

Multiplicación de Matrices con Memoria Compartida

Norman Patrick Harvey Arce

Julio 04, 2017

Docente Alvaro Mamani Aliaga

1 Objetivos

Para el producto de 2 de matrices tenemos la matriz A y B, en la que cada bloque de hilos realizará un cálculo para cada “sub-matriz” o *tile* (mosaico) lo que producirá una matriz resultado C.

Esto permitirá reducir el cuello de botella del ancho de banda de la memoria. Además nos obligará a realizar algunas sincronizaciones de los hilos dentro de un bloque.

La memoria compartida dentro de cada procesador se utilizará para almacenar cada sub-matriz antes de los cálculos, lo que acelera el acceso a la memoria global.

2 Características

Arquitectura del GPU: GeForce GT 620 (OEM)

GPU Engine Specs

- CUDA Cores: 48 cores
- Graphics Clock (MHz) 810 MHz
- Processor Clock (MHz) 1620 MHz
- Texture Fill Rate (billion/sec) 6.5

Memory Specs

- Memory Clock Up to 898MHz
- Standard Memory Config 512 MB or 1GB
- Memory Interface DDR3
- Memory Interface Width 64bit
- Memory Bandwidth (GB/sec) 14.4

3 Capacidad de Computación

Compute Capability: 2.1

Technical specifications	2.x
Maximum number of resident grids per device (Concurrent Kernel Execution)	16
Maximum dimensionality of grid of thread blocks	3
Maximum x-dimension of a grid of thread blocks	$2^{31} - 1$
Maximum y-, or z-dimension of a grid of thread blocks	65535
Maximum dimensionality of thread block	3
Maximum x- or y-dimension of a block	1024
Maximum z-dimension of a block	64
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident blocks per multiprocessor	8
Maximum number of resident warps per multiprocessor	48
Maximum number of resident threads per multiprocessor	1536
Number of 32-bit registers per multiprocessor	32K
Maximum number of 32-bit registers per thread block	32K
Maximum number of 32-bit registers per thread	63
Maximum amount of shared memory per multiprocessor	48KB
Maximum amount of shared memory per thread block	48KB
Number of shared memory banks	32
Amount of local memory per thread	512KB
Constant memory size	64KB
Cache working set per multiprocessor for constant memory	8KB
Cache working set per multiprocessor for texture memory	12KB
Maximum width for 1D texture reference bound to a CUDA array	65536
Maximum width for 1D texture reference bound to linear memory	2^{27}
Maximum width and number of layers for a 1D layered texture reference	16384×2048
Maximum width and height for 2D texture reference bound to a CUDA array	65536×65535
Maximum width and height for 2D texture reference bound to a linear memory	65000^2
Maximum width and height for 2D texture reference bound to a CUDA array supporting texture gather	16384^2
Maximum width, height, and number of layers for a 2D layered texture reference	$16384 \times 16384 \times 2048$
Maximum width, height and depth for a 3D texture reference bound to linear memory or a CUDA array	4096^3
Maximum width and number of layers for a cubemap layered texture reference	16384×2046
Maximum number of textures that can be bound to a kernel	256
Maximum width for a 1D surface reference bound to a CUDA array	65536
Maximum width and number of layers for a 1D layered surface reference	65536×2048
Maximum width and height for a 2D surface reference bound to a CUDA array	65536×32768
Maximum width, height, and number of layers for a 2D layered surface reference	$65536 \times 32768 \times 2048$
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array	$65536 \times 32768 \times 2048$
Maximum width and number of layers for a cubemap layered surface reference	32768×2048
Maximum number of surfaces that can be bound to a kernel	16
Maximum number of instructions per kernel	512 Millions

4 Código

```
#define TILE_WIDTH M //El tamanho del mosaico depende del programador
#include <iostream>

using namespace std;

__global__ void matrixMultiply(float * A, float * B, float * C, int tam)
{
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty, Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;

    for (int ph = 0; ph < tam/TILE_WIDTH; ++ph)
    {
        ds_A[ty][tx] = A[Row*tam + ph*TILE_WIDTH + tx];

        ds_B[ty][tx] = B[(ph*TILE_WIDTH + ty)*tam + Col];

        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
        {
            Pvalue += ds_A[ty][k] * ds_B[k][tx];
        }

        __syncthreads();
    }
    C[Row*tam + Col] = Pvalue;
}

int main(int argc, char ** argv) {

    float * hostA; // The A matrix
    float * hostB; // The B matrix
    float * hostC; // The output C matrix
    float * deviceA;
    float * deviceB;
    float * deviceC;

    int tam=16;

    hostA = (float *)malloc(sizeof(float) * tam * tam);
    hostB = (float *)malloc(sizeof(float) * tam * tam);
    hostC = (float *)malloc(sizeof(float) * tam * tam);
```

```

for (int i=0; i<tam; i++)
{
    hostA[i]=i;
    hostB[i]=tam-i;
}

cudaMalloc(&deviceA, sizeof(float) * tam * tam);
cudaMalloc(&deviceB, sizeof(float) * tam * tam);
cudaMalloc(&deviceC, sizeof(float) * tam * tam);

cudaMemcpy(deviceA, hostA, sizeof(float) * tam * tam, cudaMemcpyHostToDevice);
cudaMemcpy(deviceB, hostB, sizeof(float) * tam * tam, cudaMemcpyHostToDevice);

dim3 dimGrid((tam-1)/TILE_WIDTH+1, (tam-1)/TILE_WIDTH+1, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

matrixMultiply<<<dimGrid, dimBlock>>>(deviceA, deviceB, deviceC, tam);

cudaThreadSynchronize();

cudaMemcpy(hostC, deviceC, sizeof(float) * tam * tam, cudaMemcpyDeviceToHost);

cudaFree(deviceA);
cudaFree(deviceB);
cudaFree(deviceC);

free(hostA);
free(hostB);
free(hostC);

return 0;
}

```

5 Análisis

Existen diferentes tipos de memoria en CUDA, por ejemplo, Memoria Global, Constante, de Textura, Compartida y Registros. Siendo la **memoria global** la más lenta y los **registros** más rápidos.

El tamaño de nuestro mosaico (*tile*) debe ser un número que sea sub-múltiplo del tamaño de la matriz ya que de no ser así podría no abarcar a toda la matriz, o desbordar la matriz. En este último caso, se recomienda llenar con ceros los espacios sobrantes.

Además, depende del tamaño del mosaico, el porcentaje en el que se reducirá el número de accesos a memoria global del dispositivo, haciendo más eficiente el performance del algoritmo y del uso del GPU.

Para hacer el cálculo de un *sub-bloque* de C , necesitamos la correspondiente fila de A y la correspondiente columna de B . Ahora, si dividimos A y B en sub-bloques A_{sub} y B_{sub} , podemos actualizar los valores en C iterativamente sumando los resultados de las multiplicaciones de $A_{\text{Sub}} \times B_{\text{Sub}}$.

Esto es importante ya que hacemos re-uso de los datos dentro de un mosaico (tile).

El objetivo principal es compartir los datos de la sub-matriz A_{sub} y B_{sub} dentro de un “grupo de trabajo” (que en CUDA llamamos *threadblock*) a través de la memoria local. Para maximizar el beneficio del re-uso, debemos hacer los mosaicos lo más grande posible.

Por último, dependiendo del tamaño del mosaico, si fuera $K=32$, nos daría un factor de reducción 32 en accesos a memoria.