Chapter 6. Example of Matrix Multiplication

6.1 Overview

The task of computing the product *C* of two matrices *A* and *B* of dimensions *(wA, hA)* and *(wB, wA)* respectively, is split among several threads in the following way:

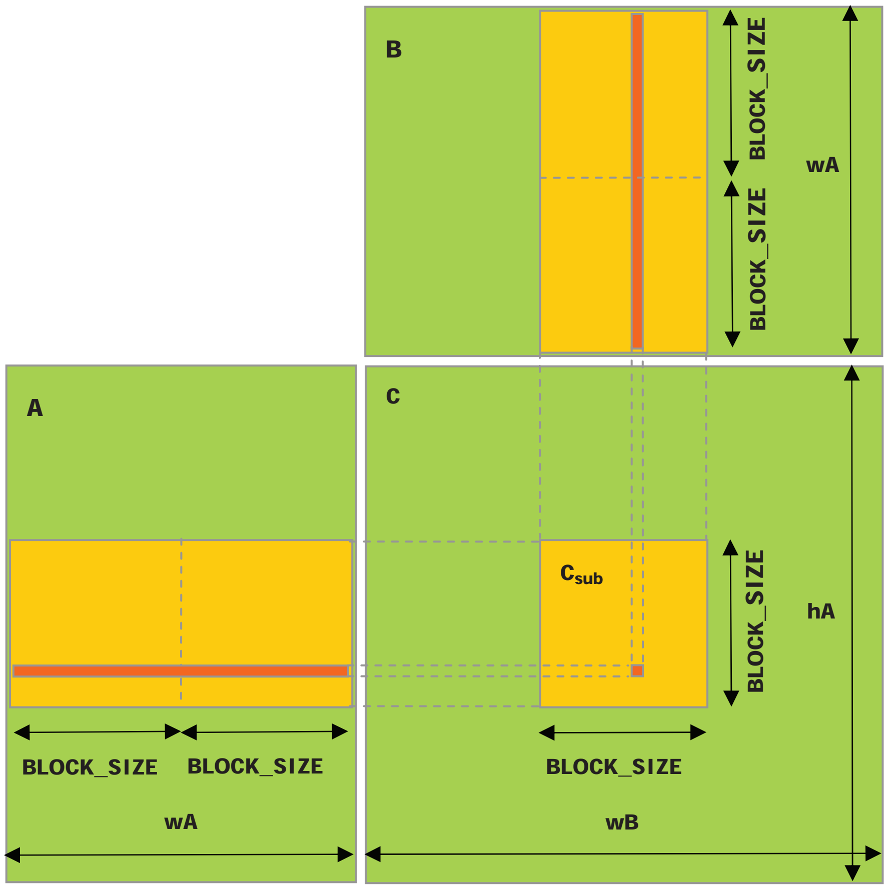
- Each thread block is responsible for computing one square sub-matrix C_{sub} of *C*;
- Each thread within the block is responsible for computing one element of C_{sub}.

The dimension *block_size* of C_{sub} is chosen equal to 16, so that the number of threads per block is a multiple of the warp size (Section 5.2) and remains below the maximum number of threads per block (Appendix A).

As illustrated in Figure 6-1, C_{sub} is equal to the product of two rectangular matrices: the sub-matrix of *A* of dimension *(wA, block_size)* that has the same line indices as C_{sub}, and the sub-matrix of *B* of dimension *(block_size, wA)* that has the same column indices as C_{sub}. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension *block_size* as necessary and C_{sub} is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since *A* and *B* are read from global memory only *(wA / block_size)* times.

Nonetheless, this example has been written for clarity of exposition to illustrate various CUDA programming principles, not with the goal of providing a high-performance kernel for generic matrix multiplication and should not be construed as such.



Each thread block computes one sub-matrix C_{sub} of C. Each thread within the block computes one element of C_{sub}.

Figure 6-1. Matrix Multiplication

6.2 Source Code Listing

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// Device multiplication function called by Mul()
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
```

```
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.3 Source Code Walkthrough

The source code contains two functions:

- **`Mul()`**, a host function serving as a wrapper to **`Muld()`**;
- **`Muld()`**, a kernel that executes the matrix multiplication on the device.

6.3.1 `Mul()`

`Mul()` takes as input:

- Two pointers to host memory that point to the elements of *A* and *B*,
- The height and width of *A* and the width of *B*,
- A pointer to host memory that points where *C* should be written.

`Mul()` performs the following operations:

- It allocates enough global memory to store *A*, *B*, and *C* using **`cudaMalloc()`**;
- It copies *A* and *B* from host memory to global memory using **`cudaMemcpy()`**;
- It calls **`Muld()`** to compute *C* on the device;
- It copies *C* from global memory to host memory using **`cudaMemcpy()`**;
- It frees the global memory allocated for *A*, *B*, and *C* using **`cudaFree()`**.

6.3.2 `Muld()`

`Muld()` has the same input as **`Mul()`**, except that pointers point to device memory instead of host memory.

For each block, **`Muld()`** iterates through all the sub-matrices of *A* and *B* required to compute C_{sub}. At each iteration:

- It loads one sub-matrix of *A* and one sub-matrix of *B* from global memory to shared memory;
- It synchronizes to make sure that both sub-matrices are fully loaded by all the threads within the block;
- It computes the product of the two sub-matrices and adds it to the product obtained during the previous iteration;

- It synchronizes again to make sure that the product of the two sub-matrices is done before starting the next iteration.

Once all sub-matrices have been handled, C_{sub} is fully computed and **`Muld()`** writes it to global memory.

`Muld()` is written to maximize memory performance according to Section 5.1.2.1 and 5.1.2.4.

Indeed, assuming that **`wA`** and **`wB`** are multiples of 16 as suggested in Section 5.1.2.1, global memory coalescing is ensured because **`a`**, **`b`**, and **`c`** are all multiples of **`BLOCK_SIZE`**, which is equal to 16.

There is also no shared memory bank conflict since for each half-warp, **`ty`** and **`k`** are the same for all threads and **`tx`** varies from **`0`** to **`15`**, so each thread accesses a different bank for the memory accesses **`As[ty][tx]`**, **`Bs[ty][tx]`**, and **`Bs[k][tx]`** and the same bank for the memory access **`As[ty][k]`**.