UNIVERSIDAD DE GRANADA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

PROGRAMACIÓN PARALELA

Memoria de prácticas:
Práctica #1: CUDA

GRADO EN INGENIERÍA INFORMÁTICA

ÍNDICE

ÍNDICE **2**

Algoritmo de Floyd **3**

Estructura Bidimensional 3

Búsqueda del mayor camino 4

Operación vectorial **6**

1. Algoritmo de Floyd

1.1. Estructura Bidimensional

En este primer ejercicio se nos proporcionó la estructura de datos para la implementación del algoritmo de Floyd así como la implementación en CPU y en GPU del algoritmo. La versión paralela usaba una estructura unidimensional de bloques de hebras unidimensionales, y se nos pedía modificar este algoritmo creando una nueva implementación del mismo estructurando el trabajo paralelo en una malla bidimensional de bloques de hebras bidimensionales (los bloques).

Para la implementación del algoritmo se han usado una gran diversidad de recursos didácticos, como los apuntes de teoría de la asignatura, la documentación oficial de CUDA, tutoriales e implementaciones de compañeros de otros años, estos últimos usados siempre como guía y en ningún momento como copia o plagio.

En primer lugar, se decidió cuál sería y cómo se implementaría esta estructura bidimensional. Los bloques, cuyo tamaño predefinido se almacena en ***blocksize*** tendrán $\sqrt{blocksize}$ hebras en su dimensión X y $\sqrt{blocksize}$ hebras en su dimensión Y, siempre redondeado al valor entero superior más cercano. De este modo cada bloque tendrá la misma cantidad de hebras que en la implementación unidimensional, pero estructuradas de forma bidimensional, en forma de matriz.

De modo parecido se define el tamaño de la malla de bloques. Como en el problema trabajamos con una matriz cuyo tamaño es $nverts2 = nverts * nverts$ tenemos que por cada fila de la matriz van a trabajar tantos bloques como defina la ecuación $blocksPerRow = nverts / threadsPerBlock$, de modo que la matriz del problema puede estructurarse como una matriz de bloques de hebras de tamaño $blocksPerRow * blocksPerRow$.

Aquí podemos ver el código resultado de este razonamiento:

```
uint threads = sqrt(blocksize);
dim3 threadsPerBlock_2D (threads, threads);
dim3 blocksPerGrid_2D (
      ceil((float)nverts/threadsPerBlock_2D.x),
      ceil((float)nverts/threadsPerBlock_2D.y)
);
for(int k = 0; k < niters; k++) {
      floyd2D  <<<blocksPerGrid_2D,  threadsPerBlock_2D>>>  (device_M,
      nverts, k);
```

El algoritmo principal no ha sufrido mucho cambio en cuanto al algoritmo original, simplemente necesitamos acomodarlo a la nueva estructura.

```
__global__ void floyd2D(int * M, const uint nverts, const uint k) {
      uint i = blockIdx.y * blockDim.y + threadIdx.y; // fila
      uint j = blockIdx.x * blockDim.x + threadIdx.x; // columna

      if (i < nverts && j < nverts) {
            int ij = i*nverts+j;     // indice de la matriz
            int Mij = M[ij];
            if (i != j && i != k && j != k){
                  uint Mikj = M[i*nverts+k] + M[k*nverts+j];
                  Mij = (Mij > Mikj) ? Mikj : Mij;
                  M[ij] = Mij;
            }
      }
}
```

Los tiempos y las gráficas de las ejecuciones pueden consultarse en la siguiente hoja de cálculo: Tiempos

En general los tiempos paralelos mejoran bastante los tiempos secuenciales para tamaños grandes del problema. Se nota también una pequeña desmejora en el algoritmo bidimensional para aquellos valores de bloque cuya raíz cuadrada no es un valor exacto y debe ser redondeado. Esto se debe a la generación de fragmentación en la estructura de hebras al final de cada malla, generado por este redondeo, necesario para implementar la estructura en estos casos no exactos.

1.2. Búsqueda del mayor camino

Como segundo ejercicio se plantea la implementación de un algoritmo que cálculo el máximo camino entre dos nodos, y para se ha usado un algoritmo de reducción de los ejemplos oficiales de CUDA el cual se ha modificado y acomodado según nuestras necesidades, las cuales son obtener el máximo valor almacenado en la matriz resultado que proporciona el algoritmo de Floyd.

La idea principal es obtener durante una primera fase el valor máximo de cada fila, almacenándolo en la primera posición de la fila. En una segunda fase, se calcula el valor máximo de la primera columna de la matriz, es decir, el máximo de los máximos locales a su fila, obteniendo así el valor máximo global de la matriz.

Aquí el código del algoritmo:

```
__global__ void reduceMax(int *g_idata, int *g_odata, const uint
size) {
      __shared__ int sdata[blocksize];
      uint tid = threadIdx.x;
      uint i = blockIdx.x * blockDim.x + threadIdx.x;

      // Cada hebra carga un elemento desde memoria global a memoria
      compartida
      if (i < size)
            sdata[tid] = (g_idata[i] > g_idata[i+blockDim.x]) ?
            g_idata[i] : g_idata[i+blockDim.x];

      __syncthreads();

      // Reducir en memoria compartida
      // Ambas dividen el tamaño de bloque por 2.
      for(uint s = blockDim.x>>1; s > 0; s >>= 1) {
            if(tid < s)
                  if(sdata[tid] < sdata[tid+s])
                        sdata[tid] = sdata[tid+s];

       // Esperar al resto de hebras para comenzar la nueva etapa
       __syncthreads();
    }

    // Escribir resultado de este bloque en memoria global
    if (tid == 0)
       g_odata[blockIdx.x] = sdata[0];
}
```

```
alex@manjaro:~/Documentos/PPR/P1/Floyd
[alex@manjaro Floyd]$ ./floyd input/input1400
Device 0: "GeForce GTX 650" with Compute 3.0 capability

Vértices: 1.400 -> 1.960.000
NumBlocks: 1.915
10,0589 1,20514 8,34663 0,878255        11,4532
[GPU] Camino más largo: 2.008
[GPU] Tiempo: 0,00486112

[CPU] Camino más largo: 2.008
[CPU] Tiempo: 0,00198507

[alex@manjaro Floyd]$
```

2. Operación vectorial

En este segundo ejercicio se plantea realizar varias operaciones aritméticas con vectores. Más concretamente lo que se nos pide es que dado dos vectores, A y B, de números en coma flotante, calculemos lo siguiente;

- ➢ Un nuevo vector C a partir de los vectores A y B, siguiendo una fórmula concreta.
- ➢ Suponiendo que el vector C está dividido en N bloques de igual tamaño M, calcular la suma de los M elementos de cada bloque y almacenarlas en un vector D de tamaño N..
- ➢ Calcular el valor máximo del vector C.

Además, el cálculo del vector C debe hacerse de dos formas distintas en CUDA, una implementación que haga uso de memoria compartida y otra que no.

Como en total se nos piden 4 cosas distintas, he realizado e implementado cada una de ellas separadamente, en funciones, de manera que cada cálculo queda completamente desacoplado de los demás. Además, se pide comprobar la exactitud de los cálculos y calcular los tiempos tanto de la versión paralela y la secuencial para poder realizar comparaciones.

El código se deja anexo a este documento para poder ser consultado, y en esta sección no voy a incorporar fragmentos de código por ser demasiado largos y redundante. Aún así voy a comentar brevemente cómo he realizado cada uno de los cálculos.

En primer lugar, tanto el cálculo del vector D como el cálculo del máximo del vector C se han realizado mediante el algoritmo de reducción visto en el ejercicio anterior, adaptados a sendos casos.

En segundo lugar, el cálculo del vector C se ha realizado siguiendo la fórmula dada en el guión de prácticas así como la implementación secuencial que se nos ha proporcionado, modificándola para convertirla en una implementación CUDA que permite el paralelismo.

```
__global__ void calcularC (const float * A, const float * B, float *
C, const int size) {
  uint block_start = blockIdx.x * blockDim.x;
  uint block_end = block_start + blockDim.x;
  uint i = block_start + threadIdx.x;

  if (i < size) {
      C[i]=0;

      for(int j=block_start; j<block_end; j++) {
      float a = A[j]*i;
```

```
        if((int) ceil(a) % 2 == 0)
        C[i] += a + B[j];
        else
        C[i] += a - B[j];
        }
    }
  }
```