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

Algoritmos Paralelos

Tema: Laboratorio Multiplicación de Matrices con CUDA (version 2)

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

11 de Julio del 2017

1. Multiplicación de matrices en CUDA

1.1. Programa 1: Método multiplicación de matrices de mosaico (tiled)

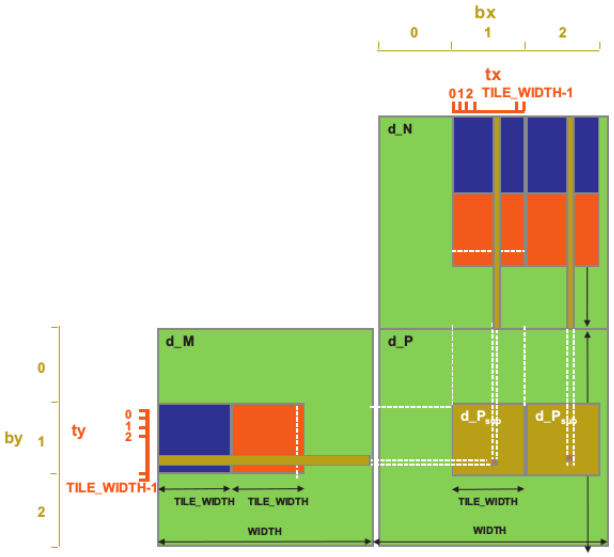
Este método de multiplicación mejora considerablemente, en cuanto a la velocidad de acceso a los datos, debido a que se aprovecha **la memoria compartida** la cual es compartida por los hilos de un bloque, esta memoria en cuanto acceso es mucho más veloz que la memoria global, pero no tiene un espacio 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) {
    Mds[ty][tx] = d_M[Row*WIDTH + ph*TILE_WIDTH + tx];
    Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*WIDTH + Col];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k) {
      Pvalue += Mds[ty][k] * Nds[k][tx];
    }
    __syncthreads();
  }
  d_P[Row*WIDTH + Col] = Pvalue;
}
```

Listing 1: Multiplicación de matrices de mosaico

1.2. Multiplicación de matrices en CUDA (version 2)

En esta nueva versión de multiplicación *tiled* en CUDA, se evita que cada bloque de hilos cargue de forma redundate tiles sobre la matriz M como lo hacía en el kernel anterior, lo podemos apreciar gráficamente en la imagen de abajo:



Multiplicación de matrices con *Tiled*

Para disminuir las veces que se carga de forma redundante sobre la matriz M, vamos a encargar a cada bloque de hilos que cargue el doble en el *tiled* para cada fase sobre la matriz N, generando esto que cada hilo calcule el valor final sobre dos elementos sobre la matriz P, que ha comparación de la imagen de arriba cada bloque de hilos se encarga de un elemento sobre la matriz P, por tanto se reduce el número total de bloques de hilos a la mitad; pero es importante mencionar que este nuevo kernel emplea mucho más registros y memoria compartida a comparación del kernel anterior.

```
__global__ void MatrixMulKernel2(float** M, float** N, float** P)
{
    __shared__ float A_b[TILE_WIDTH][TILE_WIDTH];
    __shared__ float B_b[TILE_WIDTH][TILE_WIDTH];
    __shared__ float B_b2[TILE_WIDTH][TILE_WIDTH];

    __shared__ float R_b[TILE_WIDTH][TILE_WIDTH];
    __shared__ float R_b2[TILE_WIDTH][TILE_WIDTH];

    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH * 2 + threadIdx.x;

   R_b[threadIdx.y][threadIdx.x] = 0.0;
   R_b2[threadIdx.y][threadIdx.x] = 0.0;
    __syncthreads();

    for(int i = 0;i < ceil(COL/(float)TILE_WIDTH);i++){
      A_b[threadIdx.y][threadIdx.x] = 0.0;
      B_b[threadIdx.y][threadIdx.x] = 0.0;
      B_b2[threadIdx.y][threadIdx.x] = 0.0;
       __syncthreads();

       if ((Row<ROW) && (i*THREADS + threadIdx.x<COL)){
          A_b[threadIdx.y][threadIdx.x] = M[Row] [i*TILE_WIDTH + threadIdx.x];
       }

       if ((i*TILE_WIDTH + threadIdx.y<COL) && (Col<COL)){
          B_b[threadIdx.y][threadIdx.x] = N[i*TILE_WIDTH + threadIdx.y][Col];
       }

       if ((i*TILE_WIDTH + threadIdx.y<COL) && (Col+TILE_WIDTH<COL)){
         B_b2[threadIdx.y][threadIdx.x] = N[i*TILE_WIDTH + threadIdx.y][Col+TILE_WIDTH];
       }

         __syncthreads();

         for (int k = 0; k < TILE_WIDTH; k++) {
             R_b[threadIdx.y][threadIdx.x] += A_b[threadIdx.y][k] * B_b[k][threadIdx.x];
             R_b2[threadIdx.y][threadIdx.x] += A_b[threadIdx.y][k] * B_b2[k][threadIdx.x];
         }
         __syncthreads();
  }

  if((Row<ROW) && (Col<COL)){
     P[Row][Col] = R_b[threadIdx.y][threadIdx.x];
  }

  if((Row<ROW) && (Col+TILE_WIDTH<COL)){
        P[Row][Col+TILE_WIDTH] = R_b2[threadIdx.y][threadIdx.x];
  }
}
```

Listing 2: Multiplicación de matrices de mosaico version 2

1.3. Comparación

Para comparar la velocidad de ambos kernels, vamos a probarlos con diferentes dimensiones para matrices de dimensiones NxN, para comodidad de las pruebas vamos a usar para bloques de dimensiones $Hilos$ x $Hilos$, valores que se sean divisibles con la dimensión de las matrices a multiplicar.

	Hilos x N						
	32 x 64	32 x 96	32 x 128	32 x 256	32 x 480	32 x 640	32 x 800
multi V1	0.06352	0.144192	0.274432	2.12768	14.1618	33.4889	69.2017
multi V2	0.0736	0.14752	0.33232	2.16093	12.5142	30.4295	56.2197

Tabla 1: Tabla de resultados de las pruebas en milisegundos

Estas pruebas fueron realizadas en el computador de la Universidad la Salle la cual tiene las siguientes características:

- 1024 hilos por bloque
- 49152 de memoria compartida

- 32 hilos por *warp*

2. Conclusiones

A partir de la tabla de resultados, podemos concluir que ambos kernels tienen similar tiempo de ejecución al inicio, pero cuando el número de elementos se incrementa de manera considerable la diferencia de eficiencia en cuanto a velocidad se hace amplia, superando el primer kernel al segundo. Esta diferencia se dio porque se cargó más elementos en memoria compartida sobre la matriz N, por lo que el bloque de hilos tuvo un universo más amplio de elementos sobre la memoria compartida haciendo que los accesos sean más rápidos sabiendo que este bloque hilos calculará el doble de lo que se hizo con el primer kernel.

Pero debemos tomar en cuenta que se gastará más memoria compartida y registros en el dispositivo, pero no deja de ser un buen método si queremos multiplicar matrices con gran número de elementos.