MUCD
20-21

Programación GPGPU

PROCESAMIENTO DE DATOS A GRAN ESCALA

ALEJANDRO CABANA SUÁREZ Y ÓSCAR GÓMEZ BORZDYNSKI

Suma de 2 matrices

Suma de vectores

En primer lugar, ejecutamos el código en serie, sin intervención de la tarjeta gráfica. Para ello, definimos la función de suma sin el modificador *__global__* y comprobamos que no se genera ningún error.

Posteriormente, ejecutamos el kernel en la GPU con un solo thread (*<<<1,1>>>*) colocando el modificador *__global__* en su definición. También debemos reservar memoria en la tarjeta gráfica mediante la función *cudaMalloc.* Con estos cambios, podemos ver que tampoco se produce ningún error.

La suma tarda más de 120ms, pero se está ejecutando en serie en un dispositivo pensado para trabajar con múltiples hilos. Para acelerar la suma de los vectores deberemos paralelizar el kernel suma.

A continuación, utilizamos 256 threads para realizar la suma, pero sin modificar el kernel. Como no estamos definiendo qué thread se va a encargar de qué parte del vector, cada thread realizará el cálculo completo y no se paraleliza el trabajo. Realmente si ejecutásemos todos los hilos en serie realizaríamos la operación 256*x + y. Al realizar este cálculo en paralelo y sobre memoria global podemos incurrir en carreras de datos al sobrescribir y con el resultado de la suma.

Ofrecemos una alternativa sin carreras de datos y con uso de bloques y threads simultáneos. Para ello dividimos el vector en bloques donde cada thread se encargará únicamente de sumar una posición del vector. Definimos el número de threads por bloque a 1024 tras ver las propiedades de la gráfica que usaremos en Google Colab. Anticipándonos a un cambio en N, utilizaremos el techo de la división del tamaño del vector entre el número de threads por bloque para definir la cantidad de bloques que usaremos. También debemos elegir qué índice va a operar cada thread mediante *threads_per_block * blockIdx.x + threadIdx.x.* En este caso evitamos las carreras de datos ya que cada hilo lee y escribe en posiciones de memoria distintas. Compilamos y ejecutamos viendo que no tenemos ningún error y que reducimos el tiempo de ejecución del kernel a 2.69ms.

Modificación a matrices

Para modificar el programa anterior y adapatarlo a matrices, decidimos utilizar el tipo de dato dim3 para identificar los threads como elementos de una matriz. Con esto, podemos acceder a las posiciones de los sumandos de manera más comoda. Definimos un tamaño de 32x32 threads por bloque para respetar el límite de 1024 hilos por bloque de la gráfica. Al igual que los hilos, los bloques también seguirán un patrón bidimensional.

Tras ejecutar el programa vemos que no comete ningún error y que podemos sumar dos matrices de tamaño máximo 2^{15}x2^{15} sin problemas de memoria en la tarjeta gráfica.

Stencil1d

En este apartado estudiamos el impacto del uso de memoria compartida en el procesamiento paralelo de la tarjeta gráfica.

Comenzaremos realizando todo el cálculo desde la memoria global del dispositivo para tener una base con la que comparar al cambiar el tipo de memoria. Para ello, modificamos ligeramente el archivo de ejemplo para que utilice memoria global.

Vemos que el kernel tarda el 25.88% del tiempo total de ejecución siendo el resto del tiempo la transferencia de datos entre el dispositivo y el host.

Utilizando memoria compartida vemos que los tiempos de ejecución se mantienen similares, pero tenemos errores al no utilizar la función *__syncthreads().* Esto sucede por tener threads que terminan su paso de datos a memoria compartida antes que el resto de threads del bloque. En ese momento, pasan a realizar el cálculo y acceden a posiciones de memoria que aún no han sido copiadas, provocando errores en el valor final. En cuanto al tiempo de ejecución, no detectamos ninguna diferencia significativa.

Finalmente, repetimos la ejecución utilizando la función *__syncthreads()* entre el paso de datos a memoria compartida y el cálculo. Observamos que ya no se producen errores pese a que el tiempo de ejecución aumente ligeramente. Ahora, la ejecución del kernel toma el 26% del tiempo, lo cual tiene sentido ya que todos los threads tienen que esperar al thread más lento.

En los casos discutidos, el cambio de utilizar memoria global a memoria compartida no es apreciable dado que sólo accedemos al cada dato 7 veces. Para comprobar si en casos con más accesos a memoria se consigue alguna mejoría, decidimos utilizar un radio mayor (1000) y aumentar el tamaño de los vectores (2^{19}). Ahora sí podemos notar una diferencia notable entre el uso de memoria compartida (de acceso más rápido) y el uso de memoria global (acceso más lento). El programa con memoria global toma 3.36ms en ejecutar el kernel, mientras que con memoria compartida tarda 2.86ms. El problema que presenta la memoria compartida es el control de las carreras de datos y la necesidad de utilizar la función *__syncthreads()* adecuadamente.

Multiplicación de matrices

En este caso multiplicamos dos matrices en la GPU, en primer lugar, utilizaremos memoria global para realizar el cálculo. Hemos decidido utilizar bloques de 32x32 threads al igual que en el ejercicio anterior. Medimos el rendimiento en GFlops sabiendo que realizamos $2N^3$ operaciones y obtenemos la siguiente tabla de rendimiento para distintos tamaños:

Sin usar memoria compartida	Tiempo de ejecución (us)				
Tamaño de la matriz	CPU -> GPU	GPU -> CPU	Ejecución kernel	Ratio comparado con 128x128	GFlops
16x16	3.23	1.72	3.24	0.009	1.24
32x32	3.77	1.82	8.47	0.05	7.11
64x64	6,33	2.52	15,31	0.23	32.49
128x128	17.76	6.59	29.22	1	139.77
512x512	180.9	80.70	532.12	3.6	503.81

Posteriormente modificamos el código para que utilice memoria compartida. Decidimos que cada bloque cargue las filas y las columnas que necesite y luego realice el cálculo desde la memoria compartida. Realizamos los mismos análisis de rendimiento obteniendo la siguiente tabla.

Usando memoria compartida	Tiempo de ejecución (us)				
Tamaño de la matriz	CPU -> GPU	GPU -> CPU	Ejecución kernel	Ratio comparado con 128x128	GFlops
16x16	3.52	1.95	4.32	0.021	1.24
32x32	3.87	2.11	20.0	0.05	3.08
64x64	5.86	3.01	37.56	0.23	13.51
128x128	14.05	7.10	71.82	1	57.41
512x512	X	X	X	X	X

Podemos apreciar que el uso de memoria compartida en este caso ralentiza la ejecución. Además, no hemos podido realizar el experimento de 512x512 por errores en el tamaño de la memoria compartida.

Pensamos que, al usar memoria global, la tarjeta gráfica de alguna manera almacena la matriz en alguna clase de memoria caché, de manera que ejecutar el kernel 1000 veces no representa el tiempo de acceso a memoria global sino a la caché. Al usar memoria compartida forzamos una copia desde la caché a la memoria compartida, ralentizando el proceso.

Posteriormente probamos a utilizar desenrollamiento de bucles mediante la utilización de *#pragma unroll* y compilando con la flag de optimización -O3. Los resultados son muy similares a los obtenidos anteriormente. Sólo hemos notado una ligera diferencia usando memoria global y con matrices grandes.