcto, se imprime por pantalla la frase `Test Passed`. En caso contrario, se imprime la frase `Test Failed`.

Sin argumentos El programa crea aleatorioamente un array como entrada a las funciones de reducción. Para ejecutarlo:

```
$> ../../bin/linux/release/vector_reduction
```

Un argumento El programa utiliza el fichero de datos asociado como entrada a las dos implementaciones de la reducción. Para ejecutarlo:

```
$> ../../bin/linux/release/vector_reduction data.txt
```

Ten en cuenta que, en este caso, los ejercicios no proporcionan información sobre tiempos de ejecución. Si lo deseas, intenta añadir esta información a tus implementaciones. De cualquier modo, extraeremos dicha temporización directamente del CUDA profiler.

La salida del programa correcto debería tener este aspecto:

```
Test PASSED
device: host: Input vector: random generated.
```

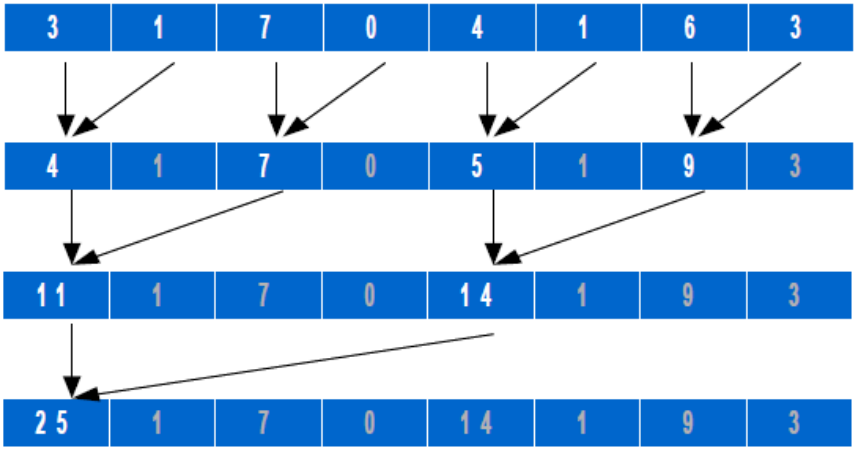

Figura 2: Segundo esquema de reducción

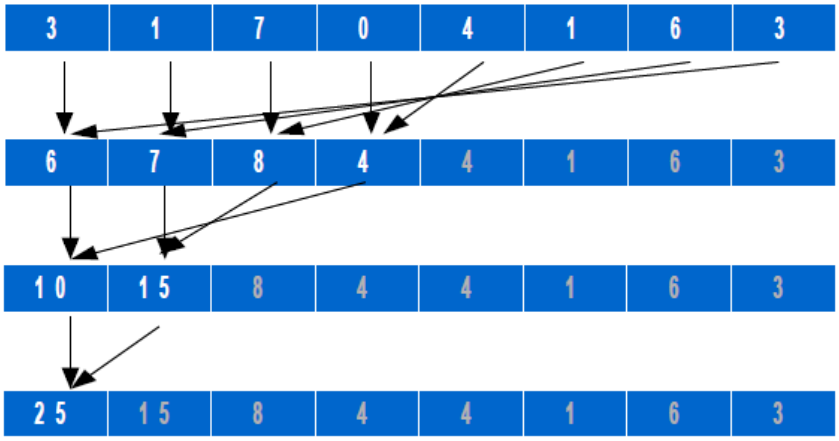

Figura 3: Tercer esquema de reducción

3.1.4. Paso 4

Una vez hayas implementado el esquema de reducción mostrado en la Figura 1, compararemos los resultados con otros dos esquemas de reducción. Edita la función reduction(...) en el fichero vector_reduction_kernel.cu, donde todavía hay dos esquemas vacíos por implementar. Los esquemas corresponden con los de las Figuras 2 y 3. En el esquema mostrado en la Figura 2, cada thread es responsable de la suma de dos nodos adyacentes en una distancia potencia de 2 para cada nivel de la operación. En la Figura 3, cada thread 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 thread número 1 suma los elementos 1 y 8, siendo la distancia entre los dos nodos $8-1=7$. El thread 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 threads número 3 y 4 son $6-3=2$ y $5-4=1$, respectivamente.

Tened en cuenta el patrón de acceso regular en ambos esquemas. La mayoría de aplicaciones utilizan patrones de este tipo para eliminar divergencias innecesarias o conflictos en accesos a bancos de memoria.

3.1.5. Paso 5

Una vez finalizado el código, utiliza la siguiente orden para activar el CUDA profiler. También existe una versión gráfica del profiler. Si tienes acceso a ella, puedes utilizarla.

```
$> CUDA_PROFILE=1 CUDA_PROFILE_CONFIG=./profile_config \
../../bin/linux/release/vector_reduction
```

Por defecto, el fichero de salida para el CUDA profiler es cuda_profile.log. Más abajo verás un ejemplo de ejecución. `gputime` muestra los microsegundos que necesita cada kernel CUDA para ser ejecutado. `warp_serialize` se refiere al número de threads en un warp ejecutados secuencialmente. En otras palabras, realmente se refiere al número de conflictos en el acceso a bancos de memoria compartida. En el ejemplo, no hay conflictos. `_Z9reductionPfi` es el nombre dado por el runtime CUDA a la función reduction(...).

```
# CUDA_PROFILE_LOG_VERSION 1.4
# CUDA_DEVICE 0 GeForce GTX 280
method,gputime,cputime,occupancy,divergent_branch,warp_serialize
method=[ _Z9reductionPfi ] gputime=[ 4.512 ] cputime=[ 46.000 ]
occupancy=[ 1.000 ] divergent_branch=[ 1 ] warp_serialize=[ 0 ]
method=[ memcopy ] gputime=[ 3.584 ] cputime=[ 19.000 ]
```

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

Esquema	warp_serialize	Tiempo de GPU (ms)
1		
2		
3		

4. Producto de matrices por bloques con memoria compartida

4.1. Objetivos

Implementaremos una versión optimizada del producto de matrices, utilizando memoria compartida y sincronizaciones entre threads dentro de un bloque. La memoria compartida dentro de cada multiprocesador se utilizará para almacenar cada submatriz antes de los cálculos. Los objetivos principales son:

- Aplicar computación por bloques sobre el producto de matrices.
- Utilizar memoria compartida en la GPU.
- Aplicar correctamente esquemas de sincronización entre threads dentro de cada bloque.

Nivel de dificultad 1 : Elige el fichero `projects/lab2.1-matrixmul.1`.

Nivel de dificultad 2 : Elige el fichero `projects/lab2.1-matrixmul.2`.

Nivel de dificultad 3 : Elige el fichero `projects/lab2.1-matrixmul.3`.

4.2. Modificación de los programas CUDA

4.2.1. Paso 1

Edita la función matrixMul(...) en el fichero `matrixmul_kernel.cu` para completar la funcionalidad del producto de matrices sobre GPU. El código de la parte del host debería estar completo. No es necesaria ninguna modificación en otras partes del código.

Primera parte del código:
Determina los valores correctos para los índices de bloque dentro del bucle.
Segunda parte del código:
Implementa el producto de matrices dentro de cada bloque.
Tercera parte del código:
Almacena los resultados en memoria global.

4.2.2. Paso 2

Compila el programa en modo emulación o modo release, en función de si tienes disponible o no una GPU en tu sistema.

Como en el caso anterior, se proporcionan cinco casos de ejemplo, que corresponden con matrices de dimensiones 8×8, 128×128, 512×512, 3072×3072 y 4096×4096.

Ejecuta el programa en modo release:

```
$> ../../bin/linux/release/lab2.1-matrixmul <8, 128, 512, 3072, 4096>
```

O en modo emulación:

```
$> ../../bin/linux/emudebug/lab2.1-matrixmul <8, 128, 512, 3072, 4096>
```

La salida del programa debería ser similar a:

```
Input matrix file name:
Setup host side environment and launch kernel:
Allocate host memory for matrices M and N.
M:
N:
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.
Setup kernel execution parameters.
# of threads in a block:
# of blocks in a grid :
Executing the kernel...
Copy result from device to host.
GPU memory access time:
GPU computation time :
GPU processing time :
Check results with those computed by CPU.
Computing reference solution.
CPU Processing time :
CPU checksum:
GPU checksum:
Comparing file lab2.1-matrixmul.bin with lab2.1-matrixmul.gold ...
Check ok? Passed.
```

Guarda los tiempos de ejecución con respecto a distintos tamaños de matriz en la siguiente tabla:

Tamaño de matriz	Transferencia a GPU (ms)	Transferencia de GPU (ms)	Ejecución (ms)	Ratio comparado con 128×128
8×8				
128×128				1
512×512				
3072×3072				
4096×4096				

¿Han mejorado los resultados en comparación a la implementación previa del producto de matrices?

4.2.3. Paso 3

Edita la macro BLOCK_SIZE en el fichero matrixmul.h para observar cómo los diferentes tamaños de bloque afectan al rendimiento sobre matrices de dimensión 4096.

Guarda los tiempos de ejecución en función de distintos tamaños de bloque siguiendo la siguiente tabla:

Tamaño de bloque	Transferencia a GPU (ms)	Transferencia de GPU (ms)	Ejecución (ms)	Ratio comparado con 4×4
4×4				
8×8				
16×16				

¿Cuáles son las conclusiones a la vista de los resultados? ¿Puedes probar otros tamaños de bloque (6, 12, 24, 32, ...?

5. Reducciones con número ilimitado de elementos

5.1. Objetivo

En este ejercicio, se implementará una solución general para la operación de reducción desarrollada en ejercicios anteriores. Esta solución es capaz de trabajar con tamaños de entrada arbitrarios. Los objetivos principales son:

- Utilizar múltiples invocaciones a kernels como medio de sincronización entre bloques de threads.
- Medir rendimientos de programas CUDA.
- Utilizar técnicas para trabajar con tamaños de problema que no son potencias de dos.

Los siguientes programas deberían compilar correctamente, aunque con warnings.

Nivel de dificultad 1 : Elige el fichero `projects/lab2.2-reduction.1`.

Nivel de dificultad 2 : Elige el fichero `projects/lab2.2-reduction.2`.

5.2. Modificación de los programas CUDA

5.2.1. Paso 1

Edita la función runtTest(...) en el fichero reduction_largearray.cu para completar la funcionalidad en la parte del host.

Primera parte del código:

- Define un timer y mide el tiempo de ejecución del kernel.

5.2.2. Paso 2

Modifica la función reductionArray(...) definida en el fichero reduction_largearray_kernel.cu para completar la funcionalidad del kernel. Ten en cuenta que en este ejercicio es necesaria una sincronización entre bloques de threads, así que es necesario invocar al kernel en repetidas ocasiones. Cada kernel sólo realiza log(BLOCK_SIZE) pasos del proceso de reducción. Puedes utilizar el array de salida para almacenar resultados intermedios. De cualquier modo, asegúrate de que la suma final queda almacenada en el elemento cero del vector, para así poder compararse correctamente con la solución de referencia calculada en la CPU.

Hay dos aspectos claves en este ejercicio:

1. Trabajar con vectores cuyo tamaño no es múltiplo del tamaño de bloque (BLOCK_SIZE).
2. Trabajar con vectores que son demasiado grandes como para caber en un único bloque.

Es aconsejable comenzar con la solución del primer punto.

Para crear tamaños de entrada homogéneos, una técnica comúnmente usada es aplicar *padding*. Esta técnica consiste en añadir elementos nulos al vector al cargarlo en memoria compartida, de forma que consigamos tantos elementos como threads hay en un bloque.

A continuación, puedes modificar la función `reductionArray(...)` para trabajar con tamaños de entrada grandes. Normalmente, es conveniente utilizar un bucle para realizar la reducción, aunque también es posible utilizar recursión. Si quieres depurar tu código, una buena idea es copiar los datos de vuelta al host tras cada nivel de la reducción, y así comprobar que los resultados intermedios son correctos. También puedes utilizar el modo emulación para mostrar dichos resultados desde el propio kernel, aunque la primera opción es más aconsejable.

Ten en cuenta que si el tamaño del vector de entrada no es múltiplo del tamaño de bloque, deberemos, en cada nivel de la reducción, reservar bloques suficientes como para cubrir la entrada completa, incluso si el último bloque posee menos elementos que el valor de BLOCK_SIZE. Por ejemplo, si el tamaño de bloque es 4 y el tamaño del vector es 10, necesitaremos generar tres bloques, no dos.

5.2.3. Paso 3

Compila y ejecuta el código tal y como has hecho en los ejercicios anteriores. El programa generará un array inicializado aleatoriamente tanto para las versiones en CPU como en GPU del código de reducción. El tamaño del array se define en la macro DEFAULT_NUM_ELEMENTS del fichero reduction_largearray.cu. El tamaño por defecto es 16000000.

Si tu implementación es correcta, la salida debería tener el siguiente aspecto:

```
Using device 0: GeForce GTX 280
**===-------------------------------------------------===**
Processing 16000000 elements...
Host CPU Processing time:
CUDA Processing time:
Speedup:
Test PASSED
device: host:
```

5.2.4. Paso 4

Prueba tu implementación con distintos tamaños de entrada. ¿A partir de qué tamaño tu kernel comienza a ser más rápido que la implementación sobre un core de la CPU?

Tamaño de entrada	Tiempo CPU (ms)	Tiempo GPU (ms)

6. Producto de matrices optimizado

6.1. Objetivo

Este ejercicio proporciona más flexibilidad para afinar el rendimiento del producto de matrices. El programa permite modificar ciertos parámetros, como el grado de desenrollado de bucles, tamaños de bloque, *register spilling*, *data prefetching*, etc. y observar las variaciones en el rendimiento.

Toma tu tiempo para observar cómo se aplican en el código fuente estas variaciones. Los programas proporcionados son ya correctos, y deberían compilar sin errores.

6.2. Modificaciones sobre el código CUDA original

6.2.1. Paso 1

Edita las definiciones y macros en el fichero matrixmul.h para variar las optimizaciones en el kernel. El código fuente no necesita ser modificado en ningún otro punto.

6.2.2. Paso 2

Compila y ejecuta como has hecho en los ejercicios anteriores. Prueba los códigos para matrices de dimensión 4096:

```
$> ../../bin/linux/release/lab3.1-matrixmul 4096
```

La salida debería ser algo de este tipo:

```
Input matrix file name:
Setup host side environment and launch kernel:
Allocate host memory for matrices M and N.
M:
N:
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.
Setup kernel execution parameters.
# of threads in a block:
# of blocks in a grid :
Executing the kernel...
Optimization parameters:
Block size:
Unrolling factor:
Working size:
Register spilling:
Data prefetch:
Copy result from device to host.
GPU memory access time:
GPU computation time :
GPU processing time :
Check results with those computed by CPU.
Computing reference solution.
CPU Processing time :
CPU checksum:
GPU checksum:
Comparing file lab3.1-matrixmul.bin with lab3.1-matrixmul.gold ...
Check ok? Passed.
```

6.2.3. Paso 3

A continuación, utilizaremos el profiler de CUDA para analizar el rendimiento del programa. Ejecuta la siguiente secuencia de órdenes desde el shell de Linux:

```
$> CUDA_PROFILE=1
$> export CUDA_PROFILE
```

Ejecuta el programa de nuevo. Verás que se ha creado un nuevo fichero en el directorio de trabajo: cuda_profile.log. Un ejemplo podría ser:

```
$> ../../bin/linux/release/lab3.1-matrixmul 4096
$> cat cuda_profile.log
# CUDA_PROFILE_LOG_VERSION 1.4
# CUDA_DEVICE 0 GeForce GTX 280
method,gputime,cputime,occupancy
method=[ memcopy ] gputime=[ 24041.824 ] cputime=[ 38837.000 ]
method=[ memcopy ] gputime=[ 24577.951 ] cputime=[ 38641.000 ]
method=[ _Z9matrixMulPfS_S_ii ] gputime=[ 1109786.625 ] cputime=[ 10.000 ]
occupancy=[ 0.500 ]
method=[ memcopy ] gputime=[ 46265.602 ] cputime=[ 108240.000 ]
```

`gputime` muestra el tiempo, en microsegundos, que ha tardado un kernel CUDA en concreto en ejecutarse (en este caso, matrixMul).

`occupancy` muestra el ratio de warps activos con respecto al máximo número de warps soportados en un multiprocesador de la GPU. En la G80, por ejemplo, el número máximo de warps es $768/32 = 24$, con 8192 registros por multiprocesador. En la GTX280, el máximo número de warps activos es de $1024/32 = 32$, con 16384 registros por multiprocesador. El número de warps activos está limitado por el número de registros y memoria compartida necesaria para cada bloque de threads.

6.2.4. Paso 4

A continuación obtendremos información sobre el uso de memoria de nuestro programa CUDA. Ejecuta el script find_mem_usage.csh. Observa las órdenes que se han ejecutado:

```
$> ./find_mem_usage.csh matrixmul.cu
nvcc -I. -I../../common/inc -I/usr/local/cuda/include -DUNIX -o
matrixmul.cu.cubin -cubin matrixmul.cu
See matrixmul.cu.cubin for register usage.
```

El fichero de salida matrixmul.cu.cubin contiene información detallada sobre uso de memoria local, registros, memoria compartida, memoria de constantes, ...

Observa el contenido de dicho fichero:

```
 $> head -n 20 matrixmul.cu.cubin
architecture {sm_10}
abiversion {1}
modname {cubin}
code {
name = _Z9matrixMulPfS_S_ii
lmem = 0
smem = 2084
reg = 9
bar = 1
const { ... }
```

A continuación, recoge información sobre las ejecuciones en función de los distintos parámetros de optimización, tal y como se muestra en la tabla:

Tam. de tile	Occupancy	Tiempo GPU (ms)	Mem. local	Mem. compartida	Registros
4×4					
8×8					
16×16					

Factor de desenrollado	Occupancy	Tiempo GPU (ms)	Mem. local	Mem. compartida	Registros
0					
2					
4					
16					

6.2.5. Paso 5

A continuación, recopila información variando los distintos parámetros de optimización, y rellena las siguientes tablas:

Tam. de tile	Occupancy	Tiempo GPU (ms)
4×4		
8×8		
16×16		

Grado de desenrollado	Occupancy	Tiempo GPU (ms)
0		
2		
4		
16		

Working size	Occupancy	Tiempo GPU (ms)
1		
2		
4		

Reg. spilling	Occupancy	Tiempo GPU (ms)
Activado		
Desactivado		

Prefetching	Occupancy	Tiempo GPU (ms)
Activado		
Desactivado		

¿Puedes encontrar la combinación de parámetros que consigue el mejor rendimiento? Trata de entender por qué dichos parámetros son óptimos:

Parámetros	Valor / resultado
Tam. de tile	
Desenrollado	
Working size	
Reg. spilling	
Prefetching	
Occupancy	
Tiempo de GPU (ms)	

7. MRI (Magnetic Resonance Imaging)

7.1. Objetivos

El objetivo de este ejercicio es afinar el rendimiento de una aplicación basada en reconstrucción de imágenes MRI. Para ello, se mejorará el nivel de ocupación de los SM cambiando el tamaño de bloque, usando registros para almacenar datos que se usan frecuentemente, etc.

Nivel de dificultad 1 : Elige el fichero `projects/lab3.2-mriQ.1`.

Nivel de dificultad 2 : Elige el fichero `projects/lab3.2-mriQ.2`.

7.2. Modificaciones sobre los programas CUDA

7.2.1. Paso 1 para nivel de dificultad 1

Edita las macros y declaraciones en el fichero computeQ.h para variar los parámetros de optimización del kernel. Puees dar diferentes valores al grado de desenrollado del bucle, tamaño de bloque, simplificación de las condicionales, uso de memoria de constantes y uso de funciones trigonométricas especiales para afinar el rendimiento. El resto del código no necesita modificaciones. En cualquier caso, busca en el código dónde y cómo se aplican dichas optimizaciones.

7.2.2. Paso 1 para nivel de dificultad 2

Edita el fichero computeQ.cu para mejorar el rendimiento de la aplicación. Una pista: el kernel computeQ_GPU toma la mayor parte del tiempo de ejecución de la aplicación, y además tiene el mayor potencial para ser optimizado. Algunas técnicas que puedes utilizar son:

- Uso de memoria de constantes para datos de sólo lectura.
- Cambio en el número de threads por bloque para mejorar la utilización del SM.
- Carga de datos en registros si éstos son frecuentemente usados.
- Desenrollado de bucles.
- Simplificación de condicionales siempre que sea posible.
- En esta aplicación en particular, el uso de la función sinconsf() puede ser útil. Busca información sobre ella en la red.

No es necesaria ninguna otra modificación para que el código funcione correctamente.

7.2.3. Paso 2

Compila y ejecuta el código tal y como has hecho en pasos anteriores. Un ejemplo de ejecución sería:

```
$> ../../bin/linux/release/mri-q -S -i
data/sutton.sos.Q64x147258.input.float -o my_output.out
```

Mientras que la salida típica del programa debe ser:

```
2097152 pixels in output; 147258 samples in trajectory; using samples
IO:
GPU:
Copy:
Compute:
```

Para verificar la corrección del programa, compara la salida con la solución de referencia que encontrarás en el fichero reference_full.out:

```
$> diff -s my_output.out data/reference_full.out
```

Como los experimentos con el conjunto completo de datos pueden llevar bastante tiempo, es recomendable experimentar con conjuntos de datos menores:

Para ejecutar el programa con 512 muestras:

```
$> ../../bin/linux/release/mri-q -S -i \
data/sutton.sos.Q64x147258.input.float -o my_output 512
```

Para ejecutar el programa con 10000 muestras:

```
$> ../../bin/linux/release/mri-q -S -i \
data/sutton.sos.Q64x147258.input.float -o my_output 10000
```

7.2.4. Paso 2 para nivel de dificultad 1

Tras experimentar con los distintos parámetros, encuentra la combinación óptima. Trata de explicar por qué dicho conjunto de parámetros es óptimo.

7.2.5. Paso 2 para nivel de dificultad 2

Tras experimentar con los distintos parámetros, encuentra la combinación óptima. Trata de explicar por qué las mejoras introducidas consiguieron mejorar el rendimiento global del programa.

Para rellenar las siguientes tablas, usa un mismo tamaño para el conjunto de datos de entrada. Utiliza 10000 muestras para que los resultados sean significativos.

Uso de registros	Tiempo de GPU (ms)
Sí	
No	

Uso de memoria de constantes	Tiempo de GPU (ms)
Sí	
No	

Simplificación de condicionales	Tiempo de GPU (ms)
Sí	
No	

Grado de desenrollado	Tiempo de GPU (ms)

Tamaño de bloque	Tiempo de GPU (ms)

Optimización trigonometría	Tiempo de GPU (ms)
Sí	
No	

A. Técnicas de optimización

A.1. Tiling

Se conoce como tile un conjunto de datos sobre el que opera una determinada unidad de procesamiento. Un tile puede referirse tanto a datos de entrada como a datos de salida. Por unidad de procesamiento, entendemos un bloque, grid, warp, ... El aumento o disminución en el tamaño del tile puede afectar al rendimiento de dos formas:

- Incrementar el tamaño del tile de salida puede mejorar la reutilización de datos de entrada en memoria compartida; esto ocurre siempre que el tamaño del tile de salida no cause que el tamaño de la entrada sea tan grande como para tener un impacto negativo en la ocupación de un SM.
- Si los datos son demasiado grandes como para almacenarse en memoria compartida, es útil reducir el tamaño de tile. Así, se aprovecha la baja latencia de la memoria compartida y se enmascara el ancho de banda con memoria global.

A.2. Desenrollado de bucles

Se entiende por desenrollado de bucles el acto de ejecutar varias iteraciones del cuerpo de un bucle original en una sola iteración del nuevo bucle desenrollado. La ventaja es la reducción en el número de instrucciones de salto, así como el número de ocasiones en las que la condición de salto debe ser evaluada. Como contrapartida, el desenrollado de bucles puede requerir un número más elevado de registros disponibles (en función del grado de desenrollado), así como una reescritura de mayor o menor complejidad del código original.

Bucle sin desenrollar:

```
for (i = 0; i < N; i++) {
  sum += array[i];
}
```

Bucle desenrollado (factor 2) cuando el número de iteraciones es múltiplo de 2:

```
for (i = 0; i < N; i=i+2) {
  sum += array[i];
  sum += array[i+1];
}
```

Bucle desenrollado (factor 2) cuando el número de iteraciones no es múltiplo de 2:

```
for (i = 0; i < N-1; i=i+2) {
  sum += array[i];
  sum += array[i+1];
}
if (N mod 2 != 0)
  sum += array[N-1];
}
```

A.3. Working size

Entendemos por *working size* la cantidad de trabajo asignada a cada thread. Normalmente se refiere al número de elementos de entrada/salida que calcula cada thread.

A.4. Register spilling

Nos referimos en este caso al uso de memoria compartida como un banco de registros extra para sustituir o ampliar el uso del banco de registros. Como el número de registros disponibles por SM es limitado, cualquier registro extra utilizado por el kernel es automáticamente almacenado en memoria local (DRAM), con la consiguiente penalización en el rendimiento. Así, es muy beneficioso, si es posible, utilizar memoria compartida para evitar esta situación.

A.5. Data prefetching

Como su propio nombre indica, consiste en traer datos desde niveles inferiores en la jerarquía de memoria antes de que sea estrictamente necesario. Normalmente, su utilidad se demuestra en bucles. En CUDA, por ejemplo, resulta de gran utilidad para solapar comunicaciones (accesos a memoria) y cálculo, ya que accesos a memoria global e instrucciones aritméticas independientes pueden realizarse en paralelo. Por ejemplo, en el siguiente ejemplo:

```
for (i = 0; i < N; i++) {
  sum += array[i];
}
```

cada suma espera que sus datos sean cargados desde memoria. En cambio, en el siguiente fragmento de código:

```
temp = array[0];
for (i = 0; i < N-1; i++) {
  temp2 = array[i+1];
  sum += temp;
  temp = temp2;
}
sum += temp;
```

el dispositivo primero carga una instrucción de carga desde memoria para la siguiente iteración, pero realiza la suma en paralelo. En efecto, la suma y la transferencia está solapándose en el tiempo. Por supuesto, esto implica un uso más intensivo del banco de registros, aspecto muy a tener en cuenta dado lo limitado de su tamaño.

A.6. Uso de registros

El uso de registros para almacenar elementos frecuentemente usados puede ahorrar ancho de banda y latencia en los accesos a memoria, incrementando el rendimiento global del sistema. En el siguiente código:

```
for (k = 0; k < N; i++){
  for (i = 0; i < M; i++) {
    sum += v1[k]*v2[i];
  }
}
```

los arrays v1 y v2 se almacenan en memoria global. En total, se realizan $(N \times M) + (N \times M)$ accesos a memoria. Pero se puede observar como v1 es utilizado M veces en el bucle interno. Por tanto, podríamos utilizar un registro temporal para eliminar dicha redundancia en el acceso a memoria:

```
for (k = 0; k < N; i++){
  temp = v1[k];
  for (i = 0; i < M; i++) {
    sum += temp*v2[i];
  }
}
```

reduciendo el número de cargas a $N + (N \times M)$.