V. OPTIMIZACIÓN PARA COMPUTACIÓN GPU EN CUDA

La arquitectura de una GPU es básicamente distinta a la de una CPU. Las GPUs están estructuradas de manera paralela y disponen de un acceso a memoria interna muy rápida, pero el tamaño de una aplicación (kernel) es limitado. Debido a que la GPU está unida a la CPU del host mediante el bus PCI-Express, el trasiego de información entre la CPU y la GPU puede resultar costoso.

Para que una aplicación sea acelerada debidamente en una GPU debe cumplir lo siguiente:

- Que las tareas sean masivamente paralelas. Pueden ser divididas en centenares o miles de tareas independientes.
- Que sean computacionalmente intensas. El tiempo usado en la computación debe superar con creces las tareas de lectura y escritura de la memoria.
- Que el tamaño del kernel sea limitado. El tamaño debe ser pequeño para ser alojado en la GPU (habitualmente unos cuantos kilobytes).

Las aplicaciones que no satisfacen estos criterios pueden llegar a ser más lentas si son ejecutadas en una GPU.

1. OPTIMIZACIÓN EN CUDA

Un kernel CUDA podrá ser ejecutado en cualquier GPU que soporte la tecnología. Sin embargo la velocidad de ejecución dependerá de cada dispositivo en particular.

Mediante la optimización nos proponemos como objetivo mejorar el rendimiento de una aplicación con respecto a versiones anteriores. Se debe llegar a un equilibrio entre la paralelización y los demás recursos puestos a nuestra disposición, para no crear cuellos de botella nuevos.

La documentación de referencia del modelo CUDA establece las siguientes recomendaciones:

RECOMENDACIONES

i. Alta prioridad

1. Para lograr el máximo rendimiento con CUDA, se debe poner especial hincapié en lograr paralelizar el código.
2. Usar como parámetro de decisión el ancho de banda efectivo para evaluar el rendimiento y los resultados de la optimización.
3. Minimizar las transferencias de datos entre host y dispositivo, incluso si es necesario ejecutar algunos kernels directamente en la GPU aunque no muestren un mayor rendimiento comparados con la ejecución en la CPU.
4. Intentar amalgamar en la medida de lo posible los accesos a la memoria Global.
5. Minimizar el uso de la memoria Global. En su lugar se usará la memoria compartida.
6. Evitar bifurcaciones o diferentes rutas de ejecución dentro del mismo "warp".

ii. Prioridad Media

1. El acceso a la memoria compartida debe ser diseñado para evitar solicitudes en serie debido a que surgen conflictos de banco.
2. Usar la memoria compartida para evitar transferencias redundantes desde la memoria global.
3. Para evitar los tiempos de espera debidos a las dependencias de los registros hay que mantener un mínimo de 25 % de ocupación en dispositivos con capacidad computacional 1.1 o inferior, y de 18,75 % en dispositivos más modernos.
4. El número de hilos por bloque debe ser múltiplo de 32 para ofrecer mayor eficiencia de cálculo y permitir el amalgamado de accesos.
5. Usar la biblioteca rápida de matemáticas siempre que la velocidad compense la precisión.

iii. baja prioridad.

1. Usar operaciones de copia cero en GPUs integradas.
2. Cuando el kernel tenga muchos argumentos se debe colocar algunos de éstos en la memoria Constante para no agotar la memoria Compartida.
3. Usar operaciones de cambio para evitar operaciones de división o módulo costosas.
4. Evitar la conversión automática de double a float.
5. Facilitar para el compilador el uso de ramificación predictiva en lugar de bucles o controles.

2. ANÁLISIS DE ALGUNAS RECOMENDACIONES

En los siguientes apartados se analizan algunos de los aspectos más importantes contenidos en éstas recomendaciones.

a. Ejecución de hilos en CUDA

Recordemos que cuando se realiza una llamada a un kernel se generan mallas de hilos que se organizan en dos niveles. Por un lado las mallas se organizan en matrices unidimensionales o bidimensionales de bloques y estos se organizan en matrices de 1, 2 ó 3 dimensiones de hilos.

Los bloques pueden ejecutar las instrucciones independientemente unos de otros. Esto es lo que da lugar a la Escalabilidad Transparente.

Teóricamente se puede suponer que los hilos también se ejecutan independientemente unos de otros dentro de un mismo bloque. Para ello debemos garantizar que la barrera de sincronización es usada para que todos los hilos concluyan una determinada fase antes de empezar la siguiente. Es decir no debemos permitir que la ejecución dependa de la sincronización estricta entre hilos.

Además recordemos que muchos dispositivos CUDA manejan hilos en forma de grupos para su ejecución. Esta estrategia de implementación impone limitaciones de rendimiento para determinados códigos.

Algunos dispositivos organizan los hilos de un mismo bloque en "warps". Esto permite disminuir los costes de fabricación y optimiza los accesos a la memoria.

Los hilos de un bloque se reparten en "warps" según el índice del hilo. Si un bloque está organizado como un array unidimensional, sólo se necesita threadIdx.x para identificar los hilos.

Como regla general para "warps" de 32 hilos, el n-ésimo "warp" empezará con el hilo 32xN y concluirá con el 32(N+1)-1.

Para bloques de 2 o 3 dimensiones, las dimensiones son proyectadas de manera lineal antes de repartir los hilos en "warps".

En un momento dado el hardware selecciona y ejecuta un "warp" cada vez. Una instrucción es ejecutada por todos los hilos del mismo warp, antes de pasar a la siguiente. Esto permite que la lectura de una instrucción y su procesamiento sean aprovechados por un gran número de hilos. Este paradigma funciona a la perfección cuando todos los hilos de un warp siguen la misma senda de control de flujo cuando tratan los datos.

Para una condición "if-then else" se produce una divergencia; unos hilos ejecutarán la sentencia "then" y otros la "else". En esta situación, la ejecución requiere varios pasos a través de las rutas de divergencia. Estos pasos son secuenciales y por tanto introducirán un retraso.

La divergencia puede surgir en otras situaciones, por ejemplo, en un bucle donde el número de iteraciones es diferente dependiendo del hilo. Si se ejecutara un bucle que diera lugar a 6, 7, 8 ó 9 iteraciones según el hilo, tendríamos que todos los hilos ejecutarían las 6 iteraciones a la vez, pero se necesitaría un paso adicional para aquellos que fueran a ejecutae el 7, otro más para 8 y finalmente otro más para el 9.

Una instrucción "if-then-else" puede también provocar divergencia si la decisión se basa en el índice del hilo, por ejemplo. Lo mismo pasa con los bucles cuando la iteración depende del índice del hilo. Este fenómeno es frecuente en muchos algoritmos paralelos, como por ejemplo el algoritmo de reducción.

Este tipo de algoritmo se basa en que se tratan todos y cada uno de los datos (maximización, minimización...). El objetivo es dividir los datos en grupos y aplicar a cada uno de ellos el algoritmo, logrando así paralelización. En una fase posterior se someten al mismo algoritmo los datos obtenidos en la fase anterior de manera paralela; y así sucesivamente hasta lograr el resultado final y así logramos aumentar la rapidez de la ejecución.

Los datos son en un principio cargados en la memoria global. Entonces son repartidos entre bloques y cada bloque carga en su memoria compartida la parte de datos que tratará. Almacenando el dato final con la instrucción "syncrthread()" garantizamos que los hilos no ejecutarán la siguiente iteración, hasta que todos hayan concluido la primera, por tanto garantizamos que para la segunda iteración todos los hilos usan el mismo resultado obtenido como resultado de la primera iteración.

b. Rendimiento de la memoria Global

Uno de los aspectos que condicionan el rendimiento de un kernel son las llamadas a la memoria global. Se utilizan varias técnicas, como el "tiling" para aprovechar de manera eficiente el trasiego de datos de la memoria global hacia las memorias compartidas y los registros.

Además se aprovecha la técnica de amalgama que permite mover datos de manera más eficiente todavía entre las distintas memorias. En general se usan ambas estrategias para lograr aprovechar de manera efectiva la limitación en cuanto a ancho de banda para el acceso a la memoria global.

La memoria global está basada en la tecnología de condensadores. El acceso a una posición de memoria implica la detección de la carga del condensador correspondiente. Como la carga y descarga de condensadores es una operación lenta limita las operaciones con memoria global. Para resolverlo se impone que cuando se accede a una posición determinada se accede a la vez a todas aquellas posiciones que forman parte del grupo y que han sido amalgamadas. Se produce entonces una lectura en paralelo lo cual permite compensar la velocidad y hacer que la transferencia de datos sea más rápida al procesador.

Si una aplicación está diseñada para acceder a posiciones de memoria consecutivas logrará una mayor velocidad que si los accesos fueran a posiciones aleatorias.

Los dispositivos GPU permiten este tipo de programación. Se aprovecha el hecho de que todos los hilos de un "warp" ejecutan la misma instrucción. Por tanto se hace que todos los hilos de un "warp" carguen datos de la memoria en el mismo momento. Además se logra una máxima eficiencia cuando la instrucción remite los hilos a posiciones adyacentes en la memoria global. Cuando el Hardware detecta que las posiciones son adyacentes, amalgama todos los accesos y se solicitan de golpe todas las posiciones de memoria. Para un "warp" de 32 hilos si al hilo 0 le corresponde la posición N, y así sucesivamente, el hardware detectará que son adyacentes, y la lectura se hará de una vez permitiendo llegar a velocidades cercanas al máximo del ancho de banda.

Recordemos que todas las posiciones de memoria forman un espacio único de direcciones consecutivas.

Las matrices son almacenadas en este espacio según el convenio de la fila mayor. Es decir los elementos de la fila 0 son colocados en posiciones consecutivas. Cuando se acaban estos elementos se pasa a colocar los elementos de la fila 1 y así sucesivamente. Las filas, son por tanto colocadas una tras otra. La denominación fila mayor se refiere a que esta disposición conserva la estructura de las filas; ya que los elementos de una misma fila son colocados en posiciones consecutivas.

Como ejemplo se ve en la figura 1 la disposición en la memoria de una matriz 4x4. Se observa claramente que los elementos M(0,0) y M(0,1), aunque son adyacentes en la estructura matricial, no lo son en el espacio de memoria.

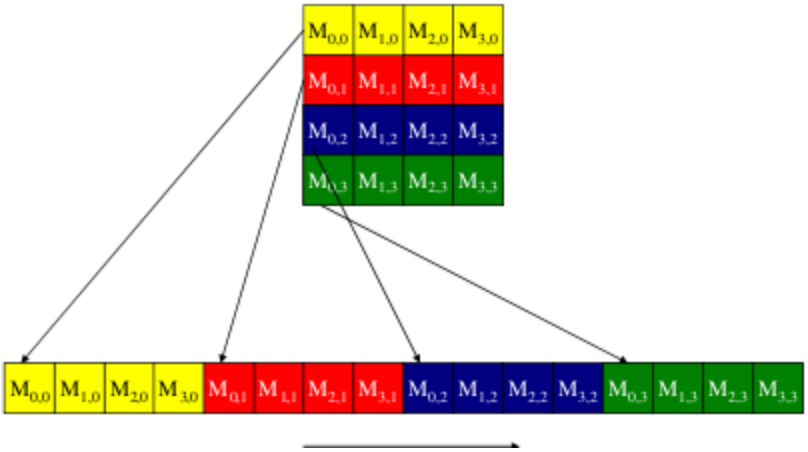

Fig 1: Disposición en la memoria de una matriz 4x4.

En conclusión los hilos se deben programar de tal modo que accedan a posiciones adyacentes de memoria en cada iteración.

c. Repartición dinámica de los recursos del SM.

Los recursos de ejecución en un Multiprocesador o SM, incluyen registros, ranuras de bloques de hilos, y ranuras de hilos. Estos recursos son asignados dinámicamente y repartidos entre los hilos para colaborar en la ejecución. En la serie G88, cada SM tiene 768 posiciones para hilos. Estas posiciones se reparten entre los bloques durante la ejecución. Hay varias posibilidades de repartir estas posiciones. Si tenemos 256 hilos por bloque, las posiciones pueden ser repartidas entre 3 bloques, en cambio con 128 hilos por bloque podremos tener 6 bloques en total. La capacidad de repartir de manera dinámica las posiciones de hilos disponibles hace que el SM sea muy versátil. Recordemos que en una tarjeta G260, disponemos de 1024 posiciones y por tanto, de hasta 32 Warps.

La estrategia de asignación estática de posiciones no permite acomodar las necesidades de los bloques que pueden variar a lo largo de la ejecución del kernel.

Para el aprovechamiento de esta estrategia se debe tener en cuenta la limitación de los SM que tan sólo pueden acomodar 8 bloques en total.

Los registros también son recursos que pueden ser compartidos de manera dinámica. El número de registros depende del Hardware. En la serie G80 hay 8K entradas de 32 bits por cada SM; mientras que en la G260 disponemos de 16K. En los registros se almacenan las variables automáticas. Son usados para datos que se usan de forma muy frecuente y para aquellos que genera el compilador. Algunos kernels pueden necesitar muchos registros; otros no. Por tanto se debe tener presente la relación entre bloques y registros a la hora de realizar la asignación.

d. Pre-captación de datos:

Se ha comentado que una gran limitación en la programación en paralelo radica en el acceso a la memoria global; en concreto al ancho de banda soportado.

En algunas situaciones se puede dar el caso de que los hilos tengan pocas instrucciones que ejecutar entre el acceso a la memoria y el consumo de datos accedidos.

Una buena estrategia es la de pre-captar los siguientes datos mientras se están consumiendo los actuales. Así aumentamos el número de instrucciones a ejecutar entre accesos a la memoria y los consumidores de datos accedidos.

e. Mezcla de Instrucciones

En las GPUs que soportan CUDA, cada procesador tiene un ancho de banda limitado para procesar instrucciones. Cada instrucción consume ancho de banda, tanto si es en coma flotante, como si es una instrucción de carga, o una bifurcación. Por ejemplo, en un bucle la variable indexadora es actualizada en cada iteración y ésta operación compite con las demás instrucciones en el consumo de ancho de banda.

f. Granularidad de los Hilos

En muchas ocasiones, es mucho más eficiente poner una gran carga de instrucciones en pocos hilos que usar la opción contraria. Esto es especialmente indicado cuando existen tareas redundantes entre los hilos.

g. Paralelismo.

La idea fundamental es estructurar un código de tal manera que muestre mucho paralelismo. Una vez puesto de manifiesto este aspecto deberá ser trasladado al Hardware de manera eficiente. Esto se cumple eligiendo la configuración de la ejecución de cada kernel (código ejecutado en la GPU). La aplicación debe, así mismo, maximizar la ejecución paralela mediante el aprovechamiento de las ejecuciones simultaneas en el dispositivo a través de los flujos de datos, así como de los trasiegos que se producen entre el Host y el dispositivo.

El uso óptimo de la memoria empieza minimizando las transferencias entre el host y el dispositivo ya que estas transferencias disponen de un ancho de banda muy inferior al disponible para transferencias dentro del dispositivo. Las llamadas del kernel a la memoria Global deben ser reemplazadas en la medida de lo posible por accesos a la memoria compartida del dispositivo. En algunos casos la mejor opción, puede llegar a ser recalcular los resultados localmente en lugar de recuperarlos de otro emplazamiento.

El ancho de banda efectivo puede disminuir en un orden de magnitud dependiendo de las pautas de acceso a cada tipo de memoria. Por tanto se debe usar la memoria según pautas de acceso óptimas. Esto es especialmente importante para la memoria Global donde los tiempos de acceso son muy largos. En cambio la memoria compartida sólo de debe optimizar cuando hay conflictos de banco importantes.

En cuanto a la optimización de las instrucciones se deben evitar aquellas que tienen bajo rendimiento y producen pocos resultados. Se debe dar prioridad a la velocidad y no a la precisión, siempre y cuando no se comprometa la exactitud de los resultados. Esto se logra usando funciones intrínsecas en lugar de las habituales y usando precisión simple en lugar de doble. Además se tiene que tener cuidado con el control de flujo de instrucciones; recordemos que estamos en un modelo Una Instrucción-Varios Hilos (SIMT).

h. Memoria de Textura

El espacio de memoria de textura es de sólo lectura y está almacenado en la caché. Por tanto acceder a los datos de la textura resulta en que el dispositivo realiza una sola lectura de la caché. Está optimizada para distribución bidimensional, por tanto los hilos de un mismo "warp" que leen posiciones contiguas de esta memoria serán muy eficientes. Esta memoria también está diseñada para capturas de "streaming" con latencia constante. Es decir una lectura de la caché reduce la demanda de ancho de banda de la DRAM, pero no reduce el tiempo de acceso.

En algunas situaciones la lectura desde la memoria de textura resulta más ventajoso que hacerlo desde la memoria Global o la Constante.

La captura a través de la memoria de textura ofrece varias ventajas con respecto a la lectura de la memoria Global:

1. La memoria de Textura está cargada en la caché y por tanto tiene mayor anchura de banda si se distribuyen en posiciones cercanas bidimensionalmente.

2. Las texturas se pueden usar para evitar cargas no amalgamadas desde la memoria Global.

3. Los datos empaquetados pueden ser desempaquetados en distintas variables en una sola operación.

4. Entradas con enteros de 8 o 16 bits pueden ser reconvertidos fácilmente en valores flotantes de 32 bits en el rango de [0.0, 1.0] o [-1.0, 1.0].