E0 – 243

Computer Architecture

Assignment 3

(IISc Bangalore)

GitHub Repository

https://github.com/karm-patel/Assignment_3

By

Yash Punjabi, Karm Patel

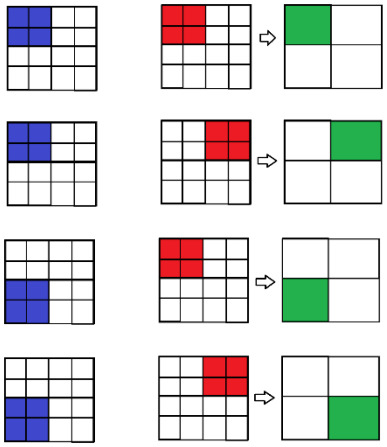
(yashpunjabi@iisc.ac.in, 20870)

(karmpatel@iisc.ac.in, 21542)

REDUCED MATRIX MULTIPLICATION

1. Introduction

We are given implementation of modified matrix multiplication. It takes two (N x N) matrices and gives an output matrix of dimension (N/2 x N/2). Here we have N = 2^k where k>=2.
It works by taking the first two rows of A and first two columns of B, multiplying them and adding all the elements and placing it into the final output matrix as one entry. This process is continued for all possible pairs of rows and columns. The illustration of above is represented below



The image shows the computation of partial answers in iteration, the same process would be followed. The movement of red block depicts inner loop iteration

Since it is a normal matrix multiplication just with reduced size and also having different blocks multiplication at every iteration which are independent of each other, there can be a potential to optimize the operations and improve the overall performance

Our task is to optimize this naive implementation and create another multi-threaded version as well

2. Processor Specifications

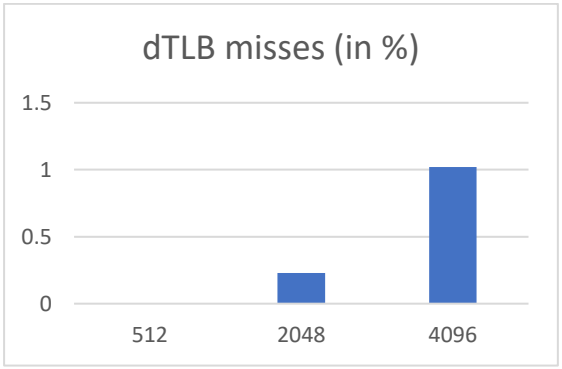
CPU	11th Gen Intel(R) Core (TM) i5-11500 @ 2.70GHz
Memory	16 GB DDR4
Cache	L1d cache = 48 KB, L1i cache = 32KB, L2 cache = 512KB, L3 cache = 12 MB
OS	Linux Ubuntu 22.04
PAGE SIZE	4096B

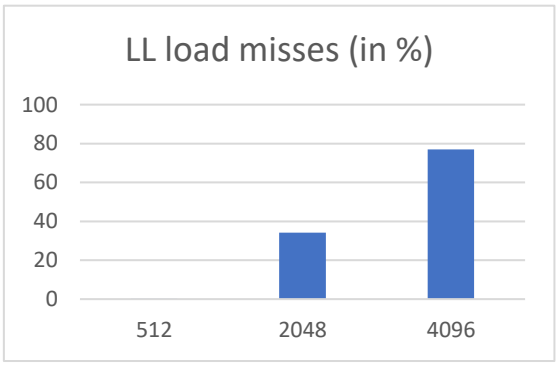
3. Analysis

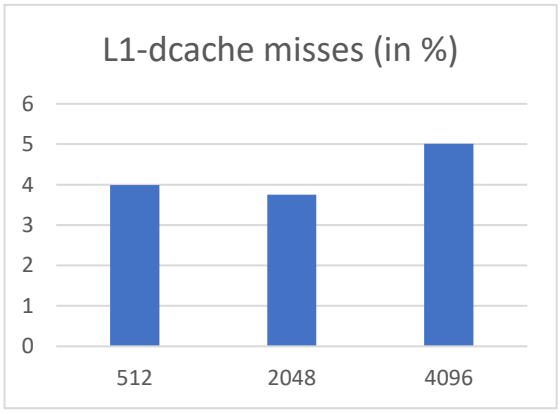
We will analyze the above implementation on various matrix sizes in order to identify the ways to optimize and reason about it to improve the overall performance
For analysis perf tool was used. The analysis was done on different input sizes varying from N = 16 to N = 16384, but for report we will show analysis on three input sizes, N = 512, 2048, 4096

3.1 Issues with reduced multiplication and need for blocking

In reduced matrix multiplication for each two rows of matrix A, matrix B two columns are accessed in column wise. But we know matrices are stored in memory in either row major or column order, since here we are trying to access matrix in both fashions, we can see the amount of miss percentage in caches.







By seeing the above graphs, it can be observed reduced matrix multiplication has high LL miss rate and L1-dcache miss rate. It is because of high number of stalls. In LL miss for 2048 we have miss rate of roughly 34% and we know accessing memory is costly, further increasing size of N will increase this value

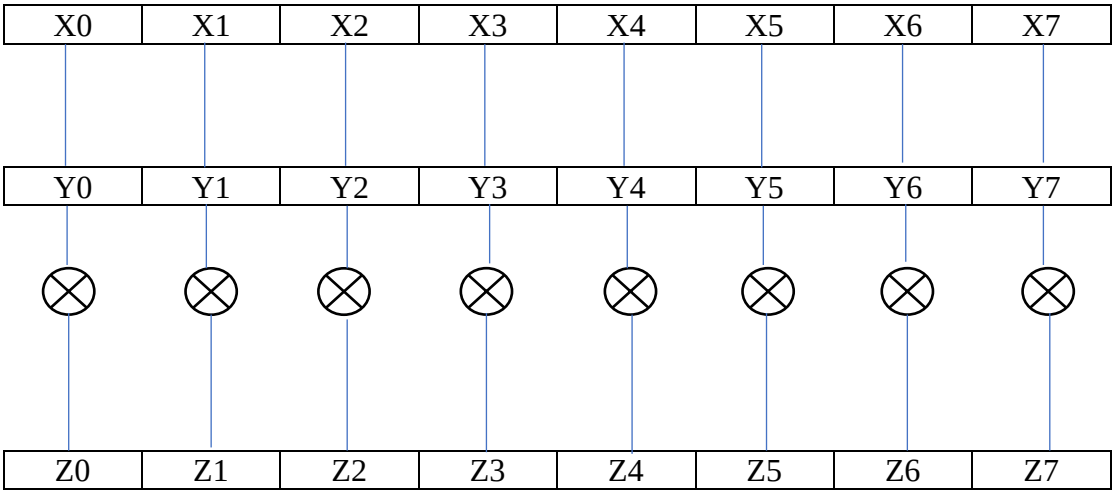
Note that dTLB miss rate is there but has low value. For 512 we have around 0% of miss. This can be probably because N is not large enough to capture the result.

3.2 Optimizing the reduced matrix multiplication

In the reduced method, elements of A were accessed in row major order and whereas elements of matrix B are accessed in column major. Also, the computation for one block to another block of matrix B is independent. Therefore, we can perform multiplicative and additive operation in parallel.

3.2.1 ***Single Thread optimization***

a. Use of Vector instructions (AVX2 of size 256 bits): For performing operations in parallel, since integer are 32 bits, we can perform 8 32-bit values in one operation



b. Ikj loop was used in order to overcome access efficiency

Approach:

First two elements of 1st row of matrix A were loaded in AVX-256 register r1, r2. Similarly next two elements of 2nd row of matrix A were loaded in AVX-256 register r3, r4. Then first two rows of matrix B were loaded on AVX register col1 and col2, following operation were performed

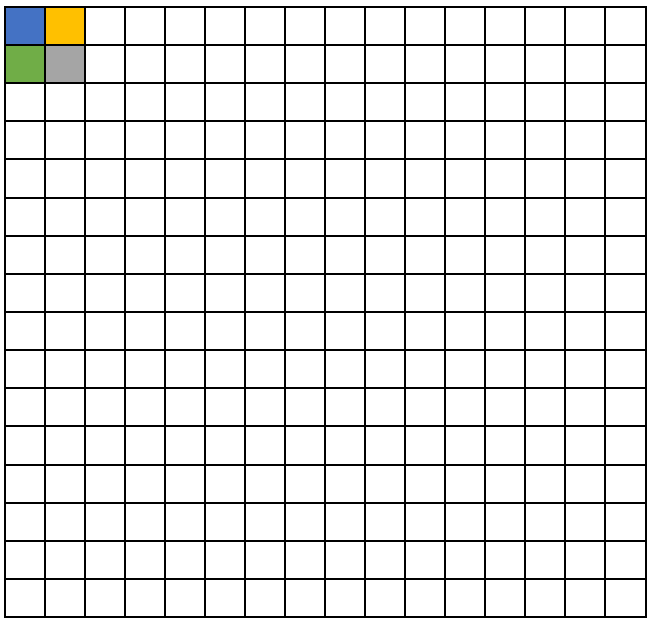
M1 = R1*col1,

M2 = R3*col1,

M3 = R2*col2,

M4 = R4*col2

Addition operation was performed along M1 and M2 giving output as AVX register of 8 values. The adjacent values of resultant array were added giving values of output index



MAT A

In first iteration AVX register

R1 contains all 8 values of shown color blue

R2 contains all 8 values of shown color yellow

R3 contains all 8 values of shown color green

R4 contains all 8 values of shown color grey

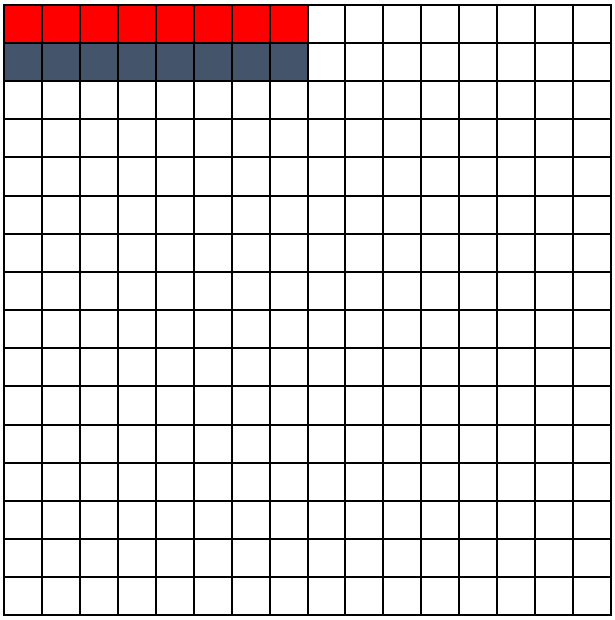
The values will change when column iteration completes

In iteration 1:

(Color shown in red) is loaded in col1 AVX register

(Color shown in dark grey) is loaded in col2 AVX register

Thus, after inner loop column iteration column will be incremented by 8 and both col1 an dcol2 will slide over while r1, r2, r3, r4 values will remain same in that iteration.



(MAT B)

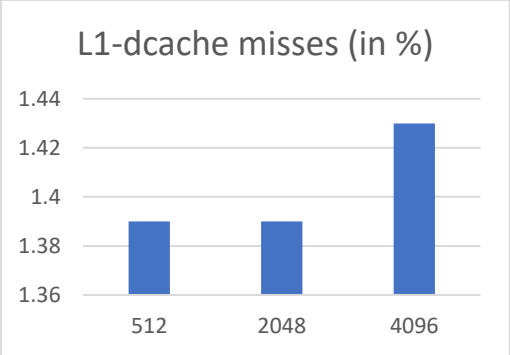
(Example for N=16)

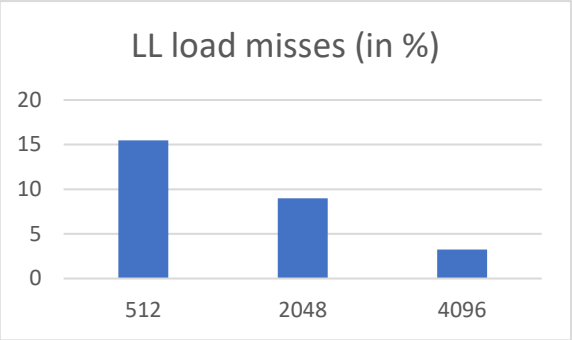
Since we have performed horizontal operations the values for output array would be partial values and would be updated in every iteration

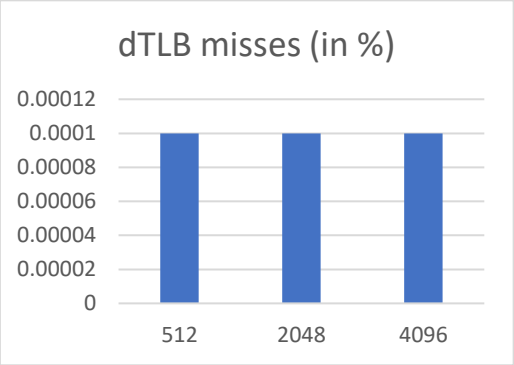
Sum [0]	Sum [1]	Sum [2]	Sum [3]	Sum [4]	Sum [5]	Sum [6]	Sum [7]

Resultant array

Cache Misses







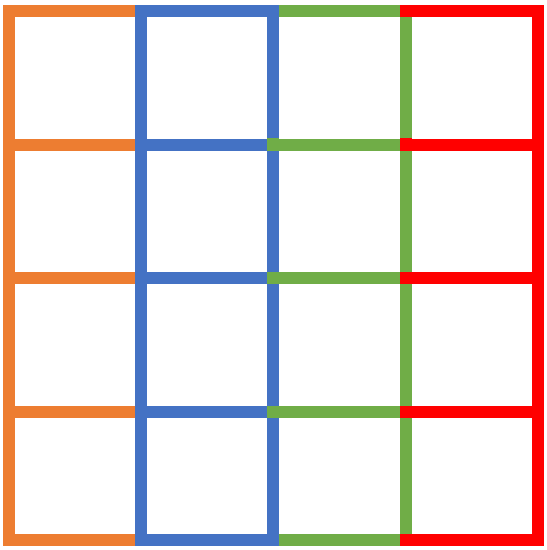
The above results show margin of values between l1-dcache misses and LL load misses. Thereby telling misses have decreased improving the overall performance (The execution time comparison shows the same).

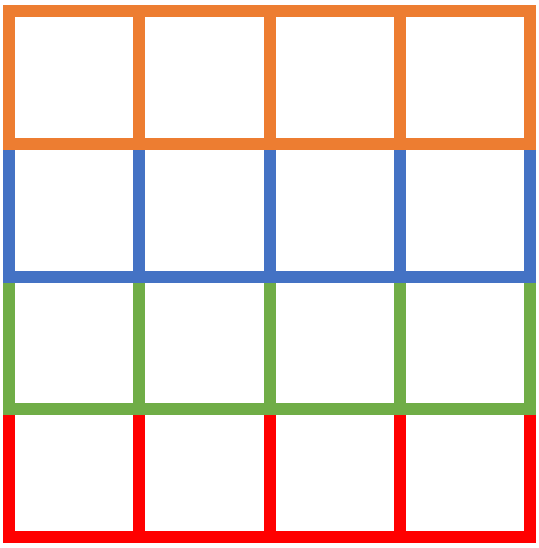
We can see for N = 4096 the LL miss rate from ~77% has reduced to 3%, telling majority of load access would have been found in the L1-dcache itself.

3.3 Multi Thread Optimization

Multithreading is the method to run different parts of program in parallel across different cores, ensuring parallelism. It does through using the help of threads which are nothing but lightweight process with the process. A thread can run on different cores leading to maximum utilization of CPU. The advantages of using multithreading

a. Resource Sharing: There are two ways for process to share resource either Message passing or shared memory. This has to be done explicitly done by programmer. Therefore, the benefit of sharing data allows process to have several threads within same address space
b. Scalability: The benefits of multithreading excel in multiprocessor architecture. Multi-threading on multiple cores increases the parallelism
c. Responsiveness: Let's say multithreading is not implemented on a system which receives a request performs a set of task and returns it sequentially. If long process comes system would be busy doing that and make other requests wait. While in case of multithreaded system may allow run a program even if a part of is blocked or doing length operation, thus increasing responsiveness





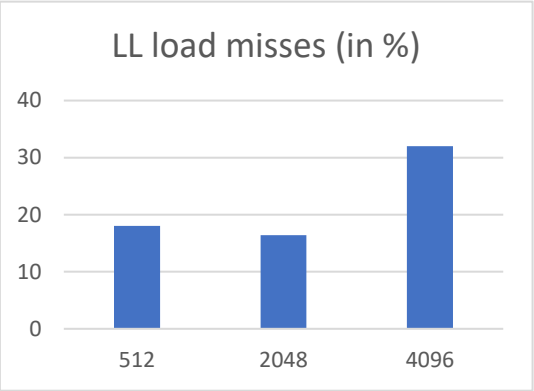
Approach:

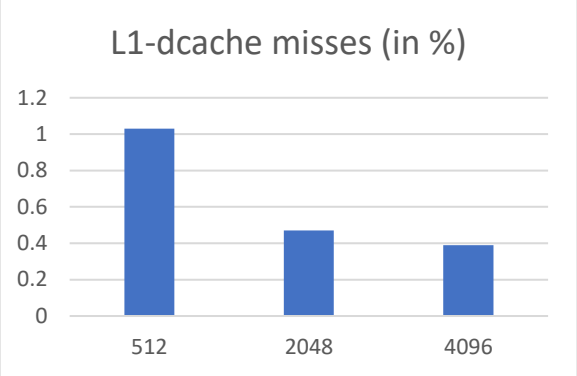
Here our program can take advantage of private caches of each core. Therefore, instead of depending on single L1 and L2 cache as in case of above we have multiple now

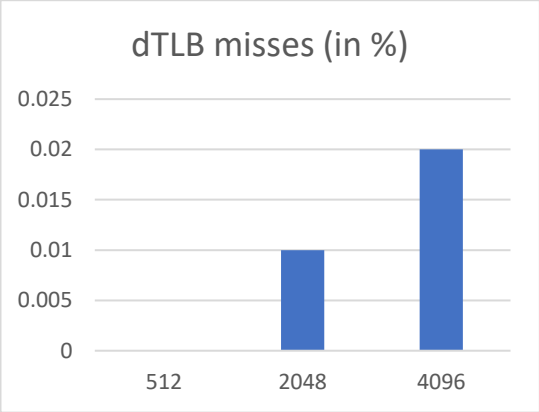
Total 64 threads are created but they are created in iteration of 4, that is 64/4 = 16, 16 threads in an iteration which computes the value of one color shown above and that is executed 4 times. Since data is independent parallelism can be achieved and there will be no need of mutex locks, since no race condition would be present

Above we have created 4 blocks in row and col that means matrix of size N is divided into N/4 rows and N/4 cols. Since the multiplication of blocks within thread (i.e N/4 * N/4) can be further optimized it is performed using tiling along with the vectorized approach which we discussed above in single thread

3.3.1 Results







The above graph shows the decrease in miss rate compared to single thread and in reference. In case of 4096 we have ~31% miss rate and in the case of L1-d cache miss rate has decline along with increase in size (for e.g., 4096 has ~ 0.4% miss rate)

dTLB has low percentage it can be due to fact that size N is not large enough to capture the details

The system on which analysis was done had 12 cores. (For size <=128 multithread was not used, since it did not provide much of performance boost compared to original version)

3.4 Conclusion

Matrix Size	Reduced Matrix (Naive version) (MS)	Single Thread (MS)	Multithread (MS)
16	0.025	0.025	0.007
128	3.134	0.988	3.776
256	24.762	7.586	4.359
512	230.973	59.933	22.515
1024	1956.72	485.665	178.589
2048	44569.6	3797.07	1300.02
4096	481741	29931	9720.1
8192	5e+06	239245	128314
16384	5.76e+7	2.07e+06	682936

Approx Speedup achieved:

a. for single Thread with respect to naive version = 27x (for size 16384)
b. for multi-Thread with respect to naive version = 84x (for size 16384)
c. for multi-Thread with respect to single Thread = 3x (for size 16384)

An ideal implementation would have achieved at least speed up of 4x when cores are increased but our implementation was not able to achieve this speedup possibly because of the kernel process running in background

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

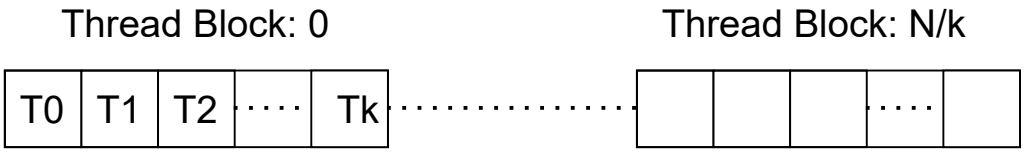


Figure 4.1: 1D thread block grid used for naive kernel of matrix multiplication. Each thread block has 1024 (k = 1024) threads, hence we require N/1024 thread blocks.

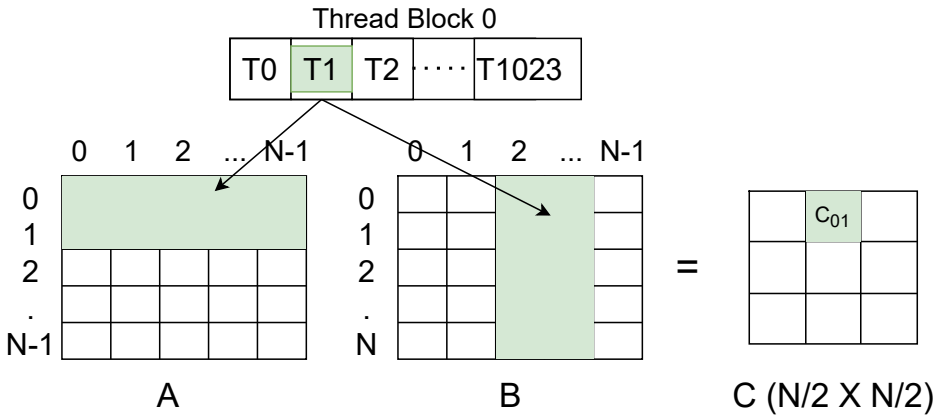


Figure 4.2: Thread work distribution in naive kernel of reduced matrix multiplication. Each thread will read two rows from matrix A and two columns of B and calculate one element of C. Hence we require N/2 X N/2 threads.

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,

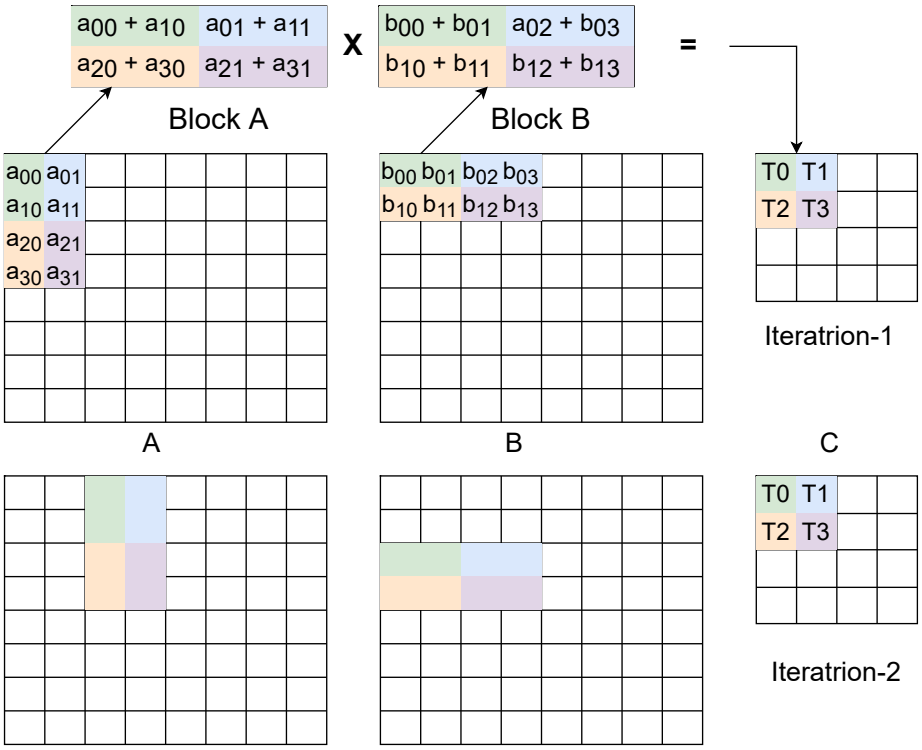


Figure 4.3: Working of tiled version of reduced matrix multiplication.

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

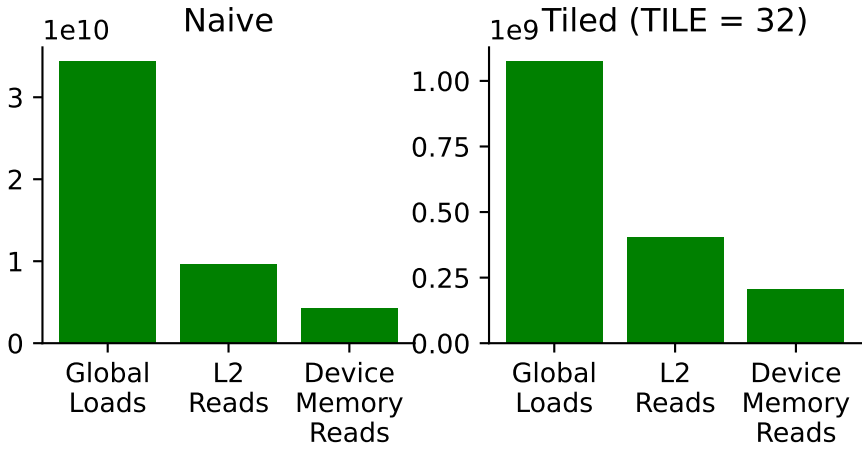


Figure 4.4: Global loads, L2 cache reads, and device memory reads for the matrix size = 4096 with naive and tiled kernel. Loads in naive version is around 30 times greater than loads by tiled version.

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.

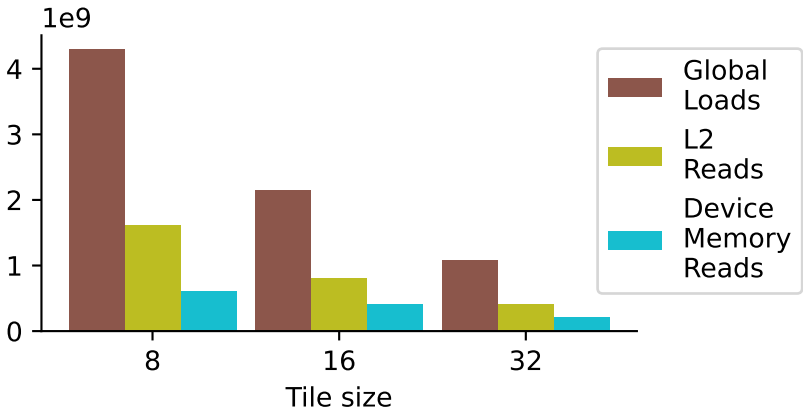


Figure 4.5: Global loads decreases as tile size increase.

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.