High Performance Computer Architecture (E0-243)
Assignment 3
M.Tech-CSA (IISc Bangalore)

Submitted By :

Dhaval Chavda (20973)

Mihir Khatri (20992)

Reduced Matrix Multiplication

1. **Github repository link**

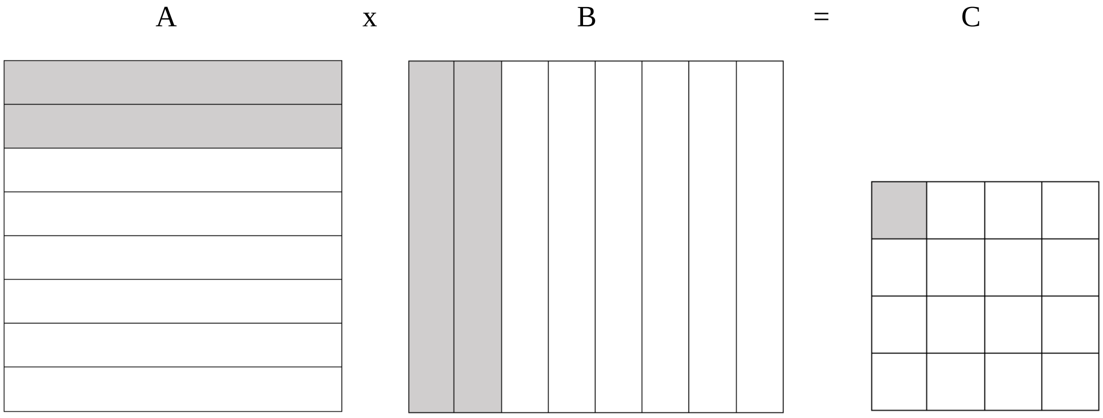
   https://github.com/Dc3337/CA_Assignment_3_Dhaval_Mihir

2. **Introduction**

   Here, we are given an implementation of reduced matrix multiplication which, as the name suggests, gives less number of cells in the final matrix.

   It takes 2 ($N \times N$) matrices as input, say A and B, where N is a matrix of size 2^k, k being a natural number and $k \geq 2$. With an output matrix C of size $\left(\frac{N}{2} \times \frac{N}{2}\right)$.

   As contrary to a normal matrix multiplication, here, a row-set is multiplied to a column-set. Where, a set is defined as the combination of two consecutive row and column respectively. Thus, we get N/2 row-set and N/2 column-set, when, each row in the row-set is multiplied with each column in the column set and their sum is added to give the output for a single cell in the output matrix.



   From the given image one can see how the reduced matrix multiplication is performed. Our goal here is to optimize the RMM.

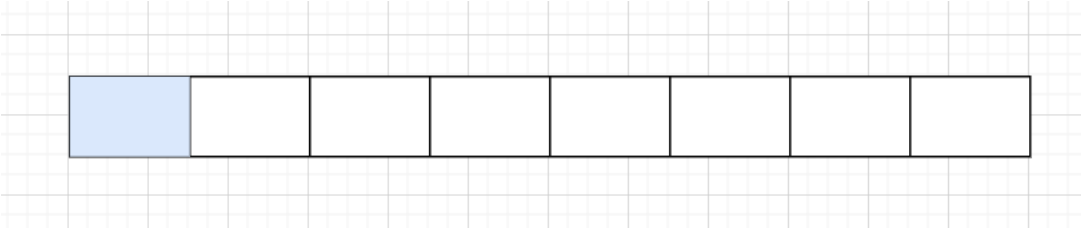
3. **Processor Specification**
   We used a system with the given specifications to execute our code and perform optimization.

CPU	11th Gen Intel(R) Core (TM) i5-11500 @ 2.70GHz
GPU	NVIDIA GeForce GTX 1050 Mobile
GPU Memory	4 GB
Memory	8 GB DDR4
CUDA	Version 11.7
Cache	L1d cache = 288 KB, L1i cache = 192 KB, L2 cache = 3 MB, L3 cache = 12 MB
OS	Linux 5.11.0-38

Core	12
Threads	2 per core

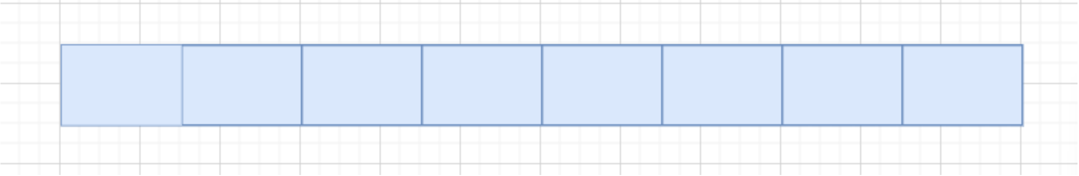
4. Analysis

We have given naïve approach for RMM. This given approach works on SISD (single instruction single data). So, here our program runs in sequence and works only single data. This approach increases the execution time.



So, here we can see that only first place is used for computation and we can not utilize the remaining part of vector. For optimization we try to utilize this blank space. For that we change our approach SISD to SIMD (Single instruction, multiple instruction).

SIMD supports instructions that can operate on vector data types. Group of array elements like a[0...3] or b[4...7] can be called vectors. Vectorization is the use of vector instructions to speed up program execution.

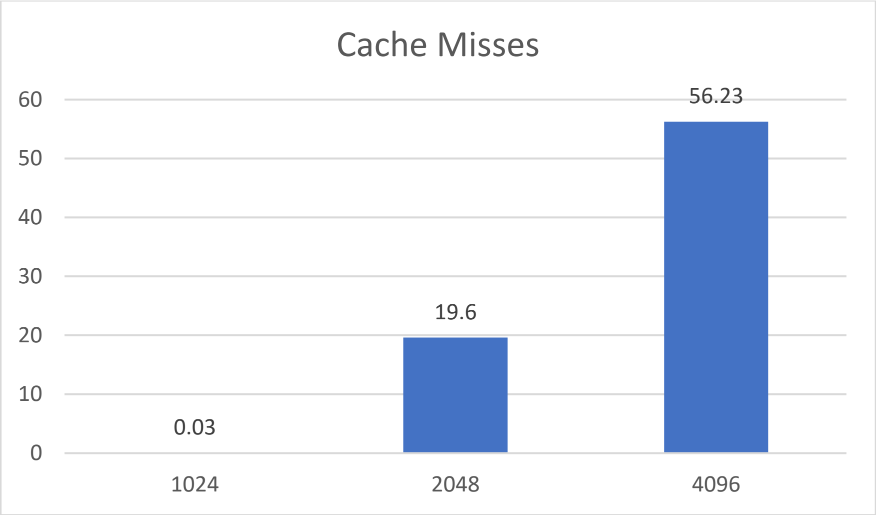


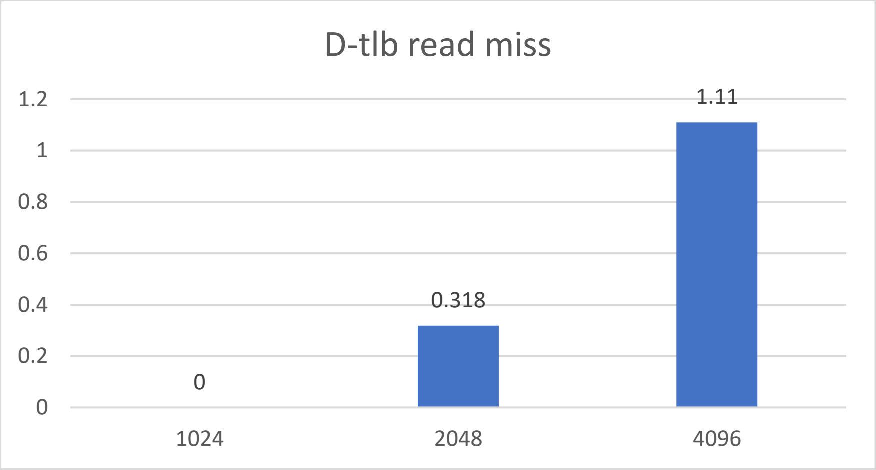
Here, we can see that all the space utilize properly. Ways to vectorize a loop computation in a C/C++ program. Programmers can use intrinsic inside the C/C++ source code to tell compilers to generate specific SIMD instructions so as to vectorize the loop computation. compilers may be setup to vectorize the loop computation automatically. This is called auto-vectorization.

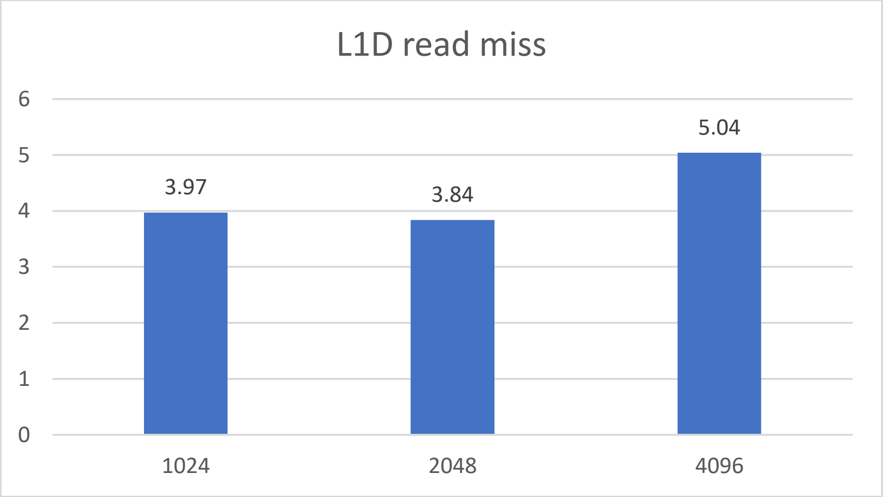
For analysis perf tool was used. The input sizes taken in consideration are 1024, 2048, 4096 and 8192.

Observation for naïve approach

We are giving a percentage of all the misses that are observed through perf evaluation and comparing to the observations found through different optimization approach.


Cache Misses
60
50
40
30
20
10
0
0.03
19.6
56.23
1024
2048
4096


D-tlb read miss
1.2
1
0.8
0.6
0.4
0.2
0
0
0.318
1.11
1024
2048
4096


L1D read miss
6
5
4
3
2
1
0
3.97
3.84
5.04
1024
2048
4096

Part A(i):

- **Why we need to optimize this?**

For given RMM code this will not utilize the memory properly. because this code run single instruction, single data approach. In one iteration only one data assign to register and remaining register space remain unutilized. We can see this problem in below given image. This is the first problem with given problem.

Second problem is order of loop execution. So, using this iteration we get high cache misses, L1, L2, L3 cache misses. Also, TLB hit is less. So, we get more execution time. We see this problem in first assignment.

- **How to optimize this code?**

To optimize this code, we can use three ways.

1. First, we change the order of loop execution. So, we had seen that order ikj run better compare to other loop execution order. So, we change the loop execution order.
2. Second, we can use the compiler optimization option.so during execution we give -O3 optimization. Using this option, compiler optimize the code by itself.
3. Above two options will not give much better result. So now we can use third way intrinsic instruction set. We add intrinsic instruction in our code to do vectorization by manually.

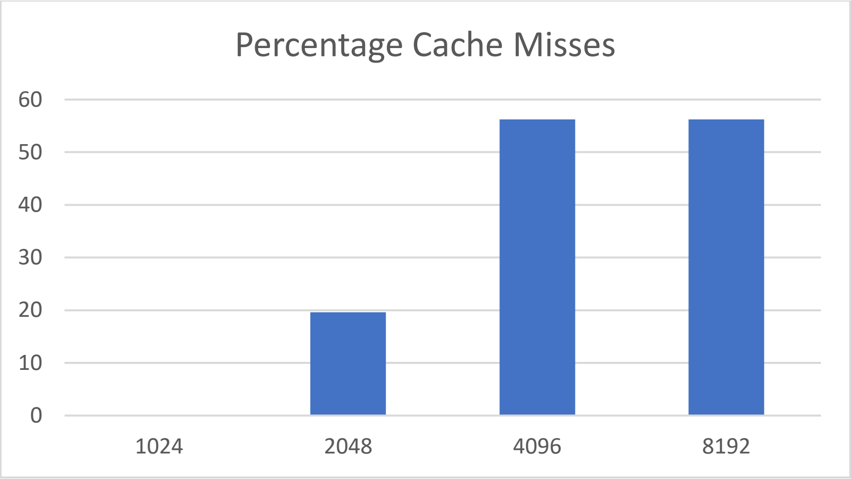
In the images above we can see that we utilize the blank space properly using vectorization.

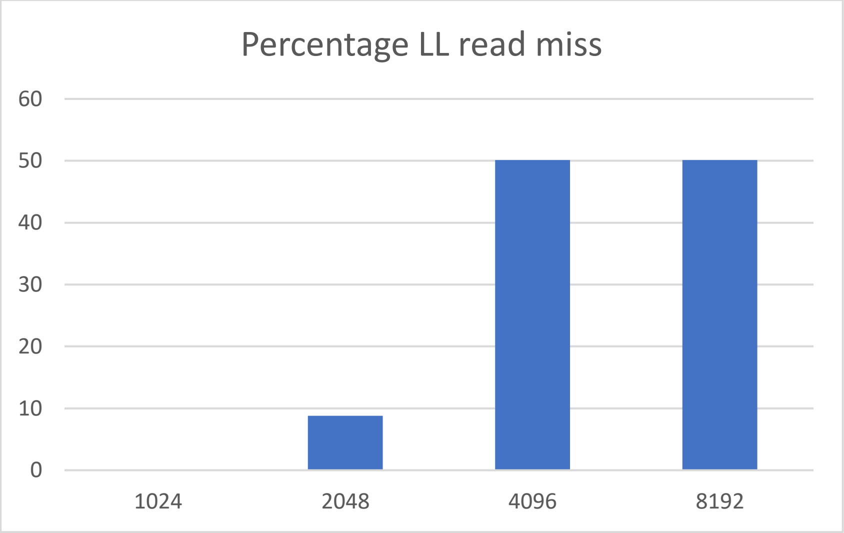
- For third way, we use Advanced vector extension (AVX) instruction set. This is x86instruction set architecture for intel and AMD.

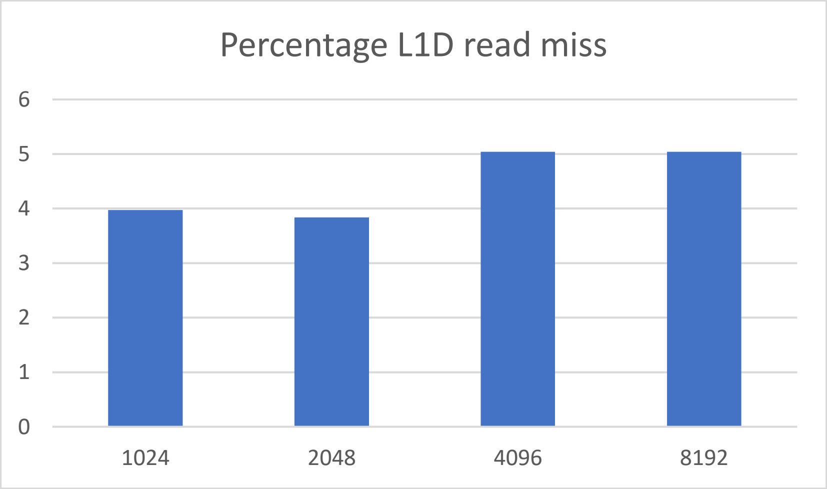
- In optimized code we use 32-bit integer vector of size 256 bit. So, in one vector we can add 8 integers in vector. Now, we can say that in one single iteration we can run operation on 8 data value. Using this we optimize the code and this optimized code give much better solution then naïve approach.

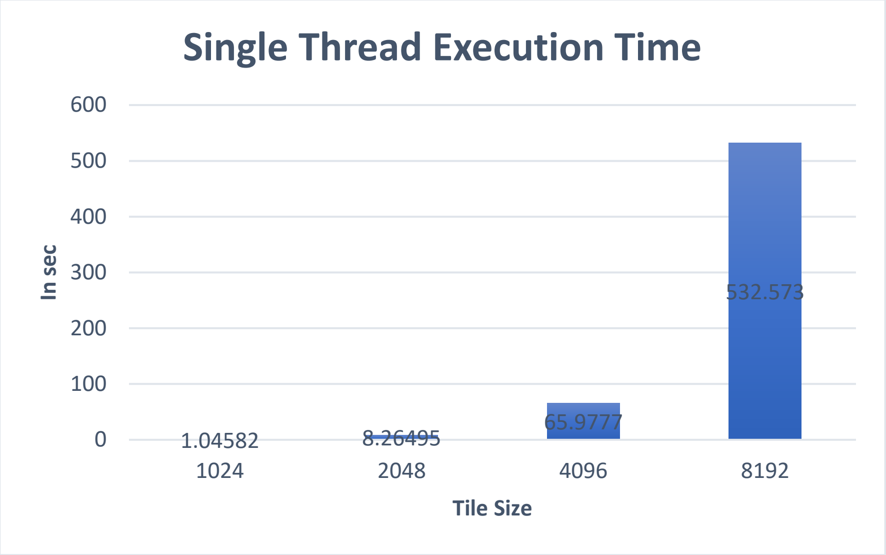
4.1 Observation with Single Thread

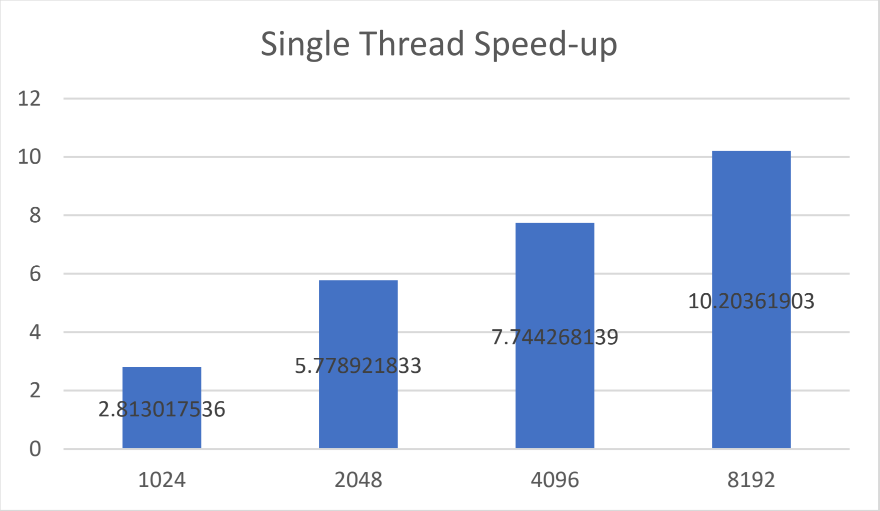
	1024	**2048**	**4096**	**8192**
Cache Misses	0.2435	17.06	31.30	36.64
L1D read miss	1.60	1.611	1.617	3.13
LL read miss	0.3755	18.66	11.48	18.22


Percentage Cache Misses
60
50
40
30
20
10
0
1024
2048
4096
8192


Percentage LL read miss
60
50
40
30
20
10
0
1024
2048
4096
8192


Percentage L1D read miss
6
5
4
3
2
1
0
1024
2048
4096
8192





From the graphs, it is observed that we get best speed-up with respect to matrix size 8192, that is approximately of 10.

Part A(ii):

- **Why we need threading?**
  Till now CPU use only one thread for whole program. So, our CPU cannot utilize full strength. Also, most of the thread in CPU remain ideal in this naïve approach. For better optimization we implement thread in our code.

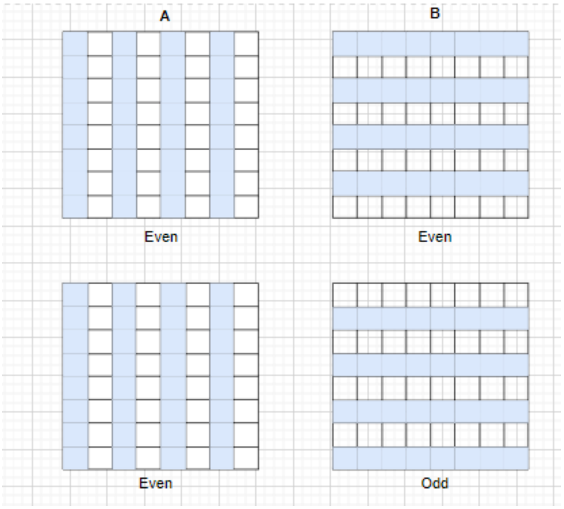
- **How to optimize the code?**

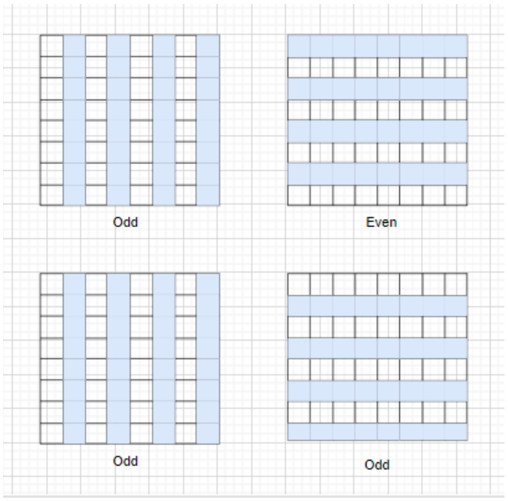
  To optimize the code, we divide the code into thread such that given thread run in parallel for better performance. In given RMM problem, we use this approach.

  Here, we create four thread which run parallel to each. In this problem, logic for RMM, We can see that for element (0,0) in output matrix we need the sum of multiplication of (Row1*Col1) + (Row2*Col1) + (Row1*Col2) +(Row2*Col2). So here we give each multiplication operation to each thread.

Data for different thread:

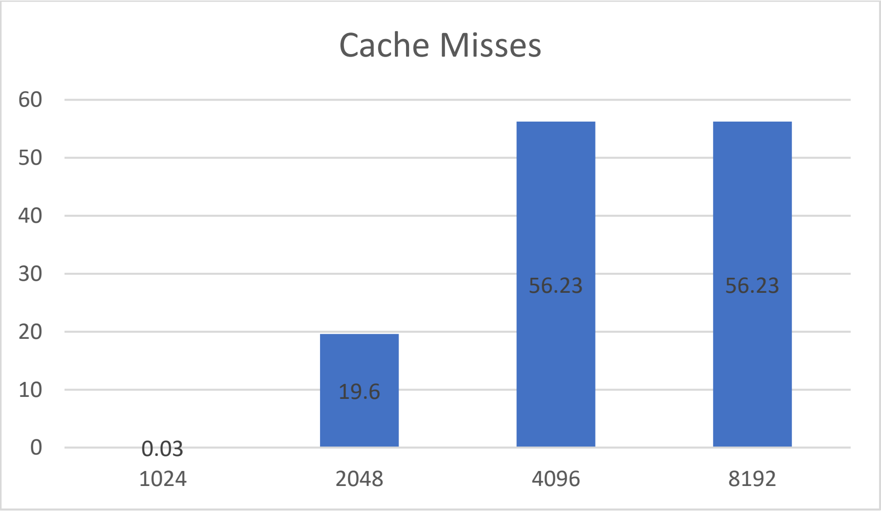
1. Even rows of matrix A and Even columns of matrix B.
2. Odd rows of matrix A and Even columns of matrix B
3. Even rows of matrix A and odd columns of matrix B
4. Odd rows of matrix A and odd columns of matrix B

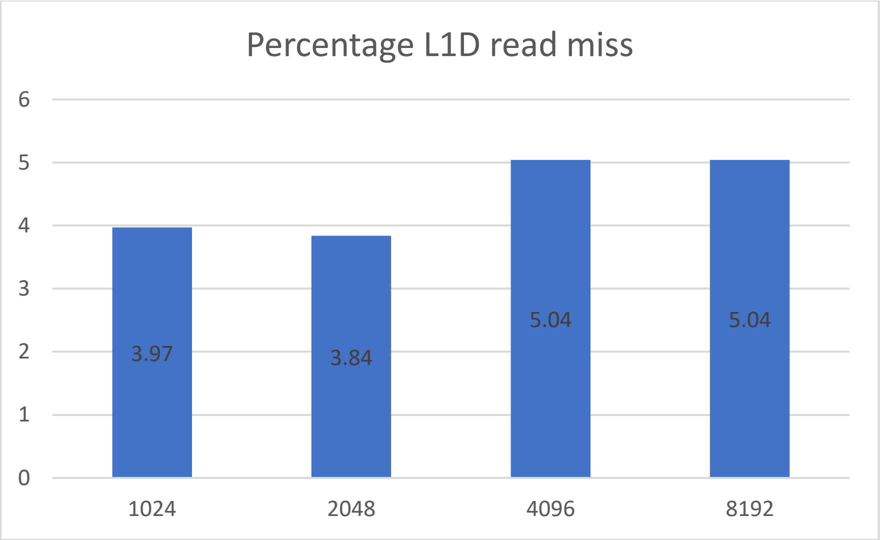


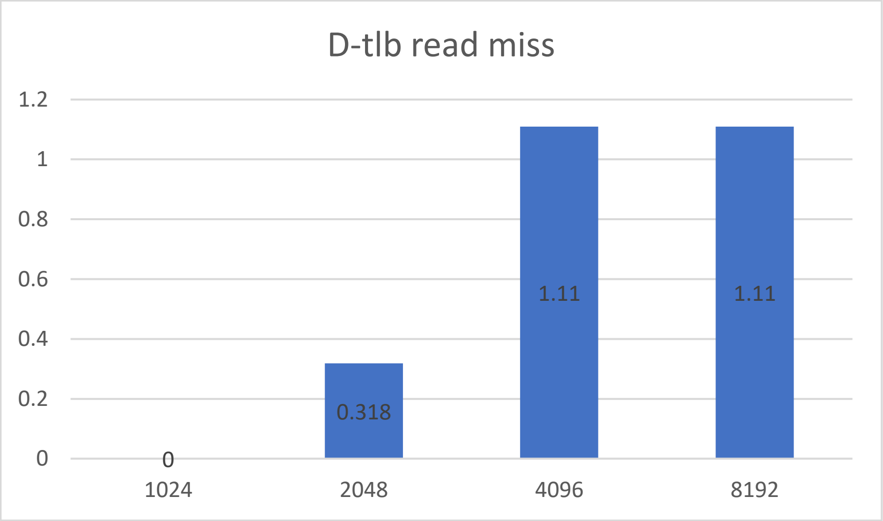


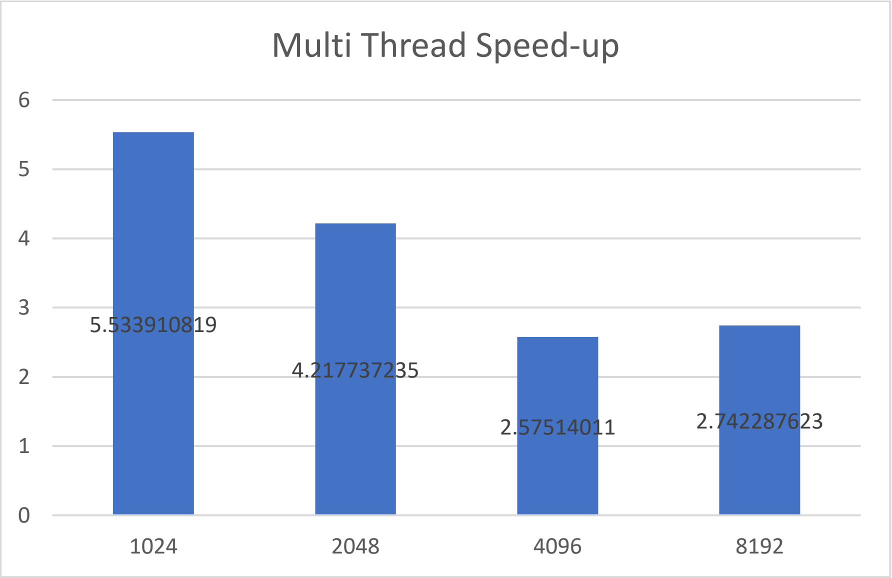
Above image will explain that what data each thread will get. After thread execution complete we get four (N/2)*(N/2) matrix. And during the thread join we sum all four matrix and get the final output matrix of size (N/2)*(N/2).

4.2 Observation with Multi-Threading









Multi-threading Speedup is observed to be 5.5 for a matrix size of 1024.

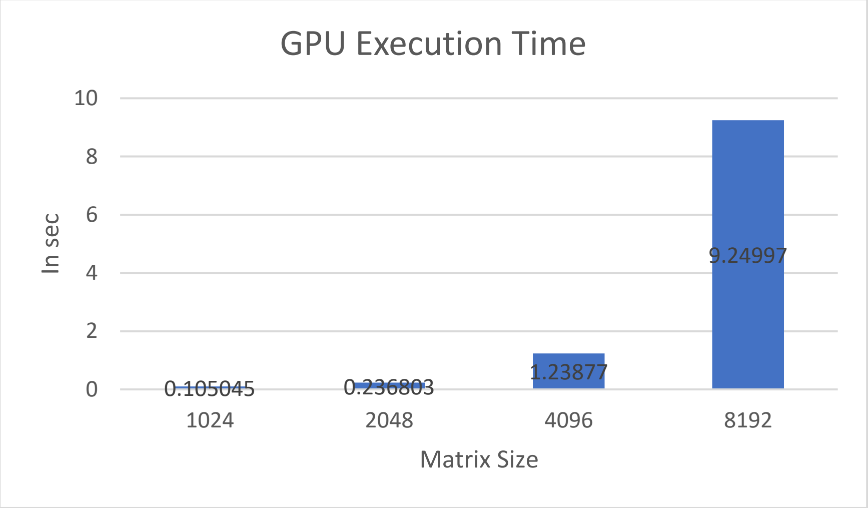
Part B: CUDA

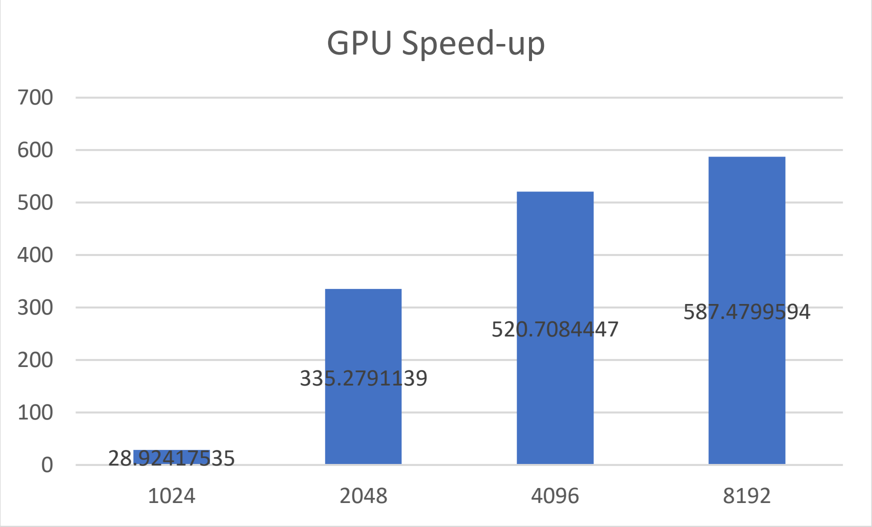
In this part we try to optimize the RMM even further in GPU. This is done using the inbuilt functions to create threads and blocks from CUDA Programming.

CUDA is a programming language to with C/C++ extensions and APIs for managing GPU and GPU memory. Here, we utilise both CPU and GPU in our program with CPU being the Host while GPU being the device. Thus, having separate memory for both the processing unit and Kernel being used as a medium to transfer the data from host to device and vice versa. The CPU is utilised for serial programming while GPUs for parallel programming.

GPU utilises the threads fully as opposed to how we did manually and adds to the performance exponentially as is observed in the given graphs.

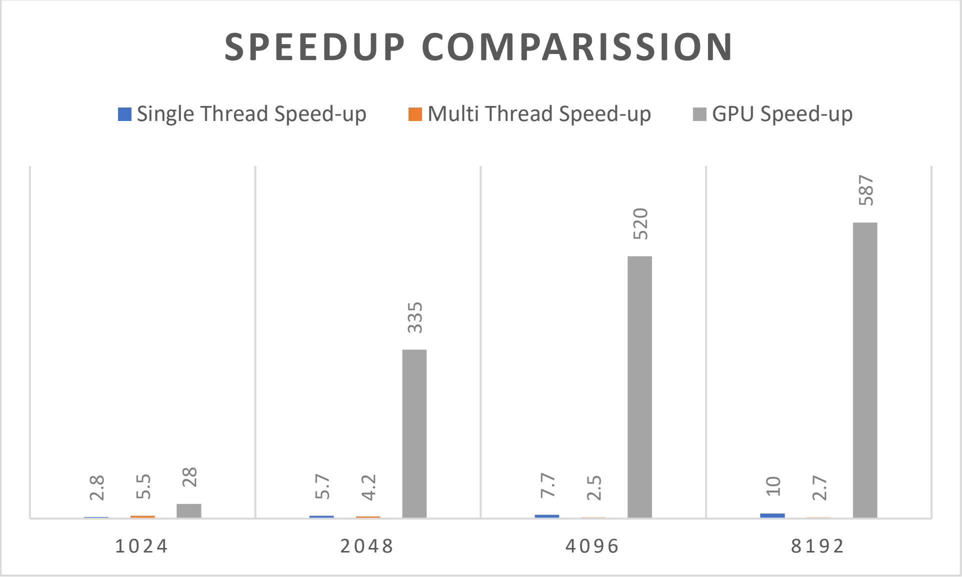
4.3 Observation with CUDA





The speed-up achieved using CUDA is as large as 587 for the matrix size of 8192 when compared with the basic RMM approach.

5. Conclusion



In naive approach we didn’t perform any parallelism and vectorization thus it gives the highest time used. In single threading we used vectorization so as to get better performance and thus the running time is decreased.

We observe the highest speedup with CUDA programming as it utilise maximum number of threads.