Práctica 1. CUDA.

¿Qué tienes que haber aprendido antes de empezar esta práctica?

Las diapositivas 1 a 37 de “Introducción a CUDA” explicadas en clase.

También debes tener conocimientos de programación en C y conocimientos básicos de Linux.

¿Cuáles son las principales competencias que debes adquirir con esta práctica (que son parte de las competencias de las que serás evaluado)?

1. Saber compilar y ejecutar un programa escrito en CUDA C.
2. Saber usar e interpretar la información ofrecida por el "visual profiler" nvvp (tiempos, anchos de banda...).
3. Saber realizar una comprobación básica de posibles errores al llamar a una función de la API de tiempo de ejecución de CUDA o a un *kernel*.
4. Elegir una configuración adecuada de la ejecución de un *kernel*:

   Elegir bien los valores de (blockDim.x, blockDim.y, blockDim.z) y de (gridDim.x, gridDim.y, gridDim.z).
5. Saber cambiar el *kernel* al cambiar la configuración de su ejecución.
6. Saber explicar las diferencias de tiempos entre diferentes programas que resuelven el mismo problema.

¿Qué documentación puedes consultar?

Hay mucha documentación en internet, pero para esta práctica debe ser suficiente con consultar las diapositivas de la “Introducción a CUDA”. Además, es recomendable solo consultar estas diapositivas ya que será la única documentación a la que tendrás acceso en el examen de CUDA.

Práctica 1

1. Descarga del espacio virtual de la asignatura el archivo `suma-vectores1.cu`. Observa el código, compila el programa y ejecuta el programa resultante de la compilación (por defecto, usa siempre como nombre del programa ejecutable el mismo que el fuente sin la extensión `cu`, en este caso, `suma-vectores1`). [competencia 1]

2. Abre el analizador de rendimiento visual (*NVIDIA Visual Profiler*) ejecutando `nvvp`, crea una nueva sesión ([File][New Session]) y selecciona el archivo ejecutable `suma-vectores1`. Después [Next>][Finish] [competencia 2]

a. Aprende, con las explicaciones y la ayuda del profesor, a familiarizarte con esta herramienta. Además, practica y recuerda estos dos detalles que pueden interesarte:

i. Para ampliar una zona en la ventana principal (*Timeline View*): mientras se pulsa [Ctrl] se arrastra el ratón por la zona.

ii. Para ajustar la línea de tiempos a la ventana pulsar el icono de lupa con el símbolo ±.

b. Queremos anotar en la siguiente tabla los tiempos que hemos obtenido en nuestra sesión con `nvvp`. Para ello, selecciona el tiempo que te interese en la ventana principal y mira el resultado en la vista *Properties* (tb. hay información en la vista *Details*).

	Tiempo		
Tiempo total de la sesión			
Kernel			
cudaMalloc DA			
cudaMalloc DB			
cudaMalloc DC			
	Latencia	Tiempo dedicado	Ancho de banda
cudaMemcpy DA ←HA			
cudaMemcpy DB ←HB			
cudaMemcpy HC ←DC			

c. Siguiendo los comentarios del profesor, observa la información que nos ofrece la vista *Analysis*. Del mismo modo, observa la información sobre las propiedades de la GPU que nos ofrece `nvvp`.

3. Aparentemente, el primer cudaMalloc() tarda más. Ello se debe a que, por ser la primera llamada a una función de la API, provoca la creación del contexto CUDA (el equivalente en la GPU a un proceso de la CPU). El tiempo dedicado a crear dicho contexto se incluye en el de este primer cudaMalloc(). [competencia 2]

a. Para separar el tiempo realmente empleado por cudaMalloc() del empleado en crear el contexto, añade al principio del programa principal la llamada cudaFree(0). Ahora será esta llamada la que provoque la creación del contexto CUDA.

b. Para quitar otros tiempos superfluos comenta el último for del programa principal una vez que está comprobado que el programa suma los vectores en el *device* correctamente.

c. Al código resultante tras estos dos cambios denomínalo `suma-vectores1a.cu`, compílalo y analiza el ejecutable con `nvvp`. ¿Cuál es la principal diferencia de tiempos con respecto a `suma-vectores1`?

.....

4. Realiza en `suma-vectores1a.cu` las siguientes modificaciones:[competencia 3]

a. Modifica lo necesario para:

i. Comprobar si se genera error al hacer cada una de las llamadas a cudaMalloc() y a cudaMemcpy().

ii. Verificar si la llamada al *kernel* produce error.

iii. Que, si en alguno de los casos anteriores se produce error, el programa use la función cudaGetErrorString() para imprimir el tipo de error producido.

b. Vuelve a habilitar el último for que comentaste.

c. Denomina `suma-vectores1b.cu` al programa resultante, compílalo y ejecútalo. Cambia el valor del número de hebras, N, a 600 y vuelve a compilar y ejecutar. ¿Hemos detectado algún error durante la ejecución?

.....

5. Modifica el programa `suma-vectores1b.cu` para obtener el programa `suma-vectores2.cu` que solo utiliza un bloque de una hebra. Modifica en consonancia el *kernel* para que el programa completo siga sumando los vectores. Comprueba que funciona correctamente. [competencias 4 y 5]

6. Modifica el programa `suma-vectores2.cu` para obtener el programa `suma-vectores3.cu` que utiliza N bloques de una hebra. Comprueba que funciona correctamente. [competencias 4 y 5]

7. [Antes de empezar con este punto, para reducir el riesgo de bloqueo de `nvvp` y de todo el ordenador, **cierra nvvp y vuelve a abrirlo**. Otra medida recomendable cuando se abren varias sesiones es poner un tiempo en "Execution timeout" (p. ej., 20 segundos) al iniciar una sesión para que la ejecución realizada por `nvvp` no sobrepase ese tiempo, pero esto no garantiza que `nvvp` no se bloquee].

Usando `nvvp`, compara los tiempos empleados por el *kernel* en los programas `suma-vectores1b`, `suma-vectores2` y `suma-vectores3`. Busca una explicación a los tiempos resultantes. [competencia 6]

Programa	Tiempo de ejecución del *kernel*
`suma-vectores1b`	
`suma-vectores2`	
`suma-vectores3`	

Explicación: .....

.....

.....

.....

.....

.....

8. Supongamos que queremos sumar dos vectores de 100000 componentes:

a. El programa `suma-vectores3` no sirve ya que, con un *grid* unidimensional, el máximo número de bloques que podemos usar es 65535.

b. Distintos motivos (entre ellos la comparación de tiempos anterior) recomiendan que los bloques no sean de una sola hebra y que el número de hebras por bloque sea múltiplo de 32. Además, con bloques grandes podremos sumar vectores más grandes.

c. Modifica el programa `suma-vectores3.cu` para obtener el programa `suma-vectores4.cu` que permita nuestro objetivo (la suma de vectores de N enteros) para valores de N mayores que 65535. [competencias 4 y 5]

   i. Para el número de hebras por bloque (tb) usa múltiplos de 32 (512, que es el valor máximo, es una buena opción). Se asumen bloques unidimensionales.

   ii. Observa que el número dg de bloques en el *grid* (que se asume unidimensional), será $\mathrm{dg} = \left\lceil \frac{N}{tb} \right\rceil = \left\lfloor \frac{N+tb-1}{tb} \right\rfloor$ (que en C se calcula con $(N + tb - 1)/tb$).

   iii. Observa también que el número total de hebras del *grid* será $\left\lceil \frac{N}{tb} \right\rceil \times tb \geq N$.

   iv. Al programar el *kernel* debes asociar a cada hebra del *grid* un índice distinto sin que falte ningún entero entre 0 y N-1, ambos inclusive, para que la suma se realice con todos los componentes de los vectores. Pero como el número de hebras puede ser mayor que N, puede haber índices fuera del rango [0, N-1] con los que hay que evitar intentar acceder a los vectores.

   v. Comprueba que `suma-vectores4` funciona correctamente.

   vi. Sube `suma-vectores4.cu` a Campus Virtual [participación].

9. Para N mayor que 65535*512 el programa `suma-vectores4` no sirve. Queremos elaborar el programa `suma-vectores.cu` que sirva para valores de N mayores. [competencias 4 y 5]

a. El *grid* sigue siendo unidimensional. El número de bloques será $\mathrm{dg} = \mathrm{mín}\left(\left\lceil\frac{N}{tb}\right\rceil, 65535\right)$ —aunque podría valer un valor menor.

b. En general, el vector se dividirá en fragmentos de tantos componentes como hebras tiene el *grid*, es decir, $gridDim.x \times blockDim.x$ componentes (habrá un solo fragmento si $gridDim.x \times blockDim.x \geq N$, es decir, si el número de hebras del *grid* no es menor que el número de componentes del vector).

c. En el *kernel*, cada hebra debe recorrer, mediante un for, los índices del vector que ocupen la misma posición en cada fragmento: el primer índice será el mismo que en el programa `suma-vectores4`, donde solo había un fragmento (blockIdx.x * blockDim.x + threadIdx.x) y el índice se incrementará en tantas posiciones como tiene un fragmento (stride = gridDim.x * blockDim.x).

d. Prueba el programa `suma-vectores.cu` con distintos valores de N (500000, 200…) para comprobar que funciona correctamente.

   i. Observa que para valores muy grandes (p. ej., 1 millón, se produce violación de segmento lo cual se debe al excesivo tamaño de los vectores en el host). Esto supone que solo probamos valores menores que 65535×512, por lo que las hebras solo realizan una iteración por el bucle for del *kernel*.

   ii. Para probar mejor el *kernel* es importante que se hagan varias pasadas por el for, lo cual se puede conseguir con un *grid* con menos hebras. Modifica el programa principal para elegir un tamaño de bloque de 32 y un número de bloques por *grid* $\mathrm{dg} = \mathrm{mín}\left(\left\lceil\frac{N}{tb}\right\rceil, 1024\right)$. Prueba de nuevo el programa con valores de N mayores y menores que $1024 \times 32 (= 32768)$. Observa que el número de iteraciones por el for del *kernel* es $\left\lceil\frac{N}{32768}\right\rceil$ para unas hebras y puede ser $\left\lfloor\frac{N}{32768}\right\rfloor$ para otras, dado que, normalmente, el último fragmento del vector tendrá menos componentes que hebras hay en el *grid*.