COMPUTACIÓN PARALELA

Optimización de una simulación de bombardeo de partículas con CUDA

Grupo 04:
Velasco Gil, Álvaro
Sanz Pérez, Alejandro

INTRODUCCIÓN

Para esta tercera práctica de *Computación Paralela* se nos ha pedido conseguir una optimización del código de una simulación un bombardeo de partículas a través de la paralelización con **CUDA**.

LABORES REALIZADAS

Una vez obtenido el código, se ha recurrido a analizar los puntos clave del programa que podrían optimizarse, encontrando 3, con los que se considera que podría pasarse el tiempo de la *Referencia 1*.

EN EL KERNEL

El primer punto, se encuentra en la función **gpu_Actualizar(...)** en el kernel, que se encarga de actualizar la capa layer de la GPU con los golpeos de las partículas, estando cada hilo modificando la posición que coincide con su identificador global. De esta forma, en vez de asignar un solo proceso los miles de golpeos, se los reparten entre todos, bajando considerablemente el tiempo, pero no siendo suficiente como para llegar a la referencia.
Se ha considerado que podría haberse mejorado esta función aplicando memoria compartida, pero no hemos sabido implementarlo de modo que mejorase el tiempo.

```
1  __global__ void gpu_Actualizar(float *layer, int posicion, float energia,int layer_size) {
2      float umbral = 0.001;
3      int gid = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y) + (threadIdx.x + blockDim.x * threadIdx.y);
4      if(gid < layer_size){
5          int distancia = posicion - gid;
6          if ( distancia < 0 ) distancia = - distancia;
7          distancia = distancia + 1;
8          float atenuacion = sqrtf( (float)distancia );
9          float energia_k = energia / atenuacion;
10         if ( energia_k >= umbral || energia_k <= -umbral ) layer[gid] = layer[gid] + energia_k;
11     }
```

El siguiente punto se trata del proceso de relajación de la capa que hay entre tormenta y tormenta. De esto se encarga la función que se realiza en la GPU que es **gpu_Relajacion**(...), en el que cada hilo actualiza el valor que coincide con su posición, asignando la media de su antiguo valor, y el de sus vecinos adyacentes.
Esta función nos ha recortado bastante tiempo, y se considera que se podía haber mejorado si se tuviesen en cuenta los warps, de modo que coincidiese con los segmentos, mejorando la coalescencia, pero tras muchas pruebas no se ha conseguido.

```
20  __global__ void gpu_Relajacion(float *layer, float *layer_copy, int layer_size) {
21      int gid = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y) + (threadIdx.x + blockDim.x * threadIdx.y);
22      if(gid>0 && gid < layer_size-1) layer[gid] = ( layer_copy[gid-1] + layer_copy[gid] + layer_copy[gid+1] ) / 3;
23  }
```

El último punto clave con el que se consideraba que se podría pasar ya la referencia, tras haber aplicado los puntos anteriores, se considera que era la obtención de los máximos. Para eso, se ha implementado la función **gpu_reduceMaximo()** que se basa en realizar una reducción de la capa, utilizando sólo un número de hilos que corresponde a la mitad del

tamaño de la capa que le entra en cada iteración. De esta forma, cada hilo se compara con el que tiene su mismo identificador más el valor de la mitad de la capa. El mayor valor se establece en la parte de la izquierda del vector, y se actualiza el vector de posiciones con su posición también.
Este método parecía eficiente, pero no cubría todos los posibles casos, por lo que se procede a crear una segunda función de kernel, llamada **gpu_reduceMaximos()**. Esta función está basada en elaborar un array en el que se establecen todos sus valores a 0, de forma paralelizada. Después, si la posición de la capa es mayor que sus dos vecinas adyacentes, escribe el valor en el otro array. De esta forma se nos elabora un array en el que si su valor no es 0, es que es un máximo relativo.
Una vez obtenido este array, tan solo hay que aplicar la primera función del kernel que creamos para encontrar los máximos, de modo que deja en la primera posición el valor del máximo, al igual que en el vector *posiciones* en su primera posición, el valor de la posición en la que estaba el susodicho máximo.

```
26  __global__ void gpu_getMaximo(float* maximos, float* positions, int size){
27
28      int gid = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y) + (threadIdx.x + blockDim.x * threadIdx.y);
29      int s = size/2;
30      if ( gid >= s ) return;
31
32      if( maximos[ gid ] < maximos[ s + gid ] ) {
33          maximos[ gid ] = maximos[ s + gid ];
34          positions[gid] = positions[ s + gid ];
35      }
36      // Extra element
37      if ( size%2 != 0 && gid == 0 ){
38          if( maximos[ 0 ] < maximos[ size - 1 ] ) {
39              maximos[ 0 ] = maximos[ size - 1 ];
40              positions[ 0 ] = size-1;
41          }
42      }
43
44  }
45
46
47  __global__ void gpu_reduceMaximos (float *layer, float *maxRelativos, int layer_size ){
48
49      int gid = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y) + (threadIdx.x + blockDim.x * threadIdx.y);
50      if ( gid > layer_size ) return;
51      maxRelativos[gid] = 0;
52      if ( gid == 0 || gid == layer_size-1 ) return;
53      if ( layer[gid] > layer[gid-1] && layer[gid] > layer[gid+1] ) maxRelativos[gid] = layer[gid];
54
55  }
```

Esta método nos ha servido con todos los test, pero el problema es que al aplicarlo con un test del leaderboard, no pasaba, ya que era demasiado grande y por el diseño de nuestro programa nos quedamos sin espacio.
Se intentó solucionar reutilizando los arrays de las otras capas y pasando los valores del HOST al DEVICE, al igual que dividiendo a la mitad algunos arrays, pero no nos dió tiempo a implementarlo de manera correcta.

También se implementó una función para copiar los datos de una capa a otra dentro de la GPU, de modo que se aprovechan los hilos para una copia más rápida.

```
14  __global__ void gpu_Copiar(float *layer, float *layer_copy,int layer_size) {
15      int gid = (blockIdx.x  + gridDim.x  * blockIdx.y) * (blockDim.x * blockDim.y) + (threadIdx.x + blockDim.x * threadIdx.y);
16      if(gid < layer_size) layer_copy[gid]=layer[gid];
17  }
```

EN EL HOST:

El primer paso y más importante era el número de hilos que queríamos asignar a cada bloque, de modo que se seleccionó 128 ya que fue el que mejor se comportaba, reduciendo los tiempos.
En el proceso de estudiar la teoría, se vió que a la hora de realizar reservas de memoria de las capas que iban a ser compartidas con la GPU, si se realizaba con la función de cuda **cudaMallocHost**, en la línea 135, se podría disminuir hasta la mitad el tiempo que se emplearía con la función malloc(...).

CONCLUSIONES

En la elaboración de esta práctica se ha aprendido el valor de seleccionar bien el número de hilos que se asigna a la hora de querer maximizar su coalescencia, al igual de cómo se realizan las transmisiones entre el host y el device, como a usar árboles de reducción.
Se nos queda la espina de no haber sabido sacar el máximo partido a la memoria compartida ya que consideramos que el programa se hubiese optimizado en gran manera, al igual que no haber conseguido reducir la memoria consumida en la GPU, porque se consiguieron tiempos que nos dejarían pasando la referencia, probando y comparando con resultados de otros test, pero debido a eso nos dió fallo de segmentación y no se consiguió actualizar el valor del leaderboard.

REFERENCIAS

- cudaMallocHost: https://stackoverflow.com/questions/23133203/cudamallochost-vs-malloc-for-better-performance-shows-no-difference
- Reduce en forma de árbol: Tarea reduce0.cu del campus virtual.
- Resto de conceptos: diapositivas de la asignatura.