1. Abstract:

Different matrix multiplication algorithms have been designed so far have different times it takes to execute. Though the time complexity analysis of those algorithms may turn out to be same, but they in practice take far longer then one another. Because of this reason for same algorithm also different hardware and coding optimizations are required. We have performed many code and strategy optimisations that can be found here.

Link to GitHub repository:

https://github.com/jaydeep-shingala/CA_Assignment03.git

2. Introduction:

2.1 Reducing matrix multiplication:

Reducing matrix multiplication is a technique of multiplying two different matrices of size say N*N and getting output as single matrix of size N/2 * N/2. Here we take two rows, then take two corresponding columns, perform their individual dot product, and then add those 4 dot products tom get final scalar answer.

Let's take 2, input matrix matA and matB of size 4X4 and look at them in this order.

row1
row2
row3
row4

*

col1	Col2	col3	col4

Then after their RMM multiplication answer is:

Row1*col1 + row1*col2 + row2*col1 + row2*col2	Row1*col3 + row1*col4 + row2*col3 + row2*col4
Row3*col1 + row4*col1 + row3*col2 + row4*col2	Row3*col3 + row3*col4 + row4*col3 + row4*col4

We wanted to optimize this given basic reducing matrix multiplication using technique of single threaded and multi-threaded optimizations. We also optimized this code for GPUs.

3. platform specifications:

The entire single threaded and multi-threaded analysis is done on the machine having following specifications:

OS	Linux
RAM	16 GB DDR4
CPU	Intel i5
cores	12 # cpu cores
Cache sizes	48 KB L1-Dcache, 3 MB L2 cache, 12 MB L3 cache
architecture	X86 64 bit

4. Analysis of the Reducing matrix multiplication:

4.1.1 Reference Program:

In the given naïve program what they are doing is simply following simple multiplication of elements by elements and then adding them. In general for n*n matrix sizes as input, we require 4*n multiplications and then 4*n + 3 addition operations for computing single entry in the output matrix.

In total number of multiplications required = 4n * n/2 * n/2

In total number of additions required = (4n+3) * n/2 * n/2

4.1.2 Optimised way to compute entries in output matrix:

One breakthrough might be that if we see this entries in output matrix carefully and expand them we get something like,

For first entry for input size 4*4:

Row1*col1 + row1*col2 + row2*col1 + row2*col2 =

(row1+row2) * (col1+col2)

Means we reduced multiplication and addition operations from, 4 and 3 to 1 and 2 respectively (considering this variables as scalars as of now for simplicity).

So, if we compute output matrix this way then for every single entry in output matrix takes,

In total number of multiplications = n * n/2 *n/2

In total number of additions = (2n+n-1) * n/2 * n/2

This way we reduced both multiplication and addition operations that are required. Even though this might not reduce addition operations too much but it reduced multiplication operations drastically, which are in general costlier to perform for computers.

4.2 Memory accesses:

4.2.1 reference execution:

In the given naïve implementation for 2 input matrix A and B, both are generated and stored as single 1-D array in row major order in the memory. When we want to access elements, matrixA accesses it in row order which benefits temporal and spatial cache locality, but matrixB is accessed in column order fashion. This does not follow cache localities and there are high chances of cache misses in consecutive memory accesses, which increases latencies a lot.

4.2.2 optimal memory access:

If we can somehow access second matrixB also in row major fashion then we can exploit cache localities in best possible way hence number of cache misses will reduce which results into lower execution time.

4.3 Dependency identifying and removal:

4.3.1 reference program:

The given naïve reference implementation is very basic C program which runs through entire input matrices sequentially and computes. There are no data dependencies except for load/stores. So on larger sizes where we have entire vectors we can add or multiply them simultaneously element wise.

For example take a vector of 8 elements,

a1	a2	a3	a4	a5	a6	a7	a8

It's multiplication/addition with another vector say,

b1	b2	b3	b4	b5	b6	b7	b8

Now instead of performing this in sequence one after another, as there is no dependency among them we can perform everything at once also.

4.3.2 vectorization of naïve program:

We have vectorized the code by AVX/AVX2 vector intrinsic instructions to parallelly perform the addition and multiplication of 8 elements in one go. Till we have elements in vector remaining to be performed we can load 8 consecutive elements from memory using,

_mm256_loadu_si256() , this method loads 8 consecutive 32 bit integers from memory into 256 bit vector.

Then add or multiply 8 elements in one go by using.

_mm256_add_epi32(), this method adds 2, 256 bit vectors each having 8, 32 bit integers.

Then can store vectors of 8 elements consecutively at given memory location using,

_mm256_storeu_si256() , and so on.

5. numerical general optimization:

5.1 using more sophisticated operations:

We can also optimise the way we perform required operation. For example when we want to divide n by 2 we can use right shift operations instead of simple division operations which are costlier. Same way we have also used left shift operations for multiplications also. We also observed that in the code there were many redundant calculations that were taking place, we also tried to optimise those more redundant operations.

6. Single threaded implementation:

6.1 Approach:

For optimising single threaded code we have used all given above different optimizations like reducing number of operations required, making memory accesses more effective, using vector intrinsic instructions, numerical optimizations etc.

6.2 optimised algorithm for single threaded implementation:

Step 1: make 2 arrays for storing added rows and columns.

We will store these added rows and columns in row major order so later when we use them, we exploit spatial locality.

Step 2: starting from first row, add all 2 consecutive rows and corresponding elements using vectorization and store it in row added array.

Step 3: starting from first column, add all 2 consecutive columns and correspond elements using vectorization and store it in column added array.

Step 4: now just perform simple matrix multiplication using vectorization and keep on storing in output array.

For different matrix input sizes when we run reference naïve implementation and out optimised implementations, we found execution times for each as given below in seconds.

size	Reference execution time	single threaded execution time
16	0.00006	0.000011
512	0.349	0.068
1024	2.061	0.316
2048	27.834	2.301
4096	998.261	19.674
8192	5699.63	108.298
16384	48989.11	1589.56

*this execution times are in seconds.

For these different sizes after optimization the expected speedup is as given below.We took formula for speed-up as:

Speed-up = Execution time for naïve implementation / execution time for optimised implementation

size	Speed up single thread execution
16	6
512	6.48
1024	6.52
2048	12.09
4096	50.75
8192	52.76

16384	37.74

7. Multi-threaded implementations:

Multithreading is the way of programming in which we try to create many threads which are independent of each other in nature and they can be executed on different cpu cores. In this way we maximise the performance by scheduling different threads on different cores.

Here our program can take advantage of private caches assigned to each core. Now instead of relying on a singleL1 and L2cache we have multiple L1 and L2 caches which can be run in parallel.

7.1 Approach:

We have used concept of single thread described above with extension to multiple threads doing those operations parallelly. We created different threads for different works like row addition, column addition and when both are done at the end multiplication. In total we created,

n/2 threads for row addition

n/2 threads for column addition

n/2 threads for multiplication of those added rows and columns.

7.2 vectorization also in multi-threading:

Those threads in within will also use vector intrinsic instructions to get best performance. Let's say one thread is adding 2 rows then it will add them in chunk of 8 elements in one go, in this way we will even optimise on those single individual threads also.

7.3 optimised algorithm for multi-threaded implementations:

Step 1: create n/2 threads, each will add different consecutive 2 rows and store results itself in array.

Step 2: wait until these threads complete their execution.

Step 3: create n/2 threads, each will add different consecutive 2 columns and store results itself in array.

Step 4. wait until these threads complete their execution.

Step 5: create n/2 threads each thread will take one added row and multiply with all other added columns and store results in output array.

Waiting in step 4 is obvious, because threads in step 5 uses data computed by threads in step 3 and step1. But waiting in step 2 is added even though there was no functional error. Because when we execute without joining in step 2, we observed that there were total of n/2+n/2 = n threads in system. Those many threads were causing higher context switching and thus higher overhead times. But when we waited and restricted number threads by n/2 we found very good around improvement of ~31X faster compared to the one in which we were not waiting.

For different matrix input sizes when we run reference naïve implementation and out optimised implementations, we found execution times for each as given below in seconds.

size	Reference execution time	Multithreaded execution time in seconds
16	0.00006	0.000627
512	0.349	0.099
1024	2.061	0.114
2048	27.834	0.746
4096	998.261	6.45
8192	5699.63	59.25
16384	48989.11	450.56

For these different sizes after optimization the expected speedup is as given below.

We took formula for speed-up as:

Speed-up = Execution time for naïve implementation / execution time for optimised implementation

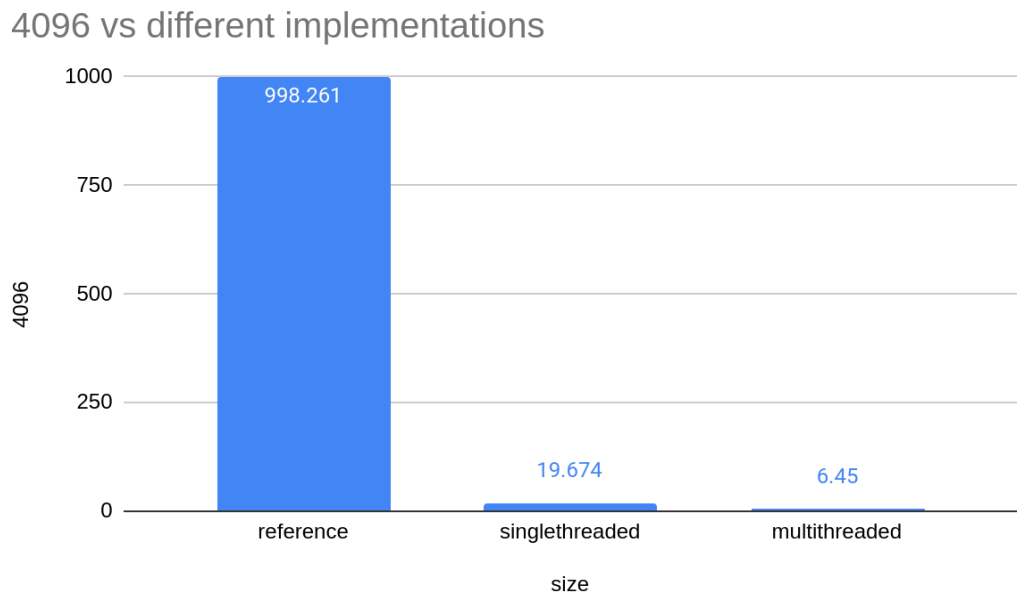
size	Speed-up for multi thread execution
16	1
512	3.52
1024	18.07
2048	37.31
4096	154.76
8192	120.59
16384	108

One important thing that we observed is that when we increased size then number of threads created is increased and then performance slightly decreased because of higher context switching overhead.

For input matrix size: 4096:

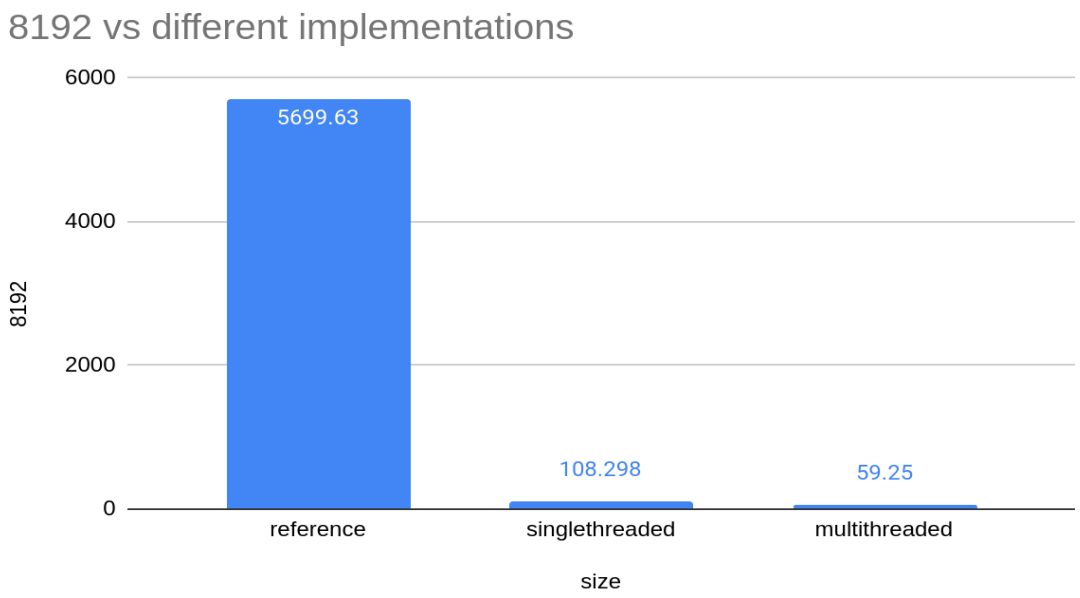
Graph 1:

Below given is the graph for input size 4096 which shows dramatic decrease in execution compared to naïve implementation. (these execution times are given in seconds)



For input matrix size: 8192:

Graph 2:



Part B: Reducing matrix multiplication using GPU:

8. Abstract

CPU are extremely powerful and are better for low latency operation where we want few serial tasks to execute in parallel. But if we have to run code which requires massively parallel processing then the CPU will take longer time to execute as it can only do because threads in CPU are heavy, and the operating system swaps the thread on and off to implement multiprocessing called context switching and it is a slow and expensive process. On the other hand, GPU have exponentially high computational throughput and bandwidth compared to CPU. GPU is better for parallel data processing. They are better for carrying out small operations like floating point arithmetic on substantial amounts of data. GPU is the best way to achieve program parallelism.

9.System Specification

CPU	Intel(R) Core i5-9300H CPU @ 2.40GHz, 8 cores
CPU Memory	8GB
GPU	NVIDIA GeForce GTX 1050 Mobile
CUDA version	11.7
GPU Memory	4GB

10. Approcah

Here we had to use two approaches. The first one is a naïve approach for reducing matrix multiplication and the second is tiled matrix multiplication.

10.1 Naïve Approach

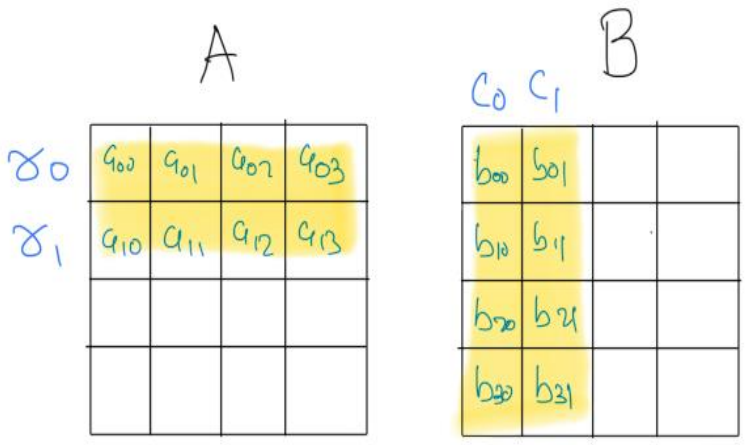
We already know that how 2 matrices A and B of size N x N are multiplied. But in RMM (reducing matrix multiplication) the output size is N/2 x N/2. And each element can be calculated independently. So, we can efficiently parallelize them. For that we make thread for each element of matrix C (which is N/2 x N/2) And each thread will find output corresponding to their element.

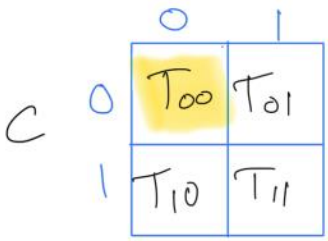
In my approach, threads are arranged in 2-D thread-blocks in a 2-D grid. CUDA provides a simple indexing mechanism to obtain the thread-ID within a thread-block (threadIdx.x, threadIdx.y and threadIdx.z) and block-ID within a grid (blockIdx.x, blockIdx.y and blockIdx.z). In our case rows are indexed in the y-dimension. To find which index of matrices we need to use we make use of thread at block index (i,j) and in that block thread index at (i,j) we used index

i = ((blockDim.y * blockIdx.y) + threadIdx.y)

j = ((blockDim.x * blockIdx.x) + threadIdx.x),

and we left shift (multiply by 2) it because we are using RMM, and that uses two rows and two columns at a time for every element.



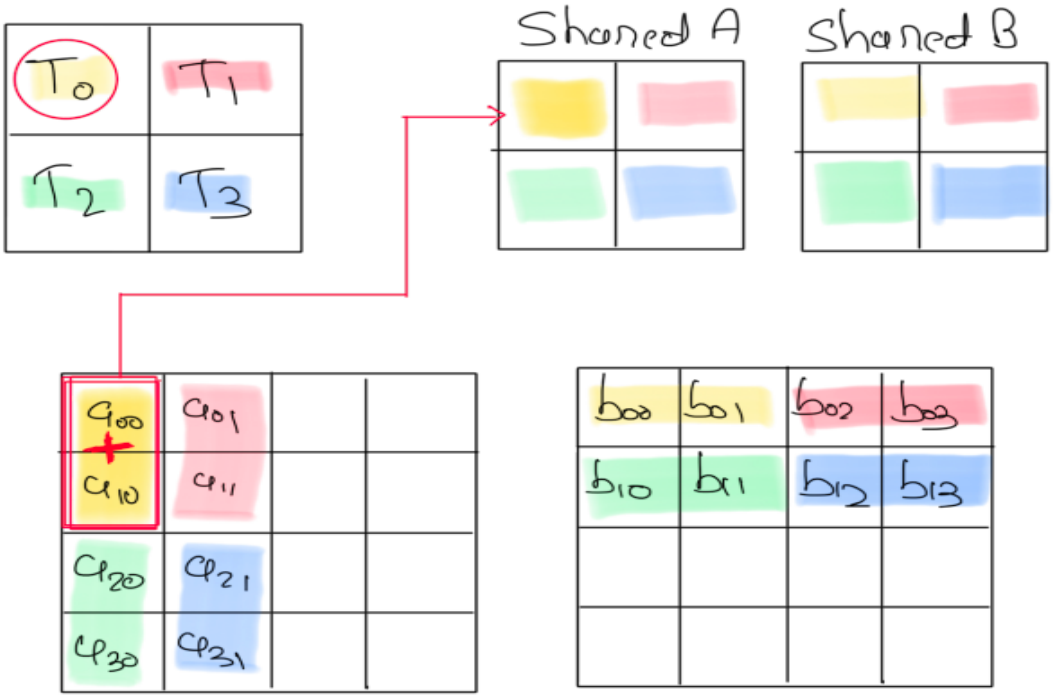


In this approach we found that the number of memory access is high we can reduce it by tiled matrix multiplication. So, we implemented tiled matrix multiplication which is the 2nd approach.

10.2 Tiled Version

In the naïve approach each thread will access 2n elements from each matrix. So, each thread will access 4n elements from global memory. Where in tiled version each thread will access (4n/tile_size) global memory and some number of shared variables which is faster to access as compared to global variables. So, Overall performance will increase.

In this approach we must take care of synchronization. Because here we define one shared variable which is accessed by all thread of thread block.



Here in the diagram, we have shown how memory accesses are taken place and where it will be stored in shared variables.

Here as shown is fig we store value of a00+a10 in sharedA[0][0] and this task will be done by thread T0.

i.e., T0: SharedA[0][0] = a[0][0] +a[1][0]

SharedB[0][0] = b[0][0] + b[0][1]

Likewise, all different threads will do their task (see distinct colors for different threads).

After storing the value in Shared variable, it must have to wait till the threads stores the value. Once all threads store the value, they can proceed further and do multiplication and addition. Again, after each iteration we must synchronize them.

11. Stats

11.1 Without tiling

Matrix size	Time (GPU kernal execution)	Time (CPU)
128	279.26ms	4.463ms
512	305.7ms	350.56ms
1024	279.28ms	3105.3ms
2048	438.33ms	79s
4096	1.5468s	10.43min
8192	10.5085s	78min
16384	88.433s	~14 hours

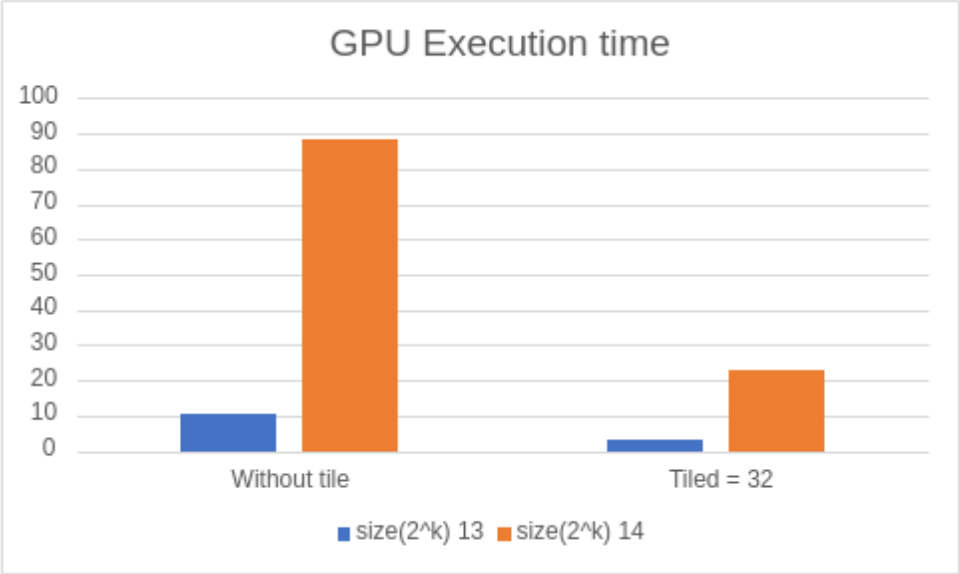
11.2 Tiled

Matrix\ Size \ Tile_ size	8	16	32
128	298.08ms	274.84ms	251.34ms

512	298.54ms	300.23ms	295.63ms
1024	286.24ms	332.83ms	295.95ms
2048	341.58ms	296.43ms	325.47ms
4096	840.33ms	650.46ms	661.24ms
8192	5.4931s	3.30721s	3.246s
16384	48.562s	23.088s	22.883s

Graph 3:

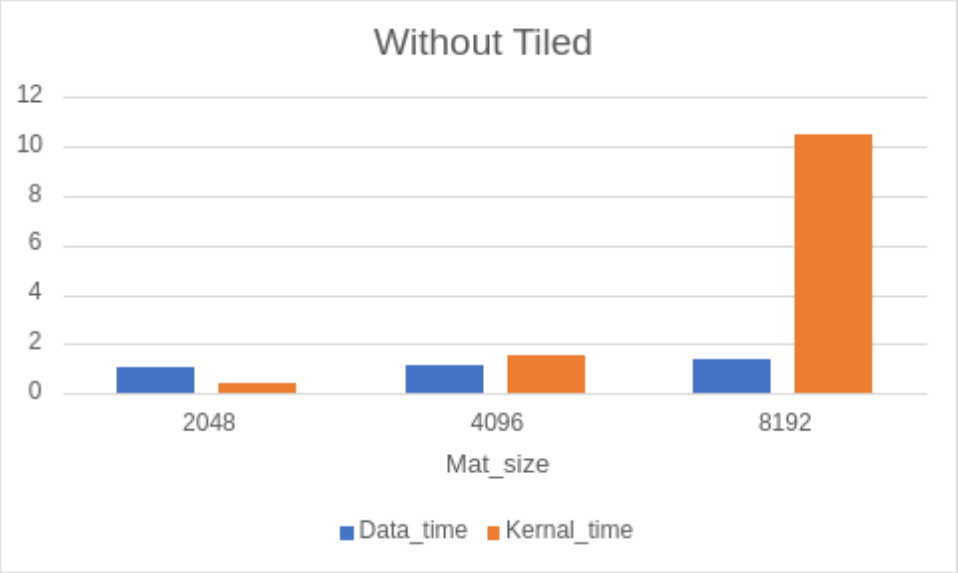
The graph of GPU execution time without tile and with tile size as 32 is given below for the matrix sizes of 2^13 and 2^14.



From the above graph we can clearly see that we get approximately 4X performance with tiling as compared to normal (without tile) matrix multiplication on GPU.

Graph 4:

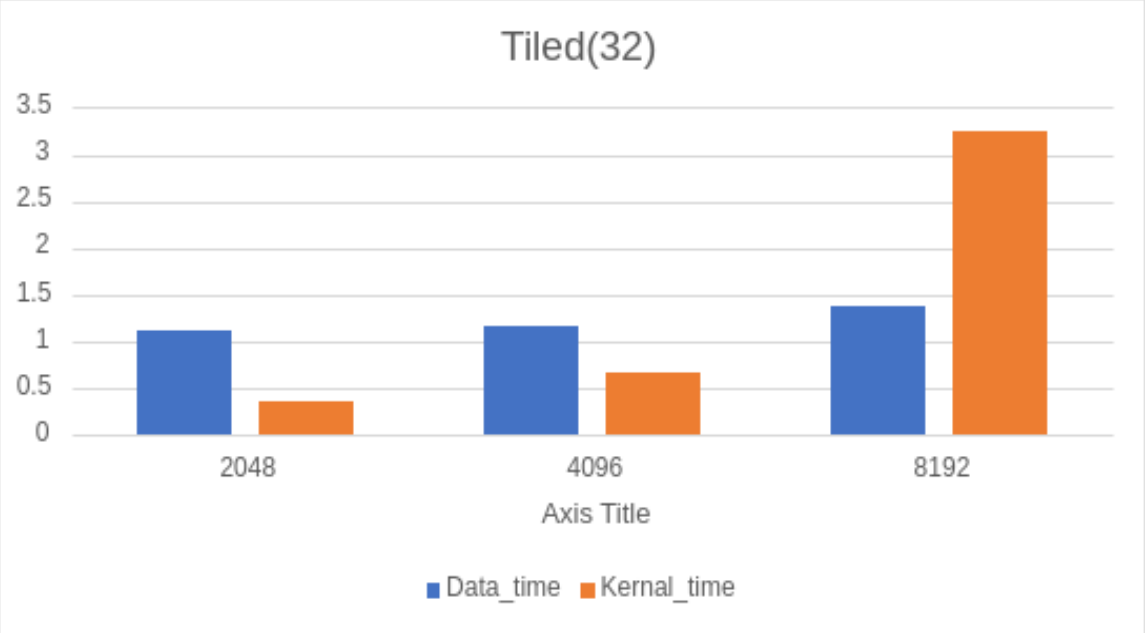
Below given is the graph of Data transfer time and Kernel execution time on GPU without tiling.



From graph we can infer that for a small size of matrix GPU takes more time for data transfer as compared to kernel execution, so data transfer time is not increasing drastically with respect to increase in matrix size.

Graph 5:

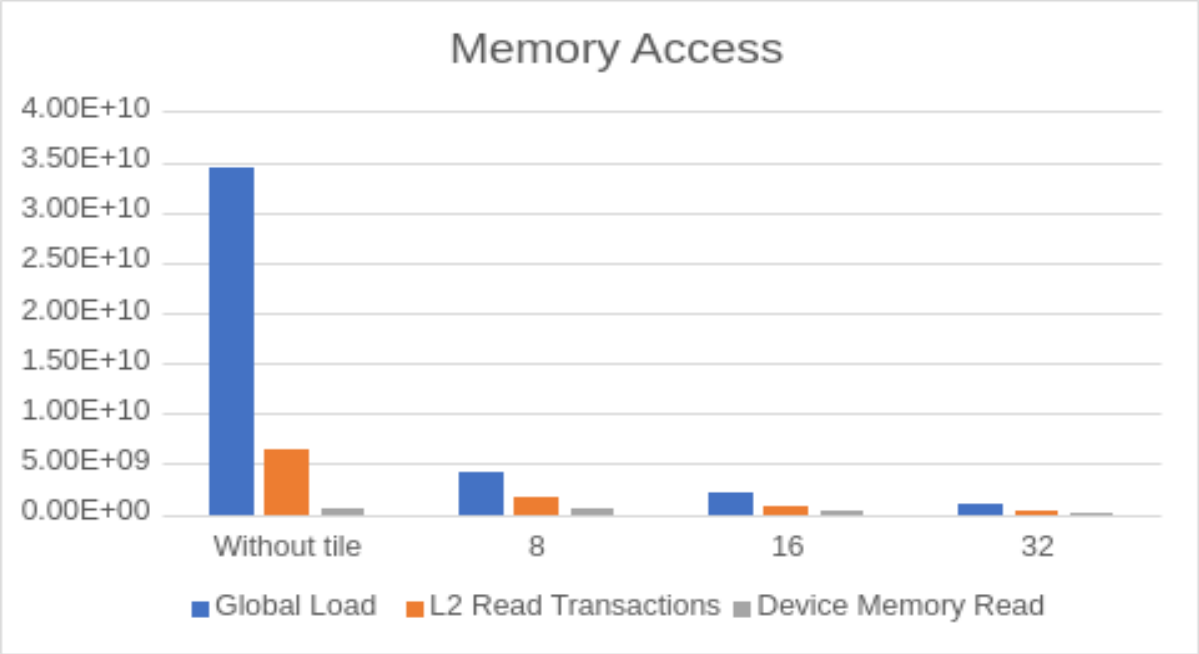
Here is the graph of Data transfer time and kernel execution time on GPU with tiling.



We can see that on GPU, for smaller matrix input sizes data transfer time is almost same as data transfer time for larger matrix input sizes.

Graph 6:

below given is the graph for number of memory access required compared to without tile, tile size as 8, as 16 and as 32.

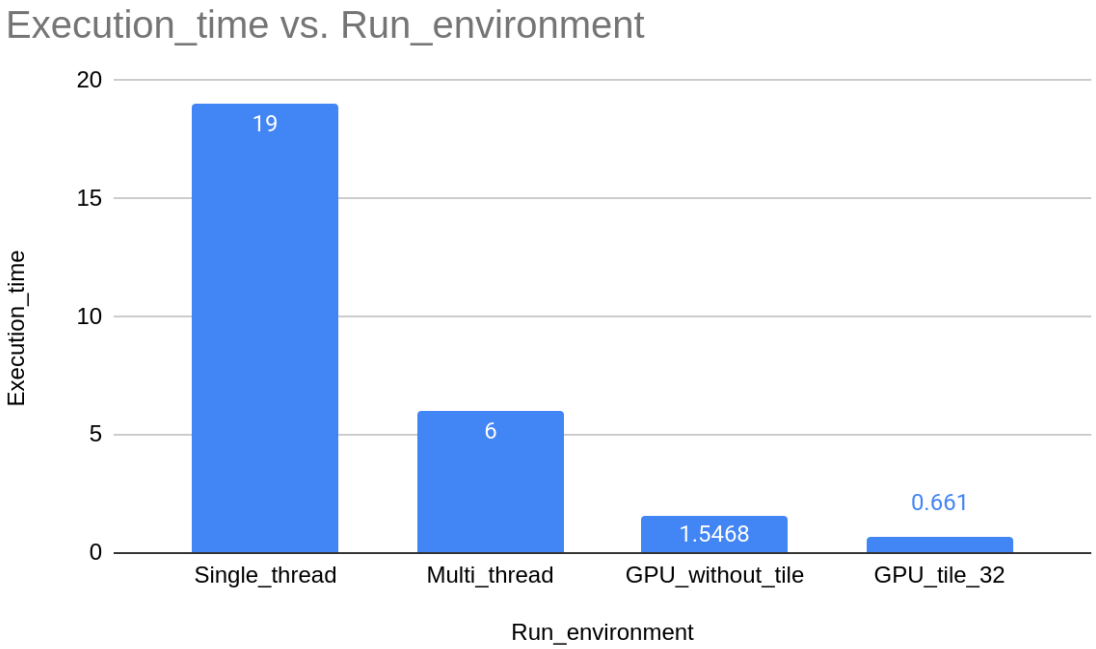


we can see that if we use tiling then number of memory access required decreases because tiling preserves spatial locality and more frequent cache accesses decreases chances of going to higher memory hierarchy and then decrease in overall execution time.

Conclusion:

For kind of applications like reducing matrix multiplication where data access and then high computation is required, we need alternate optimal ways to access data and compute faster. When input sizes are smaller or even bigger up to some threshold CPUs gives very good performance when optimised with vectorizations and multiple threads. But when input size increases drastically then CPUs more or less performs bad. At that time when input sizes are higher GPU gives best performance. Also with GPU when we use tiles it gives even good performance because of exploitation of spatial locality.

In given below graph execution times are given in seconds.


Execution_time vs. Run_environment
20
15
10
5
0
Execution_time
19
6
1.5468
0.661
Single_thread
Multi_thread
GPU_without_tile
GPU_tile_32
Run_environment