Práctica 4

Metodología de la Programación Paralela
Curso 2024/25

Introducción a la Programación de una GPU con CUDA

Miembros del Grupo:
Miguel Saez Martinez miguel.saezm@um.es
Juan Hernández Acosta juan.hernandeza@um.es

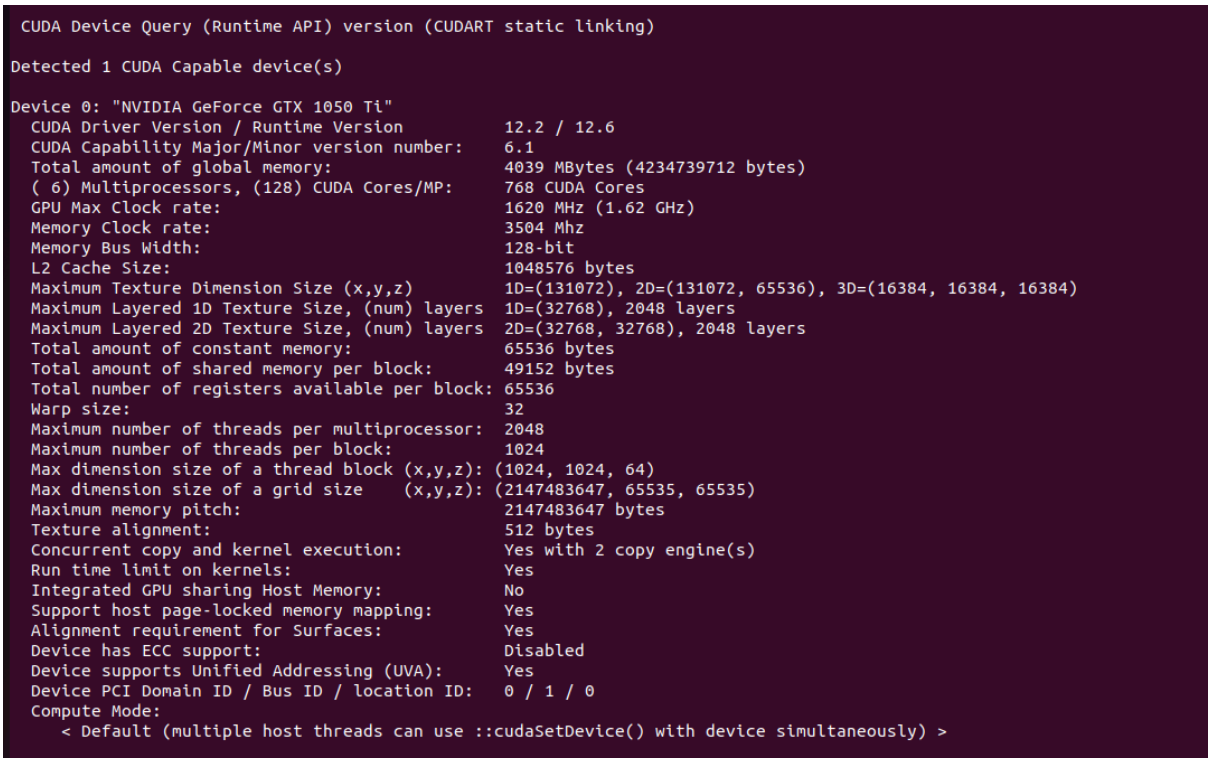
Índice

Cuestión 1.

Ejecuta el programa deviceQuery y confecciona una tabla con la siguiente información para la GPU:

Número de Streaming Multiprocessors (SM).	6
Número de Streaming Processors (SP).	768 (SP=CUDA Cores)
Total de CUDA Cores.	768
Número Máximo de Threads por SM.	2048
Número Máximo de Threads por Bloque.	1024
Cantidad de Memoria Global.	4039 MB
Registros Disponibles por Bloque.	65536
Cantidad de Memoria Compartida por Bloque.	49152 bytes
Dimensiones Máximas del Grid.	(x,y,z): (2147483647, 65535, 65535)
Dimensiones Máximas del Bloque de Threads.	(x,y,z): (1024, 1024, 64)

Indica el modelo de tarjeta gráfica donde se han llevado a cabo los experimentos, por ejemplo, con una captura de los resultados de la ejecución.



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 1050 Ti"
  CUDA Driver Version / Runtime Version          12.2 / 12.6
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 4039 MBytes (4234739712 bytes)
  ( 6) Multiprocessors, (128) CUDA Cores/MP:     768 CUDA Cores
  GPU Max Clock rate:                            1620 MHz (1.62 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Cuestión 2.

Utilizando el programa bandwidthTest:
a) Representa, en varias series de datos dentro de una misma 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".

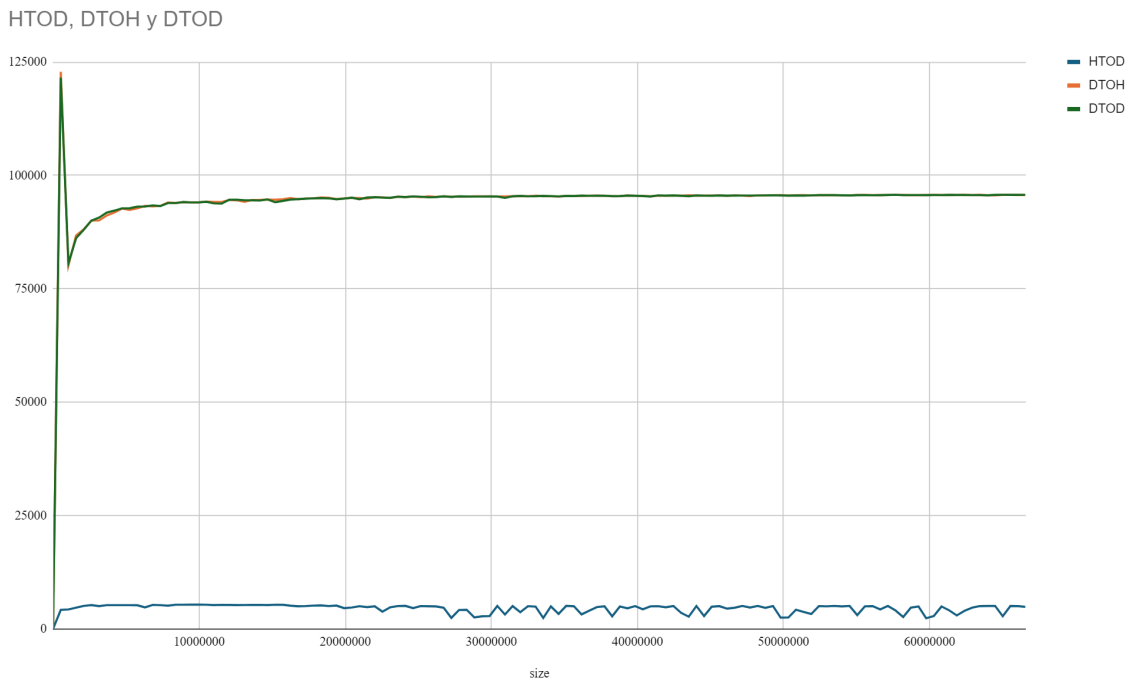
Host a dispositivo (HTOD)
Ejecutamos el comando:
./bandwidthTest --device=0 --memory=pageable --mode=range --start=1024 --end=67108864 --increment=524288 --htod

Dentro del dispositivo (DTOD)
Ejecutamos el comando:
./bandwidthTest --device=0 --memory=pageable --mode=range --start=1024 --end=67108864 --increment=524288 --dtod

Dispositivo a host (DTOH)
Ejecutamos el comando:
./bandwidthTest --device=0 --memory=pageable --mode=range --start=1024 --end=67108864 --increment=524288 --dtoh

Los resultados de las distintas ejecuciones se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p4ej2** o puede acceder directamente usando este enlace:.

Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

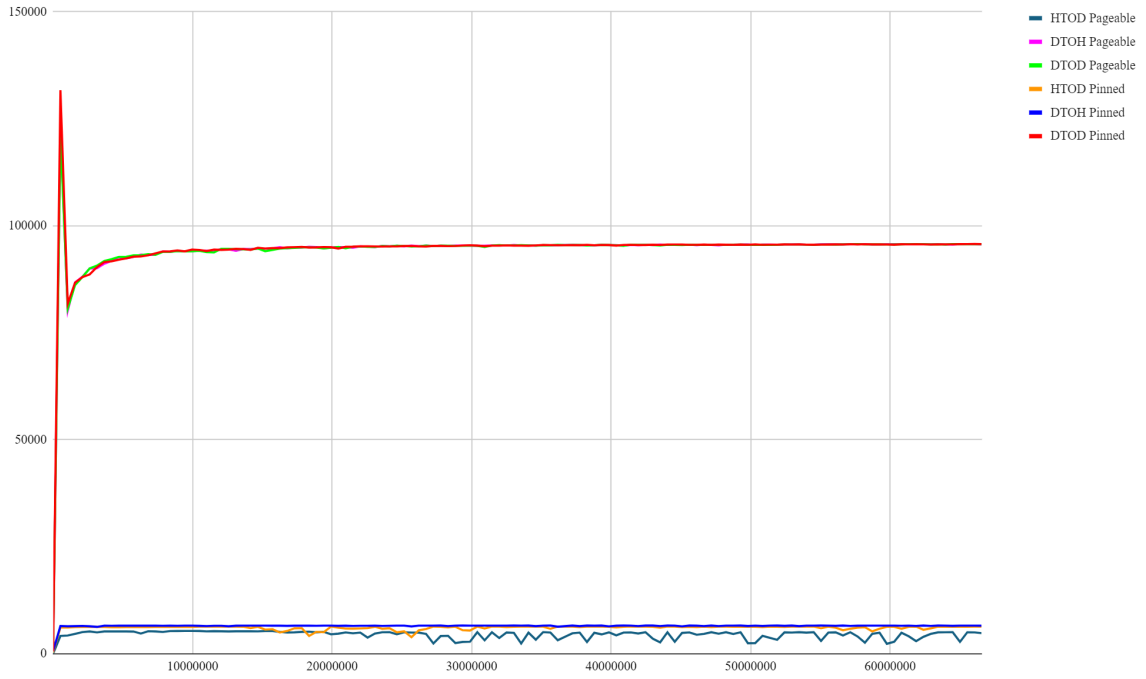


b) Repite el experimento haciendo uso de “pinned memory” y explica las diferencias obtenidas teniendo en cuanta el tipo de memoria y de transferencia realizada.

Usamos el mismo comando que antes pero modificando la parte “--memory=pageable” por “--memory=pinned”

Los resultados de las distintas ejecuciones se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p4ej2** o puede acceder directamente usando este enlace:.

Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx



Además de los resultados de a) y b), para comparar numéricamente el ancho de banda, construye una sola tabla resumen con los valores medios alcanzados tanto para “pageable” como para “pinned” con sus 3 tipos de transferencia correspondientes (un solo valor para cada uno).

Tipo de Transferencia	Pageable Memory (GB/s)	Pinned Memory (GB/s)
Host a Dispositivo	4494.510156	6055.010156
Dentro de Dispositivo	94244.492969	94337.725781
Dispositivo a Host	94258.703125	6495.103125

Cuestión 3.

Ejecuta el programa cudaTemplate con diferentes tamaños de grid y bloques de threads
y analiza los resultados obtenidos.

Los resultados de las distintas ejecuciones se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p4ej3_2D** o puede acceder directamente usando este enlace:.

Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx

a) ¿Qué hace el código del kernel?

El programa cudaTemplate_kernel.cu tiene una función que básicamente realiza una operación de copia de datos desde la memoria constante a la memoria global, pasando por la memoria compartida. Para ello sigue los siguientes paso:

1º Accede a los datos globales de memoria (el gid_d en la memoria del dispositivo) usando el identificador global tidg, que está compuesto por el índice del bloque (blockIdx) y el índice del hilo dentro del bloque (threadIdx).

2º Los datos correspondientes a cada hilo se copian en memoria compartida (shared_mem) para que los hilos de un mismo bloque puedan compartirlos de manera eficiente.

3º Sincroniza los hilos.

4º Realiza un cálculo en memoria compartida: el valor de shared_mem[tidb] se incrementa con un valor calculado como la suma del índice global tidg y un valor constante tomado de la memoria constante const_d (dependiendo del índice modulado por CT_MEM_SIZE).

5º Sincroniza nuevamente los hilos

6º El valor calculado en la memoria compartida se copida de vuelta a la memoria global gid_d.

b) 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 kernel utiliza memoria compartida para almacenar los datos de gid_d localmente en cada bloque. La memoria compartida se usa para mejorar la eficiencia, ya que el acceso a la memoria compartida es mucho más rápido que el acceso a la memoria global.

Tamaño de la memoria compartida:

El tamaño de la memoria compartida se calcula como block.x * block.y * sizeof(int). Esto representa el número total de elementos en memoria compartida que un bloque necesita (una cantidad de int por cada hilo dentro del bloque).

Se utiliza assert(shared_mem_size <= SH_MEM_SIZE) para asegurar que el tamaño de la memoria compartida no supere el límite definido por SH_MEM_SIZE, que es de 16 KB.

Verificación del tamaño:

El cálculo del tamaño de la memoria compartida (shared_mem_size) depende de las dimensiones del bloque (block.x * block.y). Si el tamaño de un bloque es mayor que lo que puede soportar la memoria compartida (16 KB), entonces el kernel fallará.

El tamaño del bloque (en términos de memoria compartida) debe ser menor o igual a 16 KB. Si no es así, tendrías que reducir el tamaño de los bloques o la cantidad de datos por hilo.

Modificación si el tamaño no es correcto:

Si el tamaño de la memoria compartida en tu configuración excede el límite de 16 KB, puedes reducir las dimensiones del bloque (dim_block_x y dim_block_y) o la cantidad de datos procesados por hilo.

Por ejemplo, si el número de hilos en un bloque es grande, puedes reducir el número de hilos por bloque (como cambiar dim_block_x y dim_block_y).

c) ¿Podría eliminarse la primera llamada a __syncthreads() en el kernel?

Esta llamada sincroniza todos los hilos dentro de un bloque después de haber copiado los valores de gid_d en shared_mem. Si se elimina, los hilos podrían intentar modificar la memoria compartida antes de que todos los hilos hayan cargado sus datos. Por tanto no se puede eliminar esta primera llamada a __syncthreads() .

¿Y la segunda?

Después de modificar los valores en shared_mem, necesitamos asegurar que todos los hilos dentro del bloque hayan completado sus cálculos antes de escribir los resultados de vuelta en la memoria global. Si se elimina esta sincronización, los hilos podrían intentar escribir en la memoria global mientras otros aún están trabajando, lo que podría provocar inconsistencias en los resultados.

d) Modifica el código para que el kernel utilice bloques de threads tridimensionales (sin modificar las dimensiones del grid), es decir, para que tenga en cuenta la coordenada z.

Los resultados de las distintas ejecuciones se encuentran dentro del documento **Resultados_Prácticas_MPP_EficienciaOrdenada** en la hoja **p4ej3_3D** o puede acceder directamente usando este enlace: *Resultados_Prácticas_MPP_EficienciaOrdenada.xlsx*

Cuestión 4.

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

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

	16	32	64	128	256
10.000	0.2528	-	-	-	-
20.000	0.3676	0.2528	-	-	-
30.000	0.2910	0.2839	0.2874	-	-
40.000	0.3393	0.3358	0.3564	0.3257	-
50.000	0.4047	0.3707	0.3779	0.3700	0.3792

Puede observarse como, generalmente, el tiempo de ejecución mejora al aumentar tanto el tamaño como el número de threads. Sin embargo, no se aprecia una mejora lineal ya que los cambios en los tiempos de ejecución parecen estabilizarse con un número bastante bajo de threads. Esto sugiere que el problema se vuelve lo suficientemente grande como para beneficiarse de la paralelización, pero después de un cierto umbral de threads (alrededor de 32-64), añadir más threads no mejora el rendimiento.

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

El programa vectorAdd_kernel.cu tiene una función que realiza la suma de dos vectores X e Y, almacenando el resultado en el vector V. El tamaño de los tres vectores es el mismo, y el kernel se ejecuta en paralelo en la GPU, donde cada hilo se encarga de sumar un elemento correspondiente de los vectores X e Y. Para ello:

1º A Cada hilo se le asigna una posición específica de los vectores X e Y equivalente a su tid.

2º Cada hilo con una posición no mayor al tamaño del vector accede a los elementos correspondientes de X e Y, los suma y almacena el resultado en V.

c) 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?

En el programa vectorAdd.cu, el número de bloques en el grid se calcula con la fórmula int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock, donde numElements es el total de elementos en el vector y threadsPerBlock es el número de threads por bloque. Esta fórmula asegura que siempre haya suficientes bloques para cubrir

todos los elementos del vector, incluso si numElements no es un múltiplo de threadsPerBlock. Al sumar threadsPerBlock - 1 antes de dividir, se realiza un redondeo hacia arriba, lo que garantiza que se lance al menos un bloque adicional si hay elementos restantes que no encajan en un número entero de bloques. Los threads que caen fuera del rango de numElements no realizarán ninguna operación en el kernel debido a la verificación if (tid < numElements), evitando así accesos fuera de los límites del vector.

d) Realiza una breve búsqueda bibliográfica sobre el concepto de warp, la unidad de planificación de hilos de CUDA. Explica cómo puede afectar al rendimiento que el bloque de threads no sea múltiplo del tamaño del warp. ¿Puede haber más causas que provoquen la caída del rendimiento relacionadas con la planificación de las ejecuciones realizada por el hardware de la GPU?

El rendimiento de una aplicación CUDA puede verse afectado significativamente si el tamaño del bloque de threads no es una potencia de 2, y esto está estrechamente relacionado con el concepto de "warp" en la arquitectura CUDA.

Un "warp" es la unidad básica de ejecución en una GPU NVIDIA. Consiste en un conjunto fijo de threads que son ejecutados en paralelo por la GPU. El tamaño de un warp es típicamente de 32 threads en la mayoría de las arquitecturas CUDA modernas.

Utilizar un tamaño de bloque que sea potencia de 2 en CUDA mejora el rendimiento ya que se alinea con el tamaño de los warps, que son grupos de 32 threads que ejecutan instrucciones simultáneamente. Si el tamaño del bloque no es potencia de 2, se pueden crear warps parcialmente llenos, resultando en una utilización ineficiente de la GPU, ya que algunos threads del warp estarían inactivos. Además, un tamaño de bloque optimizado para warps facilita el acceso coalescente a la memoria, donde los threads acceden a direcciones contiguas, reduciendo la divergencia en la ejecución del warp y mejorando así el rendimiento general.

Cuestión 5.

Incluir un apartado final con las conclusiones generales y valoración personal sobre el trabajo realizado.

En este trabajo hemos realizado un análisis detallado de varios aspectos de la programación CUDA usada en las tarjetas gráficas Nvidia. Ejecutando distintas pruebas sobre el ancho de banda de las transferencias o sumas de vectores mientras modificamos el tamaño y dimensiones de los bloques o cambiando entre memoria pinned o pageable hemos podido entender cómo estas diferentes configuraciones impactan sobre la eficiencia de la GPU. Esto ha ayudado a reforzar nuestro entendimiento sobre las técnicas de optimización para cálculos paralelos.

En un tono más personal podemos afirmar que esta última práctica ha sido bastante más ligera que las anteriores pues no ha requerido tanto tiempo para completarla como las anteriores, especialmente a la hora de la ejecución, y nos ha resultado bastante interesante poder ver las capacidades y posibilidades de los equipos con los que trabajamos a diario.