Computación Paralela
Simulación extinción de incendios
Memoria práctica CUDA

Andrés Cabero Mata
Fernando Gigosos Ronda

Planteamiento inicial:

Motivados por el poco tiempo que teníamos para realizar esta práctica, acudimos directamente a las mejoras secuenciales inmediatas que ya teníamos en previas prácticas e inmediatamente después aplicamos a los diferentes bucles una paralelización de las operaciones que se realizan en sus iteraciones. Al no tener estas condiciones de carrera o al menos no excesivamente irresolubles, quisimos hacer una paralelización con el máximo número de operaciones posibles, evitando así una ralentización excesiva debido al tiempo perdido en copiar la información del host a la tarjeta.

Para el tamaño de bloque del grid hemos optado por 256 hilos, pues es el que en el último archivo es el que mejor rendimiento ha dado en la cola (comprobando diferentes tamaños muy distintos en la máquina local).

La naturaleza de la solución aplicada consiste principalmente en sustituir apartados (bucles) o varios de ellos por llamadas a funciones definidas para la tarjeta (kernels) y por lo tanto la información de esta memoria atañe a los kernels que se utilizan en los que previamente eran diferentes apartados:

Apartado 3:

En la inicialización de la matriz (*surface*), en un principio intentamos trasladar la inicialización en bucle a una llamada a un kernel, el cual una vez establecido un espacio de memoria para surface.

```
103  __global__ void initializeMatriz(float *surface, int size){
104    int IDX_Thread = threadIdx.x;
105    int IDX_Block =  blockIdx.x;
106    int threads_per_block = blockDim.x;
107    int i = IDX_Block * threads_per_block + IDX_Thread;
108    if(i<size)
109      surface[i] = 0.0;
110  }
```

Más tarde tras de completar otros apartados a la hora de buscar la solución a otros problemas encontramos la llamada *cudaMemset*, la cual, aunque desconocemos realmente si es más rápida, parece aconsejada por su sencillez.

```
380    cudaMemset(dSurface, 0.0, sizeof(float)* (rows*columns));
```

Apartado 4:

Apartado 4.1:

En el principio de lo que ya viene a ser la simulación en sí misma, primero debemos resolver el bucle que dentro de las iteraciones espera a que al menos uno de los focos se active, para esto creamos una lista del mismo tamaño que la lista de focos en la memoria

CUDA, y acto seguido llamamos a una función kernel del dispositivo que ejecuta paralelamente el que anteriormente era ese bucle en recorrido de la lista de focos.

Al terminar esa función, copiamos de vuelta la estructura de memoria de los focos y realizamos una comprobación de cual está ya activado, y de estarlo cambiamos el valor de *first_activation*, iniciando así el comienzo del cálculo de la propagación del calor.

```
396   for( iter=0; iter<max_iter && ! flag_stability; iter++ ) {
397
398     activateFocalPoints<<<blocks_per_grid_focal, threads_per_block>>>(dS
399     cudaMemcpy(hNum_deactivated, dNum_deactivated, sizeof(int)* (num_foc
400     cudaMemcpy(hFirst_activation, dFirst_activation, sizeof(int)* (num_f
401
402     num_deactivated=0;
403     first_activation=0;
404     for(i=0; i<num_focal; i++){
405       num_deactivated += hNum_deactivated[i];
406       first_activation += hFirst_activation[i];
407     }
408     if(!first_activation) continue;
```

Al terminar esa función, copiamos de vuelta la estructura de memoria de los focos y realizamos una comprobación de cual está ya activado, y de estarlo cambiamos el valor de *first_activation*, iniciando así el comienzo del cálculo de la propagación del calor

```
112 __global__ void activateFocalPoints(float *surface, FocalPoint *focal, int size, int iter, int *num_deactivated, int *first_a
113   int IDX_Thread = threadIdx.x;
114   int IDX_Block =  blockIdx.x;
115   int threads_per_block = blockDim.x;
116   int i = IDX_Block * threads_per_block + IDX_Thread;
117   if(i<size){
118     if(focal[i].start == iter){
119       focal[i].active = 1;
120       first_activation[i]=1;
121     } else if( focal[i].active==2) num_deactivated[i]=1;
122   }
123 }
```

Sabemos que esta solución del apartado 3 no es la más óptima ni mucho menos, pero debido a sugerencias de compañeros y a la falta de tiempo, supimos que realizar una reducción paralela no iba a manifestarse en el mayor de los cambios

Apartado 4.2:

La primera modificación hecha en este apartado fue la mejora secuencial de las anteriores prácticas, hacer distintas cosas según si el paso es par o impar. Tras hacer esto empezamos a paralelizar los distintos apartados de forma singular, cambiando la matriz para la cual se hace entre *surface* y *surfaceCopy* (*dSurface* y *dSurfaceCopy* en el device).

Apartado 4.2.1:

Esta paralelización corresponde a asignar a la casilla del foco la temperatura del foco en cuestión, este bucle originalmente recorre la lista de focos y, en su posición correspondiente, si están encendidos, coloca su valor de temperatura.

```
__global__ void updateHeat(float *surface, FocalPoint *focal, int size, int rows, int columns){
  int IDX_Thread = threadIdx.x;
  int IDX_Block =  blockIdx.x;
  int threads_per_block = blockDim.x;
  int i = IDX_Block * threads_per_block + IDX_Thread;
  if(i<size)
    if ( focal[i].active == 1 ){
      accessMat( surface, focal[i].x, focal[i].y ) = focal[i].heat;
    }
}
```

Con esta función kernel, el dispositivo genera el índice correspondiente dentro de la lista de focos, sin condiciones de carrera no hay ningún problema en la total paralelización de este bucle, en este caso no se copia de vuelta la matriz de surface, ya que la siguiente función que se ejecuta (kernel) trabaja con la información tal y como esta función *updateHeat* la ha dejado.

Apartado 4.2.2:

Este apartado no se hace al hacer la división según si el paso es par o impar, por lo cual es eliminado.

Apartado 4.2.3:

Para paralelizar la expansión del calor ejecutamos un kernel con el tamaño de la tabla, en el cual, tras calcular el id global de cada hilo, y con el y la información del número de filas y columnas que componen a la matriz podemos recalcular el i y el j para la posición de la matriz. El código que ejecuta cada hilo es por tanto casi una copia del que se haría en secuencial.

```
__global__ void stepKernel(float *surfaceA, float *surfaceB, int rows, int columns){

  int IDX_Thread = threadIdx.x;
  int IDX_Block =  blockIdx.x;
  int threads_per_block = blockDim.x;
  int gid = IDX_Block * threads_per_block + IDX_Thread;

  int i = gid/columns;
  int j = gid%columns;

  if(i<rows*columns)
    if(i>0)
      if(j>0)
        if(i<rows-1)
          if(j<columns-1)
            accessMat( surfaceA, i, j ) = (
              accessMat( surfaceB, i-1, j ) +
              accessMat( surfaceB, i+1, j ) +
              accessMat( surfaceB, i, j-1 ) +
              accessMat( surfaceB, i, j+1 ) ) / 4;
}
```

Apartado 4.2.4:

Este apartado, que implica una gran carga al programa de manera secuencial, fue paralelizado después de haber entrado ya el código en leaderboard, consiguiendo tras su implementación una mejora de más de 7 segundos (en nuestro código, evidentemente).
Su funcionamiento se basa en una “flag” que indica si alguna de las diferencias entre una casilla de una iteración y la anterior son mayores que THRESHOLD, debido a esta flag y realmente a la propia naturaleza del problema, no se trata realmente de una plena función de reducción. y sin embargo es aún más eficiente que una, que como ya digo, debido a la naturaleza del código original o el problema que resuelve, esta no es necesaria.

```
157 __global__ void reductionGlobalResidual( float *surface, float *surfaceCopy, int rows, int columns, float *global ){
158   int IDX_Thread = threadIdx.x;
159   int IDX_Block =  blockIdx.x;
160   int threads_per_block = blockDim.x;
161   int gid = IDX_Block * threads_per_block + IDX_Thread;
162   int i = gid/columns;
163   int j = gid%columns;
164
165   if(i<rows*columns)
166     if ( fabs( accessMat( surfaceCopy, i, j ) - accessMat( surface, i, j ) ) >= THRESHOLD )
167       *global=THRESHOLD;
168 }
```

Este es el kernel utilizado, extremadamente sencillo, la variable global se copia de vuelta y se analiza su resultado.

Apartado 4.3:

Este apartado planeabamos ejecutarlo entero en un kernel, y como mejora futura, mover el bucle interno que busca entre todos los focos a otro kernel (“__device__”), que se ejecutaría para cada uno de los hilos. A la hora de hacer la primera parte, no logramos hacer que funcionara, pese a que parecía correcto el código, y era lo mismo que hacían otros compañeros, ese error nos impidió hacer la paralelización de este apartado.
La única mejora (secuencial) que hay es ejecutar el apartado (mover los equipos) solo si hay focos activos, con una condición antes del bucle.

Apartado 4.4:

No nos dió tiempo a paralelizar este apartado, pensamos paralelizar el bucle interior entero en un kernel, teniendo en cuenta que es necesaria una operación atómica. Hemos hecho las mismas mejoras secuenciales que la anterior práctica (Mips), cambiado las comprobaciones del radio del bucle más interno al bucle inmediatamente anterior.

Bibliografía:

- Temas 1 a 5 CUDA, Grupo Trasgo, Ingeniería Informática, UVA
- https://stackoverflow.com/questions/2619296/how-to-return-a-single-variable-from-a-cuda-kernel-function
- https://stackoverflow.com/questions/11747265/calling-a-kernel-from-a-kernel

- https://stackoverflow.com/questions/13387101/cudamemset-does-it-set-bytes-or-integers
- https://devtalk.nvidia.com/default/topic/477182/how-can-i-return-a-float-in-cuda-kernel-call-/
- El compañero de informática y amigo cercano Eduardo García Asensio :D