Algoritmos Paralelos- Uso de tile modificado

Luis Gustavo Cáceres Zegarra

July 14, 2017

1 Granularidad de Threads

Es muy importante el uso correcto de los threads. Es muy ventajoso poner mas trabajo en cada thread y tambien utilizar la menor cantidad de ellos. Esto reduce el trabajo redundante que realizan. En la actual generación dedispositivos CUDA, cada SM(streamming multiprocessor) tiene un limitado ancho de banda de instrucciones que puede realizar. Cada instrucción consume ese ancho de banda, ya sea que sea una instrucción de calculo de punto flotante, de carga, o un paquete de instrucciones. Eliminando el trabajo redundante puede reducir el uso del ancho de banda de una manera considerable.

En capitulos anteriores vimos como utilizamos el uso de tiles para el cálculo de la multiplicación de dos matrices como se muestra en la siguiente figura.

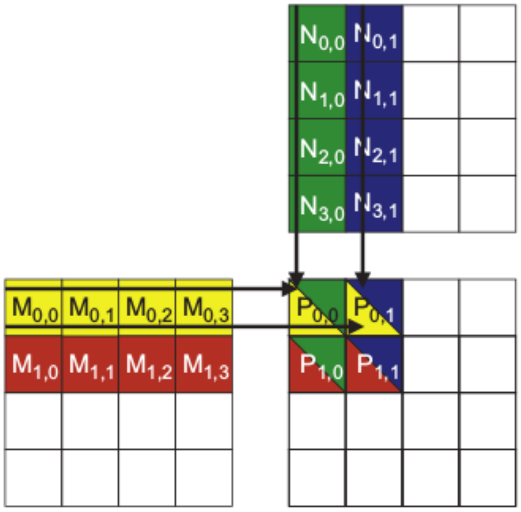

Figure 1: Multiplicación usando tiles.

Se puede ver como un thread es encargado de cargar los datos de la fila 1 de la matriz **M** en la menoria comparatida. La desvantaja de este uso es que cada vez que se quiera computar un valor de la matriz **P** se tiene que hacer la misma carga. Como vimos el concepto de granularidad de threads nos indica que debemos darle la mayor cantidad de trabajo a cada threads y reducir la cantidad que usamos de ellos tambien. Con esto en mente se propone utilizar la carga de threads de la siguiente manera.

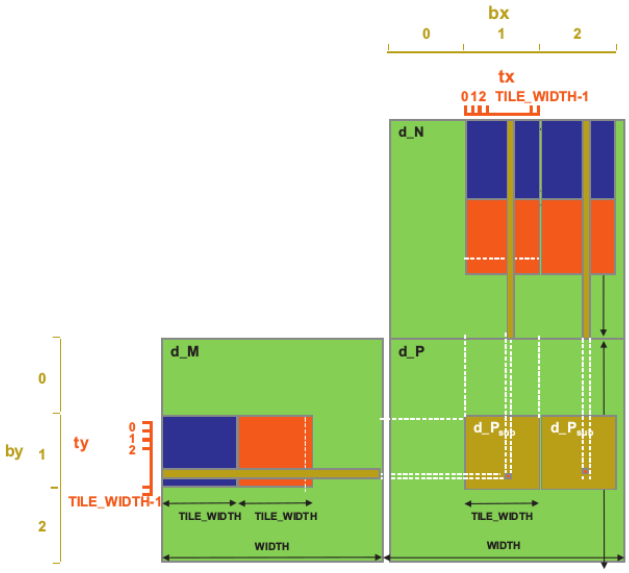

Figure 2: Aumentar la granularidad de threads con tiles rectangulares.

Como se muestra en la figura anterior, el cálculo de dos elementos de **P** en tiles adyacentes usan la misma fila de **M**. Con el algoritmo original usando tiles, la misma fila **M** es cargada dos veces para calcular de valores de la matriz **P** eso genera un trabajo redundante. Esto se logra calculando el producto punto utilizando la misma fila de la matriz **M**. Esto reduce el acceso a la memoria global en 1/4.

El siguiente código muestra la función de cálculo de la matriz.

```
__global__
void  mult_mat_rectangular(int *d_M, int *d_N, int *p,int N){
        __shared__ int Mds[THREAD_PER_BLOCK][THREAD_PER_BLOCK];
        __shared__ int Nds[THREAD_PER_BLOCK][THREAD_PER_BLOCK];

        int bx = blockIdx.x;
        int by = blockIdx.y;

        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int Row = by*THREAD_PER_BLOCK + ty;
        int Col = bx*2*THREAD_PER_BLOCK + tx;

        int Col2 = (bx*2 + 1)*THREAD_PER_BLOCK + tx;

        int p1 = 0;
        int p2 = 0;

        int k = 0;
        int prefM = d_M[Row*N + k*THREAD_PER_BLOCK + tx];
        int prefN = d_N[(k*THREAD_PER_BLOCK + ty)*N + Col];

        int prefN2 = d_N[(k*THREAD_PER_BLOCK + ty)*N + Col2];

        Mds[ty][tx] = prefM;
        Nds[ty][tx] = prefN;
        __syncthreads();
```

```
        for(int m = 0; m < N/THREAD_PER_BLOCK ; ++m){

                prefM = d_M[Row*N + m*THREAD_PER_BLOCK + tx];
                prefN = d_N[(m*THREAD_PER_BLOCK + ty)*N + Col];

                for(int k = 0; k < THREAD_PER_BLOCK; k++){
                        p1 += Mds[ty][k] * Nds[k][tx];
                }

                __syncthreads();

                Nds[ty][tx] = prefN2;

                __syncthreads();

                prefN2 = d_N[(m*THREAD_PER_BLOCK + ty)*N + Col2];

                for(int k = 0; k < THREAD_PER_BLOCK; k++){
                        p2 += Mds[ty][k] * Nds[k][tx];
                }
                __syncthreads();


                Mds[ty][tx] = prefM;
                Nds[ty][tx] = prefN;

        }
        p[Row*N + Col] = p1;
        p[Row*N + Col2] = p2;
}
```

2 Conclusiones

En esta sección analizaremos los resultados obtenidos por los dos algoritmos. El primero es aquel que necesita cargar la fila de la matriz **M** dos veces para calcular dos valores de **P**. El segundo es usando el concepto de **granularidad de threads** para el calculo de los valores **P**, que como se vio en los conceptos teóricos debería reducir en cuarto el acceso a la memoria global.

Algoritmo	Matriz 1000	Matriz 2000
Tile normal	55.3245	382.7642
Usando Granularidad	46.3241	261.4366

Table 1: Tabla de resultados. Tiempo expresado en milisegundos.

Uno de los problemas es que la nueva función kernel ahora usa mas registros y memoria compartida. Como se vio en cápitulos anteriores, el número de bloques que pueden ser ejecutados en un SM(stremming multiprocessor) puede decaer. Para determinado tamaño, esto tambien puede reducir el tamaño de threads por bloque a la mitad, lo cual puede resultar en una paralelización insuficiente en matrices pequeñas. En la practica utilizar hasta 4 bloques adyacentes mejora el desempeño en la multiplicación de matrices.