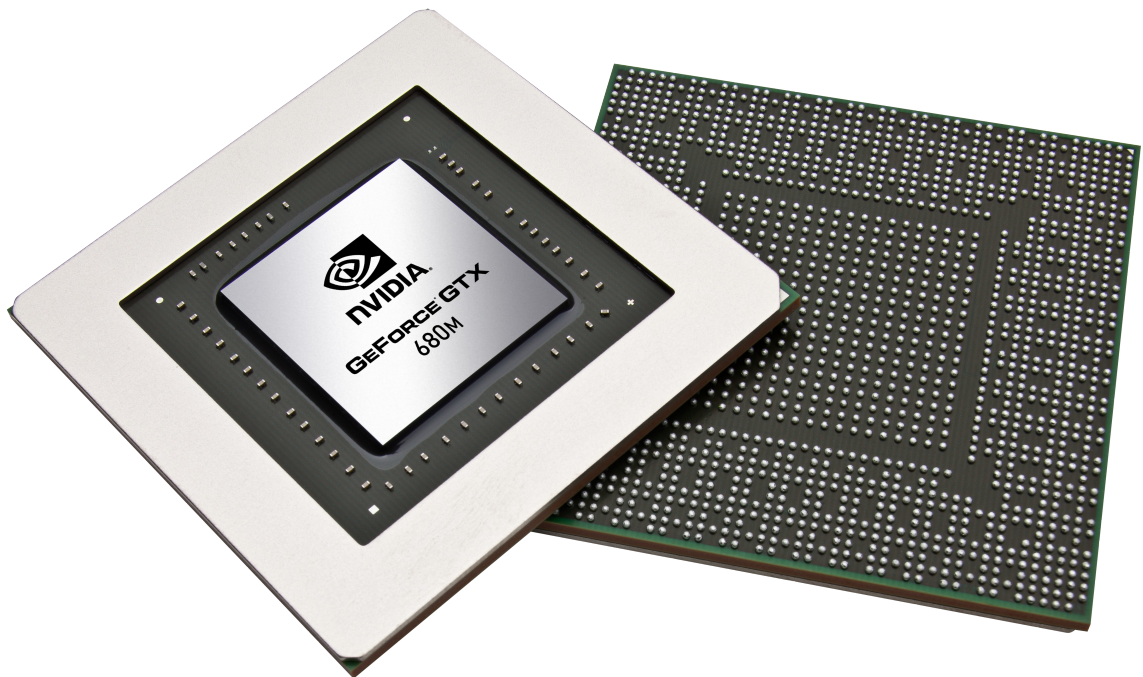


# Práctica 7

## Programación de GPUs. Kernel



**Héctor Garbisu Arocha**

Curso 2015/16

Métodos Numéricos para la Computación

Grado en Ingeniería Informática

Escuela de Ingeniería Informática

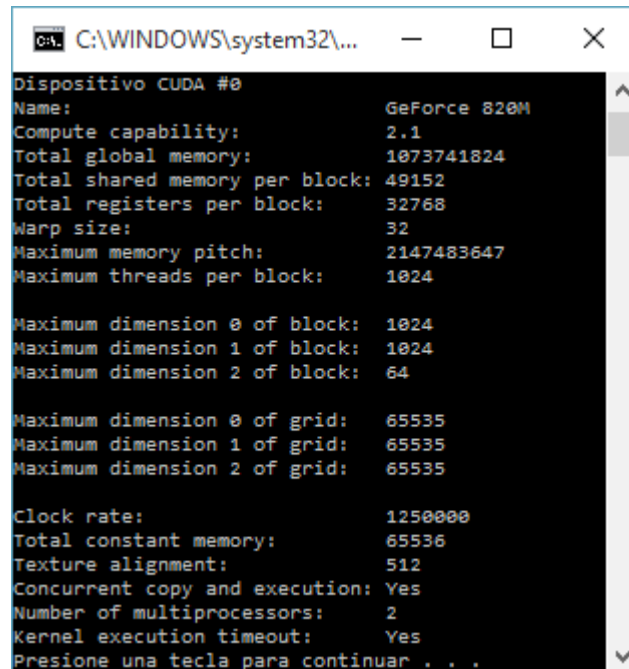
Universidad de Las Palmas de Gran Canaria

# Índice

1. miGPU .....	pág. 3
2. Suma de Matrices .....	pág. 4
3. Pinned Memory .....	pág. 7

## 1. Tarea 1. MiGPU

El primer ejercicio consiste en tomar contacto con CUDA preguntando qué características tienen los dispositivos compatibles del sistema. El programa suministrado imprime toda la información relativa al objeto `cudaDeviceProp` que se obtiene con `cudaGetDeviceProperties()`.



```
Dispositivo CUDA #0
Name: GeForce 820M
Compute capability: 2.1
Total global memory: 1073741824
Total shared memory per block: 49152
Total registers per block: 32768
Warp size: 32
Maximum memory pitch: 2147483647
Maximum threads per block: 1024

Maximum dimension 0 of block: 1024
Maximum dimension 1 of block: 1024
Maximum dimension 2 of block: 64

Maximum dimension 0 of grid: 65535
Maximum dimension 1 of grid: 65535
Maximum dimension 2 of grid: 65535

Clock rate: 1250000
Total constant memory: 65536
Texture alignment: 512
Concurrent copy and execution: Yes
Number of multiprocessors: 2
Kernel execution timeout: Yes
Presione una tecla para continuar . . .
```

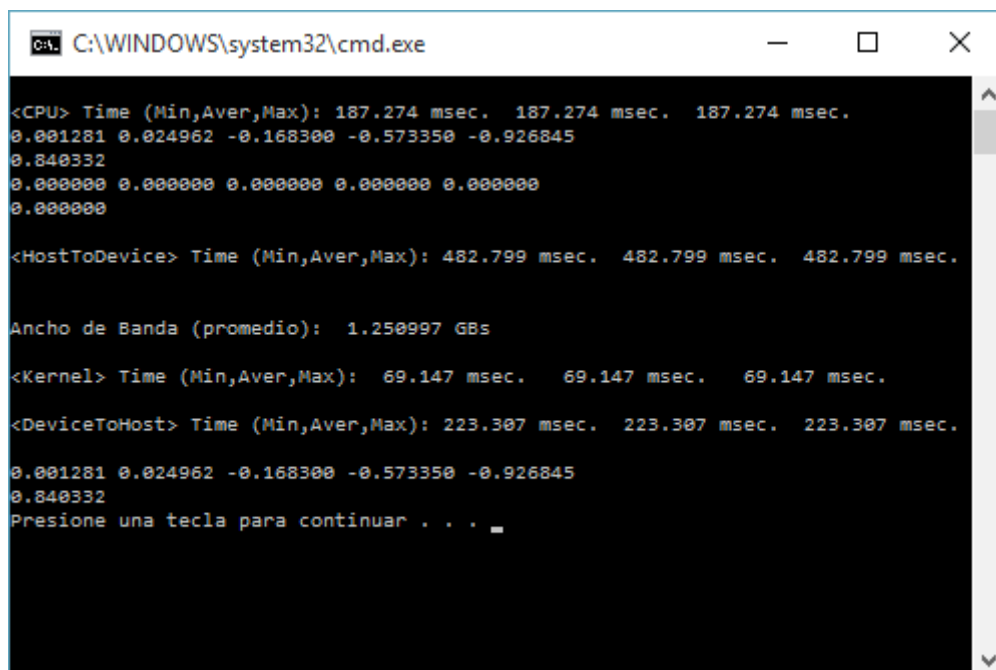
En mi portátil, donde hice la prueba, se ve que no hay una tarjeta gráfica de muy altas prestaciones (1.25Mh y 1Gb de memoria) pero es compatible con CUDA, que es lo que interesa.

## 2. Tarea 2. Suma de Matrices

En este ejercicio utilizaremos la GPU para hacer una operación matricial con un alto grado de paralelismo para comprobar la eficiencia del GPGPU. La operación es la suma de  $a*A+b*B$ , siendo  $a$  y  $b$  escalares, y  $A$  y  $B$  matrices.

Las tres operaciones se pueden hacer celda a celda, así que el kernel del programa consistirá en hacerlas en una sola celda cuya posición dependerá de información propia de cada hilo.

Las acciones que debe hacer el programa principal son crear las matrices  $A$ ,  $B$ ,  $C$ , hacer el cálculo con CPU como control, reservar la memoria de la GPU para las matrices, transferir las matrices a la GPU, dimensionar el Grid y los bloques, lanzar el kernel y volver a transferir los datos. Cada operación será cronometrada.



```
C:\WINDOWS\system32\cmd.exe

<CPU> Time (Min,Aver,Max): 187.274 msec. 187.274 msec. 187.274 msec.
0.001281 0.024962 -0.168300 -0.573350 -0.926845
0.840332
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000

<HostToDevice> Time (Min,Aver,Max): 482.799 msec. 482.799 msec. 482.799 msec.

Ancho de Banda (promedio): 1.250997 GB/s

<Kernel> Time (Min,Aver,Max): 69.147 msec. 69.147 msec. 69.147 msec.

<DeviceToHost> Time (Min,Aver,Max): 223.307 msec. 223.307 msec. 223.307 msec.
0.001281 0.024962 -0.168300 -0.573350 -0.926845
0.840332
Presione una tecla para continuar . . .
```

Se puede comprobar cómo en mi portátil la transferencia ocupa una porción de tiempo mucho mayor, llegando a ser más costosa que realizar toda la operación en la propia CPU. El tiempo del kernel es mucho menor, como esperábamos, pero los tiempos de transferencia hacen que no valga la pena para esta operación y estos tamaños de matriz.

Para calcular la tasa de ocupación de los Warps debemos usar la herramienta disponible en la hoja de cálculo escondida por los directorios de la instalación de CUDA. Los campos en verde se pueden rellenar con la información obtenida en el primer ejercicio

1.) Select Compute Capability (click):	2,1
1.b) Select Shared Memory Size Config (bytes)	49152

Los campos en naranja los podemos rellenar calculando los valores, que dependen del programa y de la información anterior.

- Threads per Block: Es la cantidad de hilos que habrá en cada bloque, que en nuestro programa es  $32 \times 16 = 512$
- Registros por hilo: No se puede saber a priori
- Shared Memory Per Block: 0 porque los hilos son independientes.

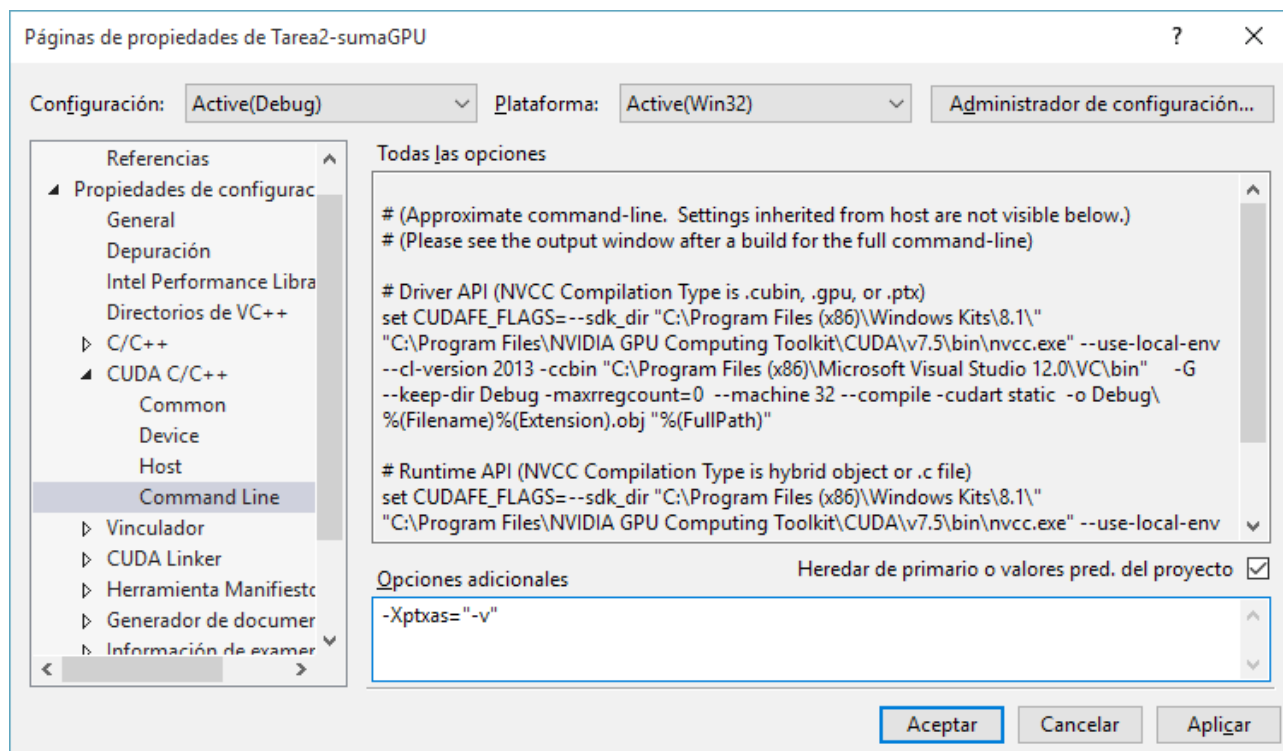
Para calcular el número de registros que se usan realmente (no el máximo) hay que tener en cuenta que el compilador hace una optimización y no podemos saber el resultado hasta después de compilar el código.

Para obtener el dato debemos compilar con la opción `-Xptxas="-v"`. Que en Visual Studio 2013 se hace entrando en Propiedades del proyecto > Propiedades de Configuración > Cuda C/C++ > Command Line, y escribiendo en "opciones adicionales" la línea `-Xptxas="-v"`.

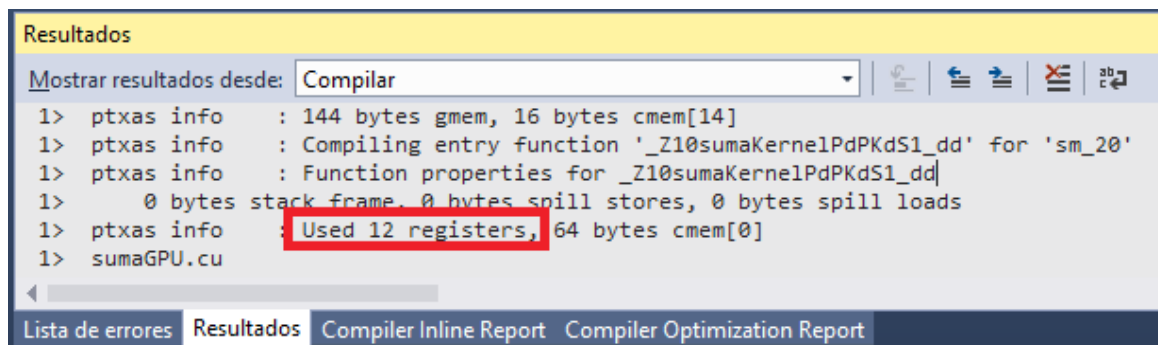
Ahora, al compilar (no hace falta ejecutar) nos dice el número de registros por hilo entre otras cosas.

(Fuente:

<https://devtalk.nvidia.com/default/topic/494306/how-to-determine-number-of-register-per-thread-how-to-determine-number-of-register-per-thread-from-a/>)



Tras la compilación:

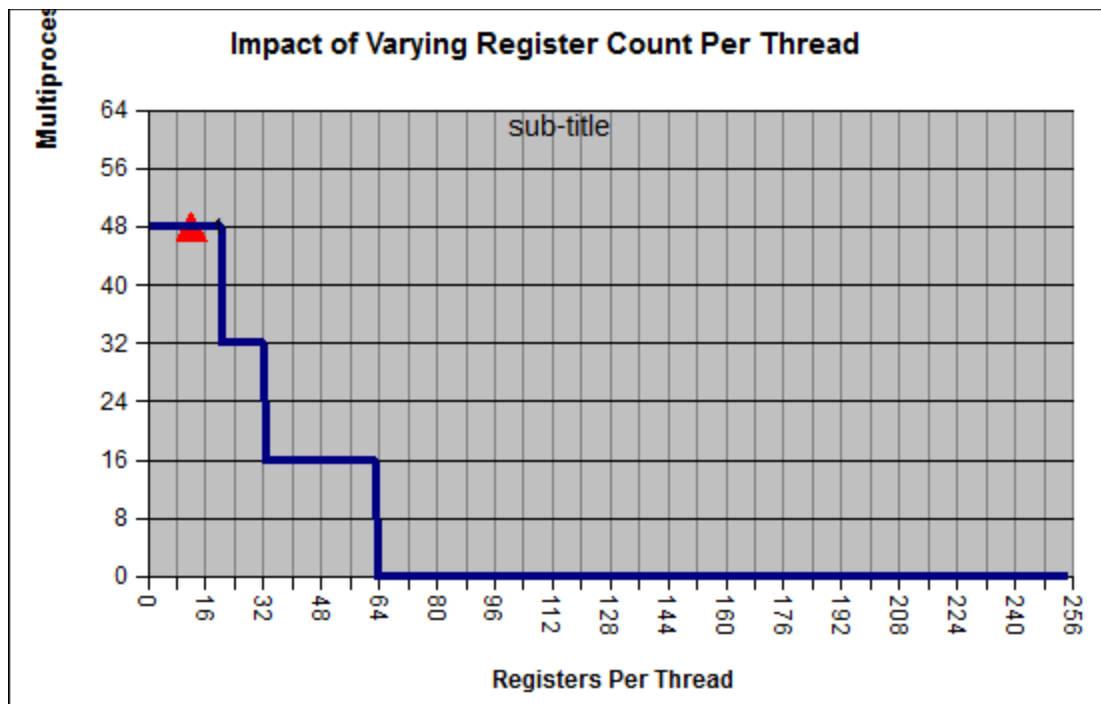
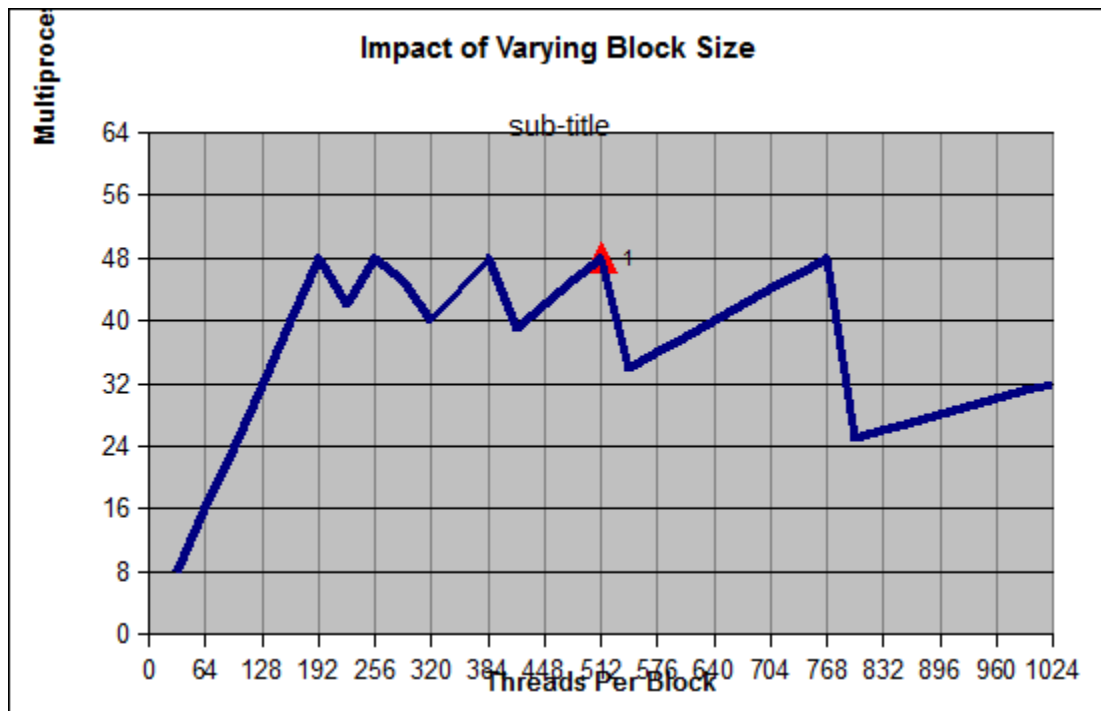


De modo que:

**2.) Enter your resource usage:**

Threads Per Block	512
Registers Per Thread	12
Shared Memory Per Block (bytes)	0

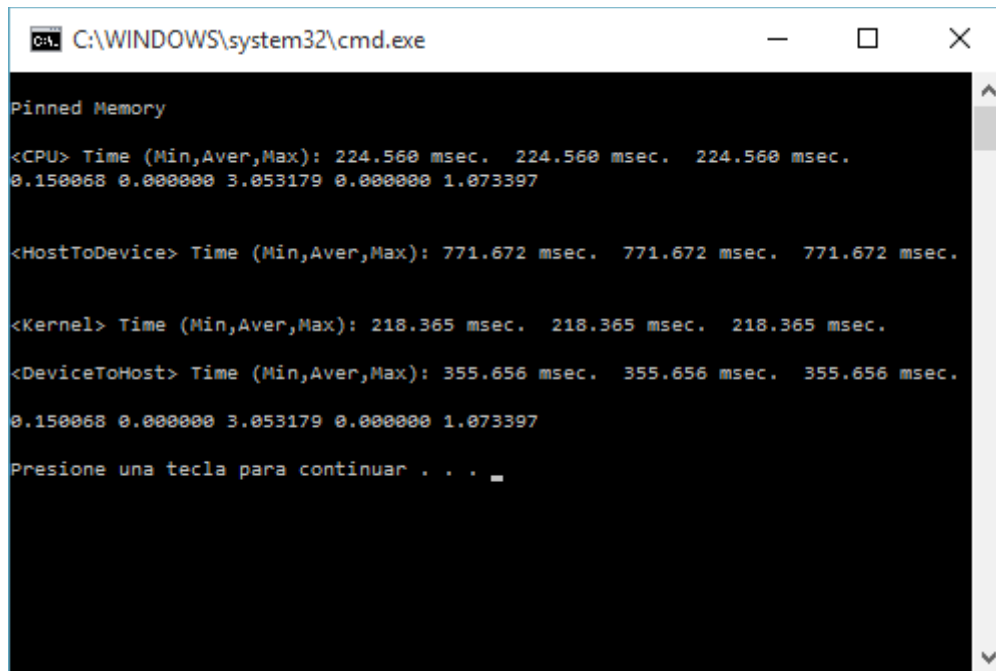
Ahora la hoja de cálculo nos confirma que teníamos unos de los posibles valores para una tasa de ocupación óptima.



### 3. Tarea 3. Pinned Memory

En esta tarea compararemos un mismo programa, usando memoria paginada y memoria “pinned”. Ejecutamos el programa que muestra los tiempos como se hacía en el ejercicio 2, pero unas veces usando memoria pinned y otras usando la normal, variando la macro del código.

Ejemplo de ejecución:



```
C:\WINDOWS\system32\cmd.exe

Pinned Memory

<CPU> Time (Min,Aver,Max): 224.560 msec. 224.560 msec. 224.560 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

<HostToDevice> Time (Min,Aver,Max): 771.672 msec. 771.672 msec. 771.672 msec.

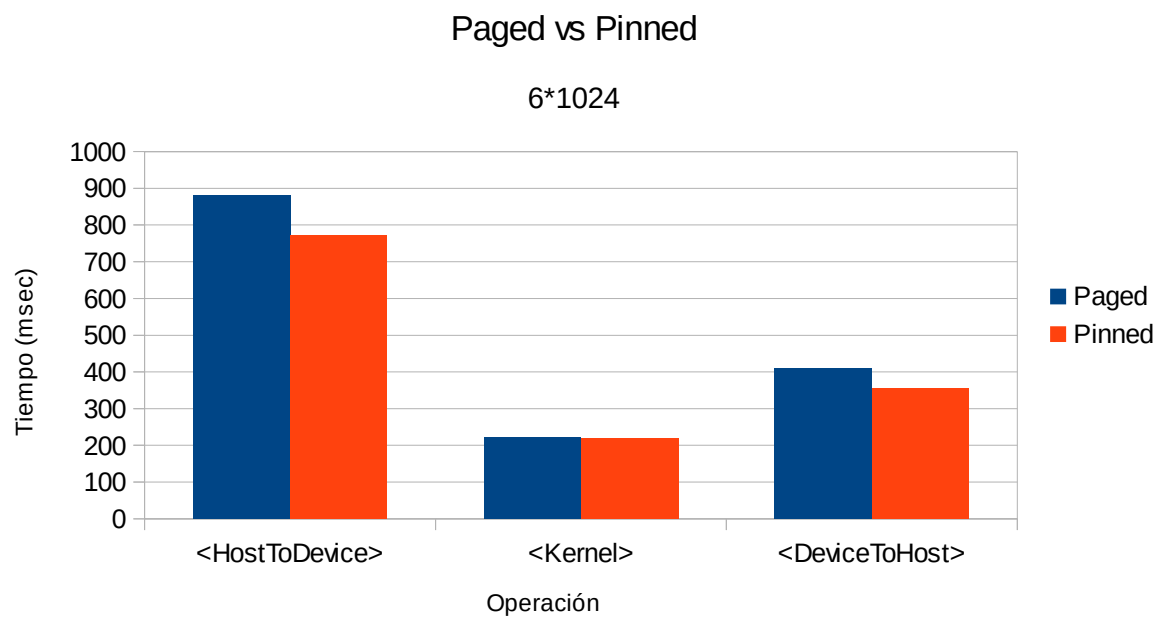
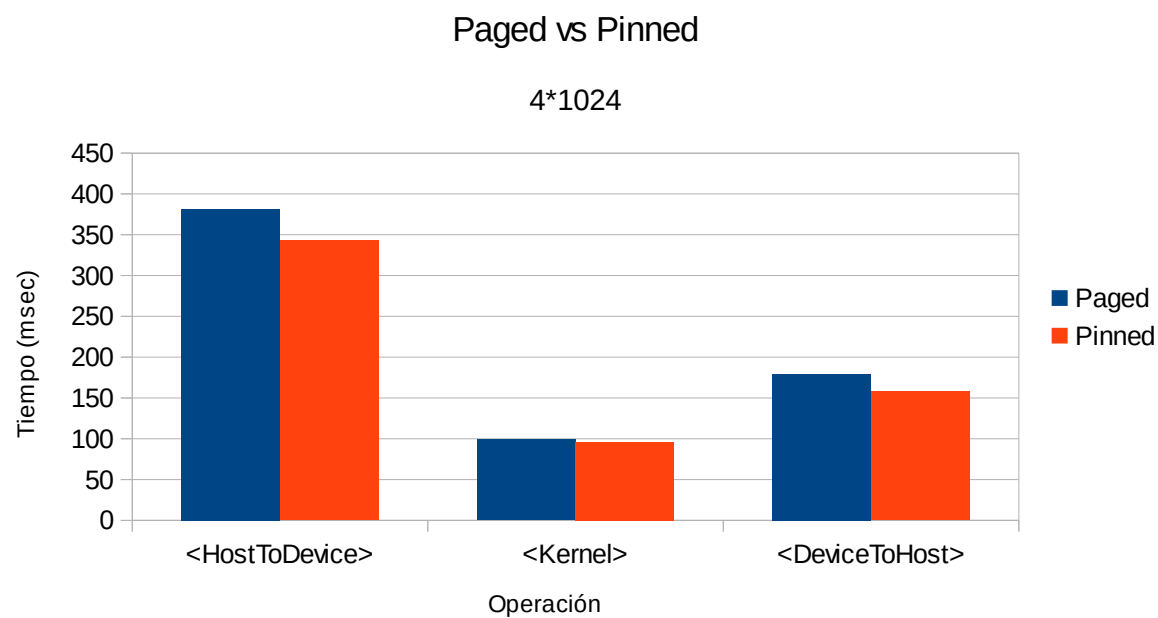
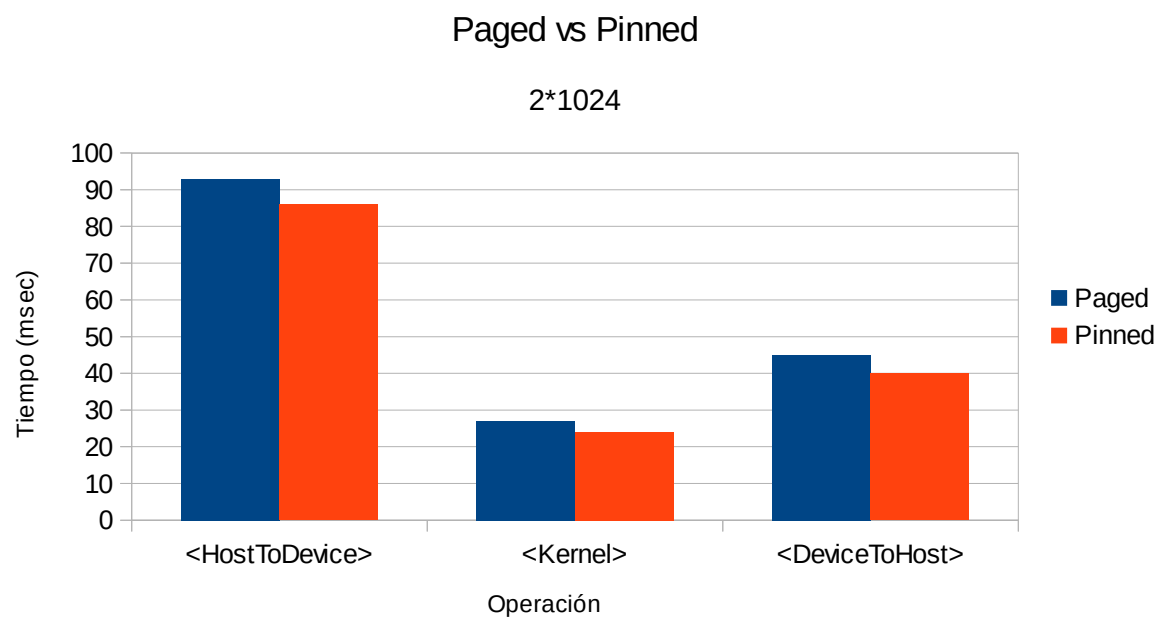
<Kernel> Time (Min,Aver,Max): 218.365 msec. 218.365 msec. 218.365 msec.

<DeviceToHost> Time (Min,Aver,Max): 355.656 msec. 355.656 msec. 355.656 msec.
0.150068 0.000000 3.053179 0.000000 1.073397

Presione una tecla para continuar . . .
```

A continuación se muestran los resultados para tres tamaños diferentes de matriz (2, 4 y 6 millares de filas), los dos tipos de memoria y tres tiempos (CPU → GPU, Kernel, GPU → CPU)

Tamaño	2*1024			4*1024			6*1024		
Operación	→	Kernel	←	→	Kernel	←	→	Kernel	←
Pinned (ms)	86	24	40	343	96	158	772	218	356
Paged (ms)	93	27	45	381	99	179	882	221	409





### Conclusión:

Los ejemplos de prueba son suficientemente grandes como para mostrar una ligera ventaja de la memoria pinned frente a la ordinaria. No he podido replicar una diferencia tan grande como la que se muestra en la imagen ejemplo del enunciado de la práctica.

Estas muestras son escasas para sacar conclusiones, realmente. Sin embargo, podemos ver cómo el tiempo de kernel se va igualando poco a poco a medida que aumenta el tamaño de la matriz, mientras que la diferencia de los tiempos en la transferencia de datos CPU  $\leftrightarrow$  GPU aumenta. Si fuera a sacar una conclusión sobre el uso de pinned memory sobre memoria paginada convencional, diría que es mejor usarla en cualquier caso donde los datos que se transfieren a la GPU sean suficientemente grandes, y que la complejidad del kernel no influye.

Para sacar mejores conclusiones habría que hacer pruebas más consistentes (ejecutar varias veces cada caso) y probar con más capacidad de memoria (yo solo dispongo de 1Gb).