Práctica 3: Computación de tipo stencil: Solución con Cuda

Daniel Ballesteros Álvarez
Hernán Maximiliano González Calderón
Universidad de Valladolid
Escuela Técnica Superior de Ingeniería Informática

14 de enero de 2014

Índice

1. Decisiones de diseño

1.1. Descripción del diseño de la solución

En la programación GPU/CPU, la GPU es utilizada como si fuera una "esclava". La CPU sólo se encarga de preparar la configuración, reservar memoria, definir las variables, comunicar los datos, lanzar las funciones realizadas por la GPU y cuando estas terminan recoger los resultados.

Por tanto para implementar el codigo Scentil para la GPU se debe de hallar una forma de maximizar la eficiencia de estas tareas, maximizando el uso de los recursos y la ocupación de la GPU y minimizando cargas innecesarias, como puedan ser las transferencias excesivas de datos entre la memoria de CPU y la de GPU. Para conseguirlo elegir una geometría de *grid* adecuada es primordial.

Se han identificado cuatro puntos parelelizables en esta práctica, la copia de los bordes o halo, el bucle de copia de la matriz de escritura en la de lectura, el cálculo de una nueva iteración y el cálculo del *checksum*. Para los dos primeros se han realizado optimizaciones no parelelas que se desarrollarán más adelante y para el resto se han diseñados *kernels* específicas que paralelizan sus tareas.

El programa está pensado para que dependiendo del tamaño de la matriz que se quiera computar se use la forma más eficiente para abordarlo. Se establecen dos casos: aquel en en el que las matrices serán suficientemente pequeñas como para que quepan en un bloque y aquellas cuyo tamaño es superior a al límite máximo de hilos por bloque, 512 para el caso de Patan. Aunque la máquina que disponía nuestro grupo tenía un límite de 1024, se ha decidido ajustarse a las especificaciones de Pegaso por razones de practicidad.

Para el segundo caso se ha decidido conservar en cada bloque matrices de 32x16 dado que así se consigue llenar un bloque completo.

Se ha minimizado las transferencias de memoria enviando la matriz leída desde fichero tan sólo una vez, trabajando desde la primera iteración siempre con ella en GPU. Sin embargo, no ha podido evitarse transmitir en cada iteración una reinicialización del residuo a 0 y su cálculo para cada vuelta. La carga de transmisión, aún así, es mínima.

Gracias a que la matriz actualizada se encuentra en GPU desde el primer momento, el cálculo del *checksum* tan sólo requiere una transmisión desde GPU a CPU al finalizar el programa, pero es una carga necesaria y pequeña.

La liberación de recursos es el último paso a realizar, dado que prácticamente en la totalidad del programa se reutilizan constantemente cada una de las variables de GPU reservadas.

1.2. Descripción de los kernels implementados

Update

Es el *kernel* encargado de calcular el valor de la matriz en cada iteración. Recibe como parámetros la matriz de lectura, la de escritura, el número de filas, el número de columnas y la variable donde deberá devolver el residuo que sirve de bandera de salida del bucle de actualización. Cada hilo, identificado

unívocamente por una tupla de valores dependientes del identificador de hilo, el identificador de bloque y su dimensión, ayudado de una buena política de aritmética que convierte la matriz en un toro, calcula el valor actualizado de la matriz sin que sea necesario que exista un halo. A su vez calcula el residuo propio de la iteración y lo compara con el residuo máximo permitido. En caso de ser mayor inicializa el valor del residuo que será devuelto a un valor mayor que el máximo permitido, concretamente `MAX_RESID + 1.0`. Se inicializa a un valor constante para evitar cálculos innecesarios, puesto que basta con que uno de los residuos sea mayor que el valor máximo permitido para forzar otra iteración del bucle. Esta implementación puede provocar una condición de carrera, pero no afecta al funcionamiento del programa, dado que en el momento de que un hilo deba actualizar ese valor una siguiente iteración debe ser lanzada y el valor concreto es irrelevante.

La identificación de los índices X e Y de las matrices viene dada por el valor concreto del hilo para esa coordenada en cada bloque, más el desplazamiento dependiente del índice de bloque para esa coordenada.

Las macros definidas son acordes a las presentes en el ejemplo base de la práctica, con la salvedad de que están definidas al estilo de CUDA, es decir con X (equivalente a i) creciendo hacia la derecha e Y (equivalente a j) creciendo hacia abajo.

Como se explicó anteriormente su geometría es dependendiente del tamaño de la matriz. Siempre y cuando el número de elementos de la matriz sea menor o igual al número máximo de hilos disponibles por bloque el cálculo se realiza con un sólo bloque. En caso contrario, se parte en trozos de 32x16 y se reparte entre bloques. La identificación de hilos hace que la geometría concreta sea transparente al programador.

```
#define m_old(i,j) (matrix_old[(i)+(j)*(cols)])
#define m_new(i,j) (matrix_new[(i)+(j)*(cols)])
```

Reduce

Es el *kernel* encargado de calcular el *checksum*, que es una reducción de suma de la última matriz generada. Recibe como parámetros la matriz de lectura y la matriz de escritura. A diferencia del anterior se usa memoria compartida entre bloques y en cada vuelta cada hilo del bloque copia en memoria compartida los datos presentes en memoria global que le corresponden y se sincroniza con el resto de hilos. Una vez sincronizados se entra en un bucle en el que de forma binaria se van sumando los elementos dos a dos. A cada vuelta sólo computan la mitad de hilos de los que computaban en la anterior. Aunque esto provoca *divergent braches*, son controladas y el rendimiento no se ve afectado. Al final de cada ejecución del kernel el hilo 0 de cada bloque copia su suma parcial en la memoria global. Repetidas ejecuciones del kernel van reduciendo el tamaño de la matriz, ahora interpretada como vector, dividiéndola por el número de hilos lanzados en cada bloque.

El método de identificación usado en este kernel es dual, por un lado se

identifica la posición en la memoria compartida, dada por el identificador de hilo en el bloque, y por otro lado identifica la posición en la memoria global, dada por el identificador de hilo en cada bloque más el desplazamiento dependiente del identificador de bloque.

La geometría de kernel depende del número máximo de hilos que tenga cada bloque, 512 en este caso, y sus ejecuciones se han planificado siguiendo ese criterio. Si la matriz tiene un número de elementos menor a 512, se usa un sólo bloque y con tantos hilos como elementos. Si no es así se lanza un kernel con una geometría con bloques de 512 elementos y tantos bloques como haga falta para cubrir toda la matriz con ese tamaño de bloque, obteniendo una matriz de tamaño 512 veces menor que la anterior. Si la matriz aún tiene más de 512 elementos se repite el proceso; en caso contrario se lanza un kernel de sólo un bloque y tantos hilos como elementos queden.

Al final en el programa principal se recuperará la primera posición de la matriz resultante.

2. Justificación del tamaño y forma de los bloques de hilos y el grid

A pesar de que ya se ha hablado de la geometría de bloques e hilos para cada *kernel*, merece la pena aclarar algunos puntos. La geometría definida para el *kernel* de actualización pretende ser óptima dado que maximiza la ocupación de cada bloque, pero es ligeramente dependiente del tamaño de las matrices y de la arquitectura. La solución propuesta no funcionará para aquellas matrices mayores a 512 elementos con menos de 32 columnas o menos de 16 filas, ni con aquellas cuyas número de columnas módulo 32 o su número de filas módulo 16 sea distinto de 0. Dado que ésto no se contempla dentro del *inputset* propuesto no se ha tenido en cuenta para esta práctica.

Así mismo la solución también es dependiente la arquitectura, una solución más correcta implicaría una consulta de número máximo de hilos por bloque para la arquitectura GPU subyacente mediante la función `cudaGetDeviceProperties` y un posterior cálculo de los valores óptimos, sin embargo por cuestiones de tiempo no se ha podido realizar esta optimización y se ha optado por ofrecer una solución altamente eficiente que contempla todo los *inputset* establecidos en la práctica para el sistema de pruebas, Patán.

En el caso de *kernel* de reducción usado para el cálculo de *checksum* sí que se ha realizado una implementación independiente del tamaño de la matriz, no así independiente de la arquitectura subyacente. Sin embargo ésto tendría fácil solución utilizando la función antes mencionada para consultar la propiedades del dispositivo.

3. Optimizaciones realizadas

Más allá de las soluciones dadas en cada *kernel* y ya expuestas detalladamente en los apartados anteriores, cabe destacar un par de optimizaciones que se decidieron implementar.

Como se comprobó en la práctica anterior, la generación de un halo es del todo innecesaria si se consigue una buena aplicación de aritmética modular que presente la matriz como un toro. Aunque supone una carga adicional en el cálculo del *kernel*, ahorra recursos de memoria, dado que no es necesario reservar espacio para los bordes y evita realizar cálculos de bordes y tareas de comunicación. El resultado, comprobado experimentalmente, es un aumento del rendimiento.

Además se eliminó el bucle de copia de matrices sustituyéndolo por un *swap* de punteros en cada iteración del búcle de actualización. No así en los bucles de reducción a la suma, dado que el algoritmo usado permitía aprovechar el mismo vector en el cálculo.

4. Resultados: tiempos de ejecución y *speedups*

Matrices	**8x8**	**16x16**	**32x32**	**256x256**
Programa completo	0,007s	0,006s	0,008s	0,152s
Bucle de cómputo	0,000093s	0,000437s	0,003023s	0,142428s

Cuadro 1: Tiempos de ejecución del código secuencial en Patán 1ª parte

Matrices	**1024x1024**	**2048x2048**	**4096x4096**	**5120x5120**
Programa completo	2,360s	10,609s	42,474s	69,946s
Bucle de cómputo	2,338792s	10,507854s	42,100687s	69,374880s

Cuadro 2: Tiempos de ejecución del código secuencial en Patán 2ª parte

Matrices	**8x8**	**16x16**	**32x32**	**256x256**
Programa completo	1,427s	1,424s	1,422s	1,416s
Bucle de cómputo	0,000674s	0,001052s	0,001239s	0,002702s
Speedup 1	0,005	0,004	0,006	0,107
Speedup 2	0,137982	0,425097	2,439871	52,712065

Cuadro 3: Tiempos de ejecución del código paralelo en Patán 1ª parte

Matrices	**1024x1024**	**2048x2048**	**4096x4096**	**5120x5120**
Programa completo	1,476s	1,627s	2,281s	2,710s
Bucle de cómputo	0,022151s	0,092112s	0,354111s	0,562706s
Speedup 1	1,599	6,521	18,621	25,810
Speedup 2	105,629182	114,076928	118,891215	123,287969

Cuadro 4: Tiempos de ejecución del código paralelo en Patán 2ª parte

Para empezar conviene destacar que se han obtenido dos valores de *speedup*, el primero sobre el programa completo y el segundo en base al tiempo obtenido para el bucle de cómputo.

Como se vio en prácticas anteriores los primeros casos no suelen presentar mejora en la paralelización puesto que la carga de comunicación adicional supone un coste mayor que el rendimiento que se gana en el cálculo paralelo. Sin embargo, con CUDA se apreciado una curva de crecimiento más pronunciada que la presente en los modelos de cómputo anteriores consiguiend un *speedup* mayor a 1 desde la matriz 32x32 y alcanzando desde matrices de tamaño 1024x1024 un *speedup* del orden de las centenas en el bucle de cómputo.

Sin embargo estos datos, aunque correctos, no son del todo útiles, dado que como puede verse si tomamos como referencias los tiempos del programa completo no se alcanza un *speedup* mayor que 1 hasta que no se ejecuta el código con una matriz de 1024x1024 alcanzando un velocidad 25 veces superior para la matriz de mayor tamaño, 5120x5120.

Ésto es un ejemplo gráfico del límite de la paralelización, dónde el código secuencial tiene mucho que ver con el crecimiento de la velocidad. En este caso las tareas que más cargan estos tiempos es la de lectura y generación de la matriz.

Con éstos resultados una buena decisión de diseño sería no utilizar un código paralelo para los casos en que no hay mejora de rendimiento.

5. Conclusiones y pruebas descartadas

En el siguiente apartado se resumen una serie de optimizaciones que por cuestiones de tiempo y/o complejidad no han podido implementarse en esta práctica.

5.1. Suma de diagonales

Una posible técnica sería aprovechar que muchas sumas se repiten. Como por ejemplo la suma de las diagonales de bloques adyacentes, las cuales se calculan para actualizar varios valores, tal como puede ver en el siguiente gráfico. En este caso en vez de un elemento se asignaría una matriz de 2x2 de elementos por hilo para las sumas parciales de las diagonales de la matriz, lo que supone un ahorro de 2 operaciones aritmeticas y 4 accesos de memoria usando dos registros.

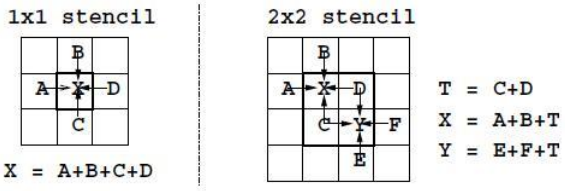

Figura 1: Suma de diagonales

Sin embargo para el *inputset* problema la complejidad de adaptar el código es suficientemente grande como para que no merezca la pena. No se niega su utilidad para matrices mayores o en casos en que los recursos de la GPU sean escasos.

5.2. Uso de memoria compartida en *kernel* de actualización

Quedó como trabajo futuro realizar un estudio más exhaustivo de la memoria compartida para aplicarla a la actualización de la matriz. Se propuso como idea dividir la matriz horizontalmente en bloques de 512 elementos que se almacenarían en memoria compartida cuando se pudiese, dado que es el límite de hilos por bloque de la tarjeta usada en las prácticas, usando la memoria global para comunicar los halos superiores, ahora sí necesarios. Dicha propuesta parte de lo experimentado en la práctica anterior con MPI. Sin embargo complejidad y falta de tiempo impidieron llevar al acto la propuesta.

5.3. Grid 3D

Como ya se ha dicho la geometría de grids y bloques afecta en gran medida al rendimiento. Podrían haberse usado estructuras de hilos de mayor dimensión para utilizar el *hardware* de manera más eficiente. Un ejemplo sería considerar una geometría tridimensional donde las diagonales formen el eje Z.

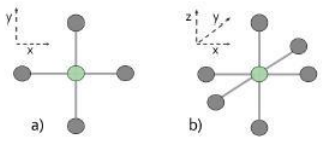

Figura 2: Eje Z

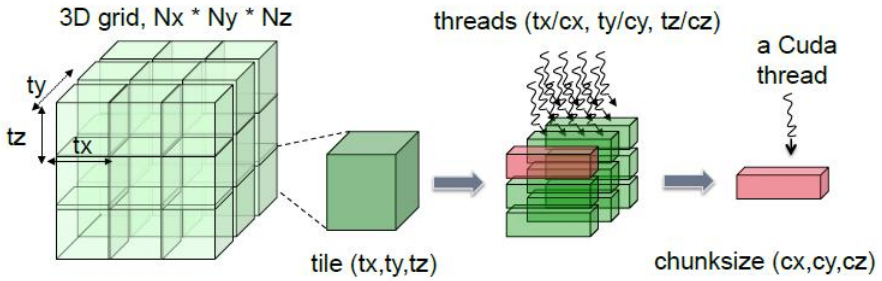

Figura 3: Grid 3D

A pesar de que hacer los calculos con la GPU aceleran el cómpute de una manera muy visible hay varios aspectos que hacen que no sea la más extendida actualmente. salvo en grandes centros de calculo. Aunque es algo que está cambiando. Muchos programas hacen uso de las GPU para acelerar sus cálculos, desde codificadores de vídeo hasta hojas de calculo, como han anunciado los autores de *LibreOffice* para su próxima versión.

Más allá del precio de las tarjetas, otro de los problemas principales de este tipo de computación es que implica un cambio de enfoque frente a la programación secuencial. Afortunadamente están surgiendo varias soluciones que hacen que la programación en las GPU sea mas accesible como "Mint", un modelo de programación basado en *pragmas*, similar a OpenMP.

Cuando escribes una función, *kernel*, no hay muchas alternativas para encontrar los fallos en el código, haciendo su depuración algo complejo.

Las transferencias entre memoria de GPU y CPU también puede volverse algo problemático y costoso en tiempo, aglo que la próxima versión de CUDA parece haber solventado ya que se publicó que podrá accederse como si fuera memoria compartida.

6. Referencias

- "Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C" - Didem Unat, Xing Cai y Scott B. Baden
- "CUDA 2D Stencil Computations for Jacobi Method" - José María Cecilia, Jose Manuel García y Manuel Ujaldrón