Memoria de la práctica de CUDA

Daniel de Vicente Garrote,Christian de la Puerta Verdejo

21 de mayo de 2018

1. Introducción

El objetivo de esta práctica,al igual que en las otras dos anteriores, consiste en paralelizar una simulación de bombardeo de partículas que emplea uno o varios ficheros de tormentas y un tamaño de array determinado por el usuario.

1.1. CUDA

CUDA es una plataforma de computación paralela desarrollada por NVIDIA, basada en el lenguaje de programación C, que permite el uso de mecanismos de paralelización basado en la creación de hilos concurrentes.

CUDA emplea el uso de kernels, que permite realizar partes del algoritmo mediante una cantidad determiada de hilos, que se distribuyen en bloques de hilos, cuyos bloques se crean dentro de un grid. Cuand se realiza una llamada al kernel, se le define el número de hilos, bloques, tamaño de memoria compartida, y otras variables previamente definidas.

2. Estructura del programa

La parte paralelizable del programa consta de cuatro partes destacables que se repiten según el número de ficheros introducidos:

- La primera parte consta del cálculo de las suma de energías de los impactos,que se almacenan en layer.
- La segunda parte solo copia el contenido de layer a layer_copy.

- La tercera parte calcula la media de cada posicion de layer_copy entre su valor anterior,su valor posterior,y su propio valor, y lo almacena en layer(no se realiza en los extremos de este).

- La última parte calcula los máximos de layer,primero lo hace buscando valores locales,comparando cada posición su valor con los de sus vecinos,si es mas grande se compara con el máximo global, y si es mas gradne lo reemplaza e indica su posicion.

3. Estrategia de paralelización

Cada elemento definido anteriormente del código se procesa en uno o varios kernels, de forma que se pueda ejecutar con varios hilos, controlando las condiciones de carrera que se puedan producir. Antes de ejecutarse los kernels, se definen bloques unidimensionales de threads de tamaño 256, calculando el número de bloques necesario para cubrir el tamaño del array(número de hilos igual o menor que e valor de layer).

En cada kernel se calcula el id global de cada hilo, basandose en su identificador de hilo y bloque, y luego evitamos que los hilos que no hagan nada se ejecuten con un return.

Antes de entrar en el bucle se van añadiendo varios cudaMalloc para crear copias de layer(layerGPU),layer_copy(layerCopyGPU),maximos(maximosGPU) y posiciones(posicionesGPU). Además se crean dos arrays especiales,tmpGPU y posGPU,que almacenan los valores de layer y sus posiciones originales, y sirven para realiar la reducción del máximo con repecto a layer, excepto los extremos que se almacenan en ini y fin,dos arrays de tamaño 1(debido al poco tiempo no se me ocurrió otro modo de declarar estas dos variables).

Por cada operación se define una variable de tipo cudaError_t, para poder comprobar si existen errores en las asignaciones de array en GPU,transfercias de memoria y liberación de memoria.
Al final del programa se copian los valores de layerGPU, layerCopyGPU, maximosGPU y posicionesGPU a sus valores correspondientes,y se liberan sus espacios de memoria correspondientes al finalizar.

int idGlobal = threadIDx.x+blockDim.x*blockIdx.x

3.1. Suma de energía de impactos

```
273         for( j=0; j<storms[i].size; j++ ) {
274             float energia = (float)storms[i].posval[j*2+1] / 1000;
275             int posicion = storms[i].posval[j*2];
276             actualiza<<<numBlocks,threads_per_block>>>(layerGPU,posicion,energia,layer_size);
277         }
```

Cada hilo calcula sus valores de energía y posición, y ejecutan el kernel de actualiza,donde se calcula el valor de layer en cada posicion asignada al hilo.Actualiza solo incluye el identificador global.
La idea era paralelizar todo este trozo del ćódigo,pero debido a problemas a la hora de mover storms a la GPU, decidimos dejarlo así para asegurarnos de que funcione bien,aunque tarde un poco mas en completar pruebas.

3.2. Copia de valores a layer_copy

```
24  __global__ void relajacionCopia(float* layerGPU,float* layerCopyGPU,int layer_size){
25      int idGlobal = threadIdx.x+blockDim.x*blockIdx.x;
26      if(idGlobal > layer_size-1) return;
27      layerCopyGPU[idGlobal] = layerGPU[idGlobal];
28  }
```

Se copian los valores de layerGPU a layerCopyGPU, cada hilo mueve una posición a su lugar correspondiente.

3.3. Actualización de la capa

```
36  __global__ void copiaAtmp(float *layerGPU,float *tmp,int layer_size,int *pos,float *ini,float *fin){
37      int idGlobal = threadIdx.x+blockDim.x*blockIdx.x;
38      if(idGlobal > layer_size-1) return;
39      if(idGlobal != 0 && idGlobal != layer_size-1){
40          tmp[idGlobal-1] = layerGPU[idGlobal];
41          pos[idGlobal-1] = idGlobal;
42      }
43      if(idGlobal == 0){
44          ini[0] = layerGPU[0];
45          fin[0] = layerGPU[layer_size-1];
46
47          if(tmp[0] <= ini[0])
48              tmp[0] = 0.0f;
49
50          if(tmp[layer_size-3] <= fin[0])
51              tmp[layer_size-3] = 0.0f;
52      }
53
54      if(idGlobal > 0 && idGlobal < layer_size-3){
55          if(tmp[idGlobal] == tmp[idGlobal+1]){
56              tmp[idGlobal] = 0.0f;
57              tmp[idGlobal+1] = 0.0f;
58          }
59          if(tmp[idGlobal] == tmp[idGlobal-1]){
60              tmp[idGlobal] = 0.0f;
61              tmp[idGlobal-1] = 0.0f;
62          }
63      }
64  }
```

Se trasladan los valores de layerCopyGPU a un array temporal creado previamente, denominado tmp(llamado tmpGPU en el código secuencial), y las posiciones originales se definen en pos(llamado posGPU fuera del kernel). Luego se descartan valores de tmp que no sirvan como máximos, por ejemplo

si el segundo valor de layer es inferior al primero se pone a cero para que no cuente.
Lo mismo para la última y penúltima posición,ademas si hay dos valores iguales consecutivos en el array también se ponen a cero, así simulamos el comportamiento de la búsqueda de máximos locales.

3.4. Busqueda del máximo

```
__global__ void reduccion(float *layerGPU,float *maximosGPU,int i,int layer_size,float *tmp,int tam,int *pos){
    int idGlobal = threadIdx.x+blockDim.x*blockIdx.x;
    if(idGlobal > (tam/2)-1) return;
        if(tmp[idGlobal] < tmp[idGlobal+(tam/2)]){
            tmp[idGlobal] = tmp[idGlobal+(tam/2)];
            pos[idGlobal] = pos[idGlobal+(tam/2)];
        }else if(tmp[idGlobal] == tmp[idGlobal+(tam/2)] && pos[idGlobal] > pos[idGlobal+(tam/2)]){
            pos[idGlobal] = pos[idGlobal+(tam/2)];
        }

    if(tam%2 != 0 && idGlobal == 0 && tmp[0] < tmp[tam-1]){
        tmp[0] = tmp[tam-1];
        pos[0] = pos[tam-1];
    }
}

__global__ void copiaAarray(float *maximosGPU,int *posicionesGPU,int i,float *tmp,int *pos,float *ini,float *fin,int layer_size,float *layerGPU){
    maximosGPU[i] = tmp[0];
    posicionesGPU[i] = pos[0];
    if(tmp[0] == 0.000000){
        posicionesGPU[i] = 0;
    }
}
```

Aquí el primer kernel se ejecuta con el tamaño inicial(layer_size) y se va dividiendo a la mitad en cada iteración,hasta que valga uno,trabajando con el array tmpGPU. En cada iteraciónse ejecutan solo la mitad de hilos de tam, y se caulcula el valor máximo entre dos elementos, uno de cada mitad del array,y se almacena en la primera mitad. Si hay dos valores iguales, se tiene en cuenta el que tenga la posicion menor, así aseguramos que si es el máximo de la oleada señale que es la primera posición en la que se encuentra.
En caso de que el tamaño sea impar, el elemento que sobra es comparado con la primera posición del array.

El segundo kernel se ejecuta con un solo hilo,y se encarga de pasar a maximosGPU y posicionesGPU el máximo y la posición de cada oleada. Si el máximo que se obtuvo es cero(no hay máximo), se pone en posicionesGPU el valor de cero.

4. Tiempos

La primera version no paralelizaba el cálculo de máximos,por lo que tardaba 1 minuto y 16 segundos en el leaderboard. La seguda versión tenía ya la parte de los máximos paralelizada y tardaba 56 segundos. Con la version final se eliminó código innecesario y se redujo el tiempo a unos 47 segundos.