Chapter 4

Reduced Matrix Multiplication using CUDA

4.1 Introduction

We are given two NxN matrix A and B and after reduced matrix multiplication we get N/2 X N/2 matrix C. GPU has multiple SMs and each SM has multiple cores. There is grid (1D, 2D or 3D) of thread blocks and each thread blocks contains grid (1D, 2D or 3D) of K threads (which we can define). We have implemented mainly two types of kernels. Kernel contains code which is executed by each thread and which ultimately contains logic of multiplication. Different types of work distribution of each threads results in various kernels.

First kernel contains naive version of reduced matrix multiplication, in which we have used 1D grid of thread blocks. Then we optimized the code and implemented tiled version of RMM, which is more cache efficient. To judge the performance we have used *nvprof* GPU profiler tool.

4.2 Processor specifications

CPU	Intel i5-9300H @2.40 GHz, 8 cores
CPU Memory	8 GB
GPU	NVIDIA GeForce GTX 1050 Mobile
CUDA Version	11.7
GPU Memory	4 GB

Table 4.1:

4.3 Working and Analysis

In this section, we discuss working of various kernels followed by it's analysis.

4.3.1 Naive Version

Working: In Naive version, each thread will calculate 1 element of C. so we require total N/2 X N/2 threads. Now we accommodate 1024 (k = 1024) threads

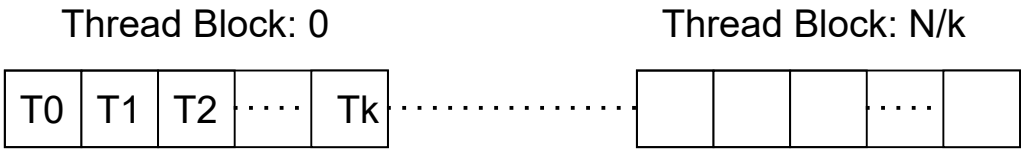

Figure 4.1: 1D thread block grid used for naive kernel of matrix multiplication. Each thread block has 1024 (k = 1024) threads, hence we require N/1024 thread blocks.

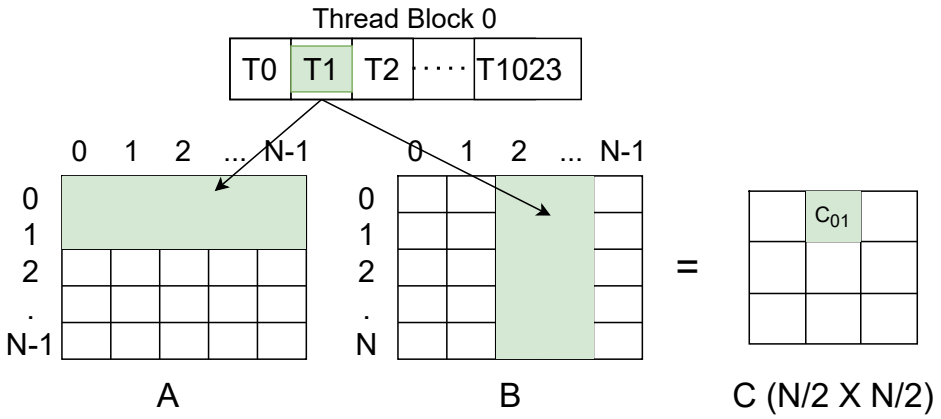

Figure 4.2: Thread work distribution in naive kernel of reduced matrix multiplication. Each thread will read two rows from matrix A and two columns of B and calculate one element of C. Hence we require N/2 X N/2 threads.

in single thread block as shown in Figure 4.1. Hence we require total N/1024 thread blocks. Note that we are storing threads in 1D manner in particular thread block and each thread block are also in 1D grid. Now, to calculate C_{ij}, we require $2i^{\text{th}}$ and $(2i+1)^{\text{th}}$ row of A and $2j^{\text{th}}$ and $(2j+1)^{\text{th}}$ column of B. For example, Figure 4.2 shows calculation of C_{01} done by thread 1 in thread block - 0. Note that each thread will calculate C_{ij} using 4p multiplications operations as shown below
$C_{01} = (0^{\text{th}}\ \text{rowA} * 2^{\text{nd}}\ \text{of colB}) + (0^{\text{th}}\ \text{rowA} * 3^{\text{rd}}\ \text{of colB}) + (1^{\text{st}}\text{rowA} * 2^{\text{nd}}\ \text{of colB}) + (1^{\text{st}}\ \text{rowA} * 3^{\text{rd}}\ \text{of colB})$

However, multiplication operation is one of costly in terms of number of cycles, so we can reduce the number of multiplications using some tricks, which we present in next section.
Analysis:
For statistical analysis, we have used 'nvprof' CLI tool. We extracted gpu-trace which contains execution time for each part of program (for example cuda memcpy time host to device and device to host, kernel execution time, etc.). Table **??** shows the comparison between execution time of unoptimzed code and GPU naive version.

4.3.2 Naive version with reduced multiplications:

We can do same calculations of C_{ij} with only p **multiplications** by first adding two rows and two columns and then multiplying it as shown following,

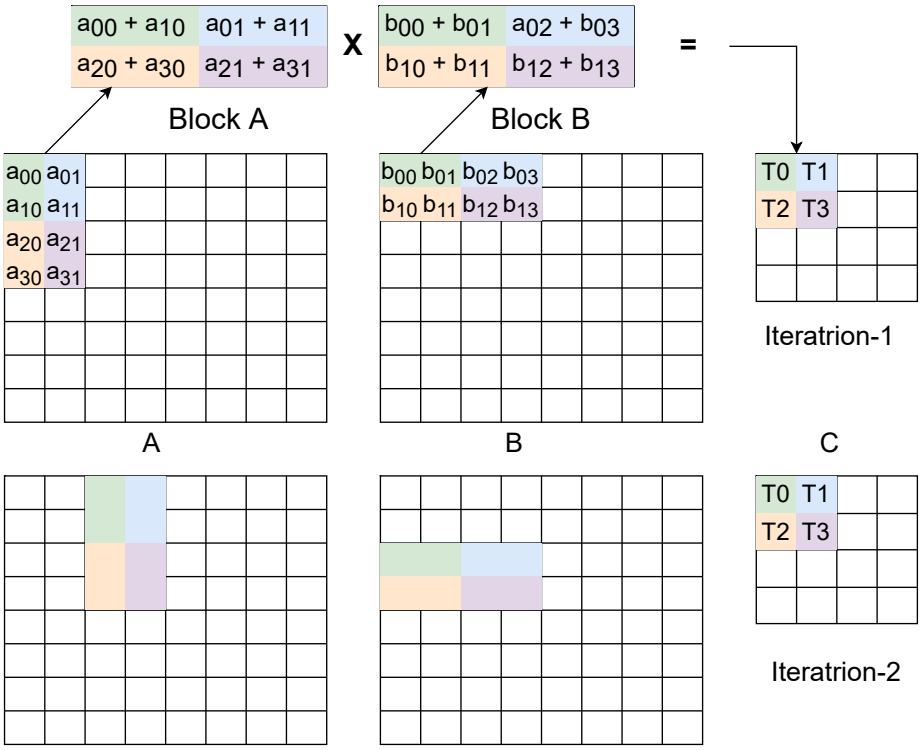

Figure 4.3: Working of tiled version of reduced matrix multiplication.

Matrix size	Un-optimized Single threaded code	Naive GPU version (4p mult.)	Naive GPU version (p mult.)	Tiled Version (Tile = 8)	Tiled Version (Tile = 16)	Tiled Version (Tile = 32)
128	5.80 ms	120.00 μs	44.80 μs	17.07 μs	13.05 μs	12.32 μs
512	630.94 ms	5.28 ms	2.01 ms	648 μs	455 μs	455 μs
2048	47.28 s	365.55 ms	145.72 ms	45.15 ms	27.4 ms	27.4 ms
8192	80 mins	24.94 s	12.08 s	4.39 s	1.98 s	1.81 s
16394	16 hrs	193.03 s	89.05 s	42.3 s	16.2 s	14.25 s

Table 4.2: Execution running time comparisons with GPU kernels which we have implemented. We can see that among these, Tiled version with tile size = 32 performs best. Note that the kernel execution time is calculated through *nvprof*.

$C_{01} = (0^{\text{th}} \text{ rowA} + 1^{\text{st}} \text{ rowA}) * (2^{\text{nd}} \text{ colB} + 3^{\text{rd}} \text{ colB})$

Analysis: Table 4.3.2 shows the execution comparison of the execution time with p multiplications and 4p multiplications. We can clearly see that just reducing number of multiplication operations effects well on execution time. For example we are getting 2X performance in case of matrix size = 16384.

4.3.3 Tiled Version

Working: To understand the working of tiling, we take example of multiplication of size 8X8 with Tile = 2, as shown in Figure 4.3. To calculate one tile of matrix C, In first iteration, 4 threads read 4X2 tile from matrix A and sum-up two vertical cells to make it 2X2 in blockA. Similarly 4 threads read 2X4 tile from matrix B and sum-up two horizontal cells to load it in 2X2 in blockB. blockA and blockB is multiplied by four threads, and temporaily stored in blockC. Note

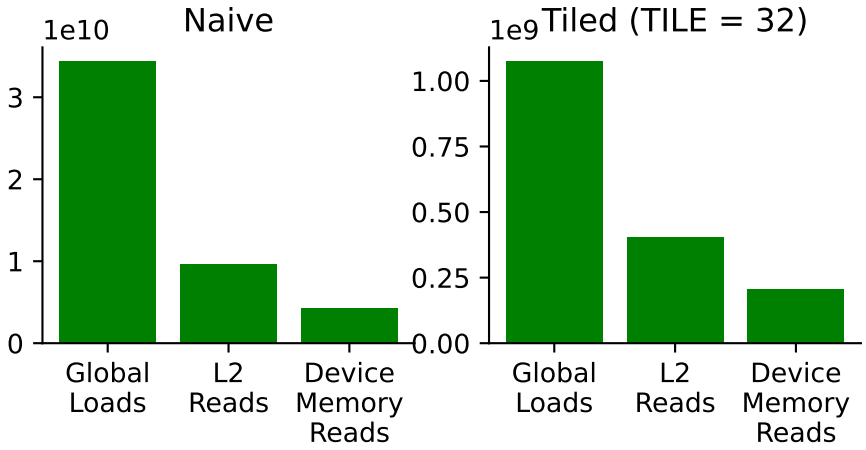

Figure 4.4: Global loads, L2 cache reads, and device memory reads for the matrix size = 4096 with naive and tiled kernel. Loads in naive version is around 30 times greater than loads by tiled version.

that this multiplication is also done in parallel like we have done in naive version. After that we will partially update matrix C by writing this 4 elements. In 2nd iteration, 4X2 tile of matrix A is shifted horizontally by 2 (tile size) and 2X4 tile of matrix B is shifted vertically by 2 (tile size). Again this data is loaded temporarily in blockA and blockB respectively and again multiplication of this two blocks is performed by 4 (tile X tile) threads. Again this result is updated in first tile of matrix C. This process runs for N/tile iterations to fill 1st tile of matrix C. Note that all tiles of matrix C is being calculated parallelly by threads.

Q: Why tile version should perform better than naive?
Intuitively, using tiles in multiplication gives advantage of cache. But theoretically tile can reduce number of global loads compared to naive version. Let's take example of reduced multiplication of two 8X8 matrices. We have already seen that number of threads in both the versions are N/2 X N/2. So we compare calculations of global loads by each thread in both the versions,

A) Naive Kernel:

Total loads by T0 = 2*row size of A + 2*column size of B
Total loads by T0 = 4*N

B) Tiled Kernel:

Total loads by T0 = no. of iterations * (2 elements from A + 2 elements from B)
Total loads by T0 = (N/TILE) * 4

We can observe that in tiled version global loads by each thread is reduced by TILE. This reduction is due to local loads in shared blockA and blockB. But access to shared memory of thread block is much faster than global memory access.

Analysis: In this section, we verify the theoretical claims given previously and compare the statistics.

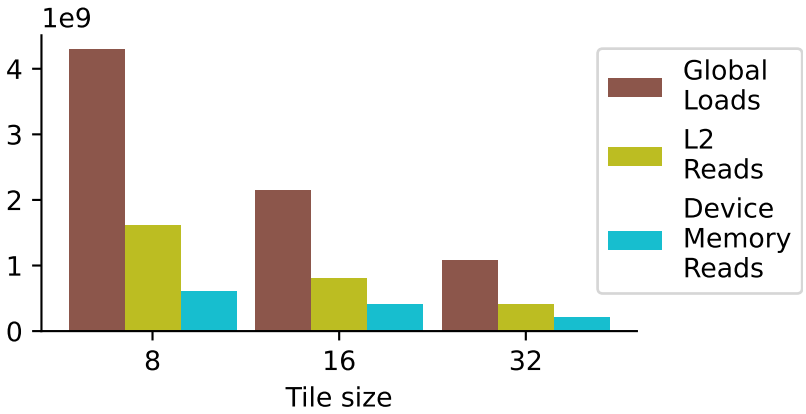

Figure 4.5: Global loads decreases as tile size increase.

First we have taken readings of global loads, L2 cache reads, and device memory reads (GPU DRAM reads). Note that from these readings we can infer important readings of memory at 3 levels (L1 cache, L2 cache and DRAM), for example every global loads will pass through L1 cache, hence L1 misses = global loads - L2 reads.

Naive vs Tiled: Figure 4.4 shows the global loads, L2 cache reads, and device memory reads for the matrix size = 4096 with naive and tiled kernel. We can observe that Naive version around 30 times greater than loads by tiled version, which matches our theory as well, since tile size = 32. Since naive version has large number of DRAM loads than tile version, it takes long time to perform operations.

Tiles comparisons: Previously we have shown that number of global loads is inversely propotional to the tile size, Also the Table 4.3.2 shows that smaller tile size takes more time than bigger tile size. From Figure 4.5, we can see that global loads decreases as tile size increase, here natural question arises that **Why we are not using more than tile size 32?**. Currently mostly all the nvidia GPUs support maximum 1024 threads per thread block. Since number of threads per thread block = TILE X TILE, so we can not have TILE greater than 32 in our implementation.