Universidad Nacional de San Agustin
Esc. Profesional de Ciencia de la Computación

Algoritmos Paralelos

Tema: Laboratorio Multiplicación de Matrices con CUDA

Profesor: Alvaro Henry Mamani Aliaga
Alumno: Diego André Ranilla Gallegos

25 de Junio del 2017

1. Multiplicación de matrices en CUDA

1.1. Programa 1: Manera tradicional

En este método de multiplicación, cada hilo tiene la tarea de obtener el resultado de la posición de la matriz que le fue asignada, para ello el hilo tiene que recorrer la fila de la primera matriz y recorrer la columna de la segunda matriz, (la posición de fila y de columna según la posición del elemento a calcular).

```
__global__ void MatrixMulKernel(float* A, float* B, float* P)
{
  int Row = blockIdx.y*blockDim.y +threadIdx.y;
  int Col = blockIdx.x*blockDim.x +threadIdx.x;

  if((Row < WIDTH) && (Col < WIDTH)){
    float Pvalue = 0.0;

    for(int k=0;k<WIDTH;k++){
      Pvalue+= A[Row*WIDTH+k] * B[k*WIDTH+Col];
    }
    P[Row*WIDTH+Col] = Pvalue;
  }
}
```

Listing 1: Multiplicación normal de matrices con CUDA

Podemos apreciar en este código escrito en CUDA, que cada elemento que se desea calcular, necesita de manera repetida las filas y columnas de las dos matrices a multiplicar, por lo que se necesita ir a las posiciones de ambas matrices para obtener ambos valores. Hay que tomar en cuenta que ambas matrices están almacenadas en la **memoria global** del dispositivo (NVDIA), así que si se necesita extraer los valores de las matrices de manera recurrente en la memoria global, la multiplicación seria ineficiente, ya que extraer un valor almacenado en la memoria global es costoso.

1.2. Programa 2: Método multiplicación de matrices de mosaico (tiled)

Este método de multiplicación mejora considerablemente, en cuanto a la velocidad de acceso a los datos, debido a que se aprovecha **la memoria compartida** la cual es compartida por los hilos de un bloque, esta memoria en cuanto acceso es mucho más veloz que la memoria global, pero no tiene un espacio almacenamiento similar o superior a esta, veamos este método de multiplicación:

```
__global__ void MatrixMulTiledKernel(float* d_M, float* d_N, float*
    d_P) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;

  for (int ph = 0; ph < WIDTH/TILE_WIDTH; ++ph) {
```

```
14      Mds[ty][tx] = d_M[Row*WIDTH + ph*TILE_WIDTH + tx];
15      Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*WIDTH + Col];
16      __syncthreads();
17
18      for (int k = 0; k < TILE_WIDTH; ++k) {
19        Pvalue += Mds[ty][k] * Nds[k][tx];
20      }
21      __syncthreads();
22    }
23    d_P[Row*WIDTH + Col] = Pvalue;
24  }
```

Listing 2: Multiplicación de matrices de mosaico

1.3. Comparación

Para comparar la velocidad de ambos métodos, vamos a probarlos con diferentes dimensiones para las matrices (NxN), para comodidad de las pruebas vamos a usar para las dimensiones de los bloque (numero de hilos), valores que se sean divisibles con la dimensión de las matrices a multiplicar, esto ayudará especialmente al segundo programa cuando divida las matrices por partes y sean almacenados en pequeñas matrices en memoria compartida.

	Hilos x N				
	5 x 500	10 x 500	20 x 500	25 x 500	50 x 500
Prog 1	1.83808	1.25312	0.891712	0.908288	0.00048
Prog 2	0.796672	0.315968	0.23552	0.28672	0.00208

Tabla 1: Tabla de resultados de las pruebas

Los siguientes resultados muestran la velocidad obtenida, por medio de los dos programas en los cuales hemos empleado una matrices de 500 x 500 elementos y dimensiones diferentes para el tamaño del bloque desde 5 hasta 50. Podemos apreciar al inicio, que al emplear pocos hilos por bloque la diferencia entre los programas se hace evidente, el segundo es más veloz que el primero, hace la multiplicación empleando la mitad del tiempo. Empleando más hilos seguimos obtenido esa diferencia, incluso se puede apreciar la velocidad del segundo programa es la tercera parte del primero cuando empleamos bloques de 10 x 10 y de 20 x 20.

Pero cuando se sigue esta tendencia de aumentar la cantidad de hilos por bloque vemos que la velocidad del segundo no sigue disminuyendo con la tendencia esperada, sino que el primero ahora es más rápido que el segundo, esto lo vemos cuando se aplica bloques de 40 x 40 hilos y sigue así cuando aplicamos con 50. Entonces la pregunta es porque sucede este fenómeno, la respuesta sería porque en el segundo programa tiene que almacenar más datos desde la memoria global hacia la memoria compartida debido al incremento de las dimensiones de las matrices *tile* (Mds y Nds en el segundo programa). Como lo hemos comentado anteriormente llamar datos que están en memoria global es costoso, por lo tanto no es recomendable aumentar demasiado el número de hilos por bloque.

2. Conclusiones

En este trabajo podemos rescatar diferentes puntos:

- La memoria compartida a comparación de la memoria global es más veloz en cuanto al acceso de datos, pero tiene una capacidad pequeña de almacenamiento a comparación de la global.
- Podemos ver que la velocidad se redujo por un factor de N en matrices *tiles* de NxN.
- Hay que tener cuidado cuando asignamos las dimensiones de la matriz *tile*, ya que a pesar que reduzca por un factor N.
- A pesar de lo dicho en el punto anterior, debemos mencionar también que no siempre el segundo programa va a superar al primero, debido a que el tamaño de la matriz *tile* podría ser una carga si su dimensión es muy grande (teniendo en cuenta que esa dimensión pueda ser cubierta por el pequeño tamaño de la memoria compartida).