E0-243: Assignment-3

Bikram Kumar Panda
17030
CDS,PhD

INTENDED BLANK SPACE

Table of Contents

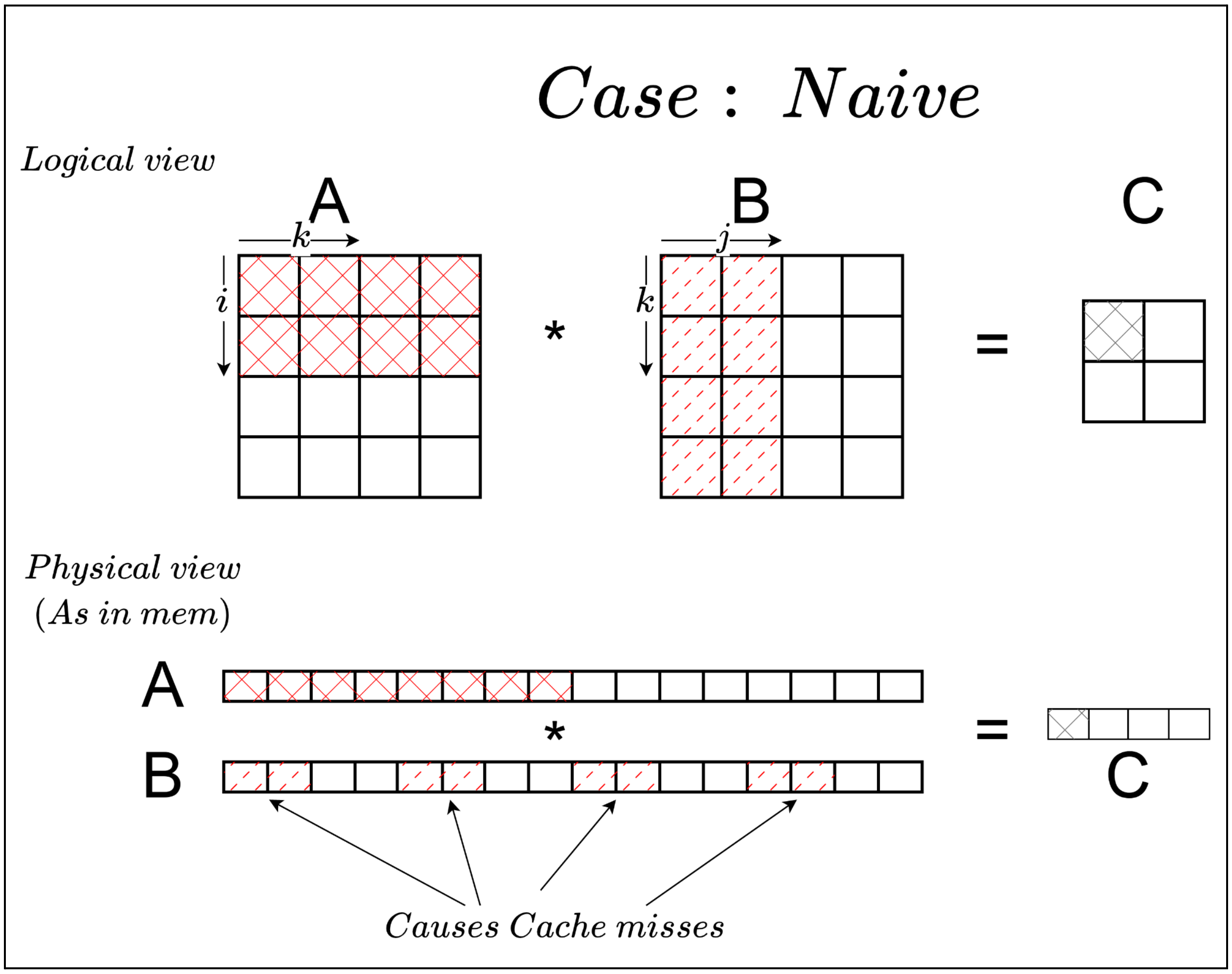
Resources and Instructions:

1. GITHUB LINK [https://github.com/bikipanda/hpca_Assignment3]
2. The single_thread.h contains the naive and the optimized routine. Please uncomment the one you would want to run.
3. The multi_threaded.h contains the routine for multithreaded code.

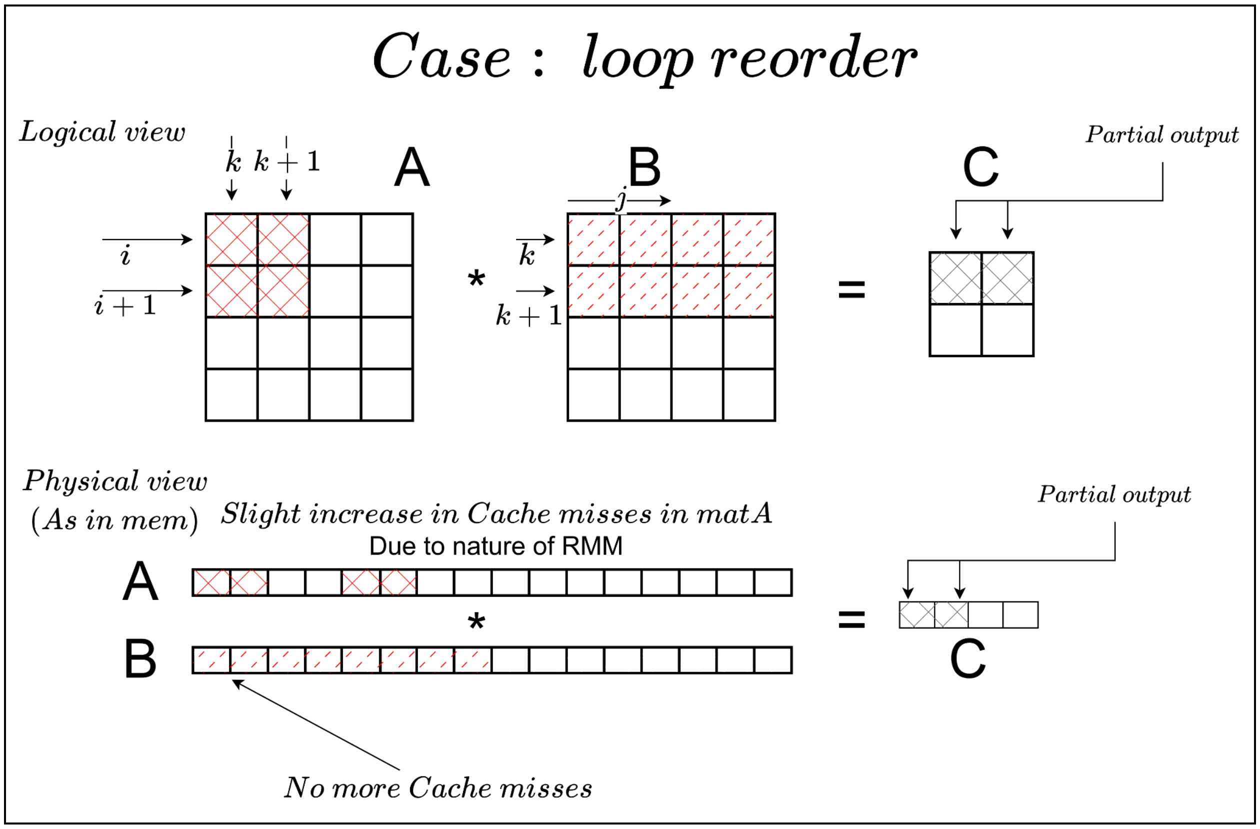
Optimising Single Threaded Implementation

Part A : Theoretical Analysis

1. We were given a naive implementation of the single thread computation of Reduced Matrix Multiplication (RMM).
2. The top loop was across rows of matrix A, next lower loop was across the cols of matrix B and the innermost loop was across the iterations required for RMM.
3. In the Figure below, we can see that **this naive configuration is bounded due to Cache misses since the memory access pattern used to access the column elements of matrix B is not spatially local.**

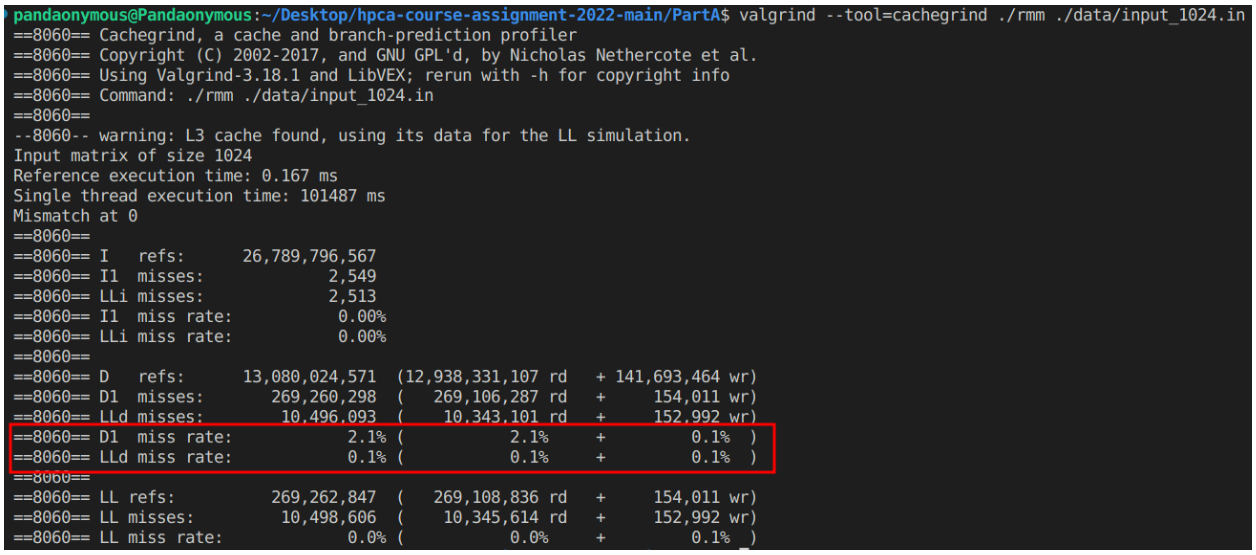


4. We can solve this issue cache misses by simply reordering our loops. The figure below shows how just by reordering the loops we can reduce the amount of cache misses.



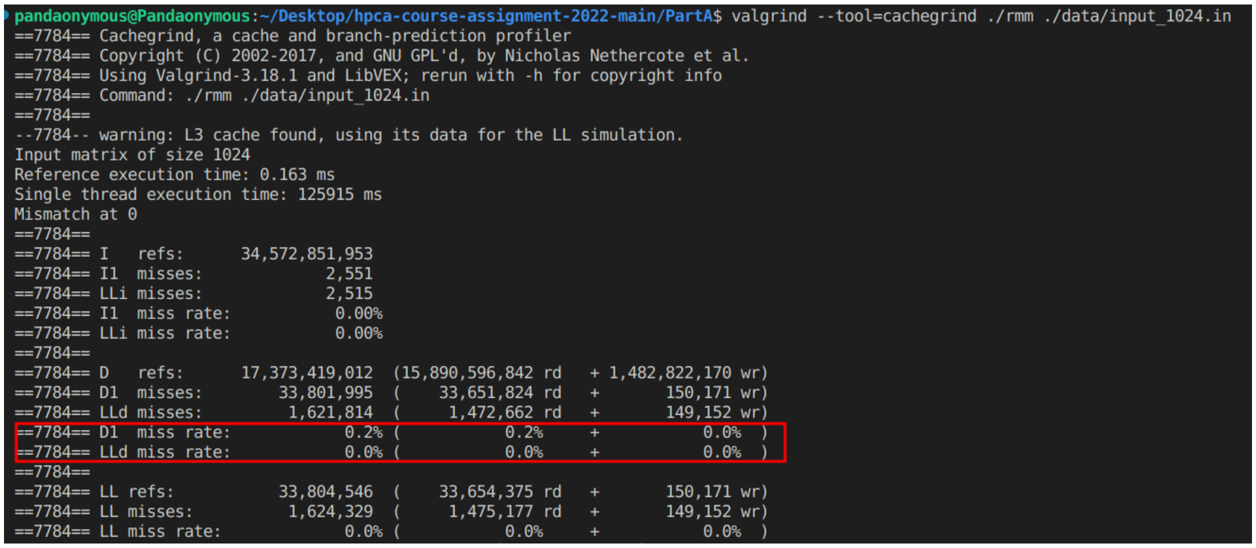
Part B : Experiment Analysis

1. We profiled both the Naive and the optimized RMM and observed that indeed after doing the optimisation we were able to reduce the Cache misses by a large amount.
2. We also observed that the cache miss rate increases as the matrix size increases since now the program is dealing with more data.
3. Cache Profile of Naive single thread RMM with N = 1024.



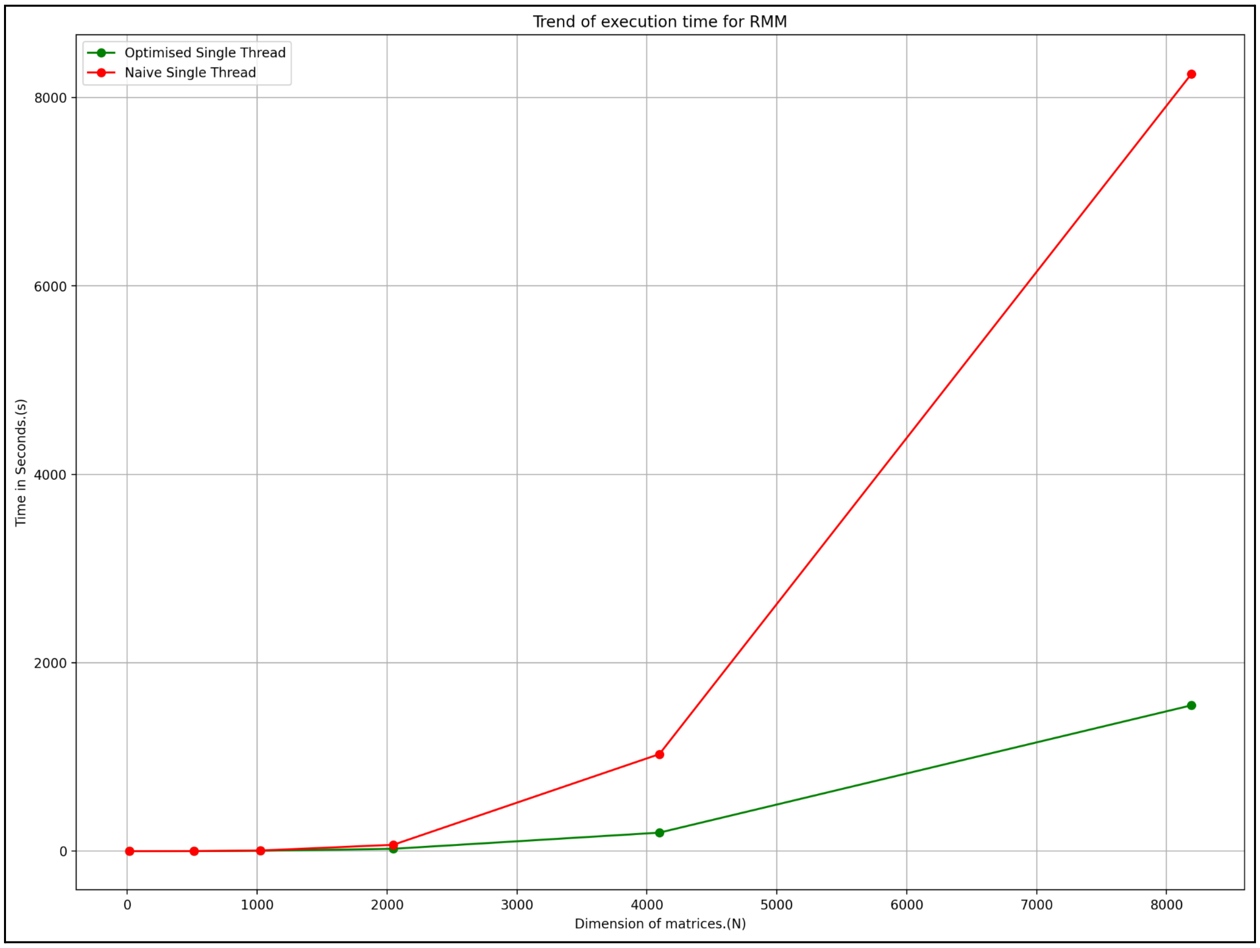
```
pandaonymous@Pandaonymous:~/Desktop/hpca-course-assignment-2022-main/PartA$ valgrind --tool=cachegrind ./rmm ./data/input_1024.in
==8060== Cachegrind, a cache and branch-prediction profiler
==8060== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==8060== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==8060== Command: ./rmm ./data/input_1024.in
==8060==
--8060-- warning: L3 cache found, using its data for the LL simulation.
Input matrix of size 1024
Reference execution time: 0.167 ms
Single thread execution time: 101487 ms
Mismatch at 0
==8060==
==8060== I   refs:      26,789,796,567
==8060== I1  misses:             2,549
==8060== LLi misses:             2,513
==8060== I1  miss rate:           0.00%
==8060== LLi miss rate:           0.00%
==8060==
==8060== D   refs:      13,080,024,571  (12,938,331,107 rd   + 141,693,464 wr)
==8060== D1  misses:       269,260,298  (   269,106,287 rd   +     154,011 wr)
==8060== LLd misses:        10,496,093  (    10,343,101 rd   +     152,992 wr)
==8060== D1  miss rate:            2.1% (           2.1%     +         0.1%  )
==8060== LLd miss rate:            0.1% (           0.1%     +         0.1%  )
==8060==
==8060== LL refs:          269,262,847  (   269,108,836 rd   +     154,011 wr)
==8060== LL misses:         10,498,606  (    10,345,614 rd   +     152,992 wr)
==8060== LL miss rate:             0.0% (           0.0%     +         0.1%  )
```

4. Cache Profile of Optimized Single Thread RMM with N= 1024.

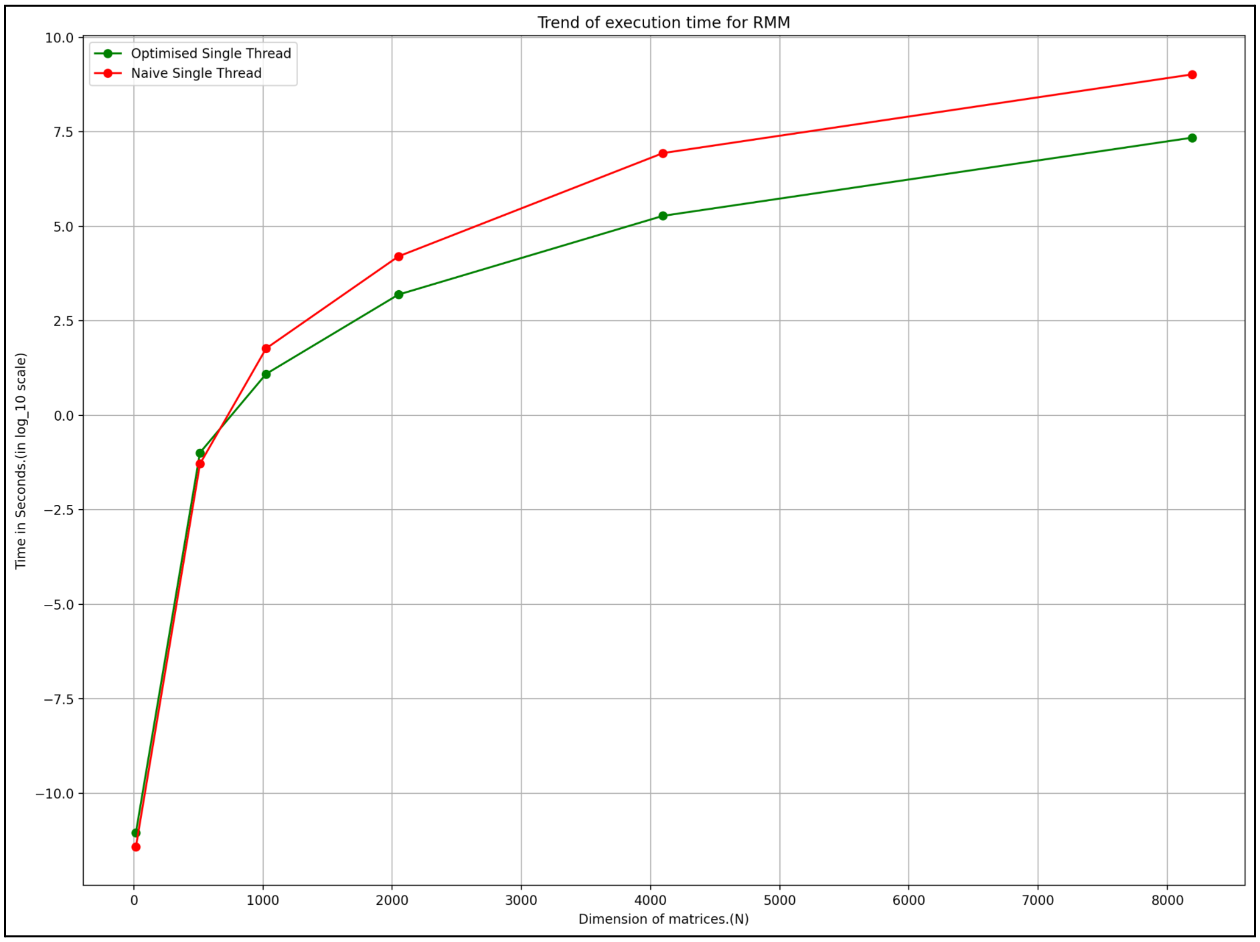


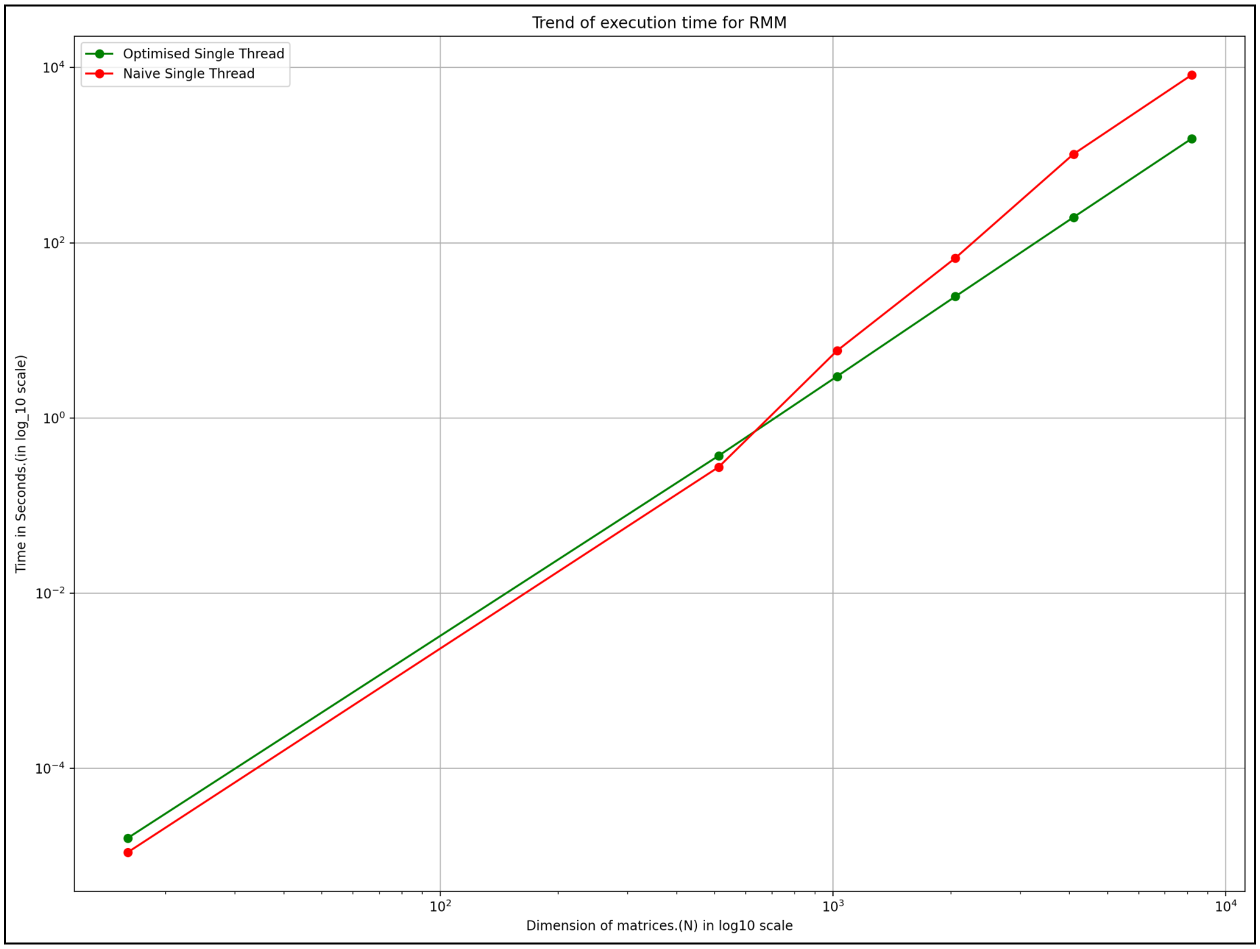
```
pandaonymous@Pandaonymous:~/Desktop/hpca-course-assignment-2022-main/PartA$ valgrind --tool=cachegrind ./rmm ./data/input_1024.in
==7784== Cachegrind, a cache and branch-prediction profiler
==7784== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==7784== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7784== Command: ./rmm ./data/input_1024.in
==7784==
--7784-- warning: L3 cache found, using its data for the LL simulation.
Input matrix of size 1024
Reference execution time: 0.163 ms
Single thread execution time: 125915 ms
Mismatch at 0
==7784==
==7784== I   refs:      34,572,851,953
==7784== I1  misses:             2,551
==7784== LLi misses:             2,515
==7784== I1  miss rate:           0.00%
==7784== LLi miss rate:           0.00%
==7784==
==7784== D   refs:      17,373,419,012  (15,890,596,842 rd   + 1,482,822,170 wr)
==7784== D1  misses:        33,801,995  (    33,651,824 rd   +       150,171 wr)
==7784== LLd misses:         1,621,814  (     1,472,662 rd   +       149,152 wr)
==7784== D1  miss rate:            0.2% (           0.2%     +           0.0%  )
==7784== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==7784==
==7784== LL refs:           33,804,546  (    33,654,375 rd   +       150,171 wr)
==7784== LL misses:          1,624,329  (     1,475,177 rd   +       149,152 wr)
==7784== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

5. We can observe that by using our optimized scheme, **we were able to reduce the data miss-rate from 2.1% to 0.2%**.
6. Plots:
    a. **Execution time Comparison**



b. Execution time comparison on log scales





Observations:

1. As we increase the N by an order of 2, the execution time of Optimised implementation is increasing by an order of 8.
2. This is why we have a linear plot when taking log scale on both x and y axis.

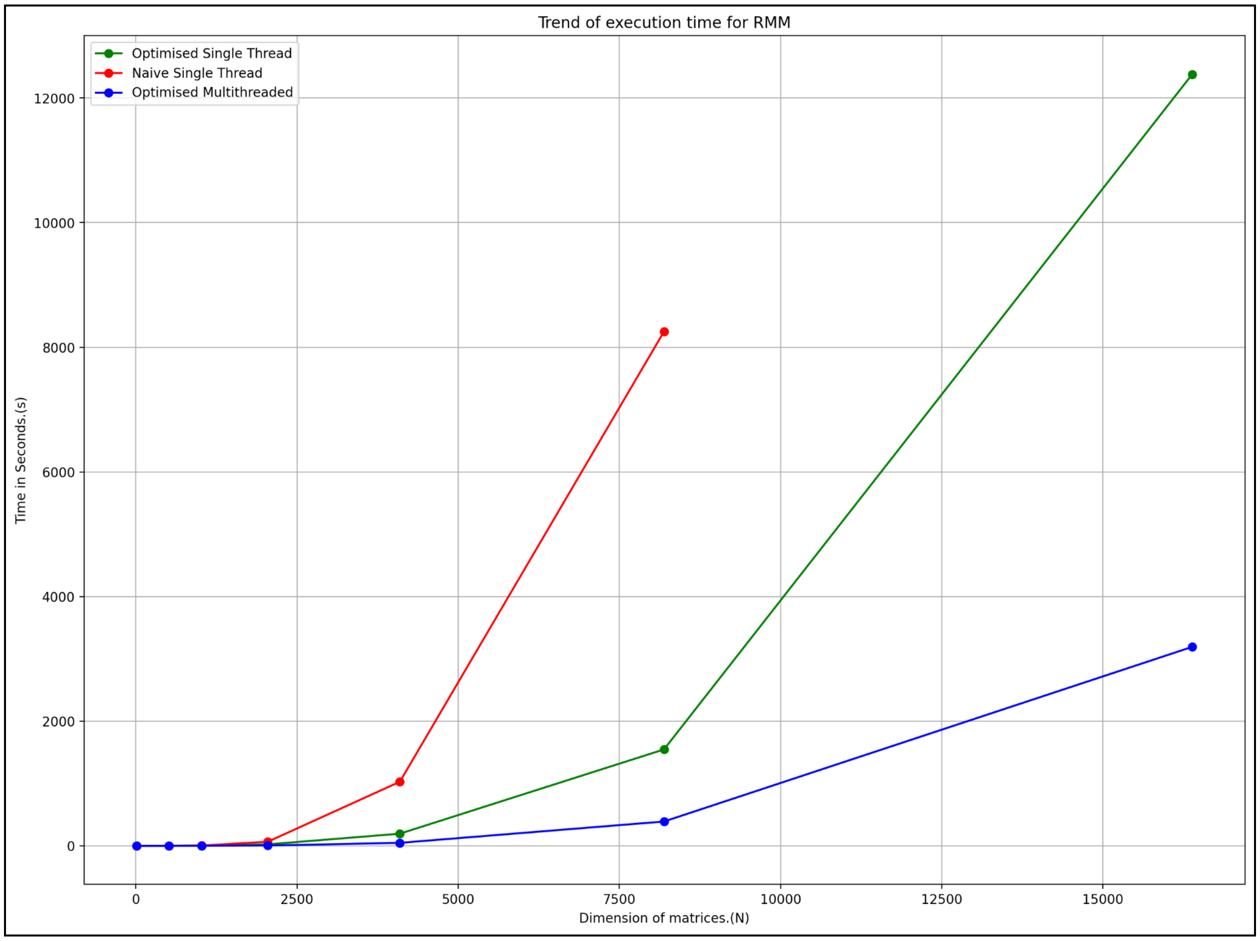
Multi-Threaded Implementation

Part A : Theoretical Analysis

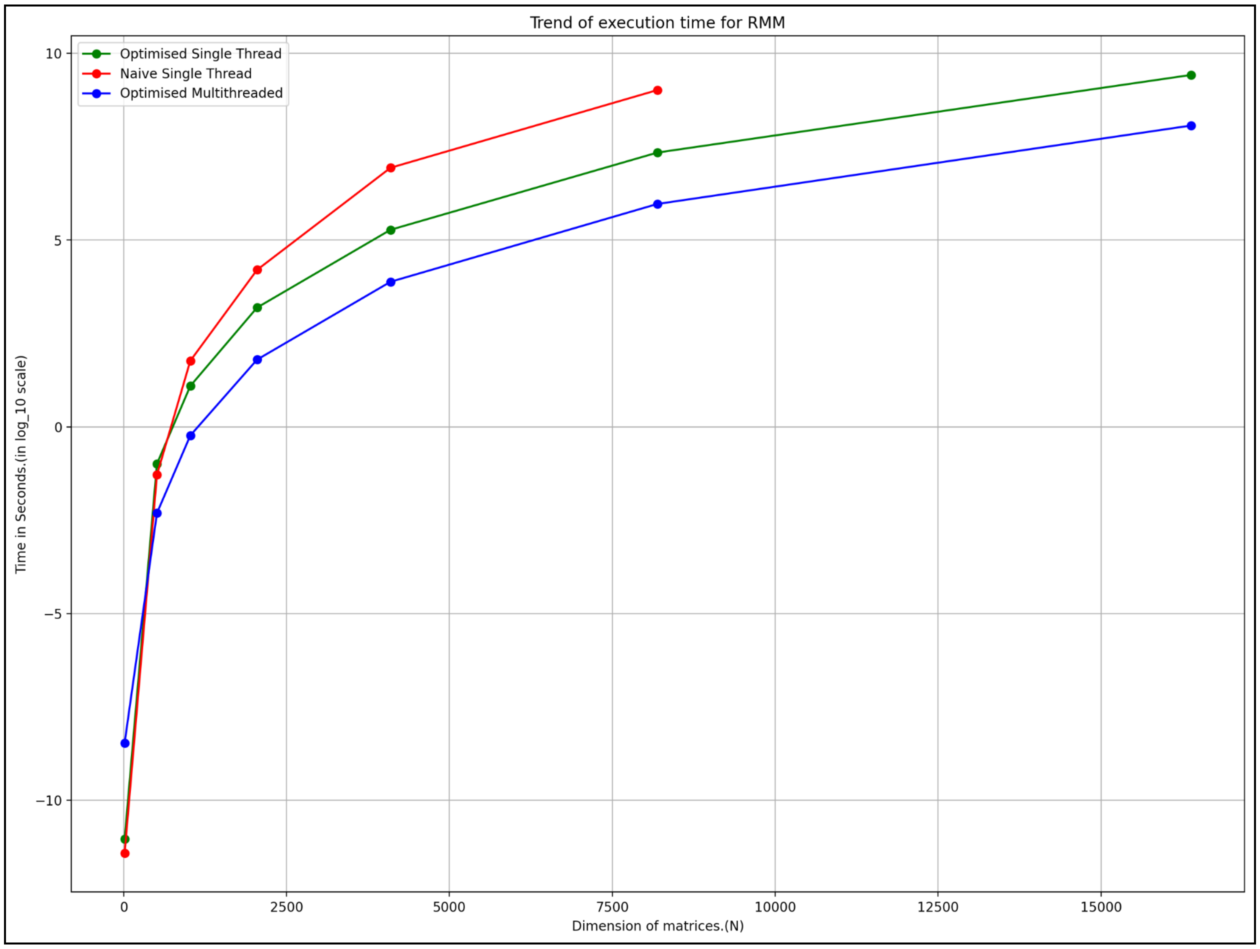
1. The benefit of working on multi thread is that we can distribute our workload onto multiple compute units which can work in parallel thus reducing the execution time.
2. Still it is not guaranteed that we will always have a better execution time in muti-threading if the algorithm itself is sequential in nature or requires communication between processes. In these scenarios the contention overhead will be high.
3. For this part, we will be multithreading the optimized single threaded code from the previous part and compare it with the previous two single threaded implementations of RMM.
4. In our implementation(which is the multi thread implementation of our loop reordering optimisation), we do not require locking and unlocking if we have an even thread count. This is due to the fact that N will always be a power of 2 and RMM requires two rows and two columns to compute one cell. Effectively that means no two threads will be contending to write onto the same index position at the same time.

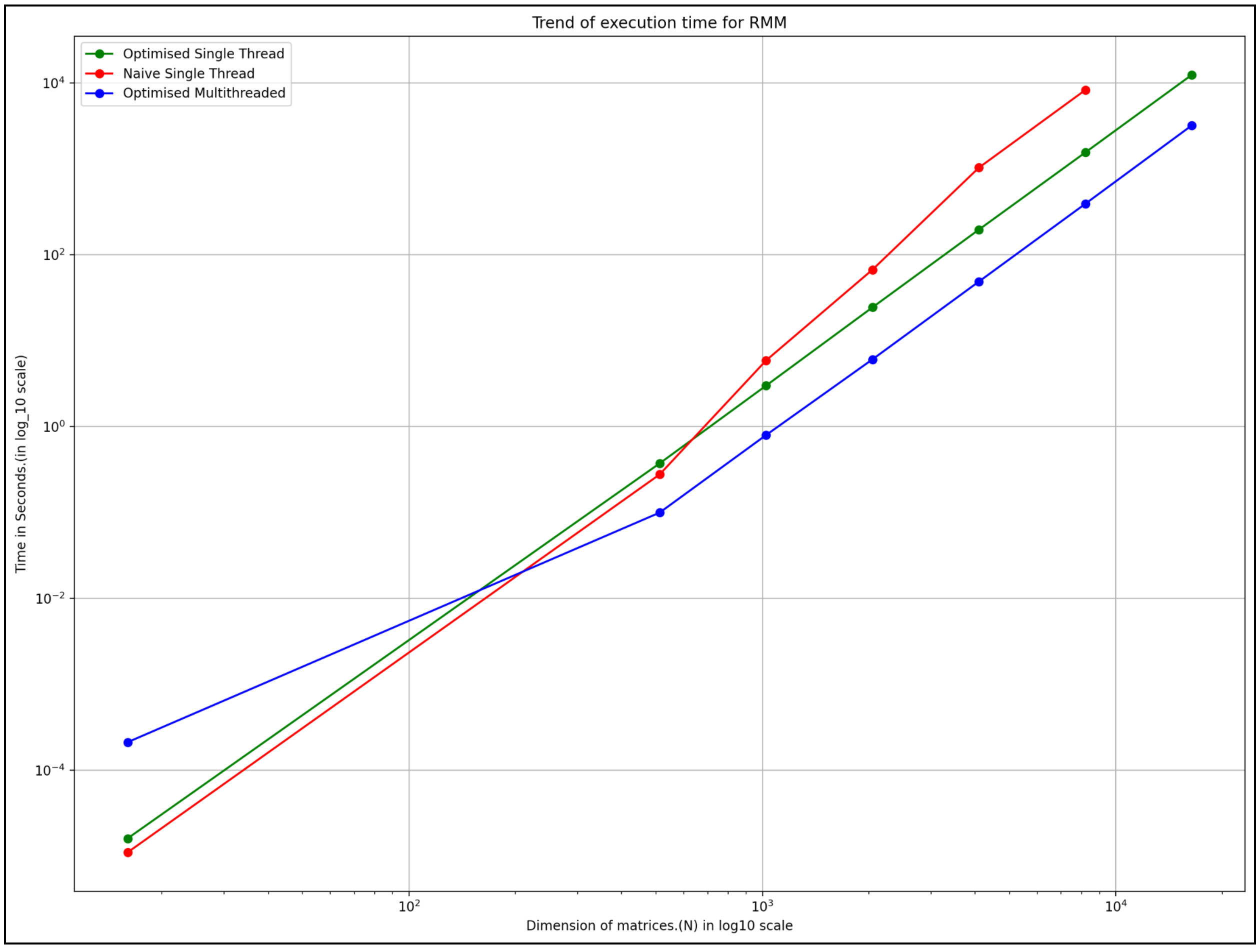
Part B : Experiment Analysis

1. **Execution Time Plots:**
   Note: I did not run the Naive single threaded for 16384, since I estimated that it would take 5-6 days to finish on my machine.
   Similarly for the Optimised Single threaded for N = 16384, but since I was able to get a consistent factor of 8 everytime the N increases by a factor of 2, I have speculated its value in the graph. All optimized Multithreaded data is from experiments.



2. Axes in log scale

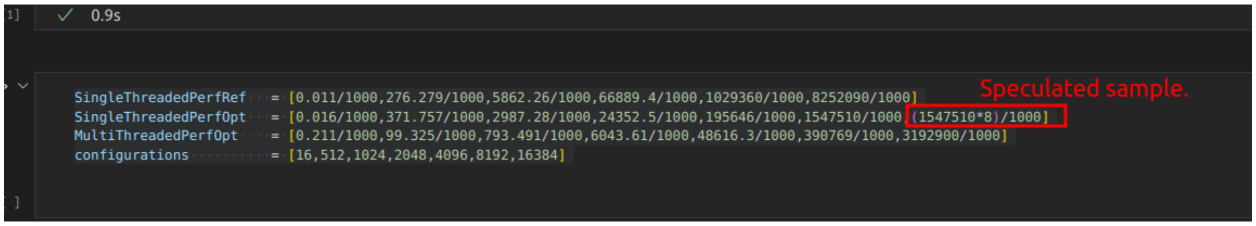

Trend of execution time for RMM
Optimised Single Thread
Naive Single Thread
Optimised Multithreaded
Time in Seconds.(in log_10 scale)
Dimension of matrices.(N)
10
5
0
−5
−10
0
2500
5000
7500
10000
12500
15000



Observations:

1. We can observe that for smaller matrix sizes , the overhead of the pthreads becomes a disadvantage. Thus Pthreads are advised if N is large.
2. Just as we saw in the optimized single threaded implementation, here also as we increase the N by an order of 2, the execution time of the optimized implementation is increasing by an order of 8. Thus the linear plot in log scales.

DATA USED FOR PLOTTING THE GRAPHS IN PART 1 AND 2



```
SingleThreadedPerfRef   = [0.011/1000,276.279/1000,5862.26/1000,66889.4/1000,1029360/1000,8252090/1000]
SingleThreadedPerfOpt   = [0.016/1000,371.757/1000,2987.28/1000,24352.5/1000,195646/1000,1547510/1000,(1547510*8)/1000]
MultiThreadedPerfOpt    = [0.211/1000,99.325/1000,793.491/1000,6043.61/1000,48616.3/1000,390769/1000,3192900/1000]
configurations          = [16,512,1024,2048,4096,8192,16384]
```

Note the speculated sample.It was taken since we saw that for a 2x increase in N there was a regular increase in execution time by 8x.

GPU implementation

Part A : Theoretical Analysis

Part B : Experiment Analysis