HPCA Assignment No. 3

Submitted By

1. Name: - Vithule Prashant Munna

   SR. No.: - 21285

2. Name: - Bhade Milin Narayan

   SR. No.: - 21404

Part A

Q1. Single thread optimization

To optimize the runtime of code, to perform reduced matrix multiplication using SIMD (single instruction, multiple data), and to perform SIMD using AVX intrinsic functions.

Link to the repository: - https://github.com/milinbhade1214/HPCA-Assignment-3

Matrix Size	Unoptimized Single thread	optimized Single thread
16	0.02 ms	0.011ms
1024	8331.74 ms	703.82 ms
2048	97.73 sec	5.47 sec
4096	983.68 sec	43.96 sec
8192	141.45 min	337.66 sec
16384	20 hr	45 min

Table 1. Optimize and unoptimized runtime of different sizes of matrix.

Cache Statistics / Matrix Size	Unoptimized L1 cache miss	Optimize L1 cache miss	Unoptimized LLC cache miss	optimized LLC cache miss	Unoptimized TLB cache miss	optimized TLB cache miss
1024	59.4K	3.29K	44.5K	1.86K	26.68K	330
2048	431K	22.9K	363.3K	12.87K	505.3K	14.43K
4096	1.28 M	*193.3K*	660K	73.3K	1.32 M	123.3K
8192	11.4 M	1.38M	7.3 M	0.37M	11.34 M	0.78M

Table 2. Optimize and unoptimized different statistics for different sizes of matrices.

Performance improvement in Optimized program:

- speed up in running time is x20-x25 from the unoptimized program for calculating reduced size matrix multiplication.

Bottleneck Observed: -

Row 1			
Row 2			
Row 3			
Row 4			
c1	c2	c3	c4

In reduced matrix multiplication we are performing operations like row1*c1, row2*c1, row1*c2, row2*c2 to find 1 entry.

Row 1= a1, a2 till a8 of matA

Col1 = b1, b2 till b8 of MatB (column entries)

a1	a2	a3	a4	a5	a6	a7	a8
b1	b2	b3	b4	b5	b6	b7	b8

We are doing multiplication like a1*b1, a2*b2, and so on. But in our unoptimized code, we do this operation one by one by one. In each iteration, we access four elements and then multiply and so in each iteration access and perform four multiplication operations. There is no dependence on each operation we perform, then we can parallelly do these operations not only 4 elements in iteration.

We are accessing 1 element from row 1 and the other from row this will again cause cache misses. But if we simply use more elements from 1 row and another row then cache misses can be significantly reduced.

Rationale for optimization: -

Inside each loop(iter) we are performing the same operation(multiplication) and each operation is independent. So, we can perform this single instruction on multiple data I.e., SIMD operation. Each iteration takes 4 elements but here in one loop performs operations on 8 elements and performs a total of 32 operations instead of 4 and that too parallelly.

So, as we are doing parallelly this will reduce the time as well we are using more elements from 1 row, and this will affect in reduction of cache misses.

a1	a2	a3	a4	a5	a6	a7	a8
X	X	X	X	X	X	X	X
b1	b2	b3	b4	b5	b6	b7	b8

All these operations are done in parallel.

In SIMD using AVX, we perform parallelly

a1*b1, a2*b2, a3*b3 and so on a8*b8.

Similarly, for row1*c2, row2*c1, and row2*c2. Performing these 8 in parallel and requiring time of 1 operation boosts up performance very much.

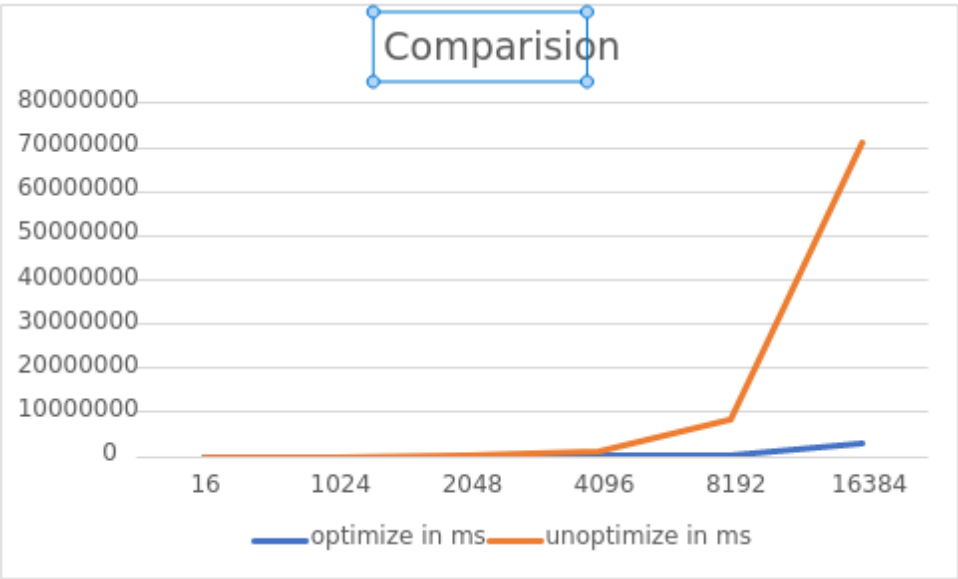
Here we are exploiting row major while putting elements in vector in both matrices (matA, matB) so cache misses are also significantly reduced.

Effect of optimization observed: -

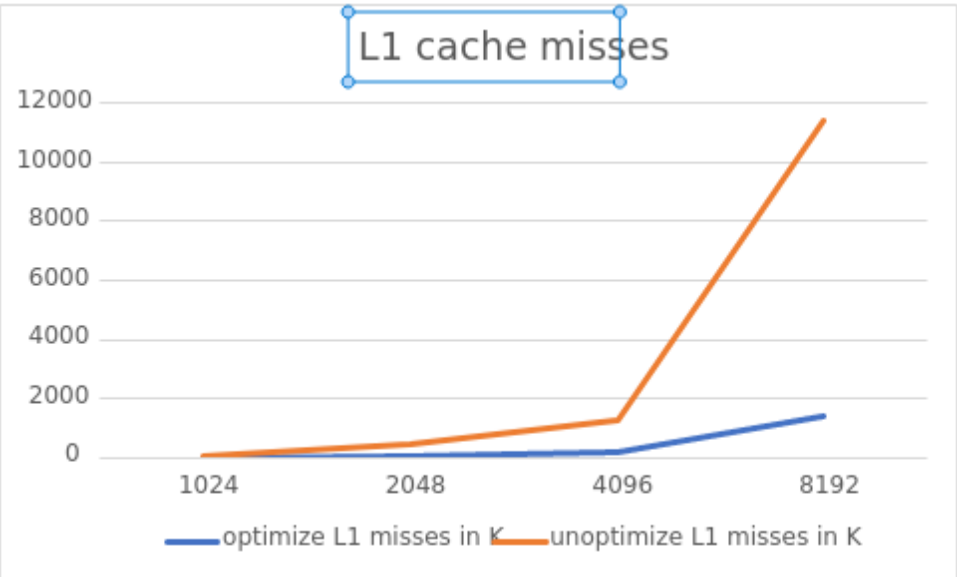
From **Table 1** we can see that for bigger matrices there is approx. 25 times reduction in running time. Graph for comparing the runtime shown below.

The data level parallelism is exploited using avx instruction. And as result, many operations are done in parallel. Similarly for the cache as we access 8 consecutive elements at once that are stored in row-major, and then the following 8 elements. This will help in exploiting row-major order and reduce the cache misses as shown in Table 2.

In optimized code TLB misses are also reduced bcz of more of row major order is exploited and less address translation required.



Time comparison Optimize vs unoptimized



L1 cache misses comparison Optimize vs unoptimized

Q.2 Multithreaded execution

We have performed the same single thread optimize code with multiple thread. Multiple thread executes code parallelly, and performance is improved.

Link to the repository: - https://github.com/milinbhade1214/HPCA-Assignment-3

No. Of threads / Matrix size	1	2	4	8	16
16	0.11 ms	0.152 ms	0.206 ms	0.45 ms	
1024	703.82 ms	367.35ms	309.356 ms	317.437 ms	299.283
2048	5.67 sec	2.85 sec	2.60 sec	2.45 sec	2.46 sec
4096	43.96 sec	23.47 sec	19.96 sec	19.43 sec	19.91 sec
8192	337.66 sec	187.96 sec	161.66 sec	157.52 sec	161.14 sec
16384	45 min	27.33 min	22.85 min	21.33 min	25.15 min

Table 3. Comparing execution time for different thread for different matrix sizes.

How Implemented?

We are dividing the rows depending on the number of threads taken. If I have taken 4 threads, then divided rows of matrix matA.

For thread 1 row 0 to row (N/4)-1, thread 2 row (N/4)-1 to row (N/2)-1, thread 3 row (N/2)-1 to row (3N/4)-1, thread 4 row (3N/4)-1 to row N-1.

Consider 16*16 matrix

R1-4 rows of matA
R2-Next 4 rows of matA
R3-Next 4 rows of matA
R4-Next 4 rows of matA

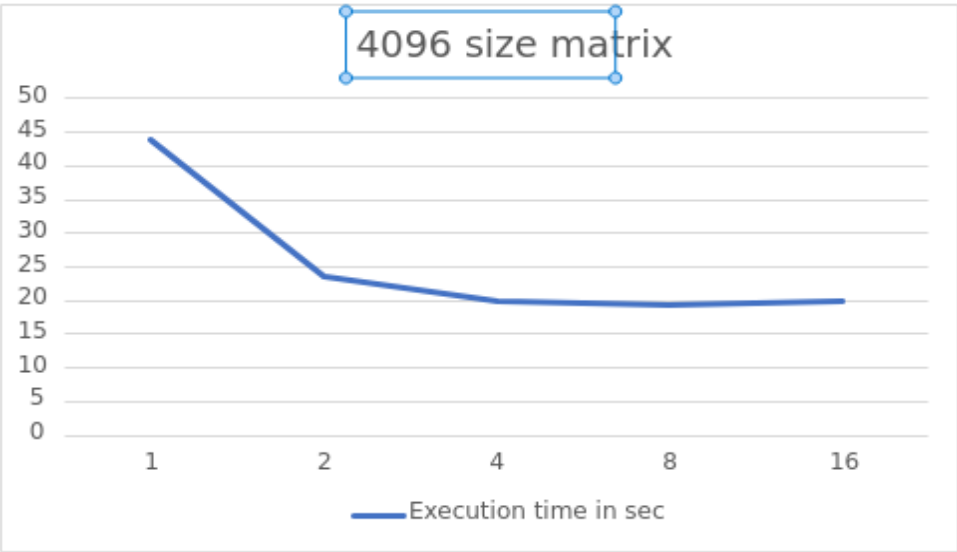
R1- rows access given to thread 1, R2- rows access given to thread 2, R3- rows access given to thread 3, R4- rows access given to thread 4. We have divided the data because there is no dependence on the operation performed and that we are exploiting more.

In this way we have implemented multi-threading.

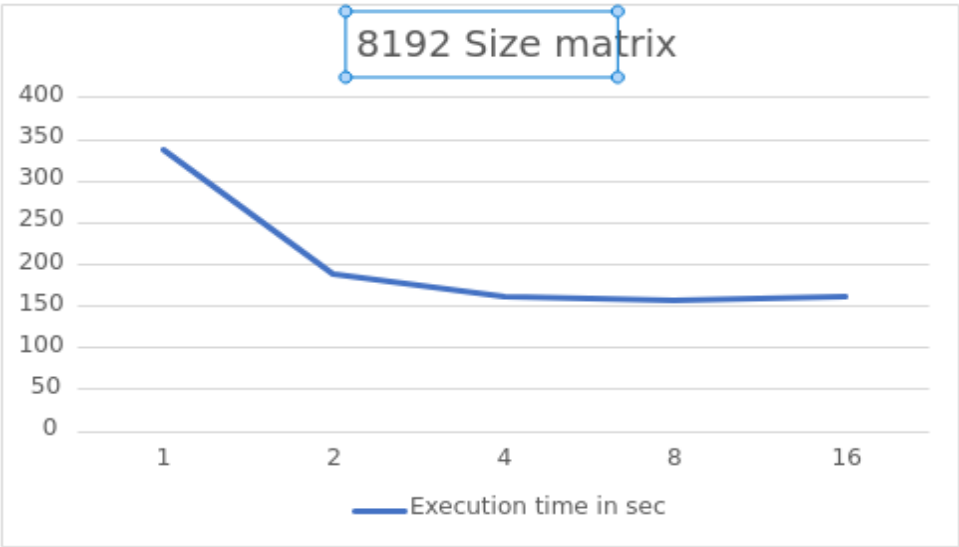
Observation: -

From table 3, we can observe that there is an X2 speed up from a single thread to multiple threads.

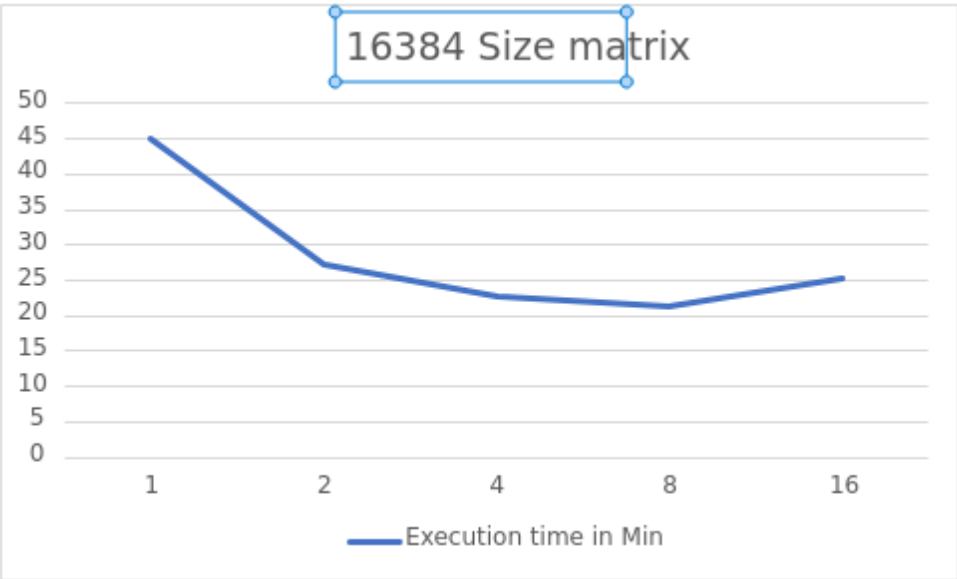
Each thread executes the single thread code parallelly. Each thread has been given the responsibility to calculate some of the entries which can be done independently as all the operations are independent. And this results in performance execution.



4K Matrix Execution time comparison for different number of threads



8K Matrix Execution time comparison for different number of threads



16K Matrix Execution time comparison for different number of threads

Cache statistics

Matrix size 1024

	2	**4**	**8**	**16**
L1 cache misses	4.8K	3.88K	3.88K	3.88K
LLC- cache misses	1.28K	590	1.32K	690
TLB cache misses	3.32K	2.55K	2.55K	1.66K

Matrix size 2048

	2	**4**	**8**	**16**
L1 cache misses	42.14K	33K	35K	35K
LLC- cache misses	15.16K	10.64K	9.94K	12.32K
TLB cache misses	29.7K	24.57K	24.3K	23.4K

Matrix size 4096

	2	**4**	**8**	**16**
L1 cache misses	330K	273K	282K	275K
LLC- cache misses	122.1K	122.3K	134.85K	133.5K
TLB cache misses	241.9K	201.4K	198.38K	188.37K

Matrix size 8192

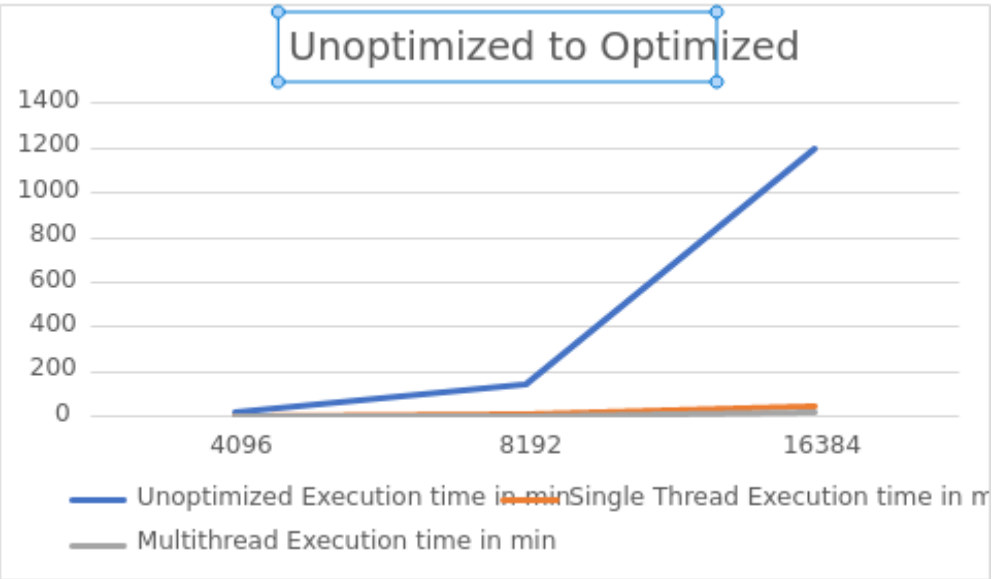
	2	**4**	**8**	**16**
L1 cache misses	1M	2M	2M	1.56M
LLC- cache misses	579K	810K	783K	530K
TLB cache misses	886.5K	920K	910K	1.06M

If we observe that there is no significant difference between a single thread and multiple threads in cache statistics. A Somewhere single thread has fewer cache misses than multithreading, somewhere multithreading. This is because we are only optimizing on the number of threads and all cache optimization is handled by single-thread optimization that's why there is not that much difference.

From my observation, we can see that for No. Of thread=8 gives the relatively low value. Not that much difference but less. After that, we increase the number of threads time increases in small numbers.

Scalability of implementation

Time Matrix size	Unoptimized RMM Implementation	Single Thread Optimization	Multi Thread Optimization
4096	983.68 sec	43.96 sec	19.43 sec
8192	141.45 min	337.66 sec	157.52 sec
16384	20 hr	45 min	21.33 min



Graph for Unoptimized, Optimized Single thread and multi-thread

From unoptimized to an optimized single thread, it is nearly X25 times sped up and from Single thread to multi-thread, it is nearly X2 times sped up. The Total speed up from unoptimized to multithreading(thread=8) is nearly X50 times. This same trend is in all matrix sizes. So, we can conclude that for a bigger size matrix it follows the X50 time speed up.

Part B) GPU- CUDA implementation of RMM

In this part, we have implemented and optimized the performance of Reduced Matrix Multiplication (RMM) with GPU (CUDA) version. CUDA is used as the GPU Library.

Following are the specifications of the system used for the comparison:

Device Specification:

NVIDIA GeForce RTX 3050 Laptop GPU"	
CUDA Driver Version / Runtime Version	12.0 / 11.8
CUDA Capability Major/Minor version number:	8.6
Total amount of global memory:	4096 MB
(016) Multiprocessors, (128) CUDA Cores/MP:	2048 CUDA Cores
GPU Max Clock rate:	1500 MHz (1.50 GHz)
Memory Clock rate:	5871 MHz
Memory Bus Width:	128-bit
L2 Cache Size:	1572864 bytes
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total shared memory per multiprocessor:	102400 bytes
Total number of registers available per block:	65536
Maximum number of threads per multiprocessor:	1536
Maximum number of threads per block:	1024
Max dimension size of a thread block (x, y, z):	(1024, 1024, 64)
Max dimension size of a grid size (x, y, z):	(2147483647, 65535, 65535)

Implementation Details:

In implementing Reduced Matrix Multiplication, we used the 2-D structure of the grids present in the GPU. Each grid consists of several thread blocks, further consisting of multiple threads. The output matrix C is divided into blocks of size (BlockSize x BlockSize). The computation of this small matrix is given to each thread block, which will then be run on each Simultaneous Multiprocessor (SMs). Each thread in thread blocks is responsible for evaluating the value of each entry of output matrix C. We tried to give as little work as possible to each thread to utilize the thread's efficiency in GP-GPU.

Following is the comparison of time required to run this computation of CPU vs. GPU:

Matrix Size (N)	Cpu Reference Time (in ms)	GPU Time (in ms)	Speedup achieved
8	0.002 ms	-	-
16	0.017 ms	0.004096 ms	4.15x
128	2.092 ms	0.037888 ms	55.2x
512	139.675 ms	0.442112 ms	315.9x
1024	1553.17 ms	2.934976 ms	529.19x
2048	35917.9 ms	22.880159 ms	1559.82x
4096	317314 ms	181.871140 ms	1744.71x
8192	3.40489e+06 ms ≈ 56.748 mins	1274.224365 ms	2672.12x
16384	5.385e+07 ms ≈ 14.95833 hrs	10236.926758 ms	5260.36x

Table 1 Comparison of CPU vs. GP-GPU time take for Reduced Matrix Multiplication

The above readings of GPU Time are observed for the block size of 16.

Observation:

1. A large speedup is obtained for matrix multiplication when performed on GP-GPU. As the matrix size increases, the difference in time taken is more apparent. For small matrices, the difference is slight. This happens due to the fact that it takes time to set up Threads, Thread Blocks, and Grid structures as specified for the computation. But for large matrices, the setup time is minimal compared to the time it takes for actual matrix multiplication calculation.
2. For block size 16, we get the best performance than other sizes. Ex. 1,2,4,8,32. This is shown in the later part.

The graph below shows the comparison between the CPU time vs. GPU time required for Reduced Matrix Multiplication (RMM):-

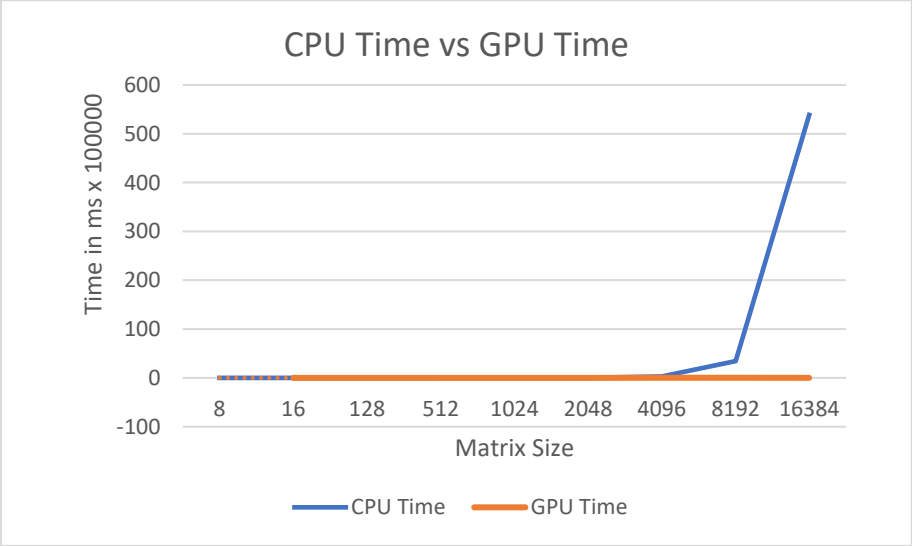


Figure 1. CPU time vs. GPU time for RMM

Bottleneck: Each Thread Block has a maximum of 1024 threads. Thus, the max block size that can be used is 32 as 32*32 = 1024.

Effect of block size on computation:

BlockSize is used to assign the number of threads in thread blocks. A single thread block can have a maximum of 1024 threads, thus restricting the value of blockSize to 32.

1. If small block size is used, then there are a small number of threads per thread block. Here we are not using thread block efficiently. As each thread block is scheduled on a single SM, thus small thread size will not use SM effectively. Also, this increases the no of SMs used for calculation, helping in better parallelism.
2. If large block size is used, then there are many threads per thread block (max up to 1024). This helps in using single SM more effectively and thread block capacity. But this also reduces the parallelism of GP-GPU.

Following is the observed time on GP-GPU for different block sizes on reduced matrix multiplication: -

Matrix size	1024	2048	4096	8192	16384
Block Size					
1	184.796997	1312.652466	10264.62695	82190.67188	666632.75
2	48.698048	365.343994	2680.375488	21056.43945	168585.6406
4	12.190048	102.895004	732.160522	5636.550781	45043.48438
8	3.6184	28.454847	228.131516	1798.215942	14510.19727
16	2.934976	22.880159	181.87114	1274.224365	10236.92676
32	3.39136	26.748447	210.913849	1464.0438	11566.29883

Table 2. running time observed on GPU for different block sizes

Note: Green to Red color shows maximum to minimum values of time taken per matrix size in columns above

The following graphs show the observation observed on different block sizes:

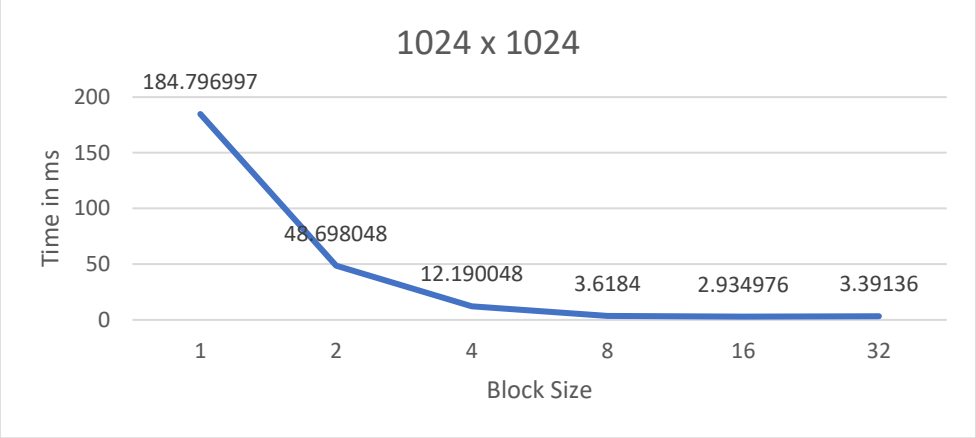


Figure 2. Running time on GP-GPU for 1024x1024 matrix for different block sizes

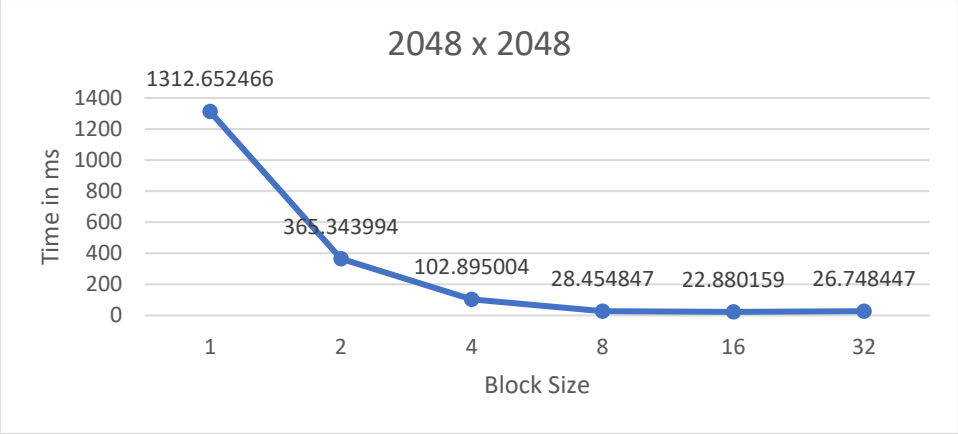


Figure 3. Running time on GP-GPU for 2048 x 2048 matrix for different block sizes

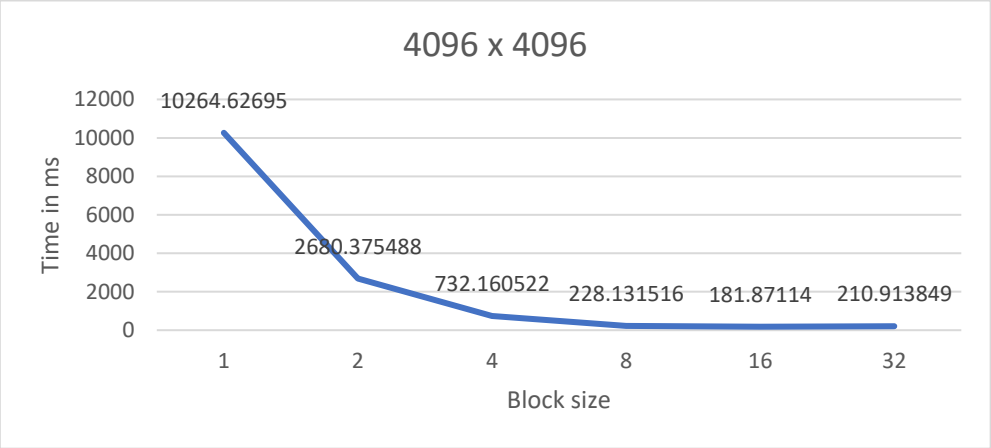


Figure 4. Running time on GP-GPU for 4096 x 4096 matrix for different block sizes

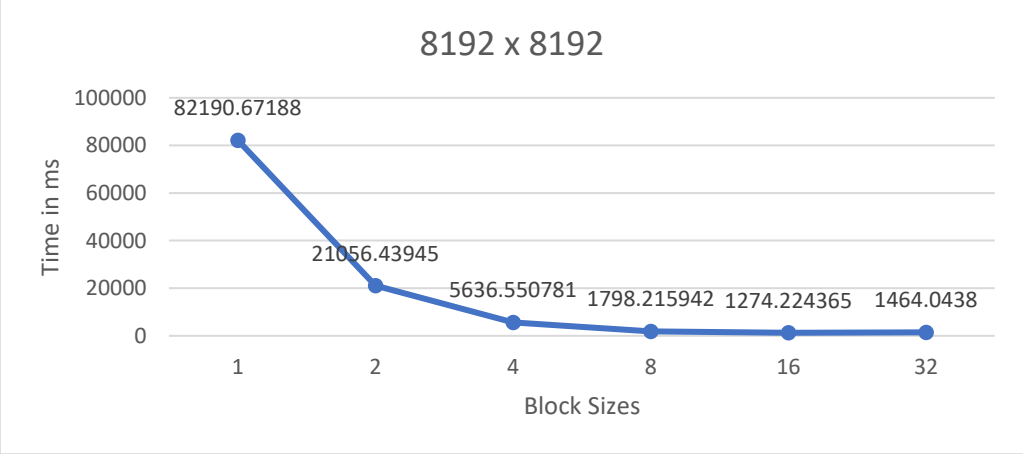


Figure 5. Running time on GP-GPU for 8192 x 8192 matrix for different block sizes

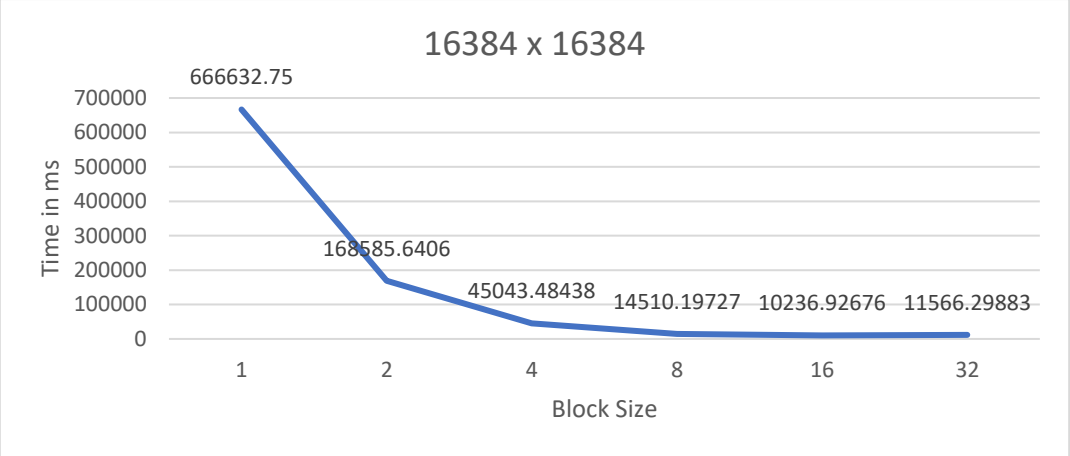


Figure 6. Running time on GP-GPU for 16384 x 16384 matrix for different block sizes

Observation:

This is observed from the data above that BlockSize 16 performs better than all other block sizes (i.e., 1, 2, 4, 8, 16, 32). As we can use a maximum of 1024 threads per thread block, i.e., block size 32, but in this way, we are using a smaller number of SMs in the GPU, thus restricting parallelism possible. Thus, block size 16 optimizes parallelism and thread capacity/ SMs capacity. It divides the work among large no of SMs and as many possible threads in the thread blocks, improving parallelism for multiplication.

Comparison of Single-Threaded Optimization, Multithreading, and GP-GPU CUDA implementation:

Time **Matrix size**	**Unoptimized RMM implementation**	**Single Thread Optimization**	**Multi Thread Optimization**	**GPU CUDA Optimization**
4096	983.68 sec	43.96 sec	19.43 sec	0.1818 sec
8192	141.45 min	337.66 sec	157.52 sec	1.274 sec
16384	20 hrs.	45 min	21.33 min	10.23 sec

Conclusion:

From the above data, it can be concluded that Reduced Matrix Multiplication can be done most efficiently in the CUDA-based GPU implementation. For large matrix sizes where CPU unoptimized implementation takes hours, GP-GPU can do this in mere seconds. Thus, GPU can improve performance for applications where small tasks can be performed in parallel, i.e., where data parallelism can be exploited effectively.