Paralelos Análisis de Multiplicación de Matrices en Cuda

Daniela Fernanda Milón FLores

June 2017

1 Introduction

A continuación procederemos a analizar las formar de multiplicar matrices en CUDA, una de ellas será la forma convencional y la segunda será una versión mejorada para aprovechar el paralelismo que nos ofrece CUDA.

2 MATRIX MULTIPLICATION KERNEL

A continuación presentamos una descripción del código que se presenta en la Figura 1.

Este código en CUDA, recibe como parámetros las dos matrices (M, N) a multiplicar. Además del tamaño de la matríz que en este caso es único, pues suponemos que son matrices cuadradas.

2.1 Análisis del Código

Como mencionamos anteriormente, esta forma de multiplicar matrices se apega mucho a las que se enseñaron en el curso. Se toma cada matríz como vector y se

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

Figure 1: Multiplicación de Matrices Tradicional. Extraído de [**?**].

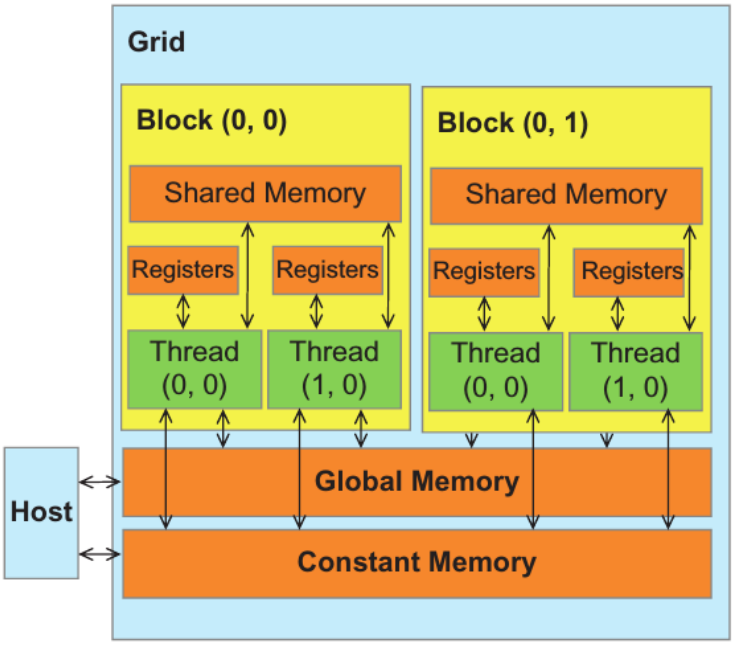

Figure 2: Tipos de Memoria. Extraído de [**?**].

comienza con el procedimiento de multiplicación, el cual consiste en multiplicar cada elemento de la primera fila de la primera matríz por cada elemento de la primera columna de la segunda matríz. Además cada hilo estará encargado de procesar un elemento resultante.

Sin embargo; este método no aprovecha de forma correcta el paralelismo que nos ofrece CUDA, pues como podemos notar, en el instante que cada thread calcula un elemento tiene que ir hasta la memoria Global para conseguir el valor que necesita. Esto no solo significa tener que ir N veces a la memoria Global, donde N es el número de elementos de cada matríz, sino que involucra el gasto de recursos y tiempo innecesario para hallar la matríz resultante. Podemos ver lo que ocurre por dentro en la Figura 2.

A continuación describiremos un método alternativo para aprovechar la correcta diposición de los tipos de memoria en CUDA.

3 A TILED MATRIX MULTIPLICATION KERNEL

Como hemos visto, en CUDA es necesario conocer de los tipos de memoria, y la versión del device que se tiene. Por ello este método de multiplicación de

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

     // Identify the row and column of the d_P element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.   float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

       // Collaborative loading of d_M and d_N tiles into shared memory
9.     Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.    Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
11.    __syncthreads();

12.    for (int k = 0; k < TILE_WIDTH; ++k) {
13.      Pvalue += Mds[ty][k] * Nds[k][tx];
       }
14.    __syncthreads();
     }
15.  d_P[Row*Width + Col] = Pvalue;
   }
```

Figure 3: Multiplicación de Matrices Tiled. Extraído de [**?**].

matrices en CUDA, tiene en cuenta la memoria Compartida y la aprovecha para un mejor performance.

3.1 Análisis del Código

Lo que se hace en este código es compartir laa matrices Mds y Nds en menoria compartida. Las cuales más adelante serán las filas y columnas de las matrices M y N respectivamente. Realizar este breve procedimeinto, nos ahorra tener que ir hasta la memoria global. Ahora cada thread se dirigi a la memoria compartida, la cuál está a más corta distancia que la memoria global.
Luego se declaran bx y tx, las cuales al estar dentro de la instancia de memoria global (shared) se compportan como variables privadas para cada thread. Lo que le permite un acceso más rápido a cada thread. Por último nos encontramos con syncthreads( ). Este comando en CUDA es como un barrier, el cuál evita que cualquier thread se ejecute hasta que todos hayan llegado a un mismo punto. De esta forma las flas y columnas podrán copiarse de forma completa de Mds y Nds.

4 Tabla de Comparación

Cant. Elementos	normal	tile
10	0,096	0,094
100	0,1035	0,9168
1000	0,089	0,087
10000	0,3492	0,3425

Figure 4: Multiplicación de Matrices Tiled. Extraído de [**?**].