Cuda

Patricia Aguado Labrador Santiago Blasco Arnaiz

Mayo 2018

---

1. Memoria

Comenzamos paralelizando las mismas secciones que en la práctica de OpenMP ya que el profesor dijo que había claras relaciones entre los bucles anidados de aquella práctica y los lanzamientos de kernel de esta. Primero definimos el tamaño de bloque, escogemos 512 hilos que es la mitad de lo que admite la arquitectura kepler que utiliza el leaderboard y calculamos el número de bloques dividiendo el número total de elementos en la matriz (rows * columns) entre el número de hilos.

Para obtener el identificador global de los hilos multiplicamos el identificador de bloque por los hilos que lo forman y a continuación le sumamos el identificador secuencial del hilo dentro del bloque, así podemos ajustar el número de hilos a los que necesitamos en cada momento, este es el código en todos los kernels:

```
int IDX_Thread   =  threadIdx.x;
int IDX_block    =  blockIdx.x;
int threads_per_block_x  =  blockDim.x;

int gid = IDX_Thread + (IDX_block * threads_per_block_x);
```

Para paralelizar la inicialización de matrices reservamos espacio en la zona de memoria global del device. Una vez hecho esto llamamos al kernel gpuFunc_inicializar que se encarga de calcular el identificador global de los hilos como hemos explicado anteriormente y de esta manera cada hilo almacena ceros en su posición correspondiente dentro de surface y surfaceCopy en el device. Por último, traemos a la CPU las matrices inicializadas.

```
gpuFunc_inicializar<<<gridShape,bloqShape>>>(cSurface,cSurfaceCopy,rows,columns);
```

A continuación, paralelizamos la sección de movimiento de los equipos, para ello asignamos a cada equipo un hilo, antes de lanzar el kernel gpuFunc_movimiento, que se encarga de ello, hacemos una copia de los arrays focal y teams en el device reservando previamente la memoria necesaria, tras la ejecución del kernel traemos de vuelta al host ambos arrays.

```
gpuFunc_movimiento(int num_teams, int num_focal, FocalPoint *focal, Team *teams)
```

Introducimos algunas mejoras secuenciales como saltarnos iteraciones si no se ha producido la primera activación, sustituir las raíces cuadradas en los dos sitios que no son necesarias, desdoblar el código de la propagación del calor entre los steps pares e impares para no tener que copiar surface y surfaceCopy, solo comprobar global_residual si estamos en el step 0 y sólo si se han desactivado todos los focos, ya que es cuando mayor será el valor de la variable que es lo que nos interesa.

Intentamos paralelizar el movimiento de los equipos, copiando los datos del array focal y surface al device y dentro del kernel asignamos a cada equipo un hilo, obtenemos resultados erróneos ya que varios hilos pueden reducir el calor en la misma posición de surface a la vez, por tanto habrá condiciones de carrera entre hilos.

Paralelizamos la sección que actualiza los valores de surface, al igual que con el movimiento de los equipos copiamos antes y después de lanzar el kernel los valores de surface para pasarnos los valores del host al device y viceversa, dentro del kernel debemos calcular para cada hilo la posición i y j, la i será el identificador global dividido entre el número de columnas y la j el resto de esa división. Los tiempos no mejoran mucho ya que estamos haciendo un número muy grande de tranferencias de memoria entre el host y el device, para notar una mejoría deberemos paralelizar todo el bucle de los steps y realizar una sola copia por iteración.

Algunos casos de prueba daban resultados erróneos y descubrimos que es por una mala elección del número de bloques, ya que la división para calcular el número de estos se hace de forma entera, si el resto no es cero tendremos menos hilos que elementos en la matriz, para solucionarlo sumamos uno al número de bloques.

Con esto obtenemos un código correcto y funcional pero que tarda más de dos minutos en ejecutarse en la máquina del leaderboard.

Decidimos variar el número de hilos por bloque, y al final decidimos quedarnos con 256 ya que notamos una leve mejoría en el tiempo de ejecución. A la hora de paralelizar la sección de los steps en la propagación del calor, necesitamos dos kernels, uno que se encargará de actualizar los focos activos y otro que actualizará a continuación de este los valores en surface, surface sólo se copiará una vez por iteración al final del bucle de los steps para que el resto de kernels puedan coger los datos de surface actualizados.

```
gpuFunc_actualizarFocos<<<gridShape,bloqShape>>>(cSurfaceCopy,cFocal,num_focal,columns)
```

Este kernel consiste en el cálculo del identificador global, y la actualización de surface si hay un foco activo.

```
gpuFunc_actualizarCalor<<<gridShape,bloqShape>>>(cSurfaceCopy,cSurface,rows,columns)
```

Este kernel consiste en el cálculo del identificador global para obtener la posición de la matriz a la que tiene que acceder cada hilo calculando los i y j

como ya hemos explicado, para así poder hacer los cálculos con surfaceCopy que almacenará en surface. Todo ello saltándose los bordes de la matriz con el fin de actualizar el calor en esta.
Ambos kernels se utilizan en los pasos impares y pares, pero en este último caso si se da que es el paso cero como ya hemos explicado, se realiza el cálculo del global_residual.

Al lanzarlo al tablón obtenemos resultados erróneos pero en local no. Gracias al compañero kikocazagansos[1] descubrimos que habíamos borrado sin querer en uno de los kernels la condición que restringe el número de hilos al número de elementos que necesitamos.
Tras arreglar este despiste conseguimos entrar en el tablón con un tiempo de 20 segundos.

Para mejorar el tiempo decidimos implementar la reducción de global_residual, creamos un kernel específico para ello al que le pasamos un array que creamos en el que almacenaremos el valor global_residual de cada hilo para más tarde hacer la reducción basada en árboles binarios que hemos visto en clase, si el tamaño del array es impar le añadimos un elemento más al final con valor 0.0 para que puedan hacer correctamente la reducción los hilos y no afecte al resultado. Obtuvimos errores relacionados con la reserva de memoria para el array de global_residual, el compañero aderator[2] nos aconseja no realizar una reducción como tal, en vez de un array con global_residual para cada hilo reservamos memoria para una variable del device que será un float cGlobal_residual q vaue tomará el valor que deba dentro del kernel y después será copiado a la variable local global_residual, esto es posible gracias a que si se cumple la condición el algún hilo cGlobal_residual pasará a valer lo que la variable THRESHOLD, por tanto sólo tiene dos posibles valores, el ya mencionado o 0.0, el cambio será observado sin condiciones de carrera por el host.
Tras la implementación de esta reducción"mejoramos un 40 % el tiempo en el leaderboard.

Referencias

[1] Francisco Ganso Gil

[2] Andrés Cabero Mata

[3] http://riubu.ubu.es/bitstream/10259/3933/1/Programacion_en_CUDA.pdf

[4] Apuntes de la asignatura Computación Paralela. Escuela de Ingeniería Informática. Universidad de Valladolid. Curso 2018-2019.

[1] kikocazagansos#0529 (Francisco Ganso Gil).
[2] aderator#1465 (Andrés Cabero Mata).