Matrix Multiplication in CUDA with Shared Memory

Paul Richmond

This document provides explanation as to how to adapt the Lab06 starting code (link) to implement a shared memory Matrix Multiplication using CUDA.

Explanation for the Starting Code

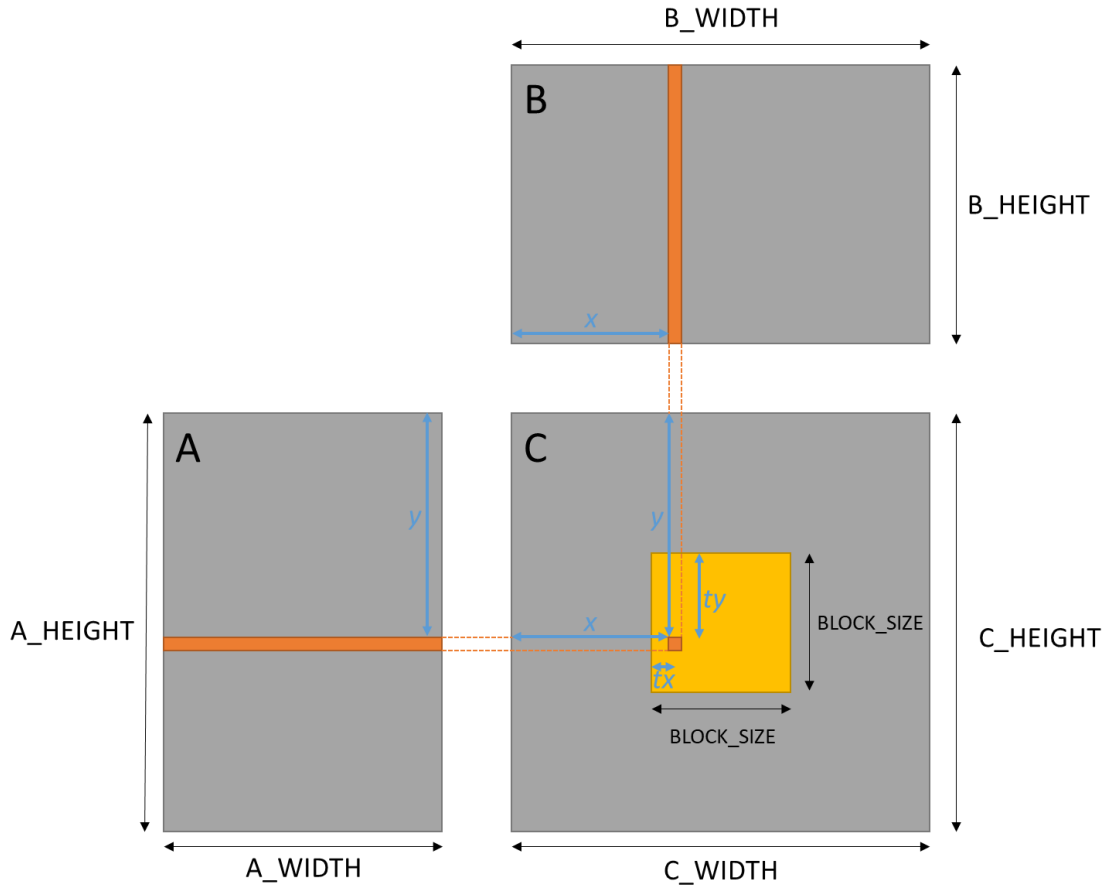


Figure 1 - Naive CUDA Matrix Multiply

Exercise one asks you to modify an implementation of a naive CUDA matrix multiply (shown in the above figure). The kernel for this is shown below.

```
__global__ void matrixMulCUDA()
{
	// Block index
	int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	int x = bx*BLOCK_SIZE + tx;
	int y = by*BLOCK_SIZE + ty;


	float Csub = 0;
	//iterate A_WIDTH (same as B_HEIGHT) to calculate the product
	for (int k = 0; k < A_WIDTH; k++){
```

```
            Csub += d_A[y][k] * d_B[k][x];
        }

        // Store the product value of C matrix
        d_C[y][x] = Csub;
}
```

The kernel is launched using the following two dimensional grid block configuration which will ensure the device has unique thread for each element of the Matrix C;

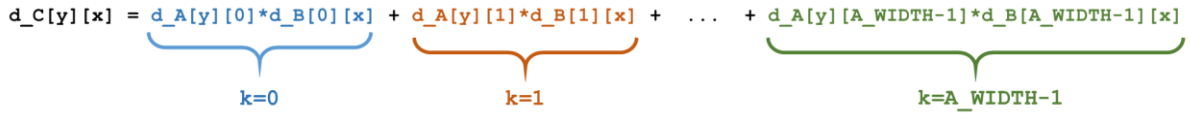
```
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(C_WIDTH / BLOCK_SIZE, C_HEIGHT / BLOCK_SIZE);
```

The implementation multiplies the Matrix A (`d_A`) by a Matrix B (`d_B`) storing the result in a Matrix C (`d_C`). All of the Matrices are stored globally on the device as two dimensional arrays. Importantly `A_WIDTH` is always equal to `B_HEIGHT`.

In this `matrixmulCUDA` kernel the variables `bx, by, tx` and `ty` are used to provide shorthand notation for `blockIdx.x, blockIdx.y, threadIdx.x` and `threadIdx.y` respectively. If we examine the inner loop we can see how the Matrix Multiplication algorithm works.

```
for (int k = 0; k < A_WIDTH; k++){
        Csub += d_A[y][k] * d_B[k][x];
}
```

The `for` loop will take pairwise elements from a row of A and multiply this with a pairwise element of a column of B at index `k` within the respective row or column. E.g. If we consider (from Figure 1) the value of Matrix C, at position `x, y` (i.e. `d_C[y][x]`) this will be the result of the following pairwise multiplications (where each pairwise multiplication is from a single iteration of the loop);



In this case the value of `x` and `y` is calculated from threads block and grid identifiers as shown in the kernel. The implementation is currently **very inefficient** as for each element of the Matrix C, each thread will perform `A_WIDTH` global memory loads from both the Matrix A and B.

If you are still unclear at this stage about how the starting code works, then you should seek further help or read up yourself on the Matrix Multiplication algorithm.

https://en.wikipedia.org/wiki/Matrix_multiplication

A Single Block Shared Memory Implementation of Matrix Multiplication

From the initial implementation we can deduce the following;

1. Each thread performs a large number of global memory loads.
2. Elements of each row in A and each column in B are loaded multiple times. E.g. every element of Matrix C with a common `x` position will load the same entire row from Matrix A.
3. Shared memory could be used to allow threads to work together to minimise the the number of global memory load. E.g. a block wise approach could be taken where threads co-operate to load values into shared memory.

Before explaining the tiled matrix multiplication approach given in the lab solution first we will consider a simple approach for very small matrices. In the case of very small matrices we can simplify the problem by implementing a single block version of the code (where only a single block of threads will be launched using the following configuration);

```
dim3 threads(C_WIDTH, C_HEIGHT);
dim3 grid(1, 1);
```

This can be represented by the following diagram (Figure 2) where the BLOCK_SIZE is equal to the square dimensions of Matrix C (i.e. BLOCK_SIZE=C_WIDTH and C_HEIGHT).

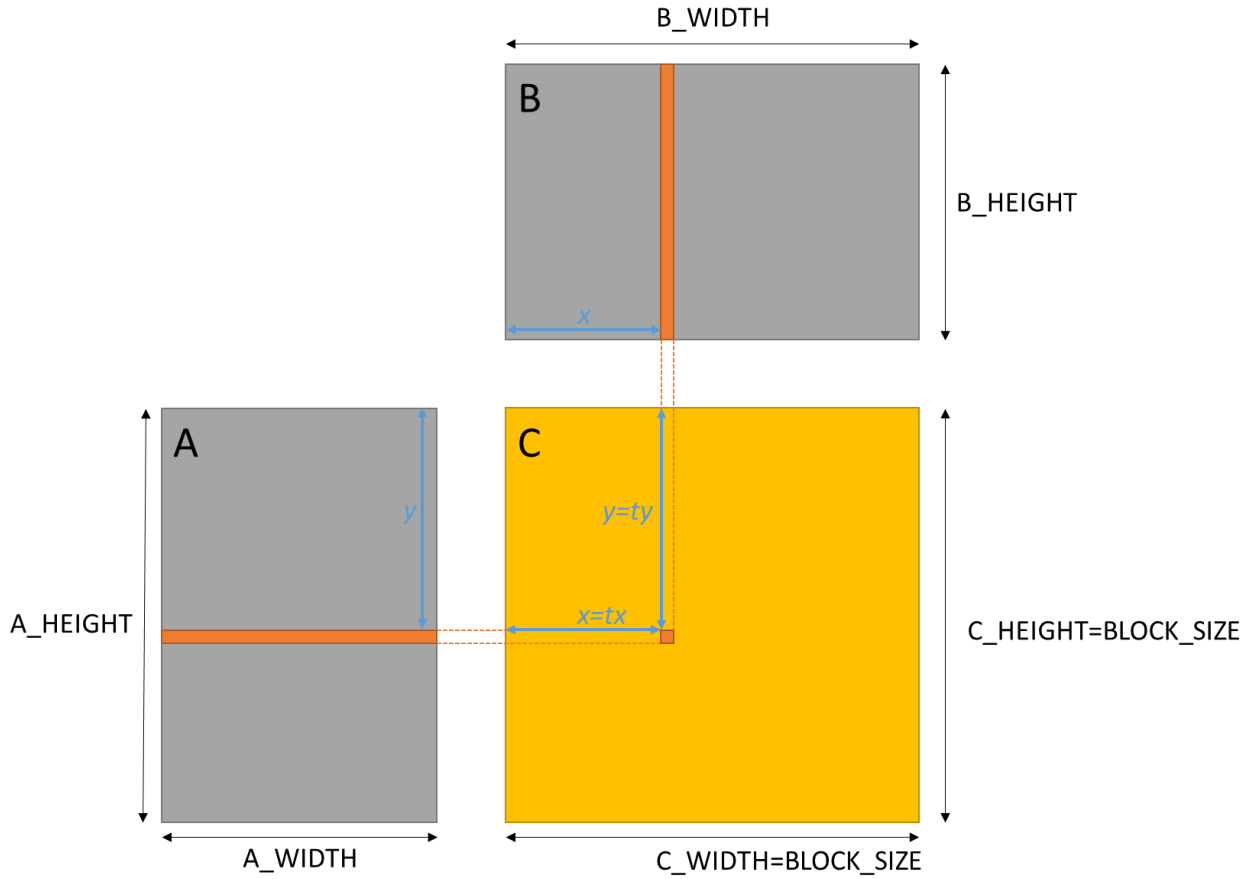


Figure 2 - A Single block representation of a shared memory matric multiplication

In the case of a single block implementation we can move the entirety of Matrix A and B into shared memory. Each thread in the block would be responsible for moving at most a single element of the Matrix A and Matrix B. E.g.

```
__global__ void matrixMulCUDASingleBlock()
{
	__shared__ float As[A_HEIGHT][A_WIDTH];
	__shared__ float Bs[B_HEIGHT][B_WIDTH];

	int x = threadIdx.x;
	int y = threadIdx.y;

	//Each thread load at most a single an element of A and B
	if (x < A_WIDTH) {
		As[y][x] = d_A[y][x];
	}
```

```
        if (y < B_HEIGHT) {
                Bs[y][x] = d_B[y][x];
        }
        // Sync to ensure each matrix is fully loaded into shared memory
        __syncthreads();

        // Sum pairwise products of Matrix A and Matrix B
        float C = 0;
        for (int k = 0; k < A_WIDTH; ++k)
        {
                C += As[y][k] * Bs[k][x];
        }

        // Store the product value in matrix C
        d_C[y][x] = C;
}
```

In the above implementation enough shared memory is allocated for the entire Matrix A and the entire Matrix B. Each thread in the block is responsible for loading at most a single element of Matrix A and a single element of Matrix B into shared memory. A check is required to ensure that the `x` position does not exceed the width of Matrix A and that the `y` position does not exceed the height of B. If both A and B were square matrices then each thread would load exactly one value from each of the matrices (A and B). After loading a value into shared memory a call to `__syncthreads` is made. This ensure that all warps which represent the thread block have reached the same position in the program. Without this call some warps may begin reading from areas of shared memory which have not yet been populated by threads in other warps. The inner loop of the kernel now performs the pairwise calculations by only reading from shared memory. The number of global memory reads has been significantly reduced as each thread is now loading values (into shared memory) which are used by other threads in the block.

A Tiled Block Shared Memory Implementation of Matrix Multiplication

The single block version of the matrix multiply has the following problems;

1. It relies on a single block of threads which will obtain very poor device utilisation. Only a single streaming multiprocessor will be used to execute the block and the rest will be idle.
2. The implementation is only suitable for very small matrices. Shared memory is a limited resource (much more limited that GPU global memory) and as such it is not possible to load the entire Matrix A and B for larger matrix sizes.

An alternative solution is required which is able to use multiple thread blocks to calculate the Matrix C but still utilise shared memory to reduce the number of global memory reads. A tiled matrix multiply can achieve this by breaking down the matrices into tiled chunks[1]. As with the naive implementation we can launch a grid of thread block which partitions the matrix C into tiles of `BLOCK_SIZE` by `BLOCK_SIZE` through the following grid block configuration.

```
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(C_WIDTH / BLOCK_SIZE, C_HEIGHT / BLOCK_SIZE);
```

If we consider the data which is required by a single thread block to calculate all of the blocks values of Matrix C it will correspond to a number of square tiles (the same size as the thread block) from

[1] We will assume for simplicity that the width of Matrix B and the height of Matrix A are also divisible by BLOCK_HEIGHT.

the Matrix A and Matrix B. E.g. If we consider a block size which is ½ of `A_WIDTH` (and `B_HEIGHT`) as shown in Figures 1 and 2 then we can describe the problem with Figure 3.

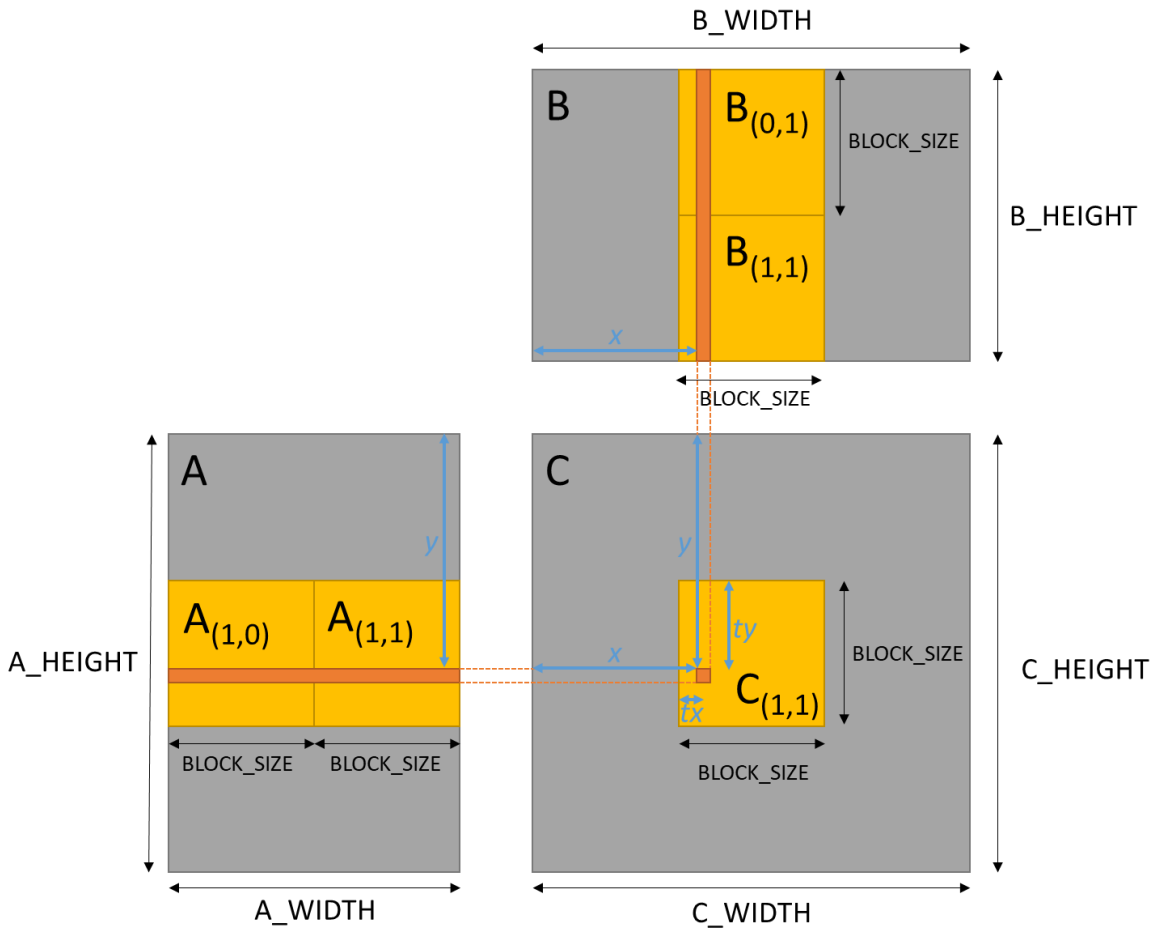


Figure 3 - A tiles shared memory matrix multiplication. Tile indices are specified in the format $A_{(y, x)}$. I.e. $A_{(1,0)}$ is the tile of matrix A at x position 0 and y position 1.

Figure 3 shows that all of the data required to calculate the values of Matrix C within the grid block (or tile location) $C_{(1,1)}$ is located within the tiles $A_{(1,0)}$ and $A_{(1,1)}$ of Matrix A and $B_{(0,1)}$ and $B_{(1,1)}$ of Matrix B. In order to perform the pairwise multiplications, we can consider a single tile from each of Matrix A and B in isolation and sum this with the results of other tile calculations. E.g. From Figure 3 consider the calculation of the value of `C[y][x]`. This requires a loop (of `BLOCK_SIZE`) through each pair of tiles ($A_{(1,0)}$ with $B_{(0,1)}$ and $A_{(1,1)}$ with $B_{(1,1)}$). The code for this calculation can be summarised as;

```
float CSum = 0;
for(k = 0; k < BLOCK_SIZE-1; k++){
    CSum += A(1,0)[ty][k]*B(0,1)[k][tx];      } Tiles A(1,0) and B(0,1)
}
for(k = 0; k < BLOCK_SIZE-1; k++){
    CSum += A(1,1)[ty][k]*B(1,1)[k][tx];      } Tiles A(1,1) and B(1,1)
}
C[y][x] = Csum;
```

To implement this as a CUDA kernel each thread block should loop through the number of tile pairs (There will always be `A_WIDTH / BLOCK_SIZE` pairs of tiles to consider) and perform the following tasks;

1. Load a subtitle of Matrix and A and B into shared memory
2. Synchronise to ensure the memory loads are complete
3. Sum the products of the pairwise elements over range of 0 to BLOCK_SIZE
4. Sum the resulting value with the results from other tile pairs.

The resulting CUDA kernel can therefore be written as;

```
__global__ void matrixMulCUDASharedMemory()
{
	__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
	__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

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

	float CSum = 0;

	// Loop over number of tiles
	for (int i = 0; i < A_WIDTH/BLOCK_SIZE; i++){
		// Calculate indices of A and B matrix to load into shared memory
		int a_x = (i*BLOCK_SIZE) + tx;
		int a_y = (by*BLOCK_SIZE) + ty;
		int b_x = (bx*BLOCK_SIZE) + tx;
		int b_y = (i*BLOCK_SIZE) + ty;

		// Load into shared memory
		As[ty][tx] = d_A[a_y][a_x];
		Bs[ty][tx] = d_B[b_y][b_x];

		// Sync to ensure matrix tiles are fully loaded
		__syncthreads();

		// Sum products of A and B matrix tiles
		for (int k = 0; k < BLOCK_SIZE; ++k)
		{
			CSum += As[ty][k] * Bs[k][tx];
		}

		// Sync to prevent blocks modifying shared memory
		__syncthreads();
	}

	// Store the result in matrix C
	int c_x = (bx*BLOCK_SIZE) + tx;
	int c_y = (by*BLOCK_SIZE) + ty;
	d_C[c_y][c_x] = CSum;
}
```

Note how within the loop over the tiles, the value of `a_x` and `b_y` varies whilst the values of `a_y` and `b_y` are constant.