Práctica evaluable – CURSO 2022/2023

Los siguientes ejercicios, relacionados con la programación en CUDA, se utilizarán para evaluar la parte de prácticas de la asignatura correspondiente al 25% de la nota (2.5 sobre 10). Para ello, a lo largo de los diferentes ejercicios se pondrán en prácticas las cuestiones estudiadas a lo largo de las sesiones 1 a 7.

El resultado evaluable consistirá en un informe donde se recogerán las respuestas a las preguntas planteadas. Algunas de las preguntas implican la generación de archivos de código que habrá que entregar junto con el informe. Es importante que los archivos los entreguéis con los nombres indicados para que así durante su corrección sean más sencillos de localizar. La entrega consistirá en un único archivo comprimido, subido en la tarea generada para tal efecto en campus virtual. El nombre del archivo será el nombre completo del estudiante.

La entrega se realizará el lunes 9 de enero de 2023 (hasta las 23:55 horas). Se sacará un listado con aquellos que tendrán que asistir al examen de prácticas bien por no haber entregado el informe o por no haber superado el mínimo de calidad del mismo. Las notas de prácticas se publicarán con las notas del examen de la convocatoria ordinaria.

1. **[0.05 puntos]** Descarga del espacio virtual de la asignatura el archivo suma- vectores1.cu Utilizando el profiler (nvvp) o cualquier otro mecanismo (el API de CUDA) rellena la siguiente tabla:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Tiempo | | |
| Tiempo total de la sesión | 278,52657 milisegundos | | |
| *Kernel* | 1,632 microsegundos | | |
| cudaMalloc DA | 256,2954 milisegundos | | |
| cudaMalloc DB | 6,076 microsegundos | | |
| cudaMalloc DC | 3,997 microsegundos | | |
|  | Latencia | Tiempo dedicado | Ancho de banda |
| cudaMemcpy DA HA | 17,327 microsegundos | 896 nanosegundos | 2,232 GB/s |
| cudaMemcpy DB HB | 6,445 microsegundos | 768 nanosegundos | 2,604 GB/s |
| cudaMemcpy HC DC | 15,018 microsegundos | 832 nanosegundos | 2,404 GB/s |

1. **[0.1 puntos]** Sobre el código del ejercicio anterior, cambiando el número de hebras, N, a 600, realiza las modificaciones que se detalla a continuación y renómbralo como suma-vectores1b.cu
   1. Comprobar si se genera error al hacer cada una de las llamadas a cudaMalloc() y a cudaMemcpy().

Para verificar si se produce un error en las llamadas a cudaMalloc() o cudaMemcpy() necesitamos añadir unas lineas de código:

-Cada operación Cuda es capaz de devolver una variable que indica si la función se ha ejecutado correctamente o ha habido un error.

-En cada operación Cuda debemos añadir "cudaError\_t err" como valor de retorno y comprobar con un condicional si esta es igual a cudaSuccess o no. En caso negativo, imprimiremos el error con: printf("%s\n", cudaGetErrorString(err))

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

Para comprobar posibles errores en la ejecución del kernel debemos comprobar tanto los sincronos como los asíncronos:

-Errores síncronos: cudaError\_t errSync = cudaGetLastError()

-Errores asíncronos: cudaError\_t errAsync = cudaDeviceSynchronize();

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

Mencionado en el apartado a).

* 1. ¿Has detectado algún error durante la ejecución?

No se han detectado errores

1. **[0.05 puntos]** 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. ¿Funciona correctamente?

Para hacer que el programa realice una suma de vectores con un bloque de un solo hilo:

Dejamos las dimensiones del bloque como estaban (N=600) pero cambiamos el lanzamiento del kernel a un solo bloque de un solo hilo (se tiene que encargar de todo el computo):

VecAdd <<<1, 1>>>(DA, DB, DC)

Y modificamos la función del kernel para que funcione correctamente:

for(int i=0; i<N; i++)

DC[i] = DA[i]+DB[i];

Con estos cambios el programa debe ejecutarse correctamente.

1. **[0.05 puntos]** Modifica el programa suma-vectores2.cu para obtener el programa suma-vectores3.cu que utiliza N bloques de una hebra. ¿Funciona correctamente?

Para hacer que el programa realice la suma de vectores con N bloques de un solo hilo:

Tenemos que modificar el lanzamiento del kernel. Considerando que N=600, usamos N como el número de bloques. Cada bloque solo tiene un hilo/hebra:

VecAdd <<<N, 1>>>(DA, DB, DC)

Para que la ejecución de la suma de vectores se haga correctamente, tenemos que modificar también el programa que ejecutará el kernel:

int i = blockIdx.x;

DC[i] = DA[i] + DB[i];

Con estos cambios el programa debe ejecutarse correctamente.

1. **[0.15 puntos**] 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.

**suma\_vectores1b.cu:** La duración del kernel (VecAdd) es de 1,569 microsegundos.

**suma\_vectores2.cu:** La duración del kernel (VecAdd) es de 102,496 microsegundos.

**suma\_vectores3.cu:** La duración del kernel (VecAdd) es de 3,456 microsegundos.

La razón por la que el primer caso (suma\_vectores1b.cu) tiene la menor duración es porque es el que mejor explota la paralelización, usando N hilos que comparten memoria.

El segundo caso (suma\_vectores2.cu) es el más lento ya que no realiza ninguna paralelización al solo usar 1 bloque de 1 solo hilo. Esto equivale a una ejecución secuencial.

El tercer caso (suma\_vectores3.cu) no es el más rápido porque la paralelización se realiza entre distintos streamming multiprocessors. Aun así es mejor opción que el segundo caso.

1. **[0.25 puntos]** Supongamos que queremos sumar dos vectores de 100.000 componentes:
   1. ¿Podemos usar el código del programa suma-vectores3? Explica el porqué y modifica el programa para obtener el programa suma- vectores4.cu que permita la suma de dos vectores de 100.000 componentes cada uno.

No podemos usar esa cantidad de componentes en los vectores ya que produce una violación de segmento. Esto es debido a que el tamaño máximo de la pila es de 8MB, y la creación de 3 vectores de 1000000 enteros supone un tamaño de 12MB. (con N=100.000 no hay ningún problema).

Para solucionar esto, crearemos los vectores en la propia memoria de la GPU, con la directiva \_\_device\_\_:

\_\_device\_\_ int DA[N];

\_\_device\_\_ int DB[N];

\_\_device\_\_ int DC[N];

Esto hace que estas variables solo puedan ser accedidas por los kernels, lo que significa que para iniciar el contenido de los vectores y para hacer la comprobación de las sumas deben hacerse en kernels distintos:

\_\_global\_\_ void initVec();

\_\_global\_\_ void checkAdd();

1. **[0.25 puntos]** Supongamos que queremos sumar dos vectores con un tamaño N mayor que N mayor que 65535\*512 componentes. NOTA: En este caso, podrás hacer que en el *kernel* se procesen más de un elemento utilizando un bucle for. El resultado de esta modificación será un programa suma- vectores5.cu que permitirá la suma de vectores de cualquier tamaño.

Para hacer el procesado de un vector de estas características (> 65535\*512 elementos) tenemos que tener en cuenta dos cosas:

- El número máximo de bloques que podemos asignar es de 65535.

- El tamaño de pila es de 8 MB, por lo que tenemos que tener en cuenta el tamaño de las variables que asignamos en el programa.

En cuanto a la primera consideración, elegimos lanzar el kernel con 65535 bloques y 512 hilos, de manera que un hilo deberá procesar más de un elemento (siendo N el número de elementos):

elPerThread = round(N/(numBlocks\*numThreads));

for(int i = blockIdx.x\*blockDim.x + threadIdx.x\*elPerThread; i<elPerThread && i<N; i+=1)

DC[i] = DA[i] + DB[i];

En cuanto a la segunda consideración, como no podemos iniciar 3

vectores de ese tamaño ya que superan el tamaño máximo de la pila,

debemos iniciarlos en la propia GPU (usamos la directiva \_\_device\_\_

para ello). Esto supone que además de un kernel para la suma de

vectores, necesitaremos otros dos: para la inicialización de los vectores y la

comprobación de la suma (initVec y checkAdd).

1. **[0.05 puntos]** Modifica el programa suma-vectores.cu de modo que se reserve memoria *pinned* para los tres vectores del host. Llama suma-vect- pinned.cu al programa resultante. No olvides comprobar si las llamadas a cudaMallocHost generan o no error. Después, completa la siguiente tabla:

|  |  |  |  |
| --- | --- | --- | --- |
| Tiempos cudaMalloc | DA | DB | DC |
| suma-vectores | 301,2168 ms | 17,744 microseg | 10,238 microseg |
| suma-vect-pinned | 232,5045 ms | 8,795 microseg | 6,078 microseg |

1. **[0.5 puntos]** El programa traspuesta1.cu obtiene la traspuesta de una matriz de F filas y C columnas usando bloques de H×K hebras. El grid también es bidimensional.
   1. Explica si hay coalescencia en las lecturas y en las escrituras en memoria global

No hay coalescencia en memoria global ya que aunque las lecturas en ‘dev\_a’ son realizadas en orden de columnas, lo que supone que los datos accedidos son contiguos en memoria y por lo cual las lecturas si serían coalescentes, a la hora de hacer escrituras en ‘dev\_b’ se tiene un stride de 43 elementos lo que supone que se debe acceder a segmentos en memoria distintos y dando lugar a que cada escritura se deba hacer en una transacción de memoria diferente.

* 1. Si no fuera así, modifica el código para que se aproveche la coalescencia en memoria global y llama al código resultantes traspuesta1- coalesc.cu.

Para poder conseguir coalescencia en memoria global tanto en escritura como en lectura y evitar los stride debemos hacer uso de la memoria compartida. Utilizaremos una matriz pequeña que nos servirá de intermediaria para los pasos de la transposición. De esta manera la lectura en ‘dev\_a’ se mantiene coalescente y, con una pequeña modificación de indices, las escrituras en ‘dev\_b’ se realizan de forma contigua, consiguiendo la coalescencia.

1. **[0.5 puntos]** Consideras necesario incluir alguna directiva para la sincronización de hilos en el bloque. Si fuera así, realiza la correspondiente modificación en un nuevo archivo de nombre traspuesta2.cu. No olvides comprobar que funciona correctamente.

Al no existir la barrera \_\_synthreads(), supondría que los hilos que se encargan de hacer la escritura de los elementos de ‘dev\_a’ en el espacio de memoria compartido no son esperados a que acaben, por lo que los hilos que en ese momento realizan la escritura de vuelta a la matriz ‘dev\_b’ en algunos casos son más rápidos que los primeros, haciendo que haya fallos en la transposición (el elemento que se escribe es 0).

Lo único que debemos hacer es añadir esa línea de código entre la lectura de ‘dev\_a’ y la escritura en ‘dev\_b’.

1. **[0.5 puntos]** Explica si las lecturas o escrituras del warp generan colisiones en los bandos de memoria. Si fuera así, realiza la modificación necesaria para que esto dejara de ocurrir y genera un archivo de nombre traspuesta3.cu.

Tenemos 16 bancos que almacenan cada 4 bytes sucesivos en bancos sucesivos. En este caso al tener un segmento compartido de 8x8 (32 elementos = 256 bytes), los elementos k y k+16 de la matriz compartida se encontrarían almacenados en el mismo banco, lo que supone que los threads que acceden a esos elementos a la vez provocarían un conflicto de recursos, esto hace que la segunda petición de acceso tenga que esperar a la primera.

Para solucionarlo se añade un padding al nº de columnas del segmento de memoria:

\_\_shared\_\_ int tile[H][K+1]

1. **[0.05 puntos]** Compara los tiempos obtenidos por los programas traspuesta1.cu, traspuesta2.cu y traspuesta3.cu. Explica los resultados obtenidos.

**transpuesta1.cu:** duración de sesión 510,88 ms

**transpuesta2.cu:** duración de sesión 428,89 ms

**transpuesta3.cu:** duración de sesión 305,32 ms

La duración disminuye con respecto a transpuesta1 debido a que el código con coalescencia implica una disminución en el número de transacciones a memoria, trayendo más datos en 1 sola transacción (mayor efectividad de bandwidth), por lo que los tiempos de kernel dedicados a acceder a los datos disminuyen.

También hay una diferencia entre transpuesta2.cu y transpuesta3.cu debido a que evitamos que sucedan los conflictos de banco. Cuando ocurría un conflicto la petición de acceso a datos se serializaba esperando a que la primera acabase lo que supone un retardo. Como en el caso de transpuesta3 esto no ocurre, el tiempo es menor que en transpuesta2.