Link to GitHub repository :
https://github.com/mani-312/Reduced_Matrix_Multiplication

This report contains information about the analysis of optimizations performed on single threaded code, multithreaded code and GPU code for calculating the Reduced Matrix Multiplication.

1. Single Threaded Implementation :

Comment of reference Single Threaded code :

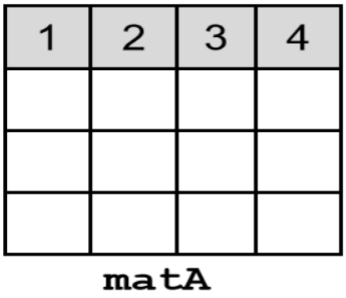
- Reference code accesses the matA row wise and accesses the matB column wise to calculate the Reduced Matrix Multiplication.
- It essentially performs the dot product of a row vector(present in contiguous memory) in matA and column vector(present in non-contiguous memory) in matB.
- It doesn't exploit the cache locality much and since the column vector of matB is non-contiguous it's not possible to vectorize the dot product operation.

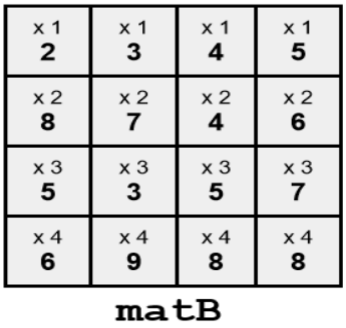
We can change the memory access pattern of accessing matB such that it allows us to exploit cache locality for the fullest and the compiler is able to generate the automatic vectorized code.

This leads to the first optimization of reference code.

1.1 Accessing the matB row wise :

- It essentially performs scalar multiplication of a vector as a base operation.
- Every element matA[rowA][rowB] is multiplied with row vector matB[rowB][:]. And the result is added to the output array correspondingly.
- This specific memory access pattern exploits the cache locality of matB. It also allows the compiler to automatically vectorize the code up to some extent.
- We can also perform loop unrolling of rowB up to some extent, which will eventually decrease the number of accesses to a specific location in array





Speedup achieved on different matrix sizes :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	3.378	0.91
1024	2015	1299.72	1.55
8192	5.23E+06	680092	7.69
16384	5.10E+07	5445500	9.37

- There is a significant speedup achieved by this specific optimization. We can also observe that as we increase the matrix size speedup is also increased.

1.2 Vector Intrinsic code :

- We can write our own vector intrinsic code to exploit more data parallelism.
- An AVX 256 bit vector is used in our code which performs 8 16-bit integer multiplications in parallel.
- Our code implements the scalar multiplication of a row vector of size 8 using AVX 256 bit operation.
- Finally, the result of 8 values is added to the output array at corresponding locations sequecially.

Speedup achieved on different matrix sizes :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	2.512	1.22
1024	2015	856.291	2.35

8192	5.23E+06	455064	11.5
16384	5.10E+07	3663670	13.9

- More speedup compared to prior optimization.

1.3 Further optimization to vector intrinsic code :

➔ We can still optimize the vector intrinsic code by vectorizing the addition of the result of scalar multiplication of a row vector to the output array.
➔ This can be done by performing two scalar multiplications at a time, it results in 16 values.
➔ These 16 values can be added to the output array parallely by writing it as a vector intrinsic operation.

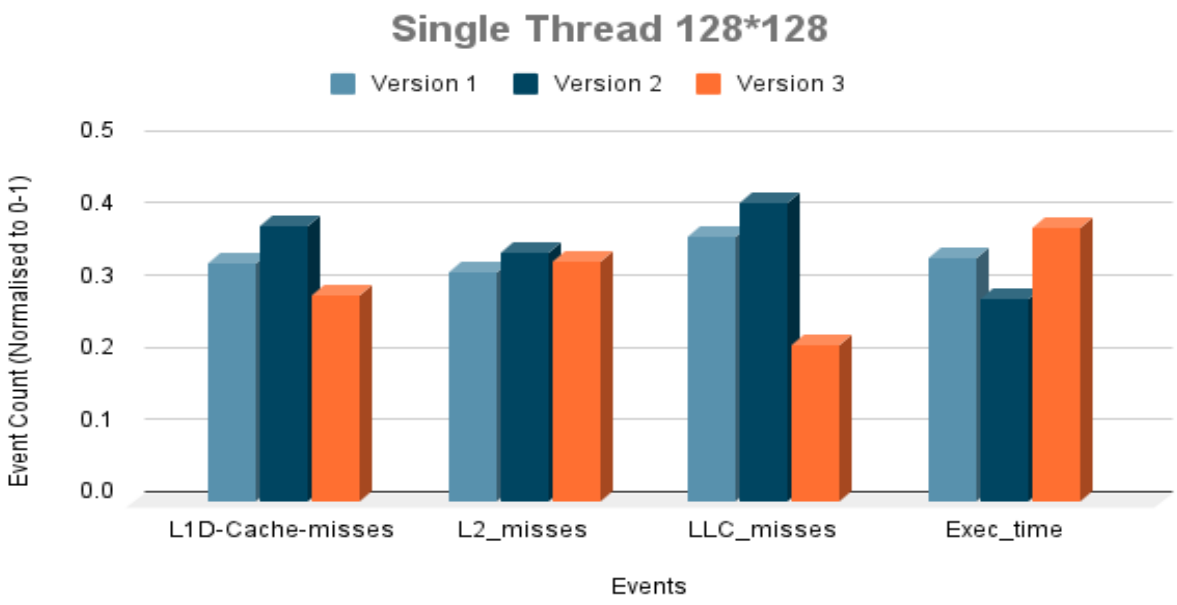
Speedup achieved on different matrix sizes :

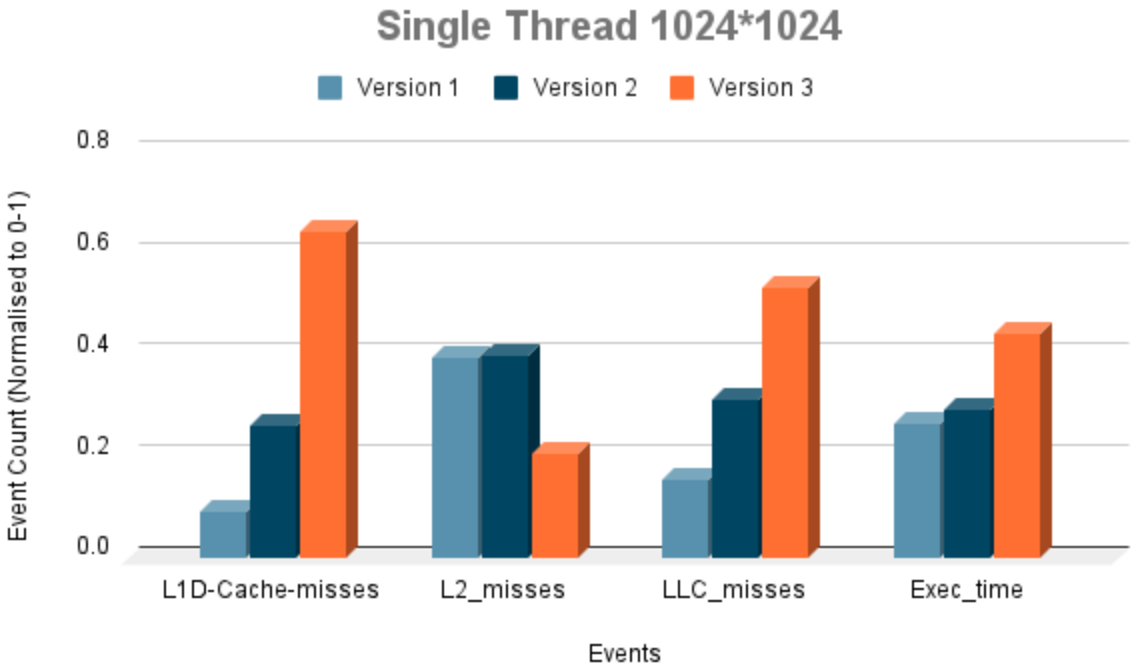
Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	3.003	1.02
1024	2015	773.392	2.60
8192	5.23E+06	404185	12.9
16384	5.10E+07	3222610	15.8

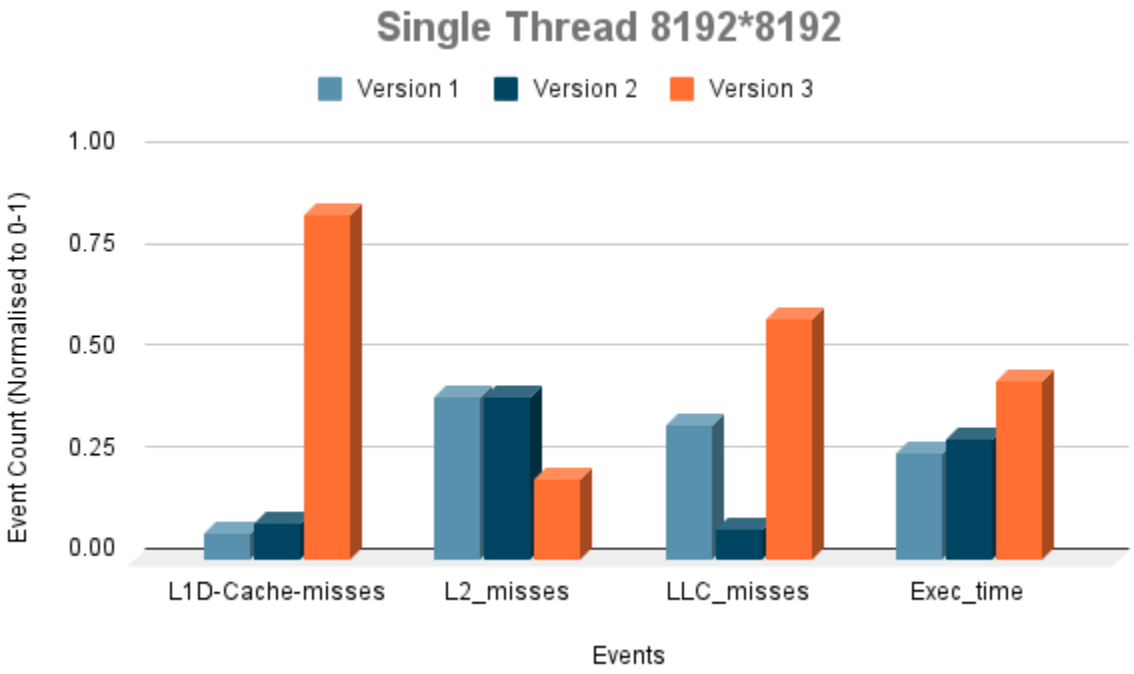
- Involving more data parallelism increased the speedup even more than the traditional vector intrinsic code that we've written.

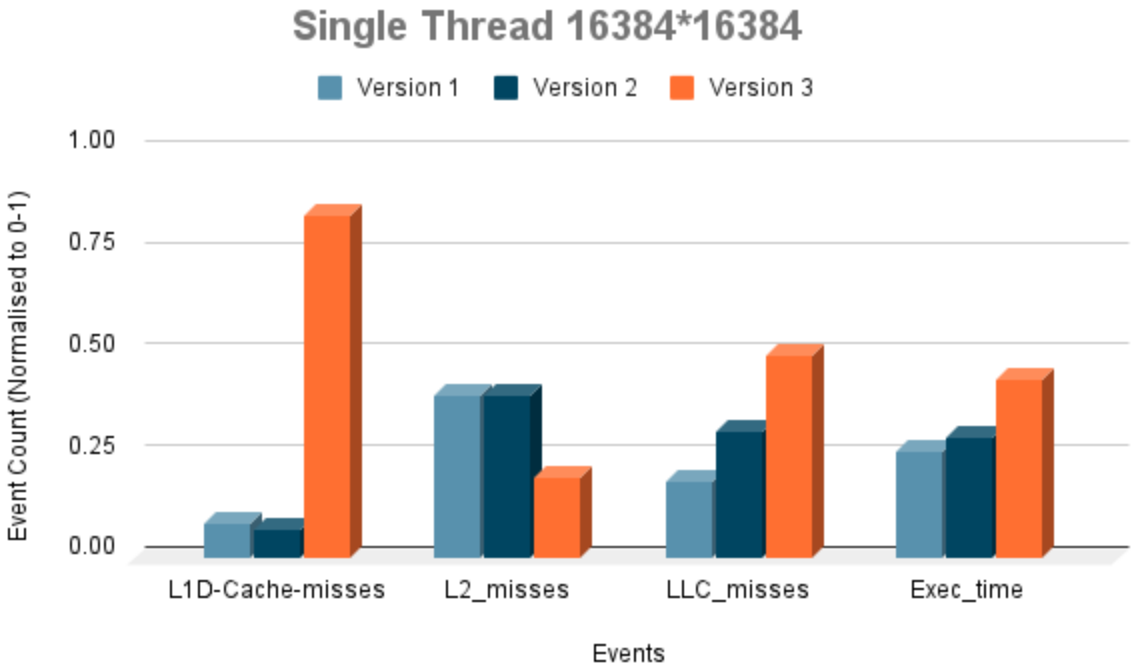
In all the below plots:

- **Version 1** refers to implementation of vector intrinsic code.
- **Version 2** refers to implementation of optimization done on vector intrinsic code.
- **Version 3** refers to implementation of row wise access of matB
- All the event counts are normalized so that they can be compared on same scale




Single Thread 1024*1024
Version 1
Version 2
Version 3
0.8
0.6
0.4
0.2
0.0
Event Count (Normalised to 0-1)
L1D-Cache-misses
L2_misses
LLC_misses
Exec_time
Events


Single Thread 8192*8192
Version 1
Version 2
Version 3
1.00
0.75
0.50
0.25
0.00
Event Count (Normalised to 0-1)
L1D-Cache-misses
L2_misses
LLC_misses
Exec_time
Events



- For a vector operation of loading the data it costs only one load instruction. Load instruction will fetch the consecutive 8 elements in the memory starting from the given memory location.

- It can be inferred that huge decrease in the cache misses led to the decrease in execution time for the vector intrinsic code.

2. Multithreaded implementation :

- We have created 16 threads in our implementation of Reduced Matrix Multiplication.
- Say the output matrix size is n * n, then each thread calculates the n/16 rows of the output array. Altogether 16 threads calculate n rows of output array.
- Same ideas as mentioned in the single threaded code are implemented in multithreaded code as individual versions.

Increasing the number of threads is not giving efficient results. Probably this is due to my system configuration supports 8*2 threads parallelly to be executed.

2.1 Row wise access of matB :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	1.519	2.02
1024	2015	228.261	8.82
4096	443818	17974.3	24.69
8192	5.23E+06	138811	37.7
16384	5.10E+07	1.11E+06	46.0

2.2 Vector intrinsic code :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	2.723	1.12
1024	2015	143.469	14.04
4096	443818	9879.56	44.92
8192	5.23E+06	102579	51.0
16384	5.10E+07	847497	60.2

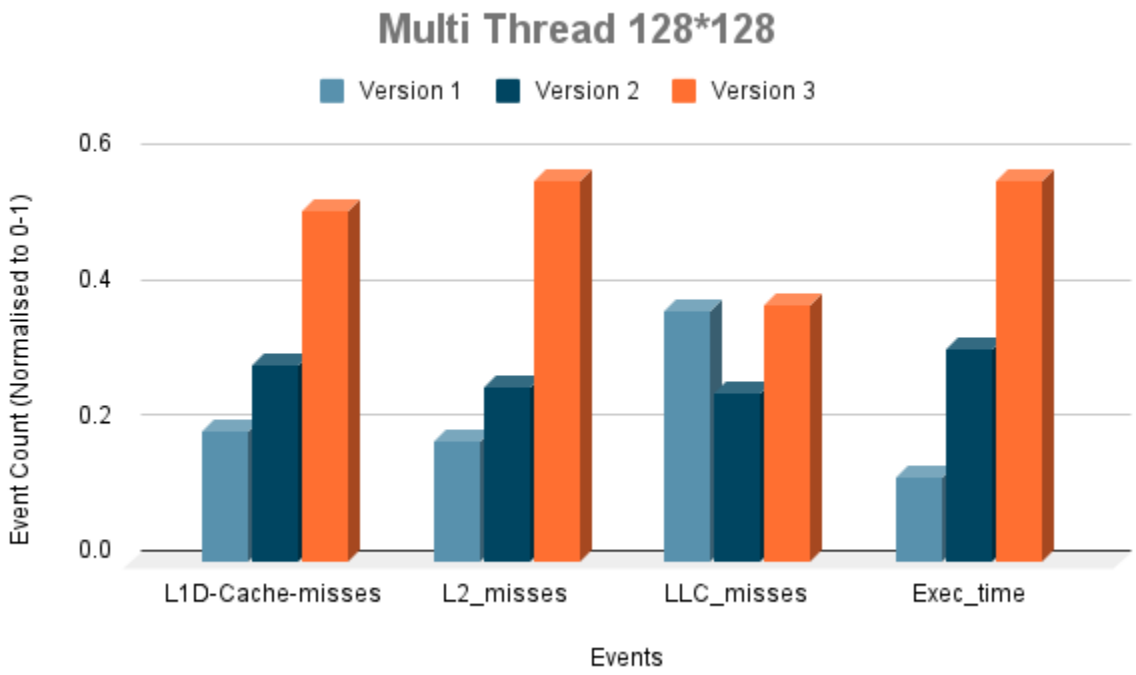
2.3 Further optimization on vector intrinsic code :

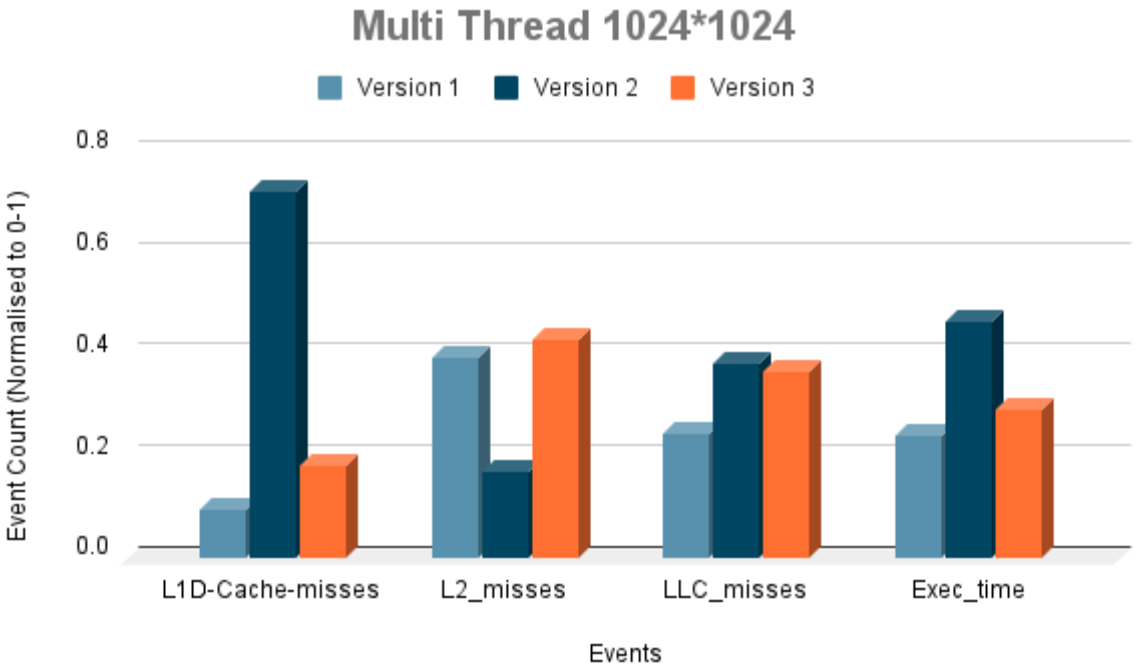
Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
128	3.074	0.612	5.02
1024	2015	117.473	17.15
4096	443818	8508.01	52.16
8192	5.23E+06	94080.8	55.6
16384	5.10E+07	830669	61.4

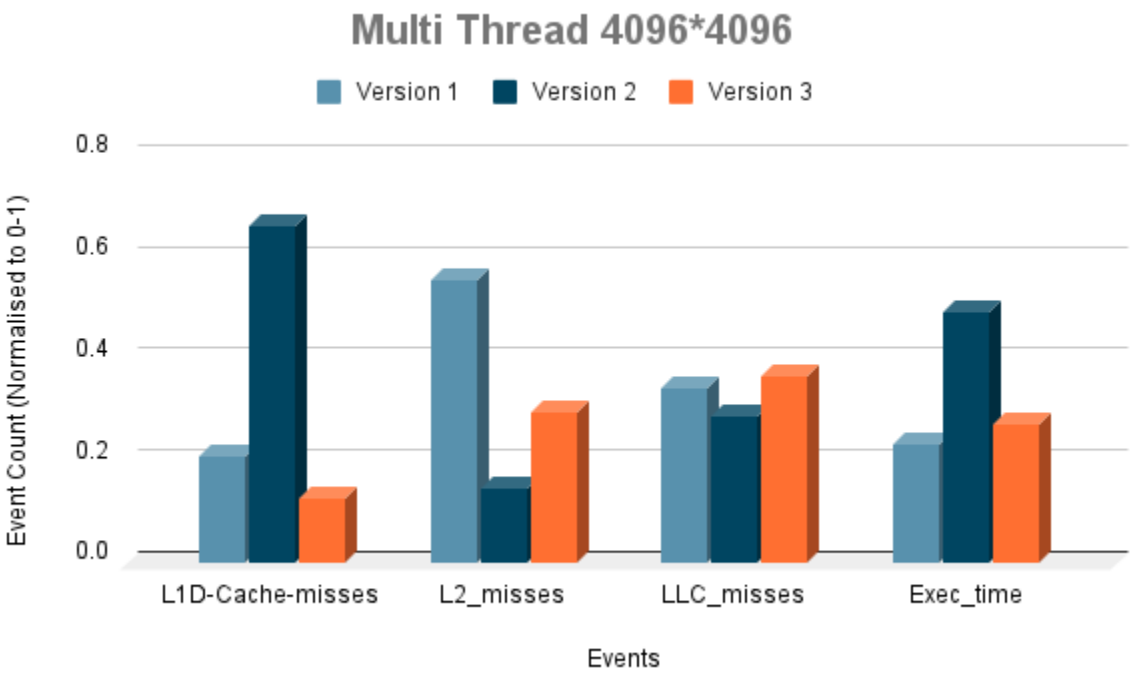
- It turns out that multithreaded implementation of RMM for 16384 matrix size is 61 times faster than the reference execution. This is due to the fact that the threads run parallely on different cores.
- Multithreaded vector intrinsic implementation is approx 4 times faster than the single threaded vector intrinsic implementation.

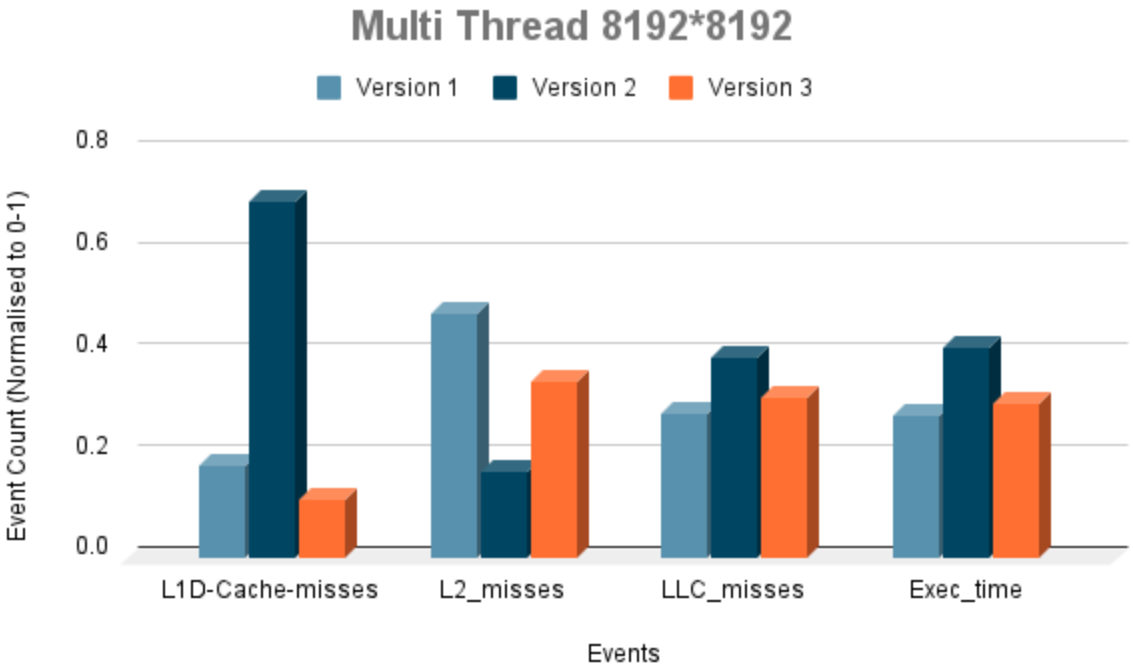
In all the below plots:

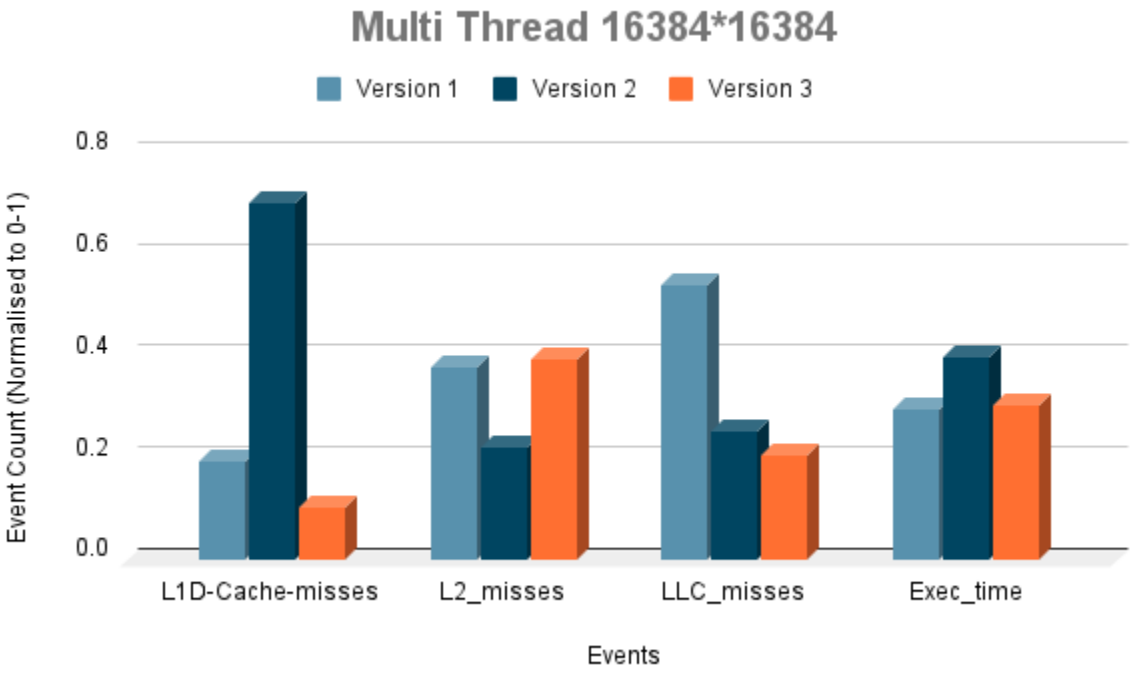
- **Version_1** refers to implementation of optimization done on vector intrinsic code.
- **Version_2** refers to implementation of row wise access of matB
- **Version_3** refers to implementation of vector intrinsic code.




Multi Thread 1024*1024
Version 1
Version 2
Version 3
0.8
0.6
0.4
0.2
0.0
Event Count (Normalised to 0-1)
L1D-Cache-misses
L2_misses
LLC_misses
Exec_time
Events


Multi Thread 4096*4096
Version 1
Version 2
Version 3
0.8
0.6
0.4
0.2
0.0
Event Count (Normalised to 0-1)
L1D-Cache-misses
L2_misses
LLC_misses
Exec_time
Events





- Multithreading implementation on smaller matrix size doesn't give much performance, because more overhead will get associated with creating, running and joining the threads.

- We can see the effect of multithreading on larger matrix sizes, a lot of improvement.

- Same as in the single thread implementation, it also outperforms in the vector intrinsic implementation.

3. GPU implementation :

The configuration of GPU system we've run the code is as follows :

Server Specification

Model	2 X AMD EPYC 7713 64-Core Processor (Total 128 cores, 256 hardware threads)
GPUs	8 X NVIDIA A100-SXM4 40GB
System Memory	1 TB DDR4 DRAM
Storage	1.8 TB

- Number of SM = 108
- Number of cores = 6912
- This specific GPU allows Max threads in a threadblock = 1024.

- To match the Reduced matrix multiplication execution, we've created 2D Thread Blocks and 2D threads in each thread block.
- Say n*n is the output array size.
- Then we create the number of threads = n*n, this allows each thread to compute a specific index value of the output array.

3.1 Number of Threads in TB = (32*32) :

- We first create a version as (32*32) threads present in each thread block.

- This specific implementation exploits the data parallelism by means of executing the same instruction by all threads in a warp(32 threads) paralely in different SIMD cores.

- Since each threadblock has several warps, it also exploits the multithreading by context switching between warps within the resident thread blocks in SM. It ensures to keep the resources busy as much as possible.

Speedup achieved on different matrix sizes :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
1024	2015	396	5.08
8192	5.23E+06	707	7355
16384	5.10E+07	2475	20606

This specific implementation is 20606 times faster than reference execution for matrices of size 16384.

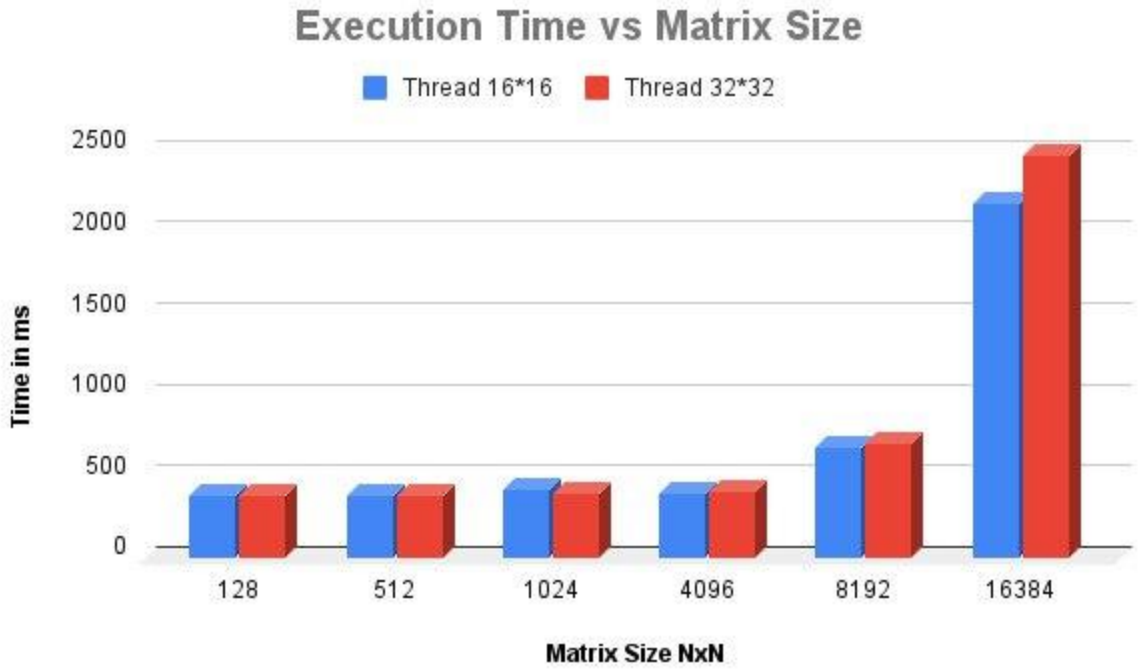
3.2 Number of Threads in TB = (16*16) :

- Another version as (16*16) threads present in each thread block.
-
- By this configuration, comparatively less number of thread blocks are resident in an SM, which allows for faster scheduling among the warps resident in the SM.

Speedup achieved on different matrix sizes :

Matrix Size	Reference Time(ms)	Optimal Time(Ms)	Speed Up
1024	2015	425	4.74
8192	5.23E+06	677	7680
16384	5.10E+07	2182	23373

This specific implementation is 23373 times faster than reference execution for matrices of size 16384.



- We can see that the configuration of 16*16 threads in a threadblock is outperforming the configuration of 32*32 threads in a thread block.