CA Assignment - 2

Kuchi Sai Teja (21317, kuchisai@iisc.ac.in)
Utkrisht Patesaria (20892, utkrishtp@iisc.ac.in)

Link to code: https://github.com/UTKRISHTPATESARIA/hpca

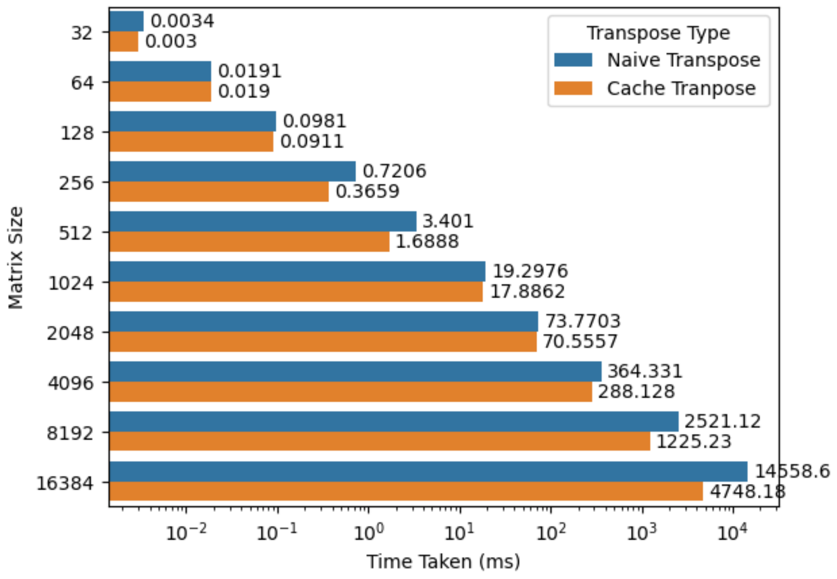
Question-1 Part-A Single Thread :-

- All graphs are log scaled.
- For sizes :- 16, 4096, 8192, 16384, the program is ran for a total of 3 times for getting stable results.
- The program is ran on dgx server which has Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz

Optimization Process :-

1. The very basic optimizations include converting multiplications to right shift and divisions to left shift which doesn't improve performance to a whole lot level but is still taken care of.

2. Generally when accessing matrices, fetching the next elements i+1, i+2 (if current index is i) is faster as matrices as stored in row-fashion order. Hence with this hunch its better to transpose second matrix when performing matrix multiplication to exploit performance.
    a. 2 different types of tranposes methods are used based on the matrix size that is passed to the program.
    b. Its observed that for smaller matrix sizes naive transpose (get_tranpose in code) is same as cache transpose hence for simplicity the former is used.
    c. But for larger matrix sizes there is an **cache oblivious transpose** approach ("Cache Oblivious Approach", n.d.) which helped achieving speedup of ~3x (-10 secs) for tranpose case for matrix size of 16k as can be seen in the below figure.



3. The problem statement was to multiply first row and second row of first matrix with first column and second column of second matrix which will be first row and second row of second matrix after transpose of second matrix. In mathematical notation its as follows
    a. Let f = first, s = second
    b. Result = A[f] * B[f] + A[f] * B[s] + A[s] * B[f] + A[s] * B[s].
    c. The above is performing 4 multiplication operations and 3 addition operations which can be reduced as there are some redundant operations.
    d. The equation in 3b can be reduced using the below equation which is just 2 additions and 1 multiplication.
    e. Result = (A[f] + A[s]) * (B[f] + B[s])
    f. This above optimization further helps improve in boosting the performance.

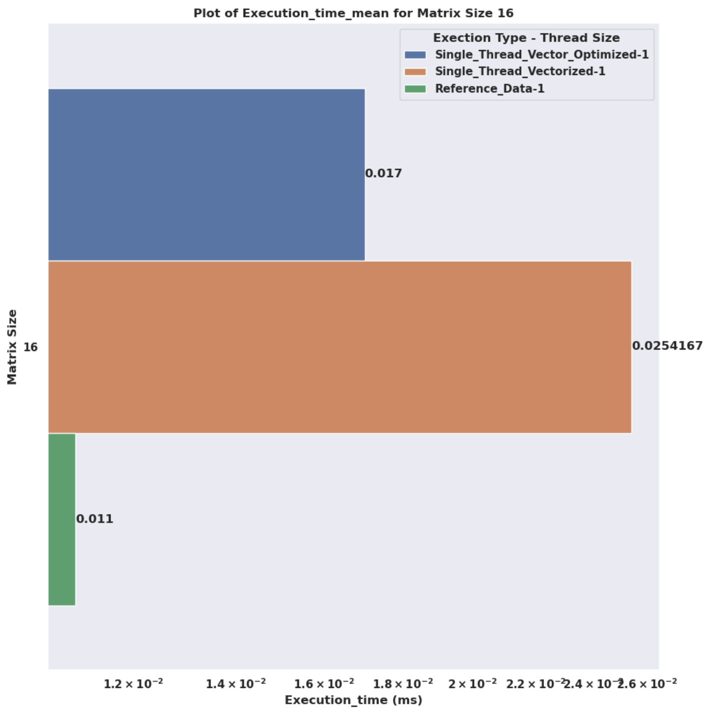
4. Further, it was observed that when performing matrix multiplication of A's first 2 rows with B's 2 rows (because of transpose), many computations are getting recalculated again and again. Hence to avoid this 2 new arrays of size N/2 * N are created which will hold the sum of consecutive rows. (0th row of resultant matrix will hold sum of 0th and 1st row of actual matrix, while 1st row of resultant matrix will hold sum of 2nd and 3rd row of actual matrix). Thus whenever matrix multiplication is being done there is no need of performing the addition of consecutive rows everytime.

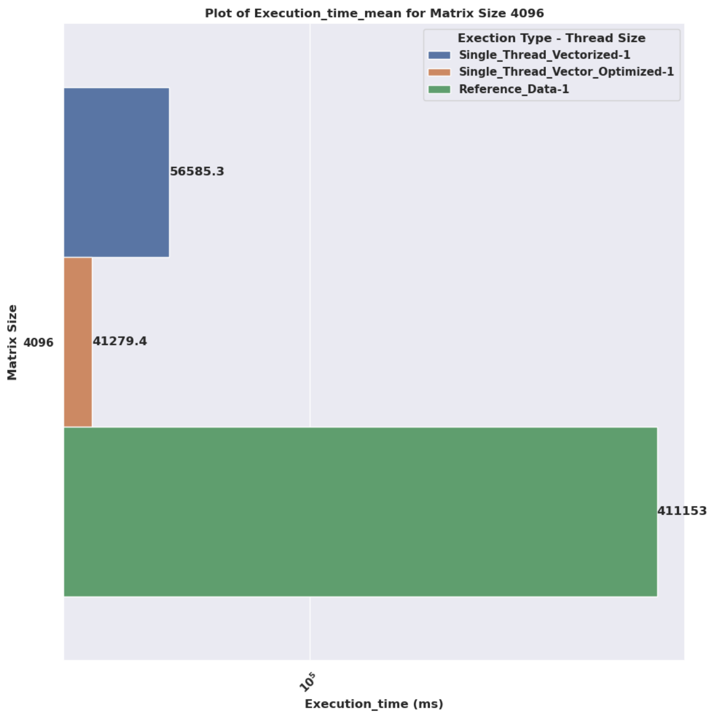
5. Current x86 ISA which is being used in the underlying hardware can only perform an operation on 2 numbers at once. Thus to further exploit performance AVX2 ("Advanced Vector Extensions", n.d.) is being used which can perform same operation on 8 integers at once is being used. There are plots below that show the difference in execution time and other parameters for implementation involving creation of the additional matrices (discussed in the above point 5) along with vectorization does help in improving performance over the implementation where plain vectorization is used.

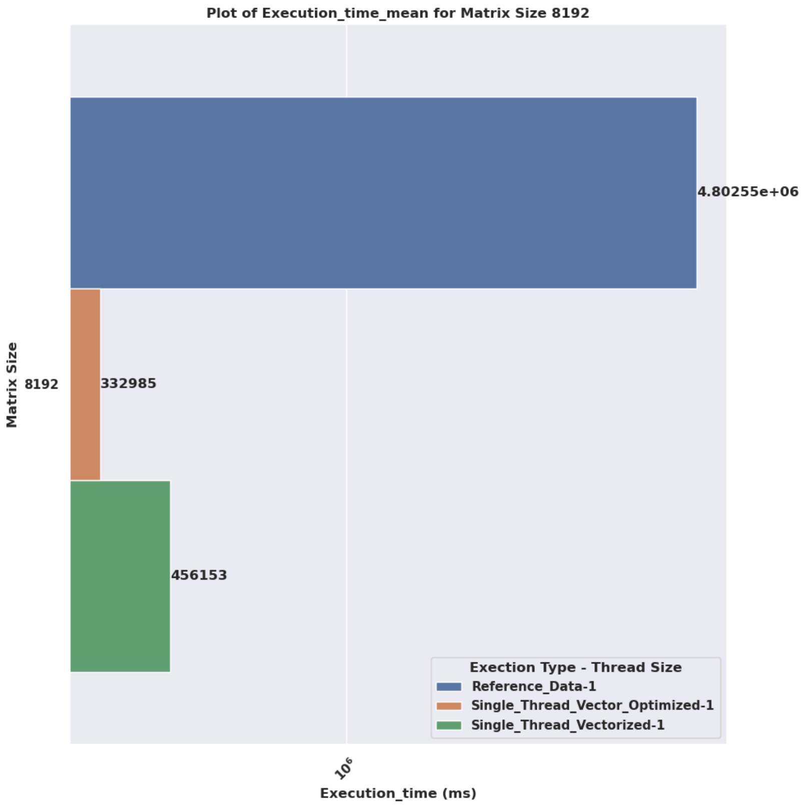
6. The primary disadvantage of it being its not reusable in the sense, if the datatype that operations to be performed changes to floats/double from integer, different methods should be used altogether. Thus a new binary needs to be created altogether by making changes to souce code.

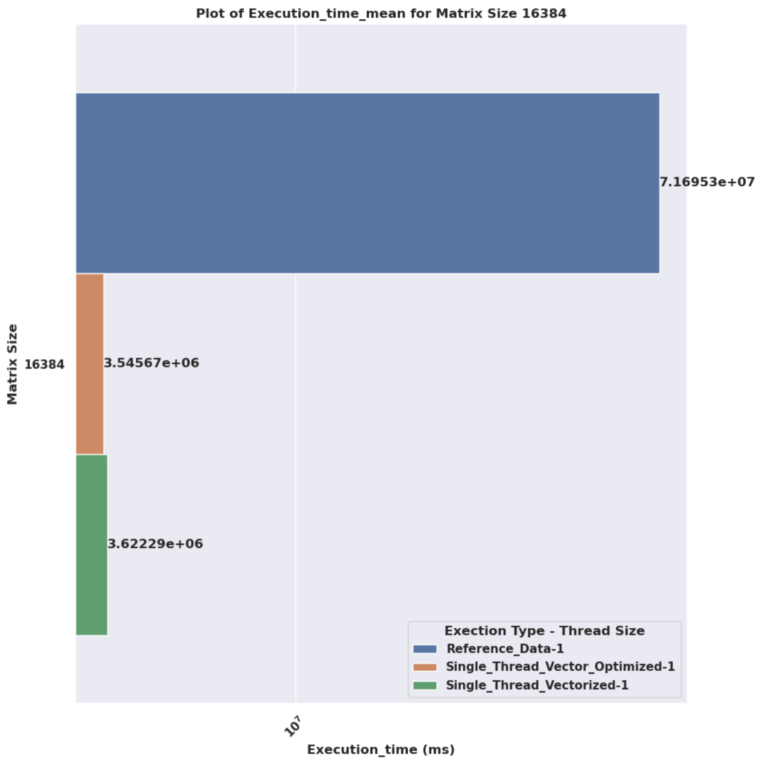
7. It was observed that not using vectorization and just interchanging loops (code present in header file) is performing better than the reference function but still slower than the vectorization functions. Its an observation that is made during the initial program building. Hence the data for the same is not captured and thus they will not be seen in any plots.

Execution Time



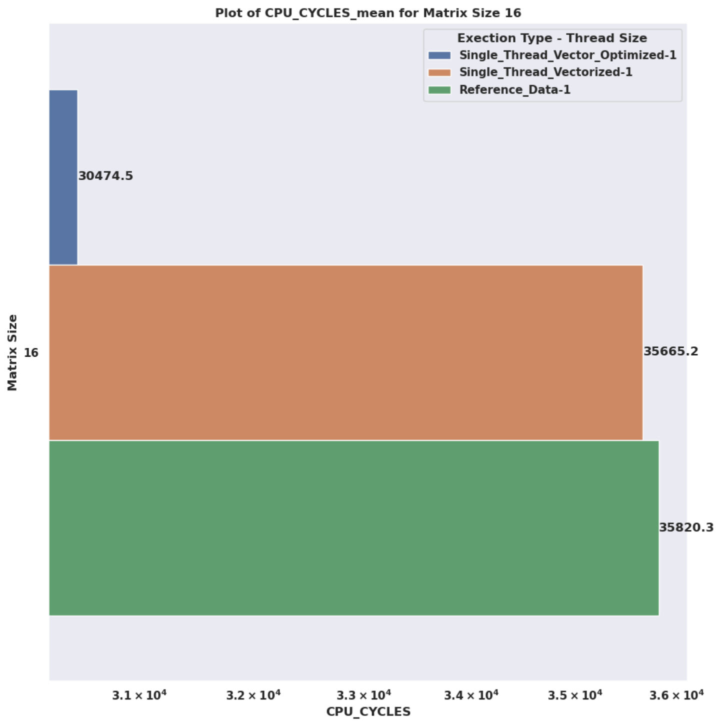


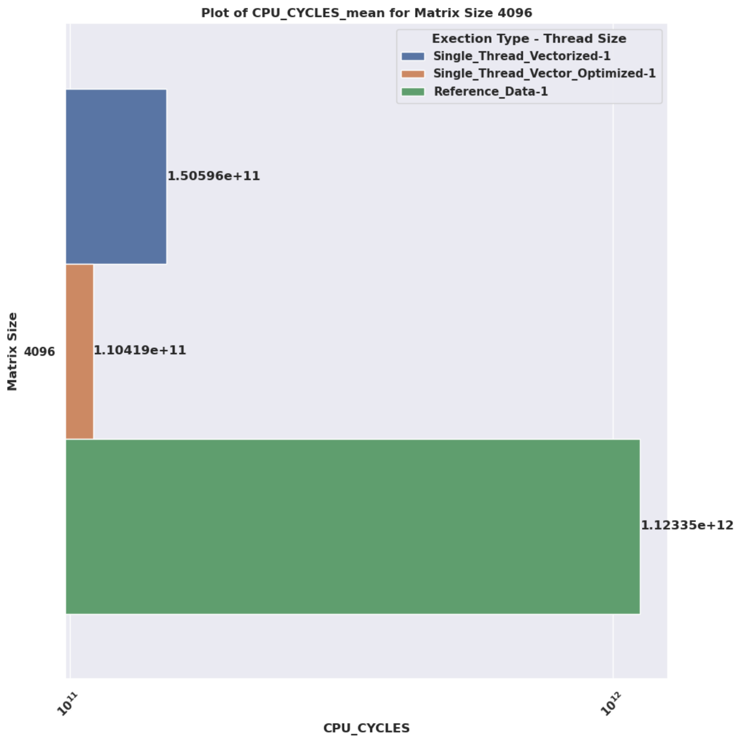


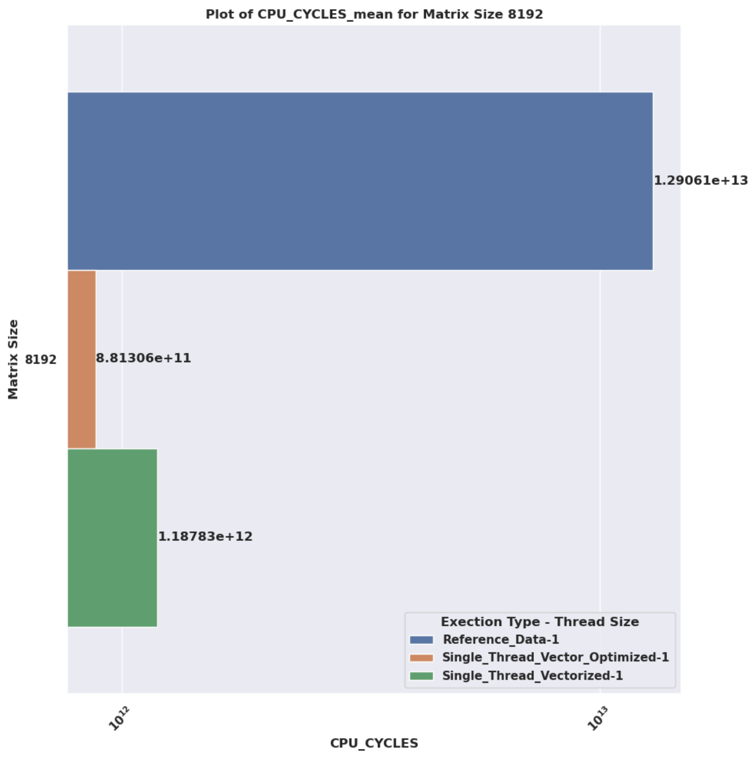


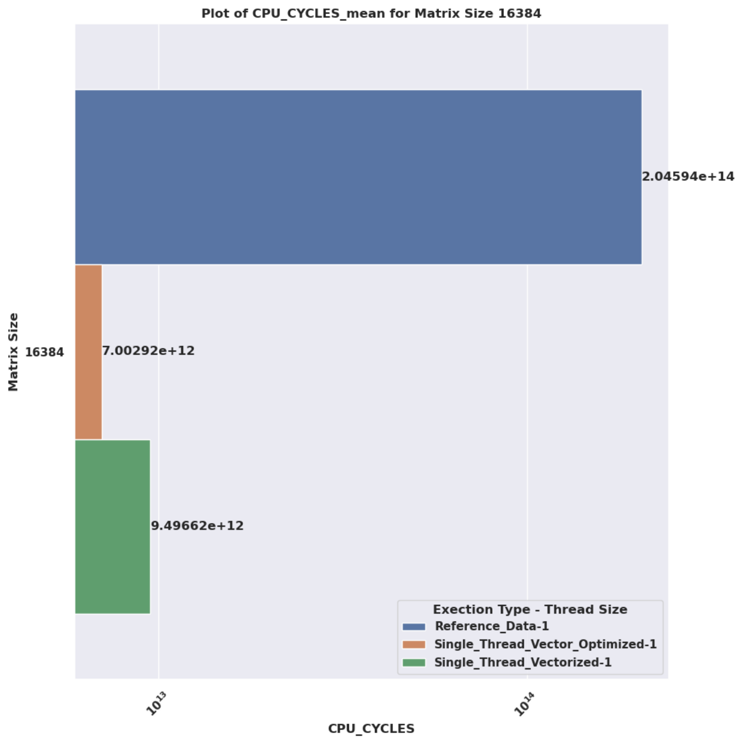
- The above 4 plots show execution time for different matrix sizes (16, 4096, 8192, 16384). Its says **mean** because an average of 3 runs time is captured.
- Reference_Data - 1 implies its the time taken by the reference function which uses a single thread to complete execution.
- Single_Thread_Vectorized-1 implies, the matrix multiplication is performed with the help of AVX using a single thread.
- Single_Thread_Vector_Optimized-1 implies before performing matrix multiplication, the optimization mentioned in point 4 is performed which creates 2 N/2*N matrices using vectorization and these matrices are then multiplied using vectorization to produce the final result. (Tranpose of the second matrix is being done in both the optimization types).
- It can be observed from the plots that while vectorization is not ideal for matrices with smaller sizes (still reasonable), but as the size increases, a speedup of ~10x, ~14.5x, ~20.2x for matrix sizes 4096, 8192, 16384 respectively for Single_Thread_Vector_Optimized-1 case and an speedup of ~7.3x, ~10.5x, ~19.8x for matrix sizes 4096, 8192, 16384 respectively for Single_Thread_Vectorized-1 case can be observed, which clearly shows that eliminating the redundant addition operations is indeed helping improve the overall performance. For 16k matrix size, it can be seen that the speedup of the 2 variants were quite close but in general this was not the case. The results that were obtained when running the 16k were contended with processes that were running from other users at the same time hence the slipup in the speedup incase of 16k.

CPU Cycles


Plot of CPU_CYCLES_mean for Matrix Size 16
Exection Type - Thread Size
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Reference_Data-1
30474.5
35665.2
35820.3
Matrix Size
16
3.1×10^4
3.2×10^4
3.3×10^4
3.4×10^4
3.5×10^4
3.6×10^4
CPU_CYCLES

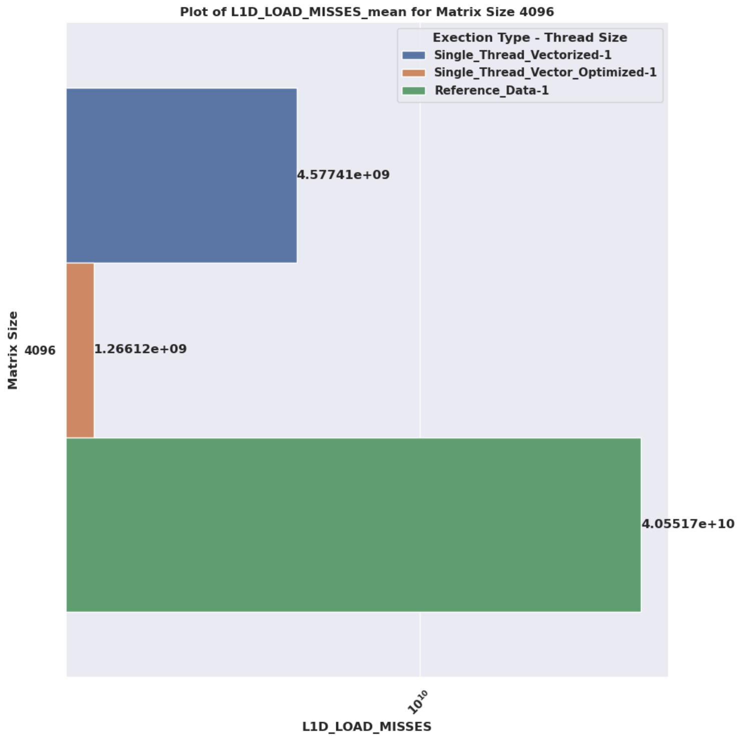

Plot of CPU_CYCLES_mean for Matrix Size 4096
Exection Type - Thread Size
Single_Thread_Vectorized-1
Single_Thread_Vector_Optimized-1
Reference_Data-1
1.50596e+11
1.10419e+11
1.12335e+12
Matrix Size
4096
10^{11}
10^{12}
CPU_CYCLES

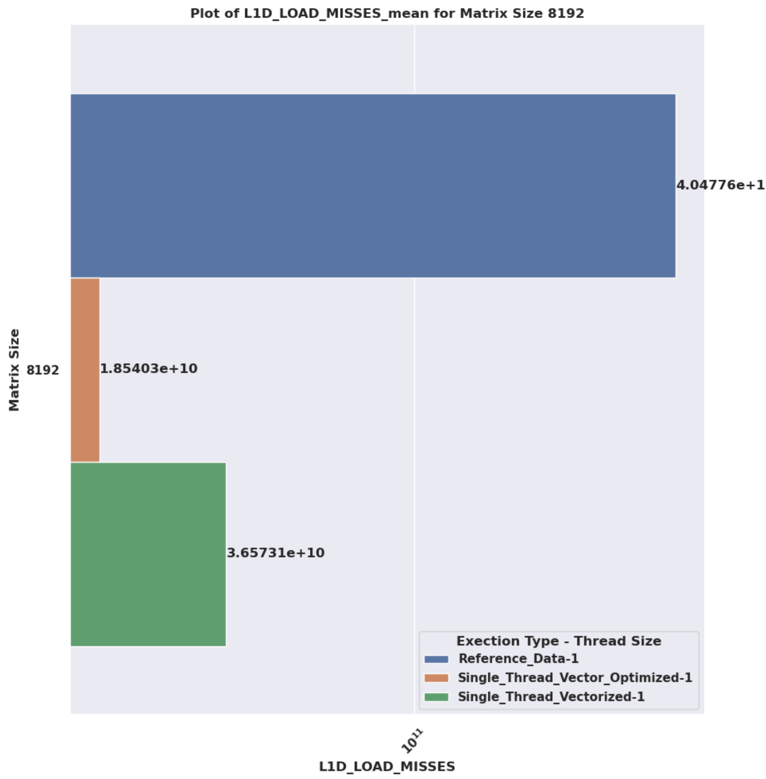

Plot of CPU_CYCLES_mean for Matrix Size 8192
1.29061e+13
8.81306e+11
1.18783e+12
Matrix Size
8192
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
10^{12}
10^{13}
CPU_CYCLES

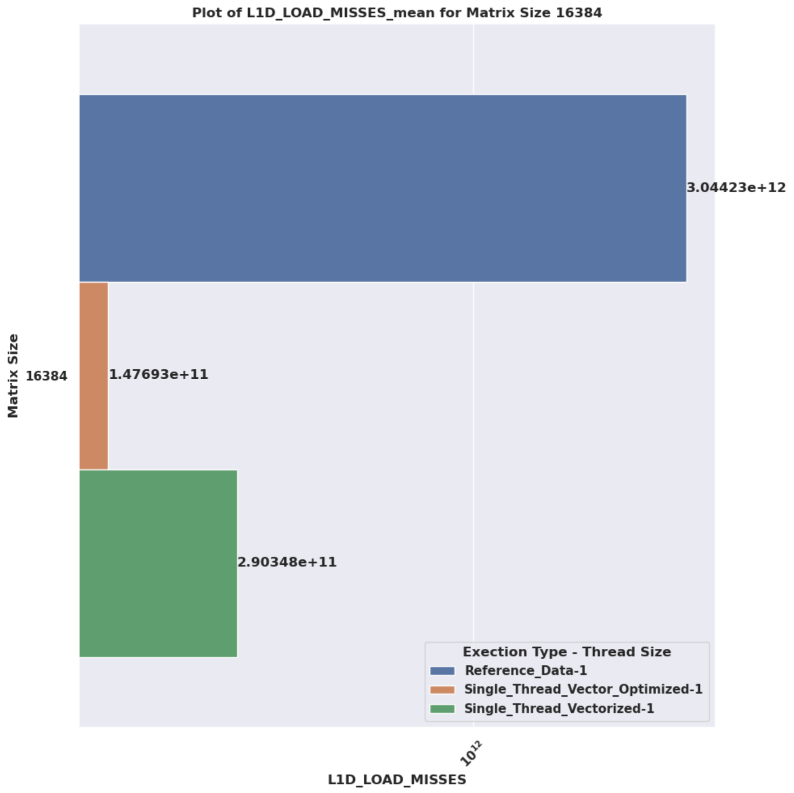

Plot of CPU_CYCLES_mean for Matrix Size 16384
2.04594e+14
7.00292e+12
9.49662e+12
Matrix Size
16384
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
10^{13}
10^{14}
CPU_CYCLES

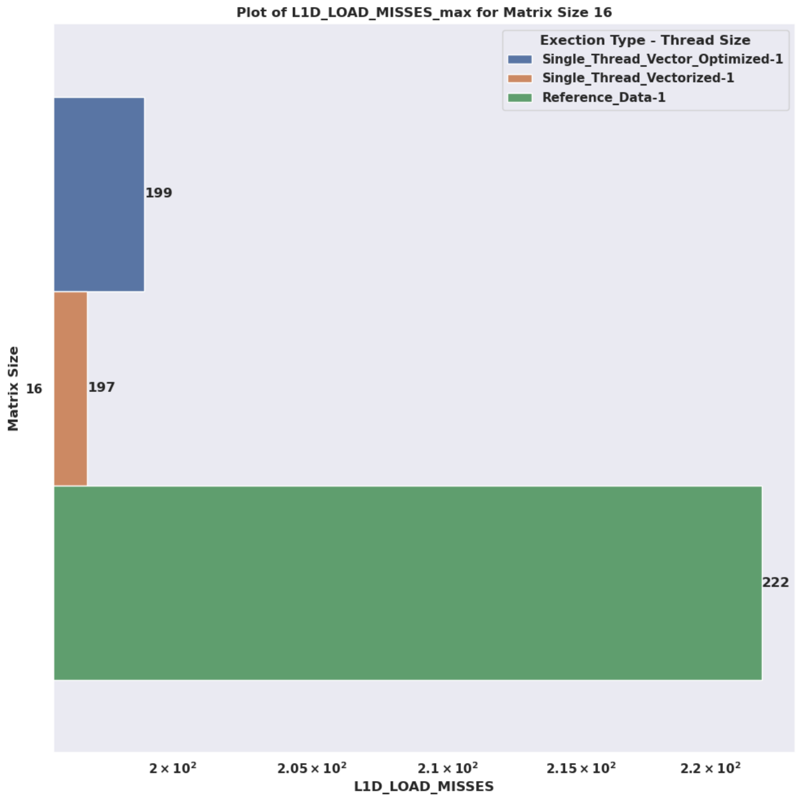
From the above plots it can observed that with the help of vectorization the number of cpu cycles are reduced as its performing operations on 8 integers at once in this case.

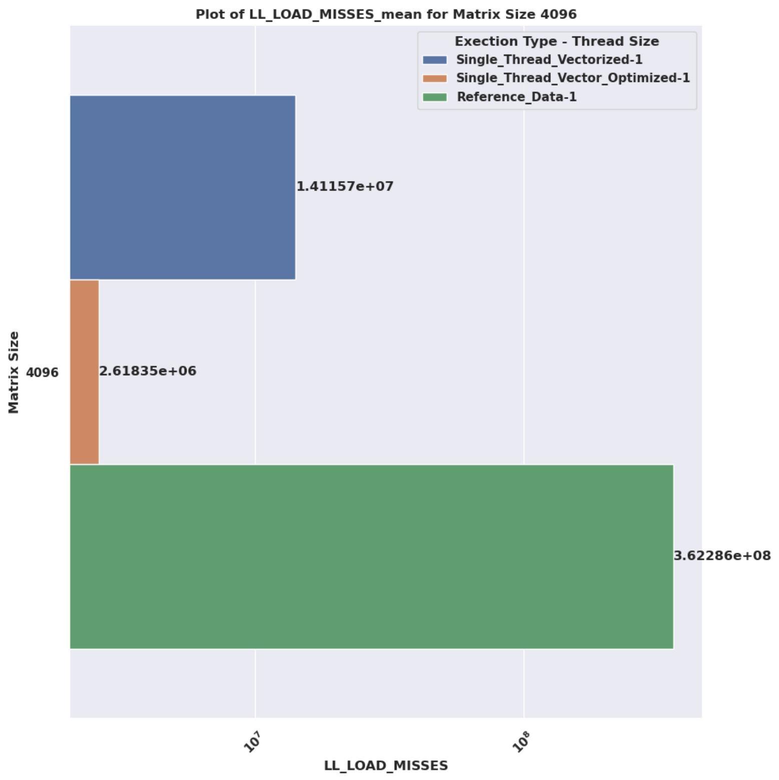
L1 Data and LLC Load misses:

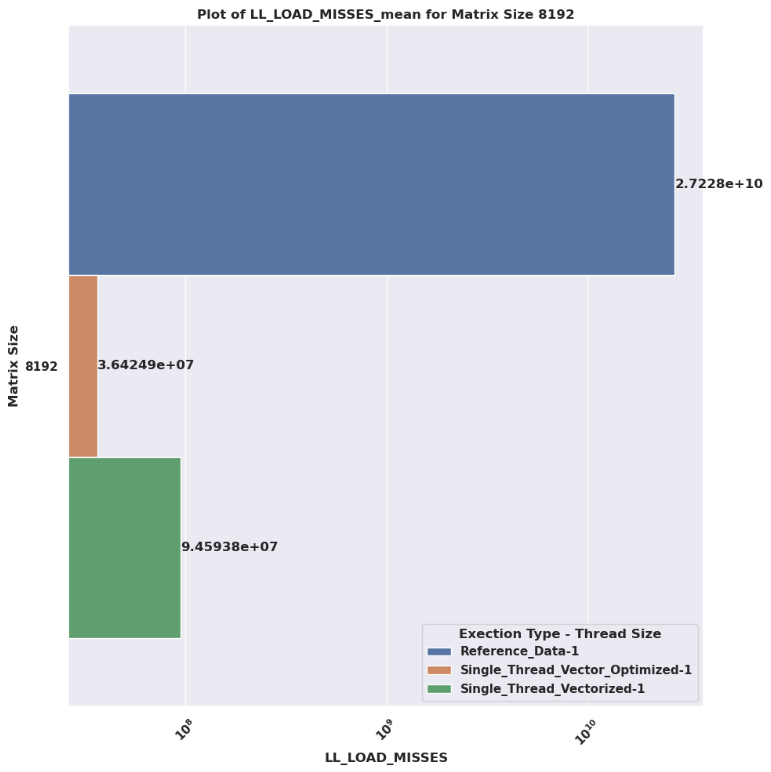

Plot of L1D_LOAD_MISSES_mean for Matrix Size 4096
Exection Type - Thread Size
Single_Thread_Vectorized-1
Single_Thread_Vector_Optimized-1
Reference_Data-1
4.57741e+09
1.26612e+09
4.05517e+10
Matrix Size
4096
10^{10}
L1D_LOAD_MISSES

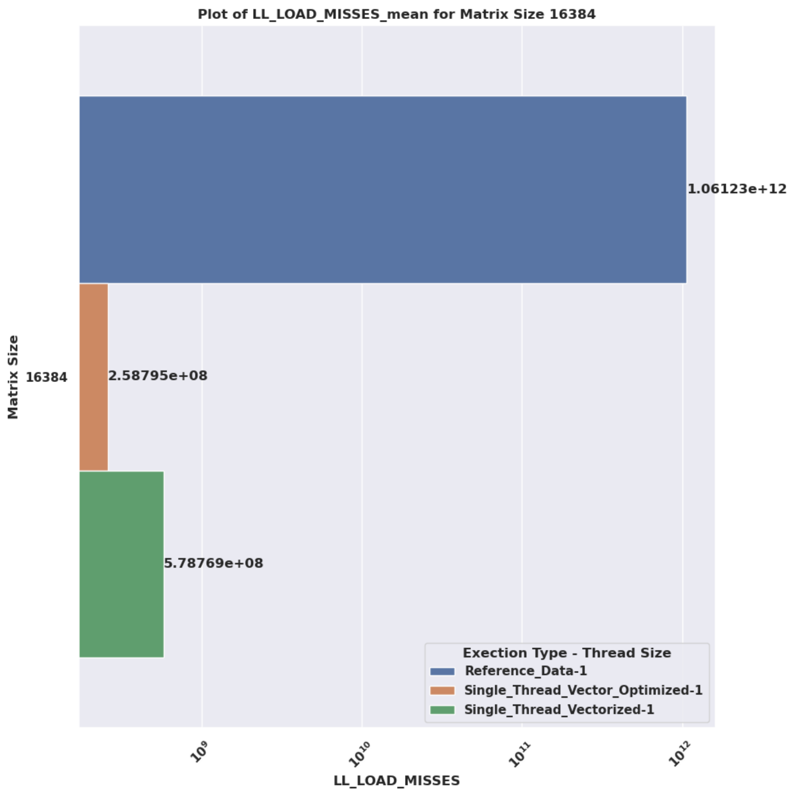

Plot of L1D_LOAD_MISSES_mean for Matrix Size 8192
4.04776e+1
1.85403e+10
3.65731e+10
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Matrix Size
8192
10^{11}
L1D_LOAD_MISSES

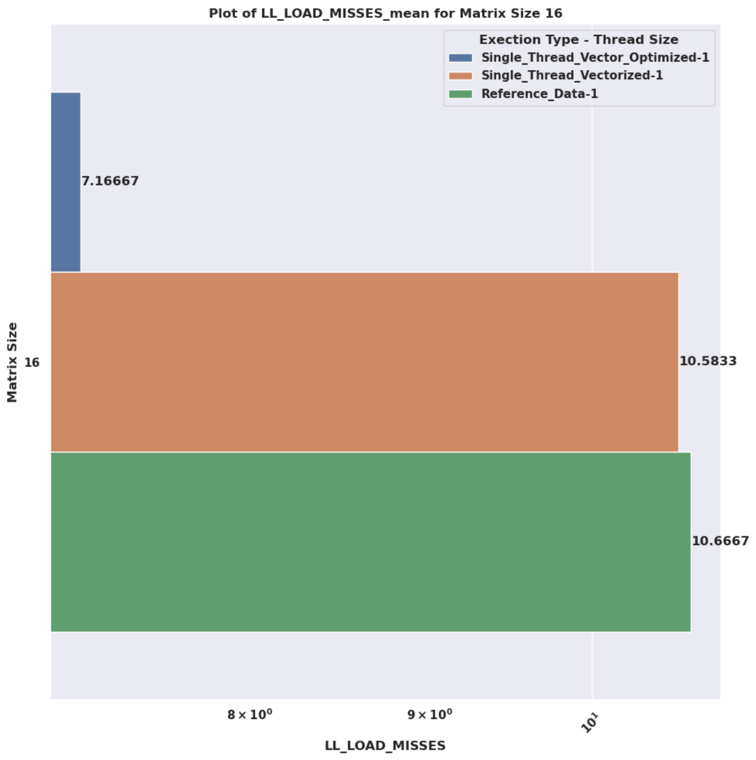

Plot of L1D_LOAD_MISSES_mean for Matrix Size 16384
3.04423e+12
1.47693e+11
2.90348e+11
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Matrix Size
16384
10^{12}
L1D_LOAD_MISSES


Plot of L1D_LOAD_MISSES_max for Matrix Size 16
Exection Type - Thread Size
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Reference_Data-1
199
197
222
Matrix Size
16
2×10^2
2.05×10^2
2.1×10^2
2.15×10^2
2.2×10^2
L1D_LOAD_MISSES


Plot of LL_LOAD_MISSES_mean for Matrix Size 4096
Exection Type - Thread Size
Single_Thread_Vectorized-1
Single_Thread_Vector_Optimized-1
Reference_Data-1
1.41157e+07
2.61835e+06
3.62286e+08
Matrix Size
4096
10^7
10^8
LL_LOAD_MISSES


Plot of LL_LOAD_MISSES_mean for Matrix Size 8192
2.7228e+10
3.64249e+07
9.45938e+07
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Matrix Size
8192
10^8
10^9
10^{10}
LL_LOAD_MISSES


Plot of LL_LOAD_MISSES_mean for Matrix Size 16384
1.06123e+12
2.58795e+08
5.78769e+08
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Matrix Size
16384
10^9
10^{10}
10^{11}
10^{12}
LL_LOAD_MISSES


Plot of LL_LOAD_MISSES_mean for Matrix Size 16
Exection Type - Thread Size
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Reference_Data-1
7.16667
10.5833
10.6667
Matrix Size
16
8×10^0
9×10^0
10^1
LL_LOAD_MISSES

The above plots show the load misses in L1 data and LLC caches. Compared to the misses in smaller sized matrices which are relatively closer, for larger sized matrices it can be clearly seen that the number of misses are not even relatively close because of proper use of locality by using both transpose and vectorization.

Concluding remarks :-

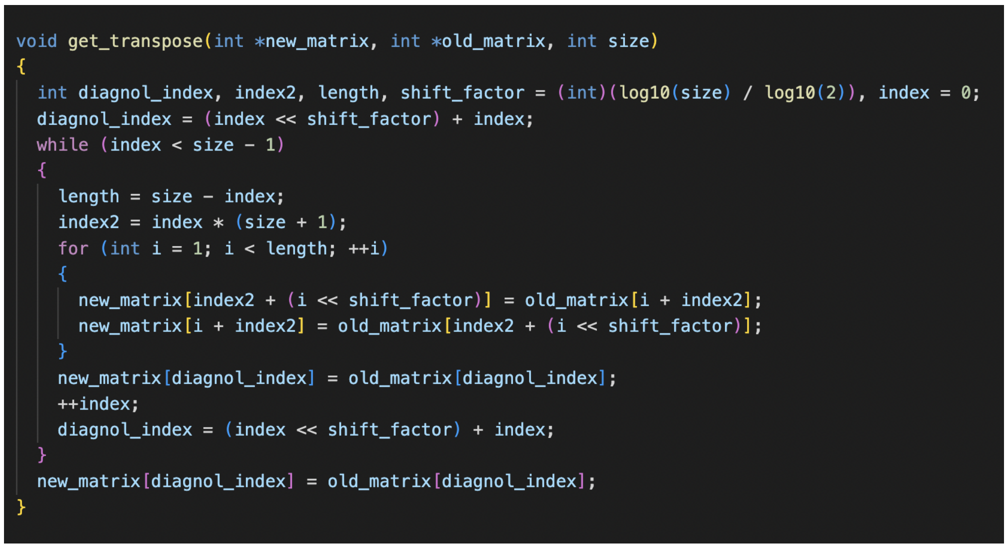
In single threaded version, to achieve as much speedup/performance as possible its better to pre-process few things along with vectorization which helps in performing operations on multiple elements at once.

Question-1 Part-A Multi-Thread :

Implementation Details :-

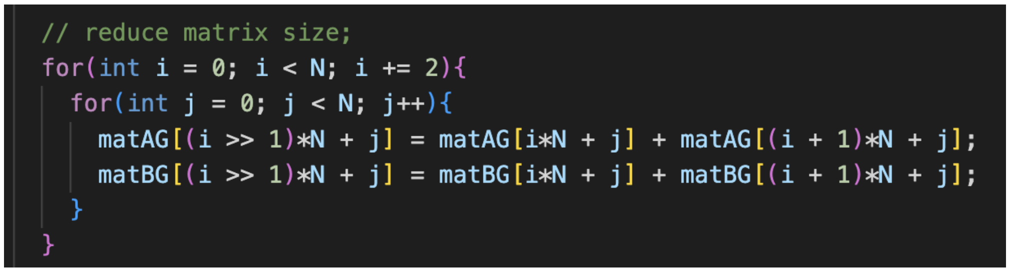
- Implemented multi-threading using pthreads -
    - Thread Counts ranging from {32, 64, 128, 256}
    - Matrix sizes {16, 4096, 8192, 16384}.

- Implementation:(Optimizations applied in multi-thread as similar to that of the optimization that are applied in single-thread part)
    - Transposed the second matrix (MatB) to convert into row major order to exploit better locality.



```
void get_transpose(int *new_matrix, int *old_matrix, int size)
{
  int diagnol_index, index2, length, shift_factor = (int)(log10(size) / log10(2)), index = 0;
  diagnol_index = (index << shift_factor) + index;
  while (index < size - 1)
  {
    length = size - index;
    index2 = index * (size + 1);
    for (int i = 1; i < length; ++i)
    {
      new_matrix[index2 + (i << shift_factor)] = old_matrix[i + index2];
      new_matrix[i + index2] = old_matrix[index2 + (i << shift_factor)];
    }
    new_matrix[diagnol_index] = old_matrix[diagnol_index];
    ++index;
    diagnol_index = (index << shift_factor) + index;
  }
  new_matrix[diagnol_index] = old_matrix[diagnol_index];
}
```

    - Below equation is used to reduce the total number of arithmetic operations:
        - {Row(i) + row (i + 1)} x {Col(i) + Col(i + 1)}



```
// reduce matrix size;
for(int i = 0; i < N; i += 2){
  for(int j = 0; j < N; j++){
    matAG[(i >> 1)*N + j] = matAG[i*N + j] + matAG[(i + 1)*N + j];
    matBG[(i >> 1)*N + j] = matBG[i*N + j] + matBG[(i + 1)*N + j];
  }
}
```

    - Pre-processed the matrix as above to add two consecutive rows and columns into one,which effectively reduced the matrix sizes to (N / 2) x N and N x (N / 2) respectively

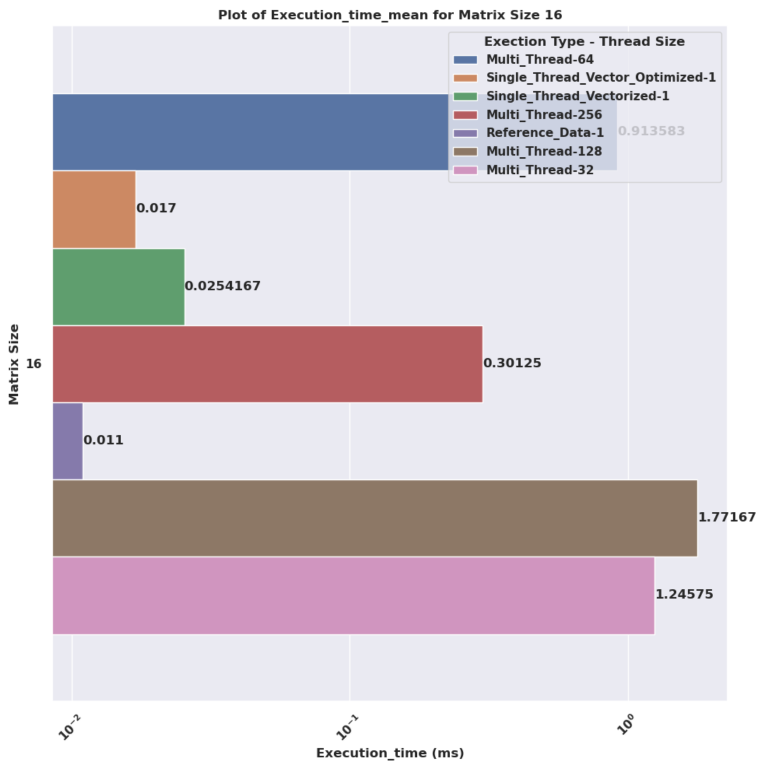
- Sub-divided the matrix multiplication operation as per the -
    - load factor(k) :- **(Input Matrix A rows / # threads)**
    - each thread will be performing **(k x (N / 2)) matrix multiplications**

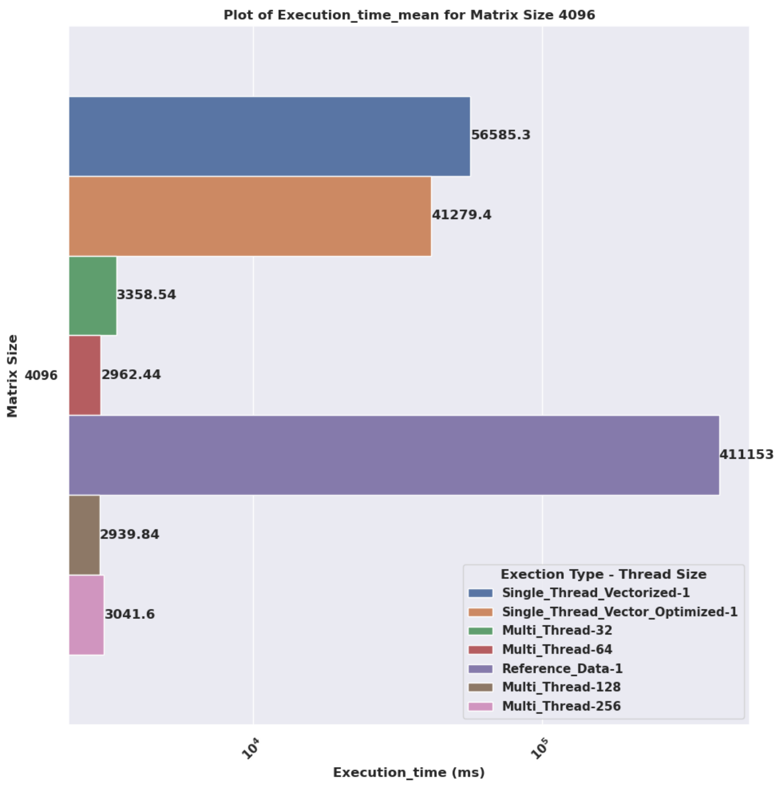
- We have keep the input matrices global for better performance, we tried implementing a version where for each pthread we sent out the sub-matrix upon which it would operate but yielded results even worse than single-thread optimizations.

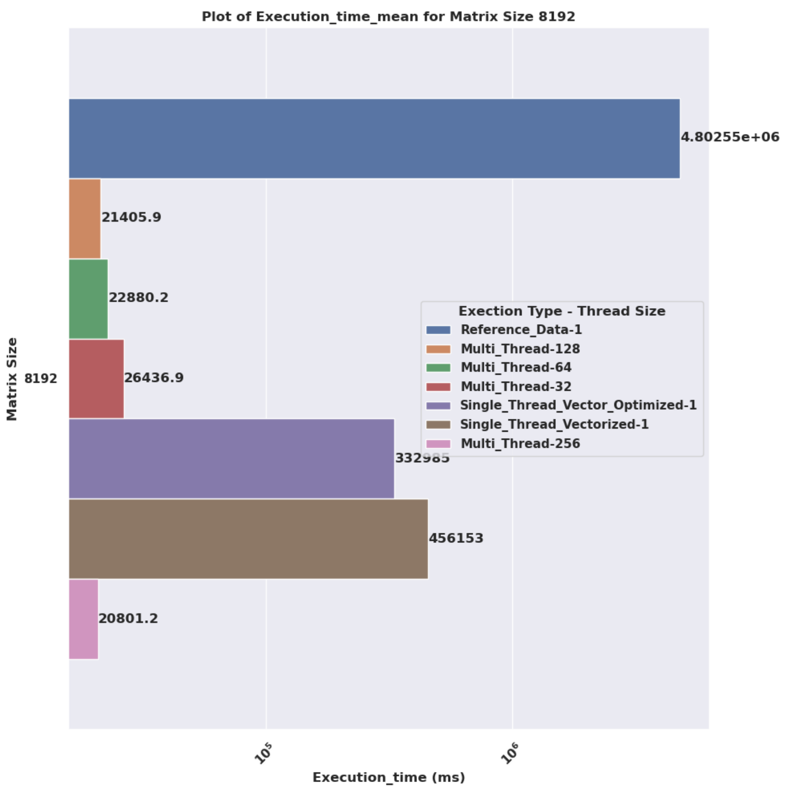
Performance Report:

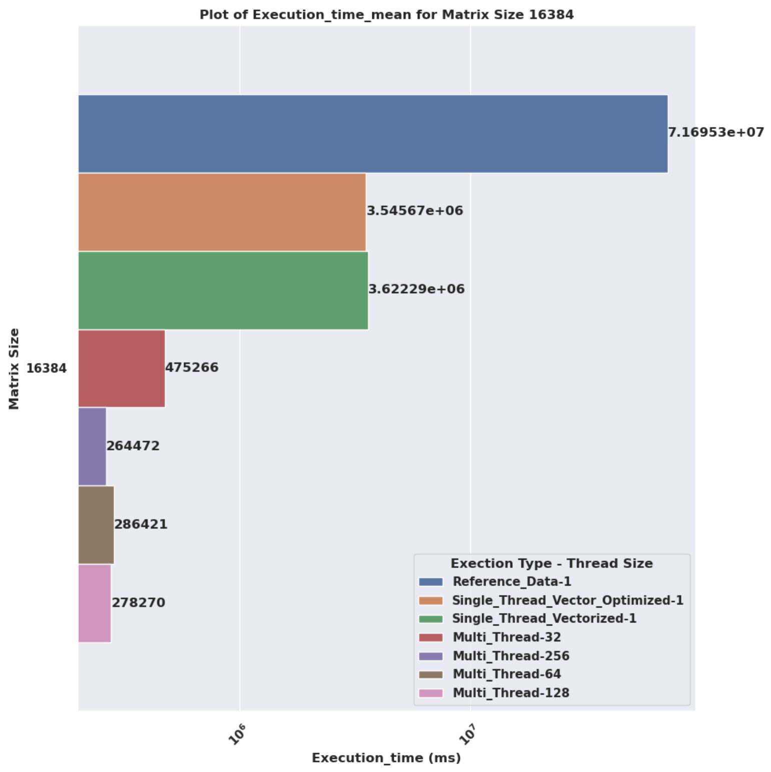
- All the results have been averaged over 3 runs for all the combinations of matrix and thread size.

Execution Times









- The above 4 plots show mean execution time for all the matrix sizes along with 4 thread combinations.

- We have compared it with reference time and single thread optimizations.

- From the above plots we can infer that as we gradually increase the thread size from 32 upto 256 we see a gradual decrease in execution times, and thread size 128 yields the best performance.

- We sampled all the params for thread size greater than 256, but either they were giving the same improvement and for larger thread sizes the run times were very high, which is justified as creation of pthreads can add a lot of overhead.

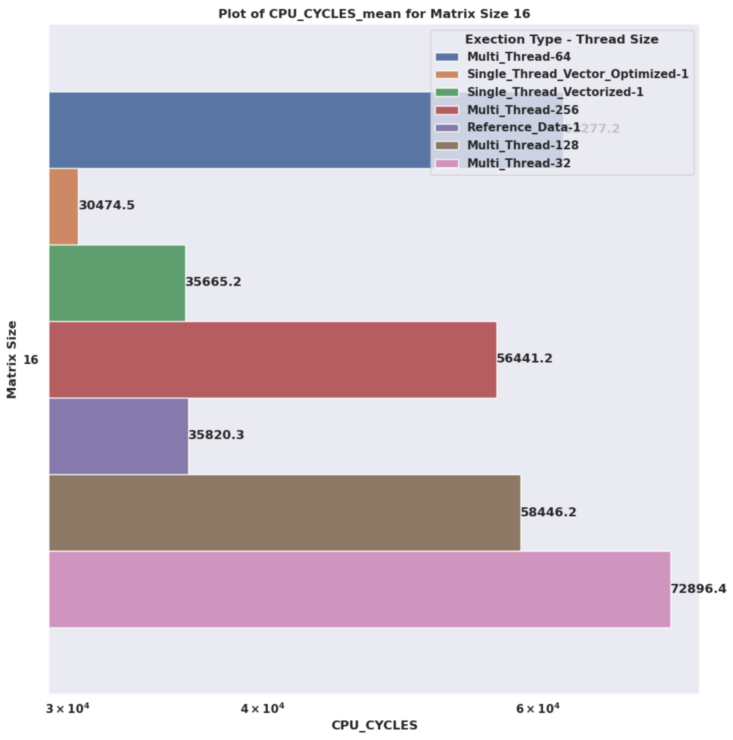
- The threads are able to exploit better parallelism as we have divided the tasks among them in such a way that each thread will operate independently. MatB will shared across all, but since it’s a read access there will be no data dependency conflicts.

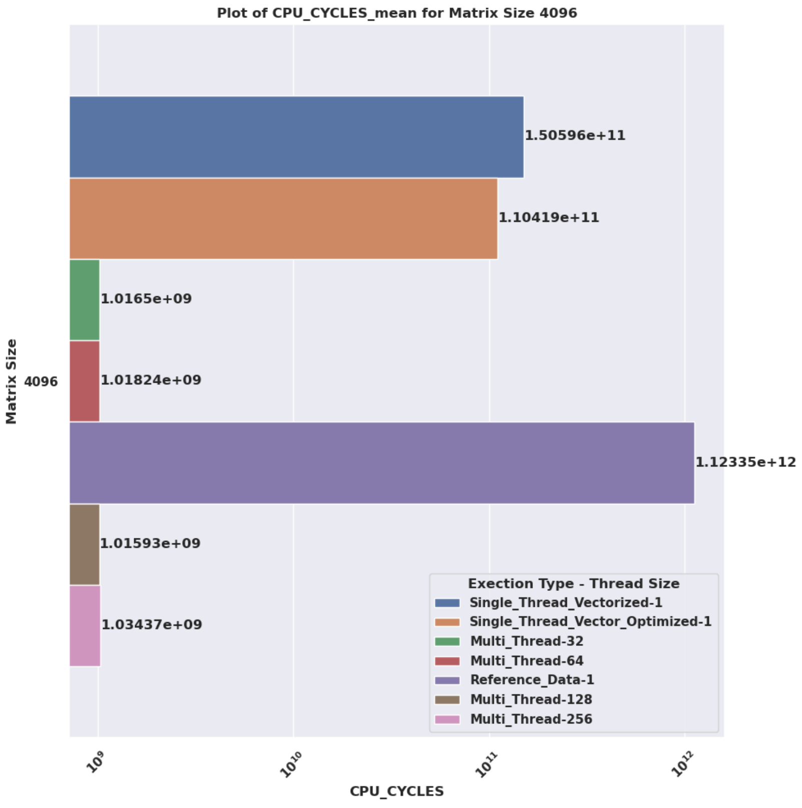
- As we can see from the plots that multi-threading is not ideal for smaller matrices because of pthread overhead, causing it to perform even worse than base line execution.

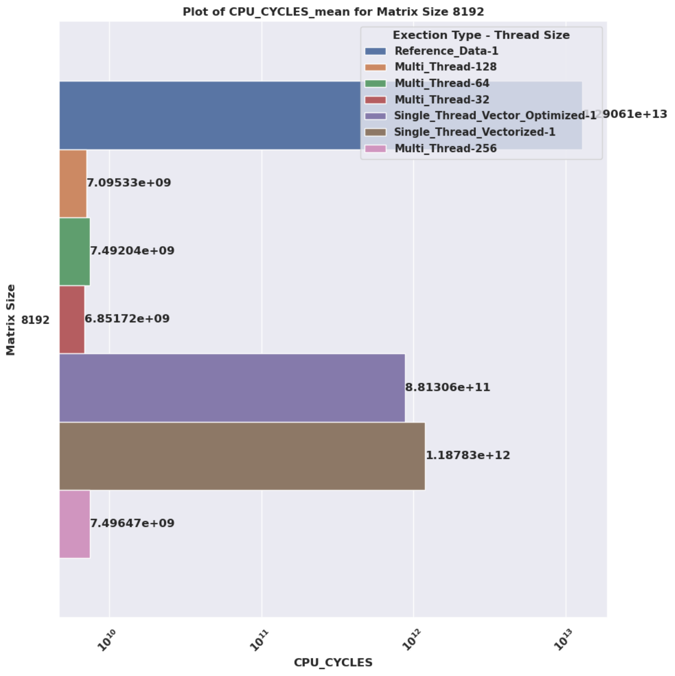
- Overall we see ~140x, ~224x, ~270x improvements in execution time over baseline and ~14x, ~15x, ~13x improvements in execution time over single-thread for matrices [4096, 8192, 16384]. We can infer that for larger matrix sizes the improvement using multi-threading is significant.

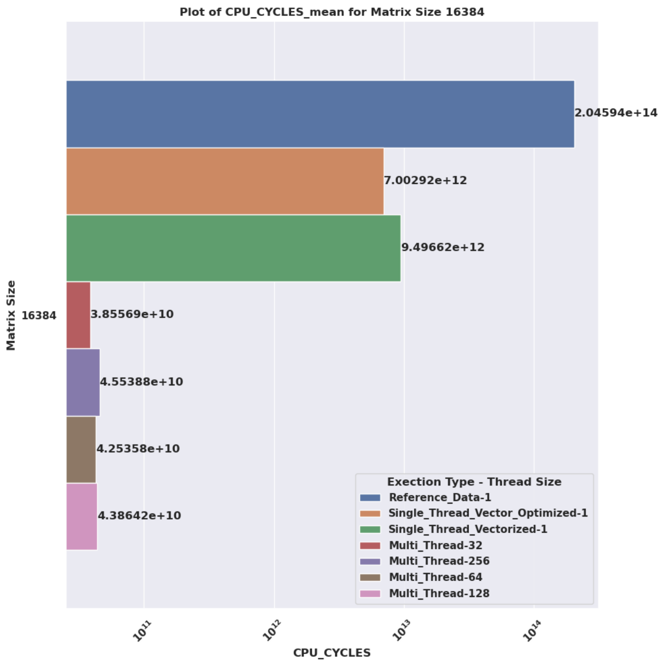
NOTE: While running the execution times for 16k, in some cases we found 540x improvement over baseline,

CPU cycles:



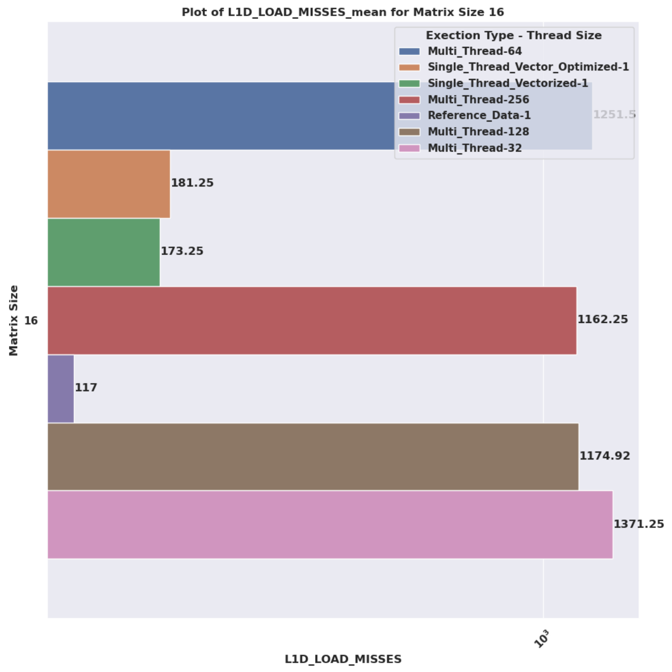


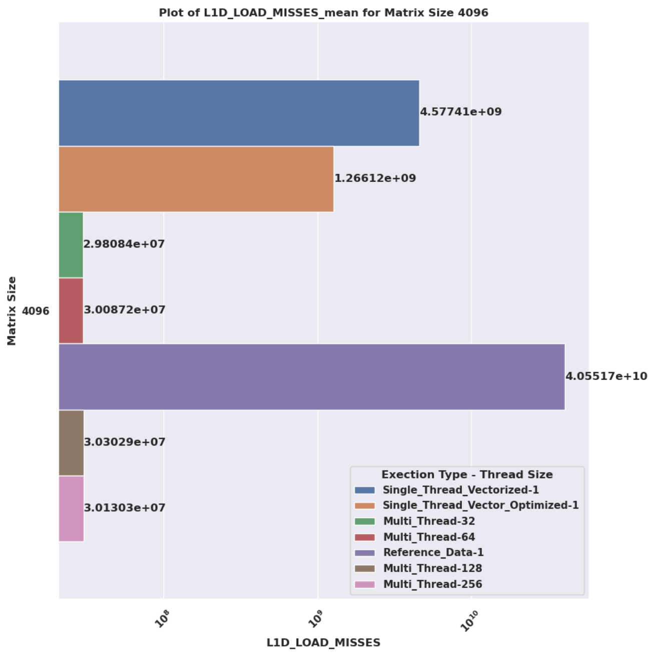


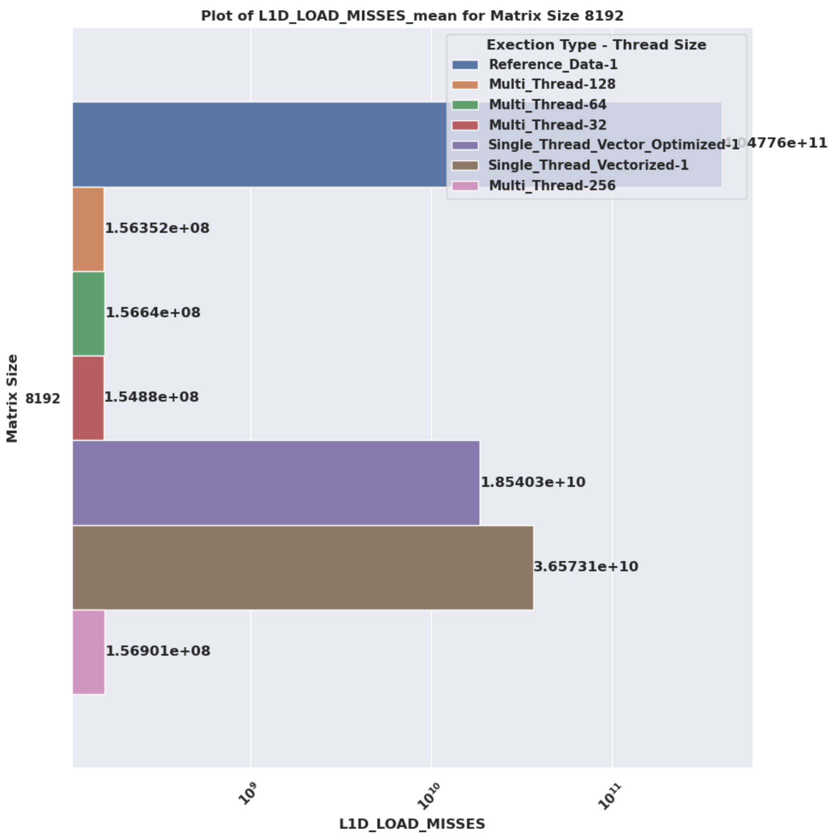


We can clearly see that the number of CPU cycles have substantially reduced in multi-thread variant as compared to others, which justifies the improvement in execution time in the previous plots.

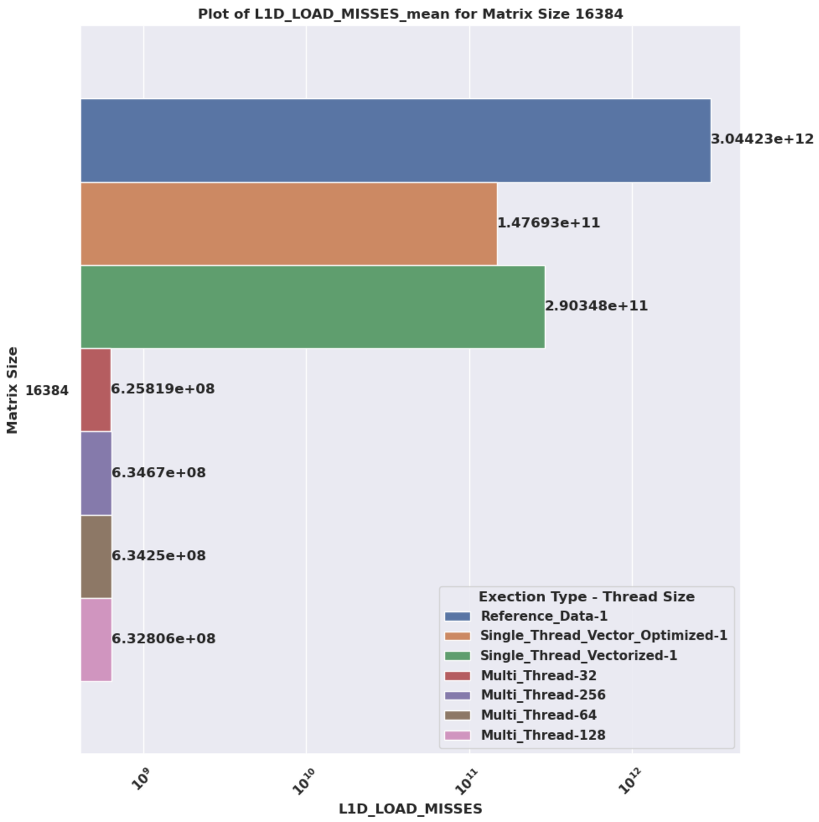
L1 Data and LLC Load Misses:



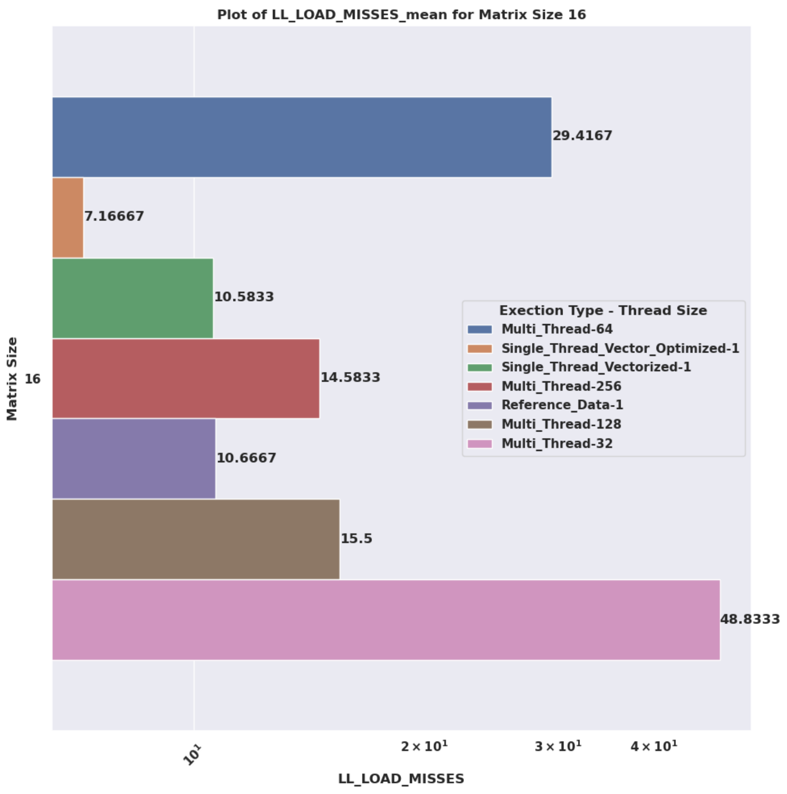


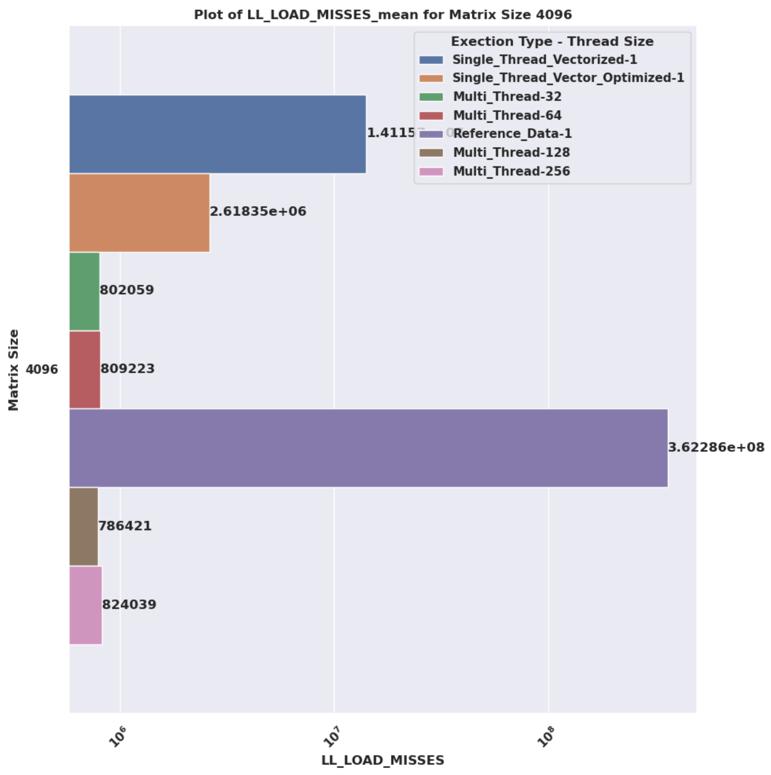


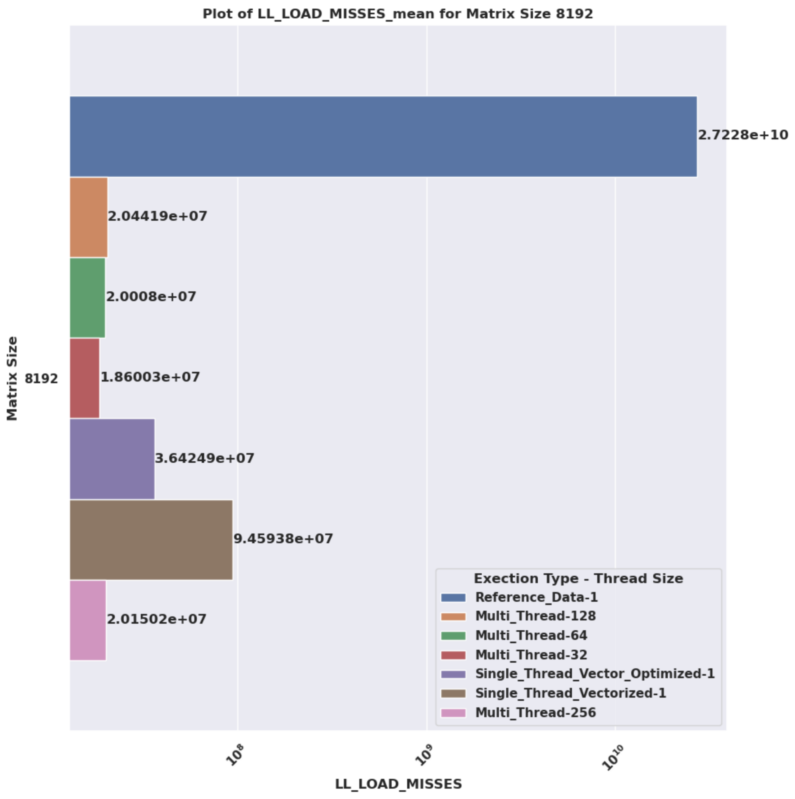
Plot of L1D_LOAD_MISSES_mean for Matrix Size 8192
Exection Type - Thread Size
Reference_Data-1
Multi_Thread-128
Multi_Thread-64
Multi_Thread-32
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Multi_Thread-256
4776e+11
1.56352e+08
1.5664e+08
1.5488e+08
1.85403e+10
3.65731e+10
1.56901e+08
Matrix Size
8192
10^9
10^{10}
10^{11}
L1D_LOAD_MISSES

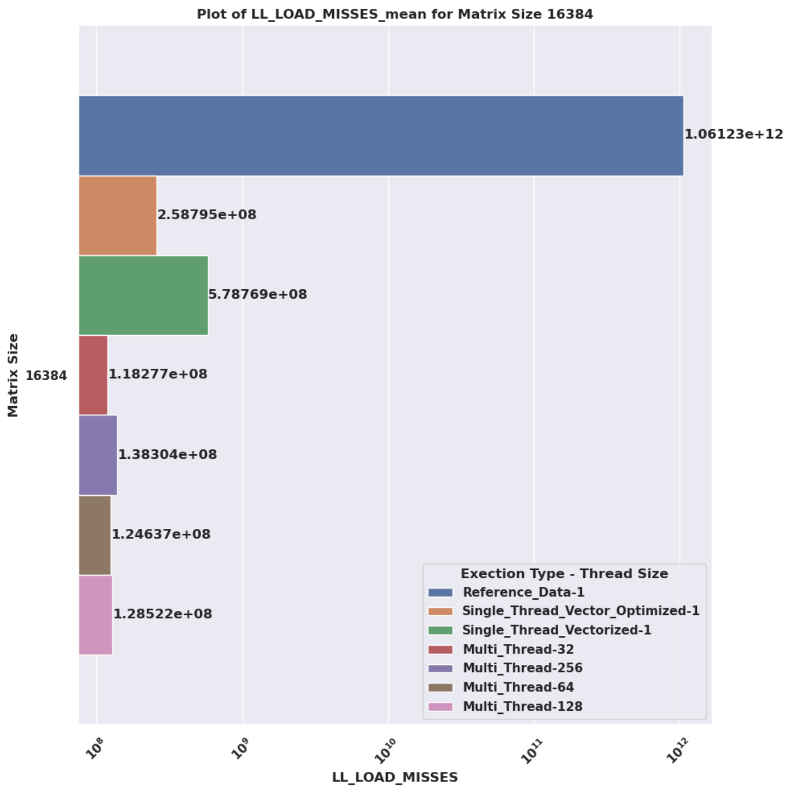


Plot of L1D_LOAD_MISSES_mean for Matrix Size 16384
3.04423e+12
1.47693e+11
2.90348e+11
6.25819e+08
6.3467e+08
6.3425e+08
6.32806e+08
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Multi_Thread-32
Multi_Thread-256
Multi_Thread-64
Multi_Thread-128
Matrix Size
16384
10^9
10^{10}
10^{11}
10^{12}
L1D_LOAD_MISSES



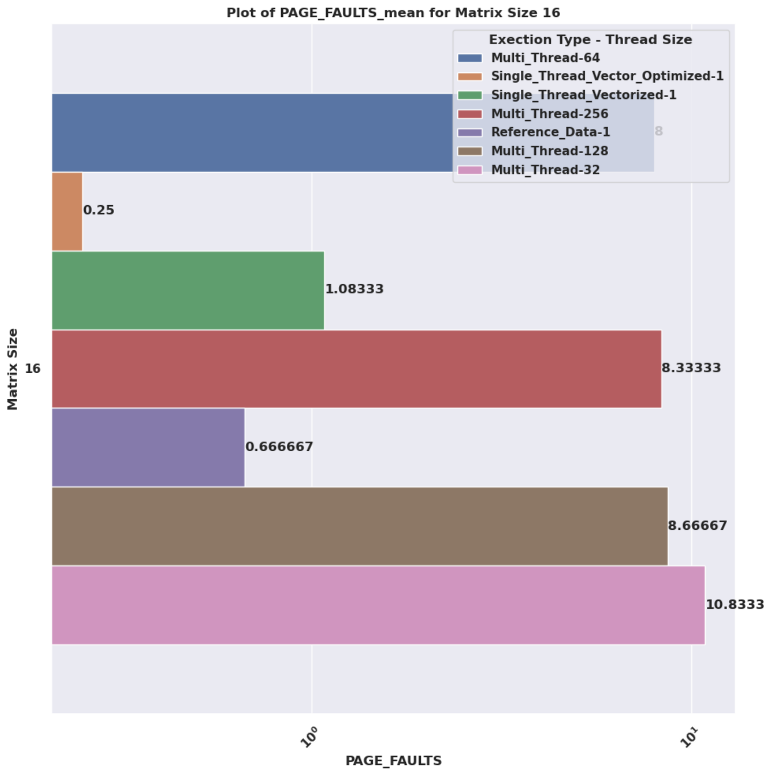


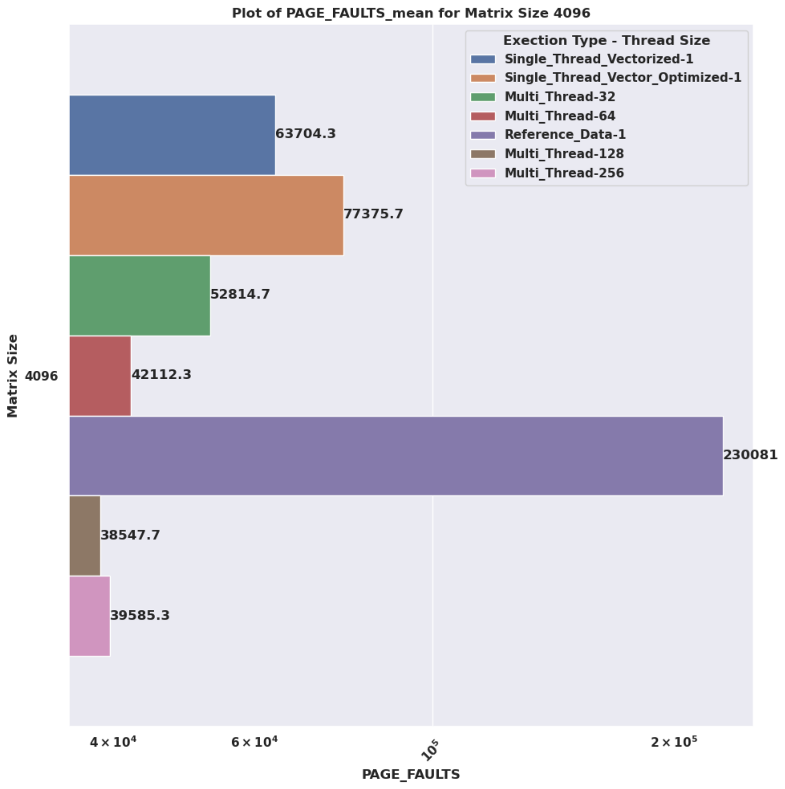


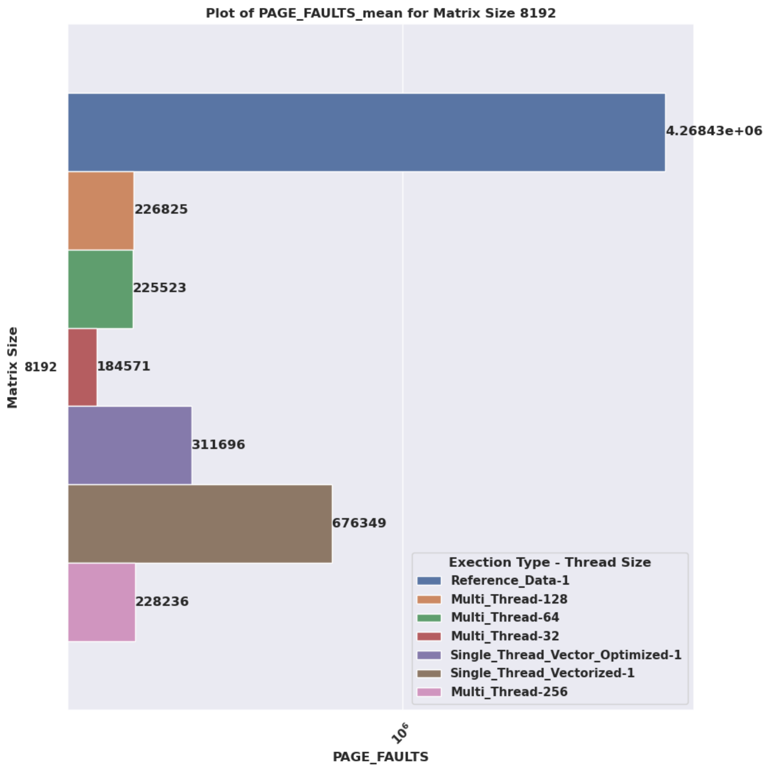


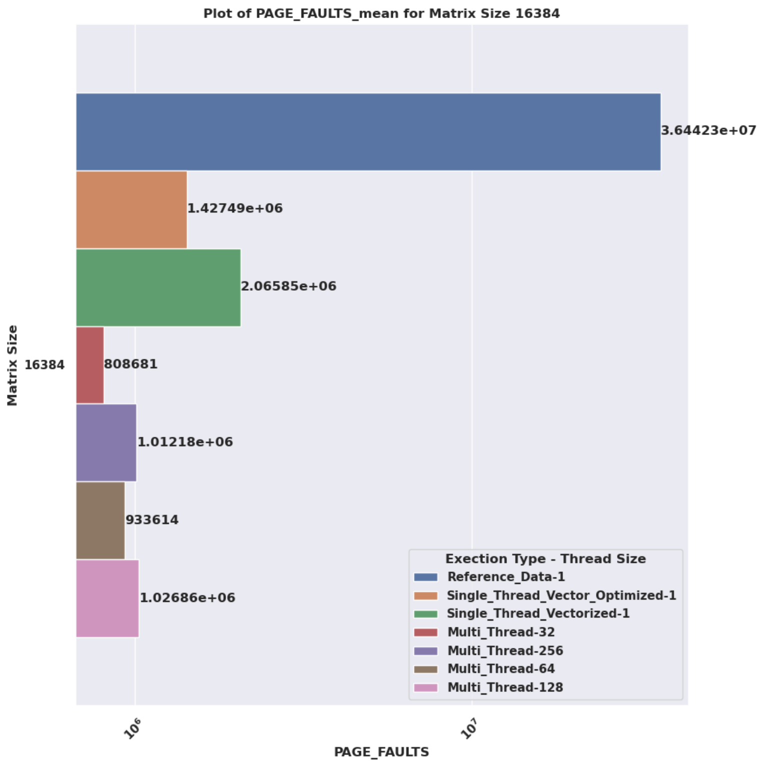
- From above plots we can see a clear reduction in L1D and LLC load misses for multi-thread variant over baseline as well as single thread. This explains multi-threading is exploiting the locality better in terms of single-thread optimized version as multiple threads are operating on independent sets of data which are already pre-loaded in cache resulting in more cache hits.

Page Faults:


Plot of PAGE_FAULTS_mean for Matrix Size 16
Exection Type - Thread Size
Multi_Thread-64
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Multi_Thread-256
Reference_Data-1
Multi_Thread-128
Multi_Thread-32
0.25
1.08333
8.33333
0.666667
8.66667
10.8333
Matrix Size
16
PAGE_FAULTS


Plot of PAGE_FAULTS_mean for Matrix Size 4096
Exection Type - Thread Size
Single_Thread_Vectorized-1
Single_Thread_Vector_Optimized-1
Multi_Thread-32
Multi_Thread-64
Reference_Data-1
Multi_Thread-128
Multi_Thread-256
63704.3
77375.7
52814.7
42112.3
230081
38547.7
39585.3
Matrix Size
4096
PAGE_FAULTS


Plot of PAGE_FAULTS_mean for Matrix Size 8192
Exection Type - Thread Size
Reference_Data-1
Multi_Thread-128
Multi_Thread-64
Multi_Thread-32
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Multi_Thread-256
4.26843e+06
226825
225523
184571
311696
676349
228236
Matrix Size
8192
PAGE_FAULTS


Plot of PAGE_FAULTS_mean for Matrix Size 16384
Exection Type - Thread Size
Reference_Data-1
Single_Thread_Vector_Optimized-1
Single_Thread_Vectorized-1
Multi_Thread-32
Multi_Thread-256
Multi_Thread-64
Multi_Thread-128
3.64423e+07
1.42749e+06
2.06585e+06
808681
1.01218e+06
933614
1.02686e+06
Matrix Size
16384
PAGE_FAULTS

- Page-Faults are lower for multi-thread variant than baseline as well as single-thread optimizations.

- Normally, if a thread takes a page fault and must wait for the page to be read from disk, the operating system runs a different thread while the I/O is occurring, this also adds up to improved execution time and hides page-fault latency in case of multi-threading.

- But in our case the overall page-faults are anyways lower, this is because multiple threads are operating on different parts of matrix atonce, which leads to lesser page-eviction to disk from current process as they are least recently used, unless some-other competing high priority process comes into play.

Concluding remarks :-

- In multi-threading we can clearly see a significant improvement in execution times, indicating that the workload we are evaluating has high parallelism.
- Parallelism can be exploited upto a certain extent as spawning very high number of threads(>256) will add to pthread management overhead.
- We are also limited by the logical cores which if not taken care of can yield poor results as most of the time threads will be competing with each other for resources.

Question-1 Part-B :

- Configurations:
    - NVIDIA Corporation TU104 [GeForce RTX 2080 Rev. A] (rev a1)
    - **nsight compute** is used to profile the performance counters.

1. Optimizations that were used in the previous parts are borrowed and the same are applied in this part aswell.

2. Transposing of second matrix is done in GPU only if it exceeds as certain size (32 in out case) otherwise the transpose is done in CPU itself as there is no point in allocating cuda memory, copying matrix and launching a kernel for smaller sized matrices when the same can be achieved in the CPU itself at a similar time.
    a. Transposing matrix in GPU is done only when matrix size is greater than 32. For the implementation part when coming to sizes greater than 32, its implemented using Coalesced Transpose Via Shared Memory. (“CUDA GPU Tranpose”, n.d.)

3. 2D blocks per grids and 2D threads per blocks are being used for every different matrix sizes for both addition and multiplication of matrices as it helps in better utilization of the overall resources incase the resources are not getting contended.

4. For a matrix of size N * N, which creates a resultant matrix of size N/2 * N/2, a total of N/2 * N threads for addition and N/2 * N/2 incase of multiplication are being spawned. so every element of resultant matrix can compute its own value without waiting for any other values. Thus for a matrix size 4096 * 4096, for multiplication phase there will be a total of 4914304 threads.

Results :-

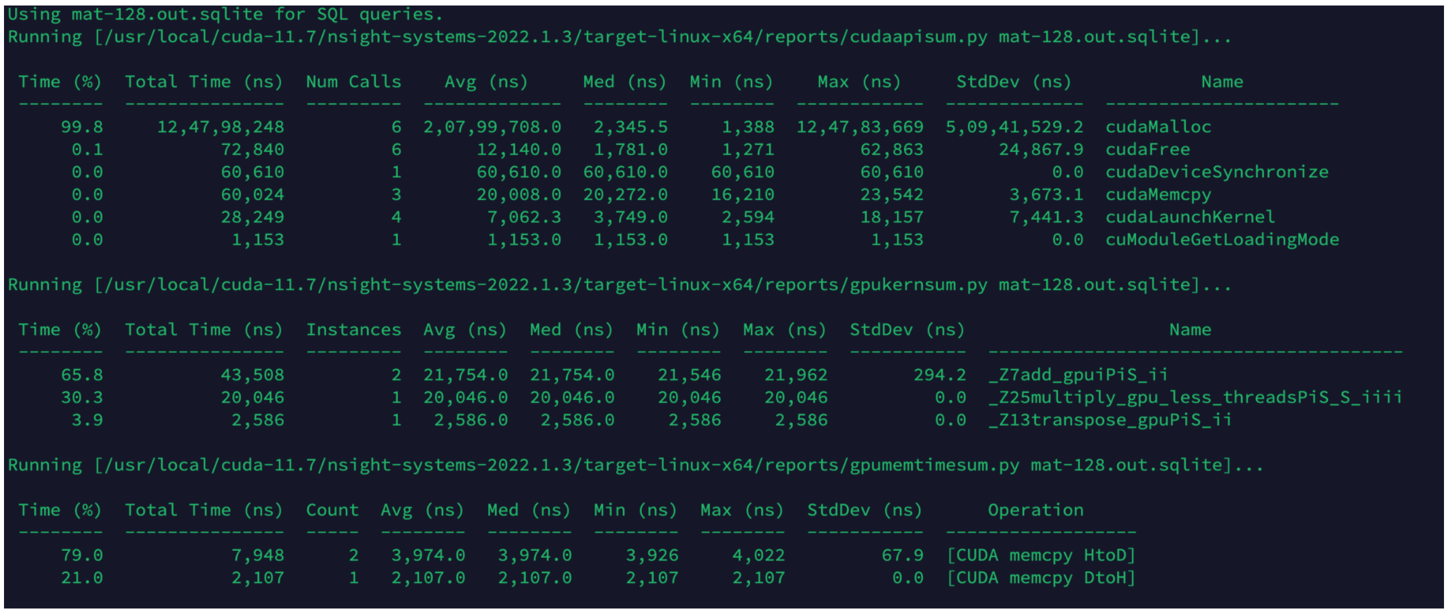
- Nsight cli provides performance counters on a per-kernel basis
- We have used a total of 4 kernel for our implementations and hence in the plotted graphs we will be splitting up the parameters for each kernel.

Execution Times:

We have used nsys for profiling timings and modeled three different times as below:

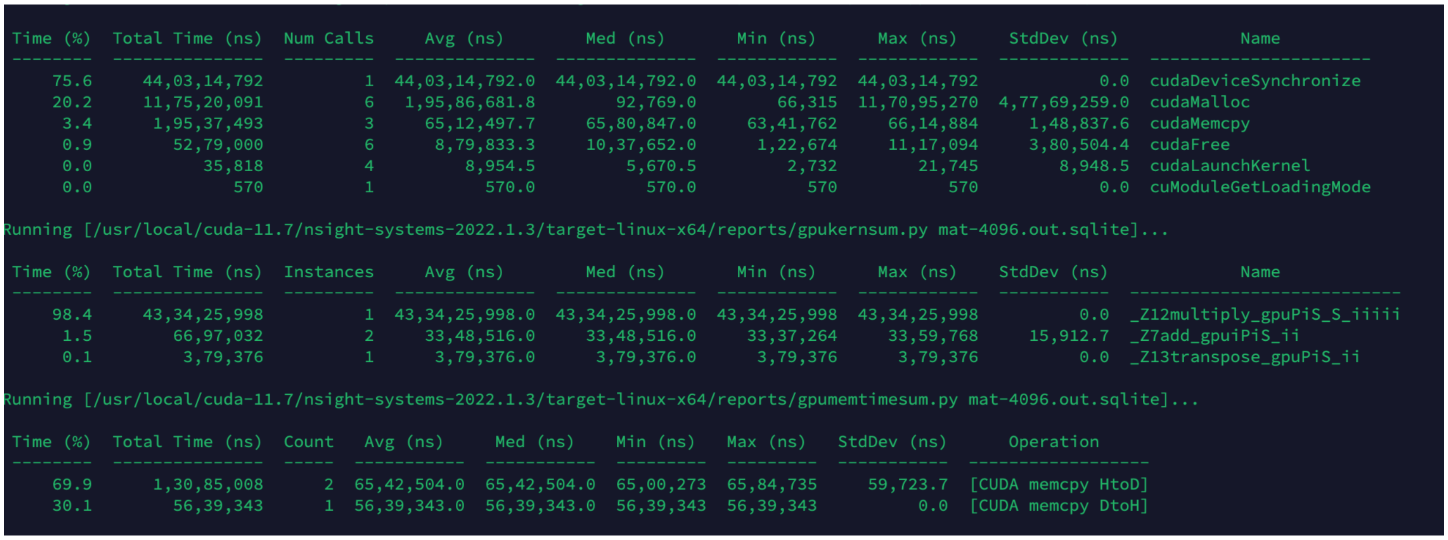
```
$ nsys stats -r cudaapisum,gpukernsum,gpumemtimesum
```

MAT 128



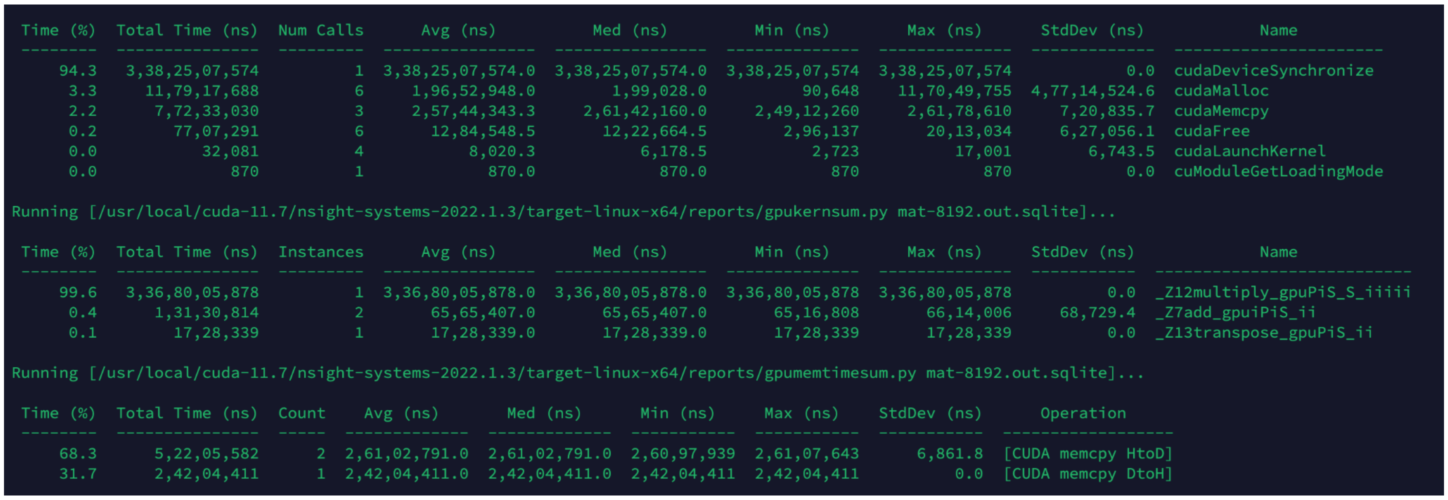
```
Using mat-128.out.sqlite for SQL queries.
Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/cudaapisum.py mat-128.out.sqlite]...

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)  Min (ns)    Max (ns)     StdDev (ns)           Name
 --------  ---------------  ---------  -------------  --------  --------  ------------  -------------  ----------------------
     99.8     12,47,98,248          6  2,07,99,708.0   2,345.5     1,388  12,47,83,669  5,09,41,529.2  cudaMalloc
      0.1           72,840          6       12,140.0   1,781.0     1,271        62,863       24,867.9  cudaFree
      0.0           60,610          1       60,610.0  60,610.0    60,610        60,610            0.0  cudaDeviceSynchronize
      0.0           60,024          3       20,008.0  20,272.0    16,210        23,542        3,673.1  cudaMemcpy
      0.0           28,249          4        7,062.3   3,749.0     2,594        18,157        7,441.3  cudaLaunchKernel
      0.0            1,153          1        1,153.0   1,153.0     1,153         1,153            0.0  cuModuleGetLoadingMode

Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpukernsum.py mat-128.out.sqlite]...

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                  Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  --------------------------------------
     65.8           43,508          2  21,754.0  21,754.0    21,546    21,962        294.2  _Z7add_gpuiPiS_ii
     30.3           20,046          1  20,046.0  20,046.0    20,046    20,046          0.0  _Z25multiply_gpu_less_threadsPiS_S_iiii
      3.9            2,586          1   2,586.0   2,586.0     2,586     2,586          0.0  _Z13transpose_gpuPiS_ii

Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpumemtimesum.py mat-128.out.sqlite]...

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)      Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ------------------
     79.0            7,948      2   3,974.0   3,974.0     3,926     4,022         67.9  [CUDA memcpy HtoD]
     21.0            2,107      1   2,107.0   2,107.0     2,107     2,107          0.0  [CUDA memcpy DtoH]
```

MAT 4096:



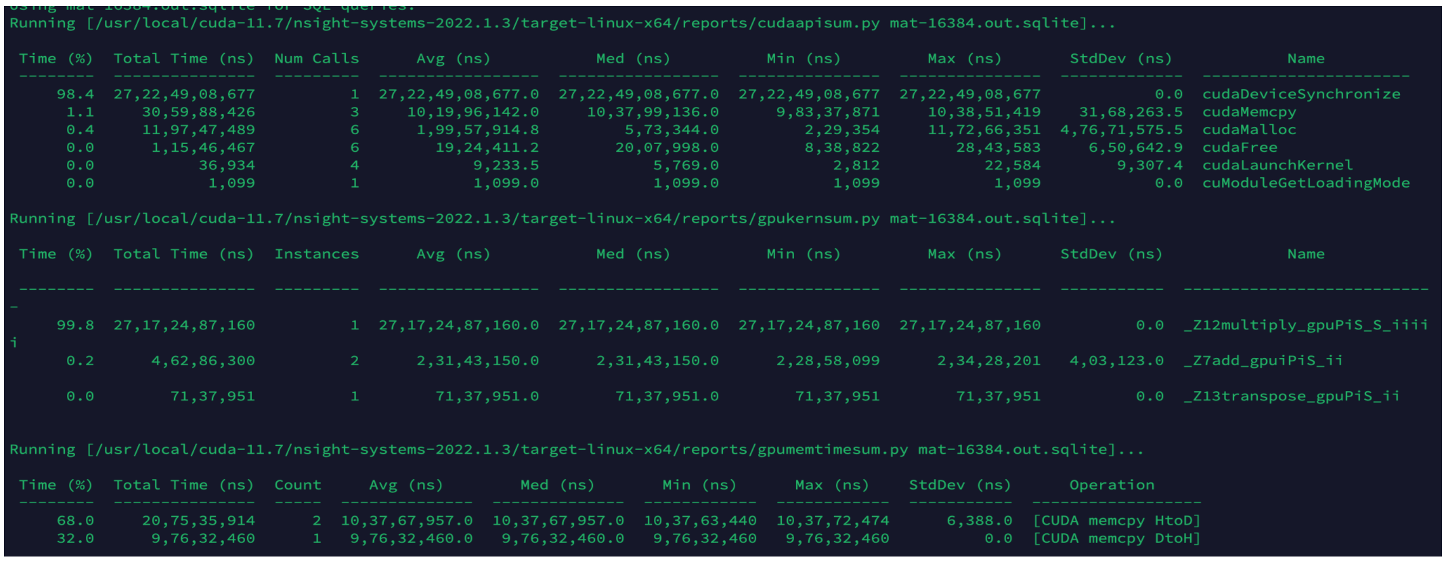
```
 Time (%)  Total Time (ns)  Num Calls     Avg (ns)        Med (ns)       Min (ns)      Max (ns)     StdDev (ns)           Name
 --------  ---------------  ---------  --------------  --------------  ------------  ------------  ------------  ----------------------
     75.6     44,03,14,792          1  44,03,14,792.0  44,03,14,792.0  44,03,14,792  44,03,14,792           0.0  cudaDeviceSynchronize
     20.2     11,75,20,091          6   1,95,86,681.8        92,769.0        66,315  11,70,95,270  4,77,69,259.0  cudaMalloc
      3.4      1,95,37,493          3     65,12,497.7     65,80,847.0     63,41,762     66,14,884    1,48,837.6  cudaMemcpy
      0.9        52,79,000          6      8,79,833.3     10,37,652.0      1,22,674     11,17,094    3,80,504.4  cudaFree
      0.0           35,818          4         8,954.5         5,670.5         2,732        21,745       8,948.5  cudaLaunchKernel
      0.0              570          1           570.0           570.0           570           570           0.0  cuModuleGetLoadingMode

Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpukernsum.py mat-4096.out.sqlite]...

 Time (%)  Total Time (ns)  Instances     Avg (ns)        Med (ns)       Min (ns)      Max (ns)     StdDev (ns)            Name
 --------  ---------------  ---------  --------------  --------------  ------------  ------------  -----------  --------------------------
     98.4     43,34,25,998          1  43,34,25,998.0  43,34,25,998.0  43,34,25,998  43,34,25,998          0.0  _Z12multiply_gpuPiS_S_iiiii
      1.5        66,97,032          2     33,48,516.0     33,48,516.0     33,37,264     33,59,768     15,912.7  _Z7add_gpuiPiS_ii
      0.1         3,79,376          1      3,79,376.0      3,79,376.0      3,79,376      3,79,376          0.0  _Z13transpose_gpuPiS_ii

Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpumemtimesum.py mat-4096.out.sqlite]...

 Time (%)  Total Time (ns)  Count    Avg (ns)     Med (ns)     Min (ns)   Max (ns)   StdDev (ns)      Operation
 --------  ---------------  -----  -----------  -----------  ---------  ---------  -----------  ------------------
     69.9      1,30,85,008      2  65,42,504.0  65,42,504.0  65,00,273  65,84,735     59,723.7  [CUDA memcpy HtoD]
     30.1        56,39,343      1  56,39,343.0  56,39,343.0  56,39,343  56,39,343          0.0  [CUDA memcpy DtoH]
```

MAT 8192



```
 Time (%)  Total Time (ns)  Num Calls      Avg (ns)          Med (ns)         Min (ns)        Max (ns)      StdDev (ns)           Name
 --------  ---------------  ---------  ----------------  ----------------  --------------  --------------  ------------  ----------------------
     94.3   3,38,25,07,574          1  3,38,25,07,574.0  3,38,25,07,574.0  3,38,25,07,574  3,38,25,07,574           0.0  cudaDeviceSynchronize
      3.3     11,79,17,688          6     1,96,52,948.0        1,99,028.0          90,648    11,70,49,755  4,77,14,524.6  cudaMalloc
      2.2      7,72,33,030          3     2,57,44,343.3     2,61,42,160.0     2,49,12,260     2,61,78,610    7,20,835.7  cudaMemcpy
      0.2        77,07,291          6       12,84,548.5       12,22,664.5        2,96,137       20,13,034    6,27,056.1  cudaFree
      0.0           32,081          4           8,020.3           6,178.5           2,723          17,001       6,743.5  cudaLaunchKernel
      0.0              870          1             870.0             870.0             870             870           0.0  cuModuleGetLoadingMode

 Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpukernsum.py mat-8192.out.sqlite]...

 Time (%)  Total Time (ns)  Instances      Avg (ns)          Med (ns)         Min (ns)        Max (ns)     StdDev (ns)             Name
 --------  ---------------  ---------  ----------------  ----------------  --------------  --------------  -----------  --------------------------
     99.6   3,36,80,05,878          1  3,36,80,05,878.0  3,36,80,05,878.0  3,36,80,05,878  3,36,80,05,878          0.0  _Z12multiply_gpuPiS_S_iiiii
      0.4      1,31,30,814          2       65,65,407.0       65,65,407.0       65,16,808       66,14,006     68,729.4  _Z7add_gpuiPiS_ii
      0.1        17,28,339          1       17,28,339.0       17,28,339.0       17,28,339       17,28,339          0.0  _Z13transpose_gpuPiS_ii

 Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpumemtimesum.py mat-8192.out.sqlite]...

 Time (%)  Total Time (ns)  Count     Avg (ns)       Med (ns)      Min (ns)     Max (ns)   StdDev (ns)      Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  -----------  ------------------
     68.3      5,22,05,582      2  2,61,02,791.0  2,61,02,791.0  2,60,97,939  2,61,07,643      6,861.8  [CUDA memcpy HtoD]
     31.7      2,42,04,411      1  2,42,04,411.0  2,42,04,411.0  2,42,04,411  2,42,04,411          0.0  [CUDA memcpy DtoH]
```

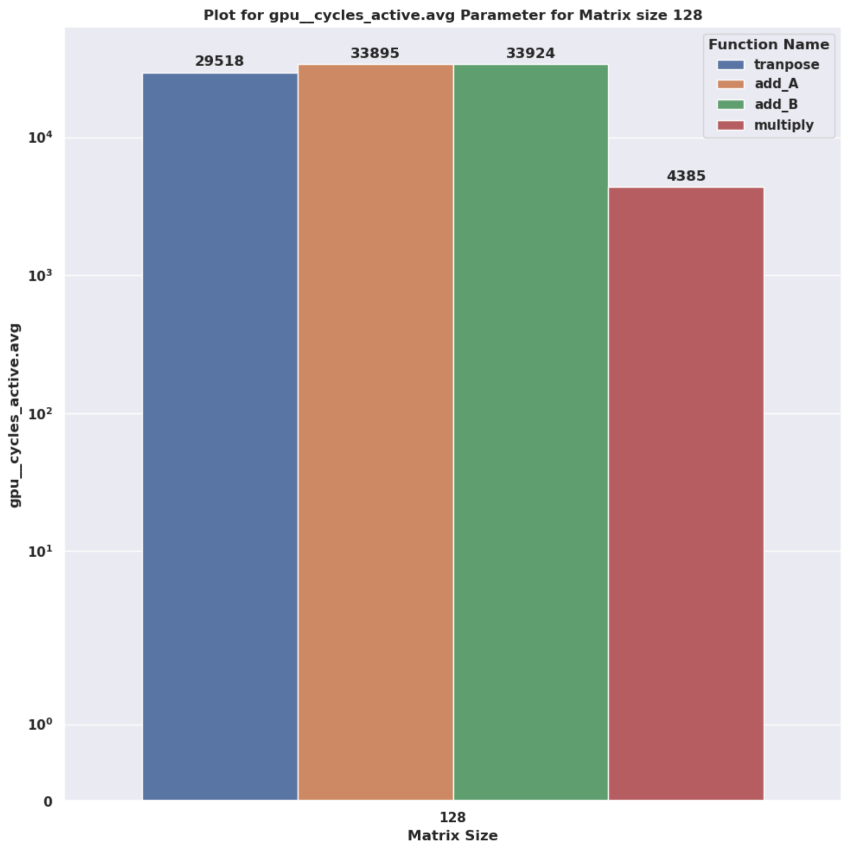
MAT 16384

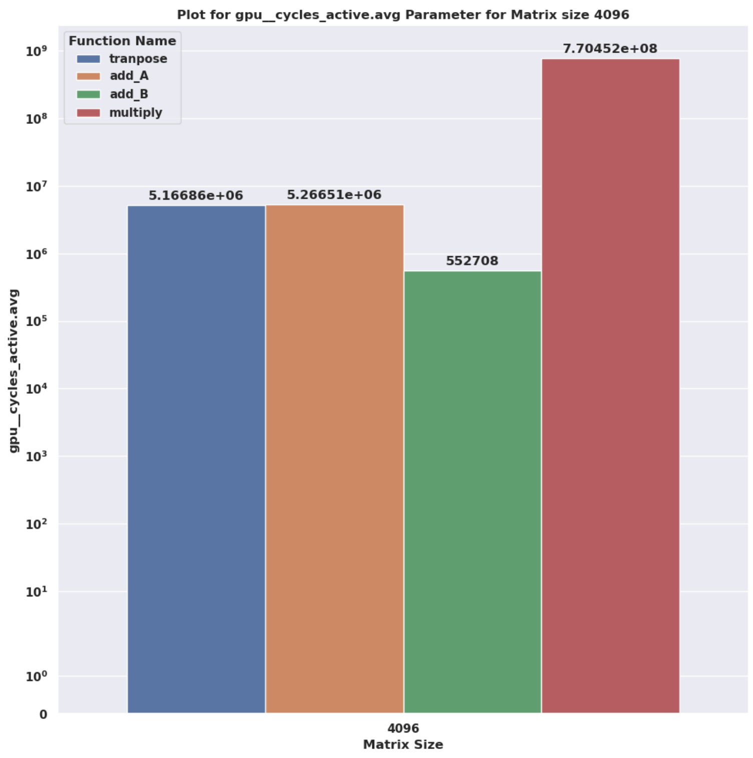


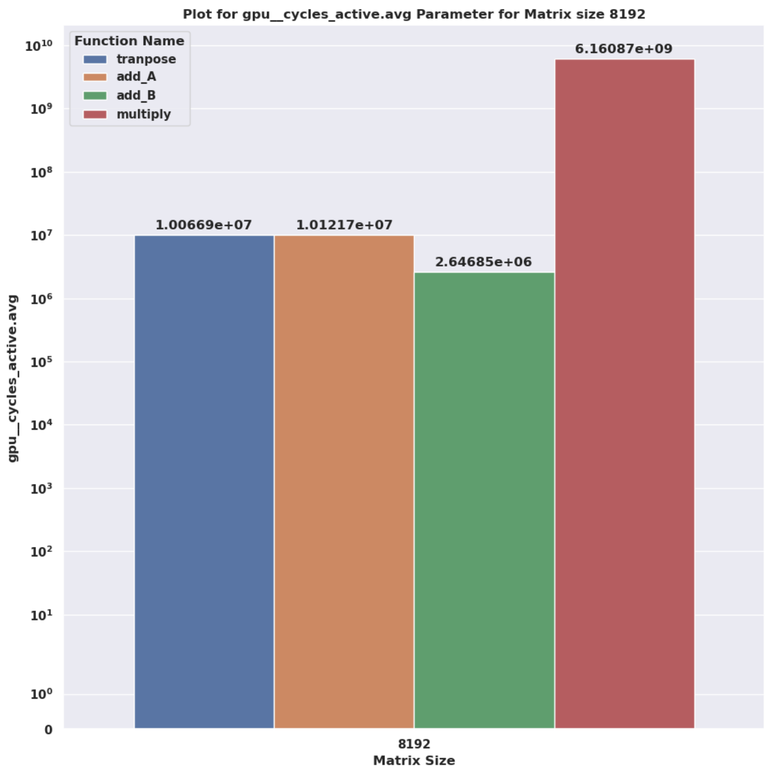
```
Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/cudaapisum.py mat-16384.out.sqlite]...

 Time (%)  Total Time (ns)  Num Calls       Avg (ns)           Med (ns)          Min (ns)         Max (ns)       StdDev (ns)           Name
 --------  ---------------  ---------  -----------------  -----------------  ---------------  ---------------  -------------  ----------------------
     98.4  27,22,49,08,677          1  27,22,49,08,677.0  27,22,49,08,677.0  27,22,49,08,677  27,22,49,08,677            0.0  cudaDeviceSynchronize
      1.1     30,59,88,426          3     10,19,96,142.0     10,37,99,136.0      9,83,37,871     10,38,51,419     31,68,263.5  cudaMemcpy
      0.4     11,97,47,489          6      1,99,57,914.8        5,73,344.0         2,29,354     11,72,66,351  4,76,71,575.5  cudaMalloc
      0.0      1,15,46,467          6        19,24,411.2       20,07,998.0         8,38,822        28,43,583     6,50,642.9  cudaFree
      0.0           36,934          4            9,233.5            5,769.0            2,812           22,584        9,307.4  cudaLaunchKernel
      0.0            1,099          1            1,099.0            1,099.0            1,099            1,099            0.0  cuModuleGetLoadingMode

Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpukernsum.py mat-16384.out.sqlite]...

 Time (%)  Total Time (ns)  Instances       Avg (ns)           Med (ns)          Min (ns)         Max (ns)      StdDev (ns)             Name

 --------  ---------------  ---------  -----------------  -----------------  ---------------  ---------------  -----------  --------------------------
-
     99.8  27,17,24,87,160          1  27,17,24,87,160.0  27,17,24,87,160.0  27,17,24,87,160  27,17,24,87,160          0.0  _Z12multiply_gpuPiS_S_iiii
i
      0.2      4,62,86,300          2      2,31,43,150.0      2,31,43,150.0      2,28,58,099      2,34,28,201  4,03,123.0  _Z7add_gpuiPiS_ii

      0.0        71,37,951          1        71,37,951.0        71,37,951.0        71,37,951        71,37,951          0.0  _Z13transpose_gpuPiS_ii


Running [/usr/local/cuda-11.7/nsight-systems-2022.1.3/target-linux-x64/reports/gpumemtimesum.py mat-16384.out.sqlite]...

 Time (%)  Total Time (ns)  Count      Avg (ns)         Med (ns)        Min (ns)       Max (ns)    StdDev (ns)      Operation
 --------  ---------------  -----  --------------  --------------  ------------  ------------  -----------  ------------------
     68.0     20,75,35,914      2  10,37,67,957.0  10,37,67,957.0  10,37,63,440  10,37,72,474      6,388.0  [CUDA memcpy HtoD]
     32.0      9,76,32,460      1   9,76,32,460.0   9,76,32,460.0   9,76,32,460   9,76,32,460          0.0  [CUDA memcpy DtoH]
```

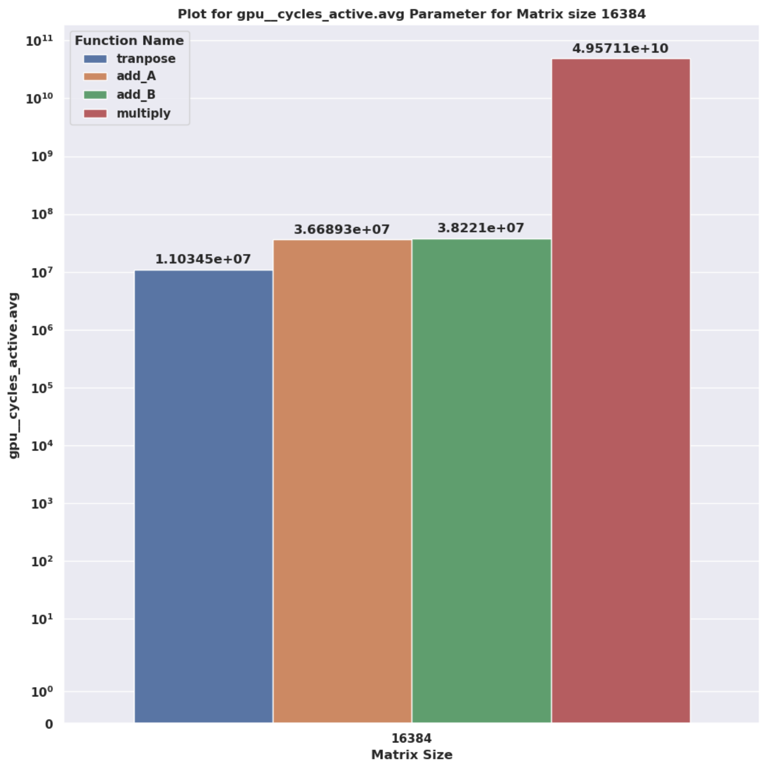
++ The above tables show the bifurcation of the times taken by cuda-api’s and kernel’s launched by us.
++ The majority of time is being consumed by the multiplication kernel itself.
++ Whereas the next top consumer is the cudamemcpy operation, in which the host to device copy is the bottleneck.

GPU Cycles:


Plot for gpu__cycles_active.avg Parameter for Matrix size 128
29518
33895
33924
4385
Function Name
tranpose
add_A
add_B
multiply
gpu__cycles_active.avg
128
Matrix Size


Plot for gpu__cycles_active.avg Parameter for Matrix size 4096
Function Name
tranpose
add_A
add_B
multiply
5.16686e+06
5.26651e+06
552708
7.70452e+08
gpu__cycles_active.avg
4096
Matrix Size

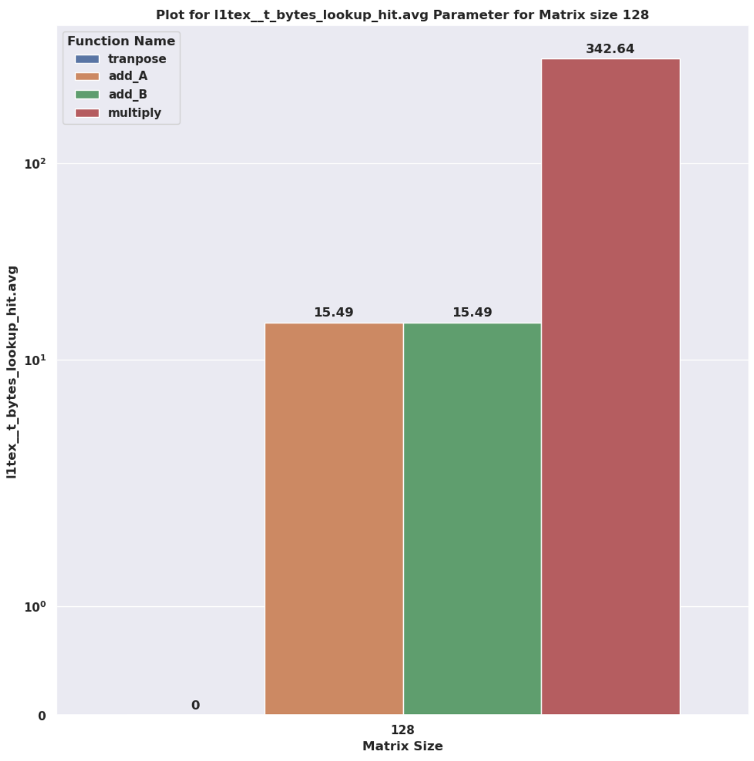


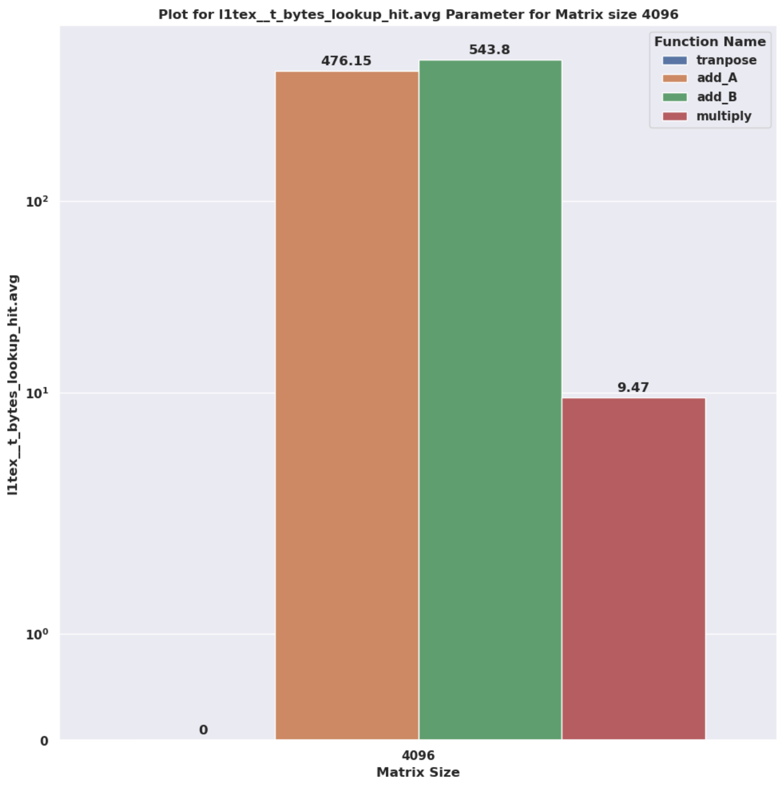


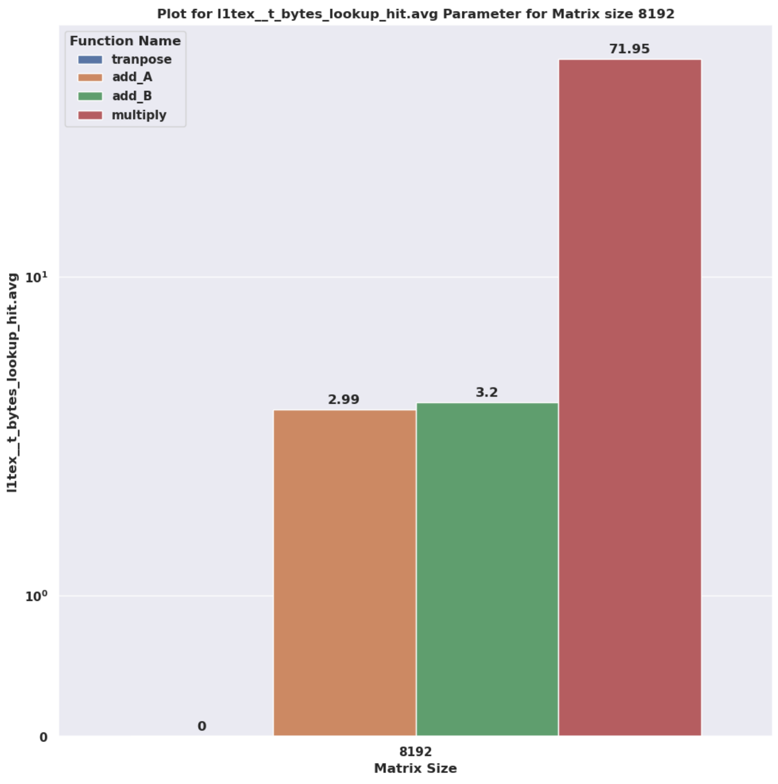
Kernel for matrix multiplication is the top consumer in terms of GPU cycles, which is justified as this where the core logic resides.

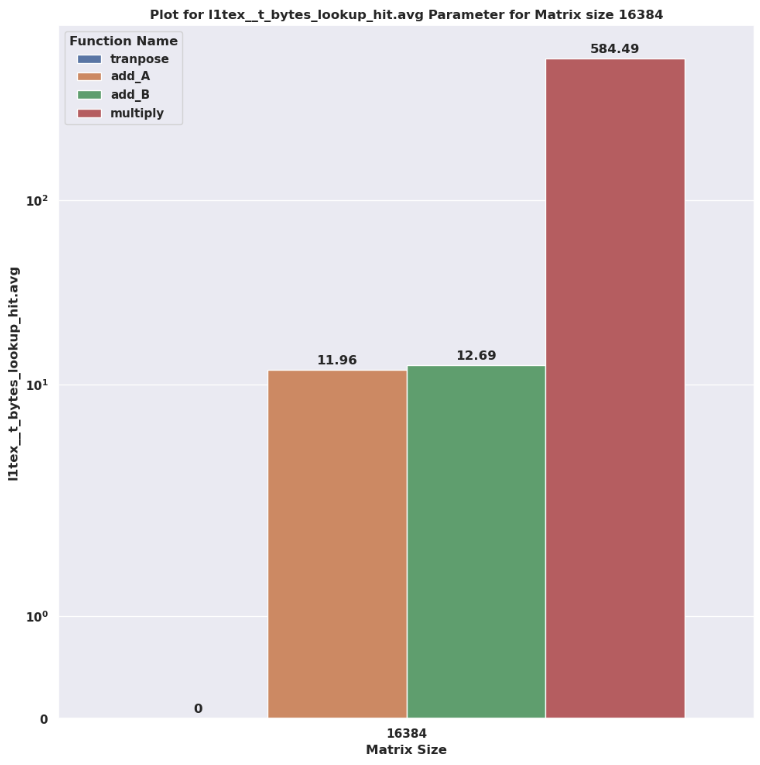
L1 Texture Cache hit rate:

The Level 1 (L1)/Texture Cache is located within the GPC. It can be used as directed-mapped shared memory and/or store global, local and texture data in its cache portion. l1tex__t refers to its Tag stage. l1tex__m refers to its Miss stage. l1tex__d refers to its Data stage.


Plot for l1tex__t_bytes_lookup_hit.avg Parameter for Matrix size 128
Function Name
tranpose
add_A
add_B
multiply
342.64
15.49
15.49
0
l1tex__t_bytes_lookup_hit.avg
Matrix Size
128

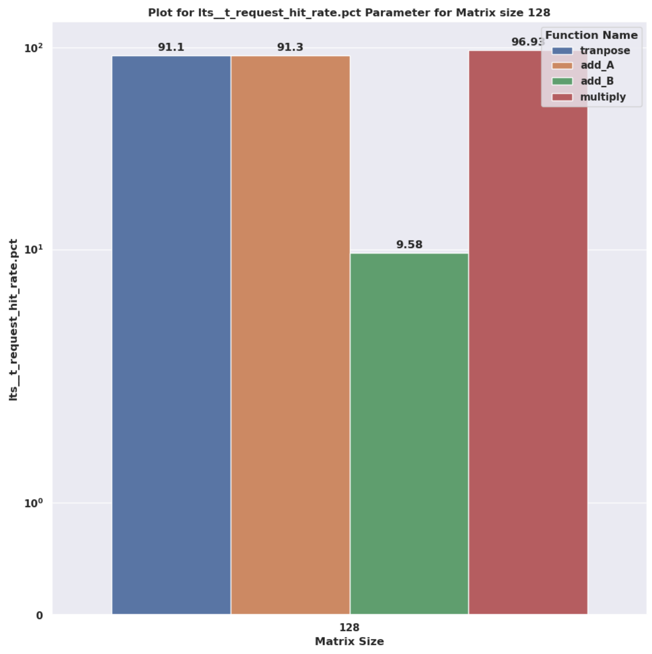

Plot for l1tex__t_bytes_lookup_hit.avg Parameter for Matrix size 4096
Function Name
tranpose
add_A
add_B
multiply
476.15
543.8
9.47
0
l1tex__t_bytes_lookup_hit.avg
Matrix Size
4096

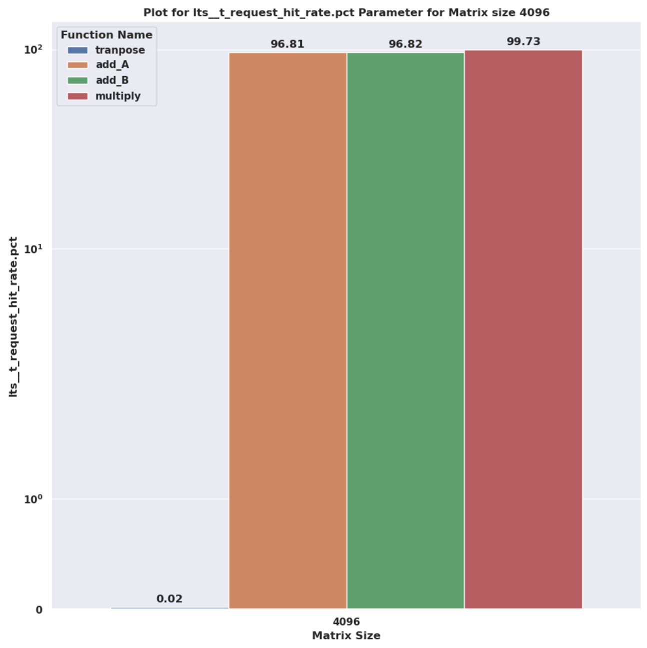

Plot for l1tex__t_bytes_lookup_hit.avg Parameter for Matrix size 8192
Function Name
tranpose
add_A
add_B
multiply
71.95
2.99
3.2
0
l1tex__t_bytes_lookup_hit.avg
Matrix Size
8192

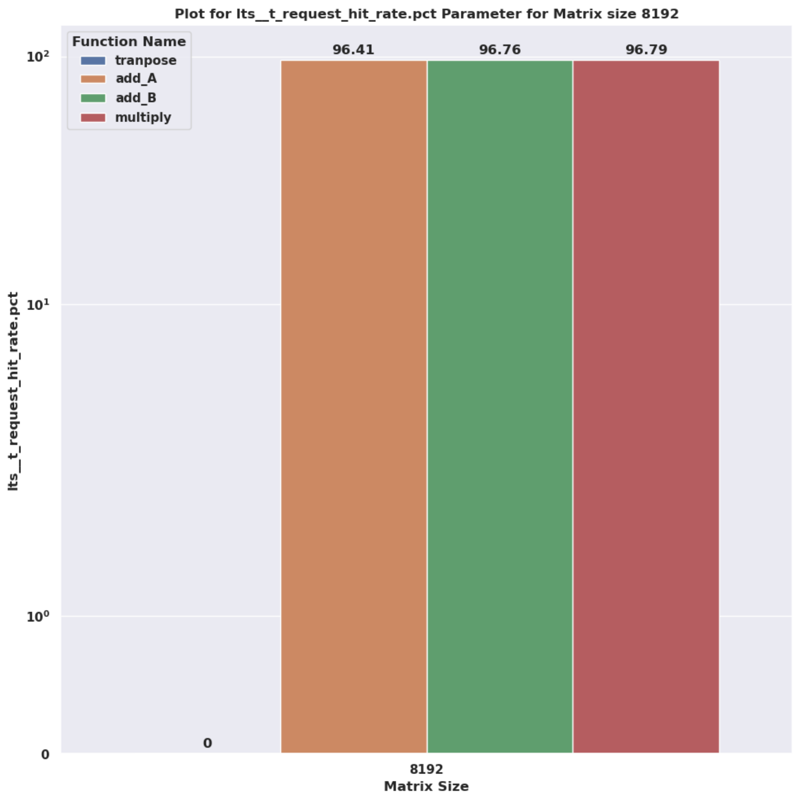

Plot for l1tex__t_bytes_lookup_hit.avg Parameter for Matrix size 16384
Function Name
tranpose
add_A
add_B
multiply
584.49
11.96
12.69
0
l1tex__t_bytes_lookup_hit.avg
Matrix Size
16384

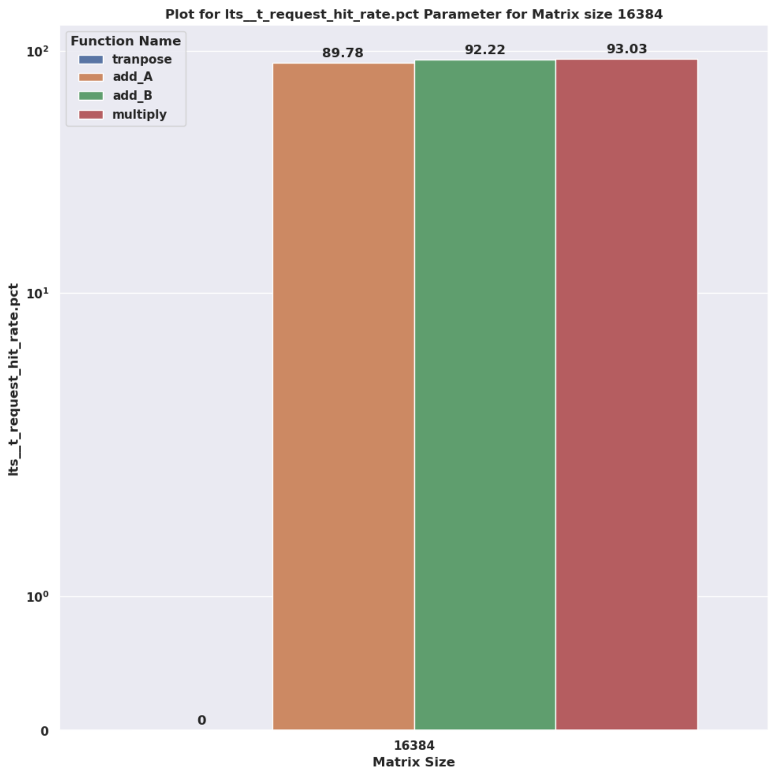
L2 Cache (lts__t):

A Level 2 (L2) Cache Slice is a sub-partition of the Level 2 cache. lts__t refers to its Tag stage.



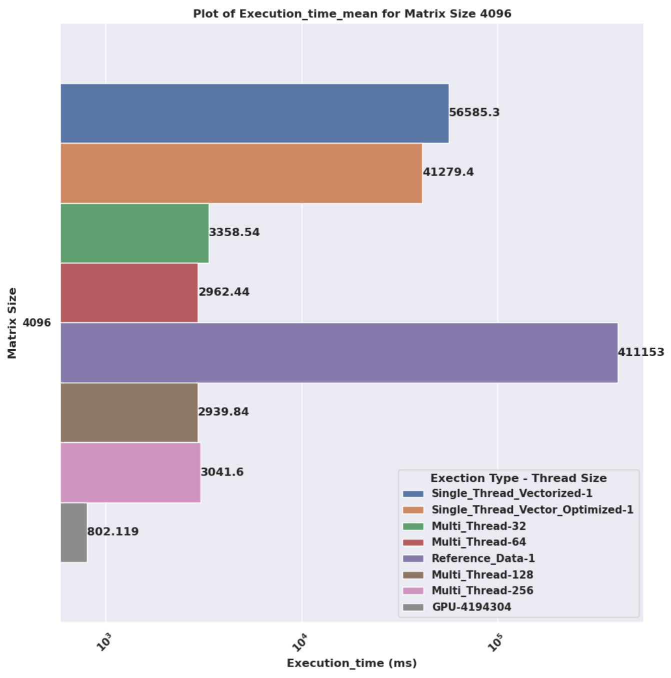


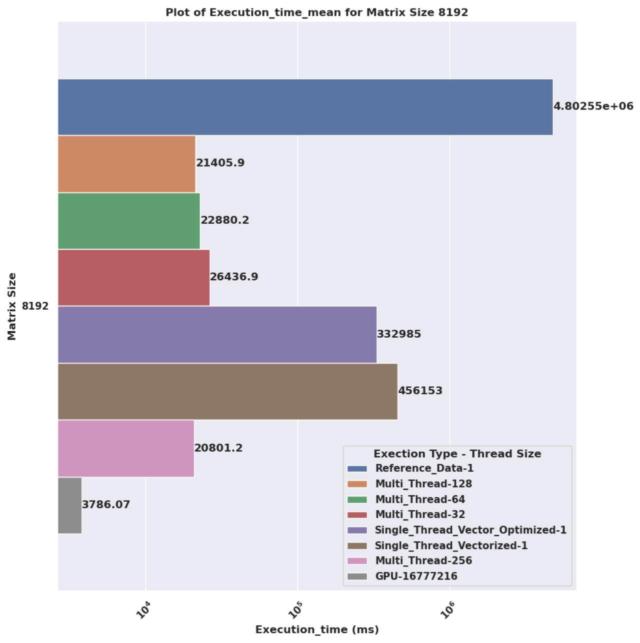


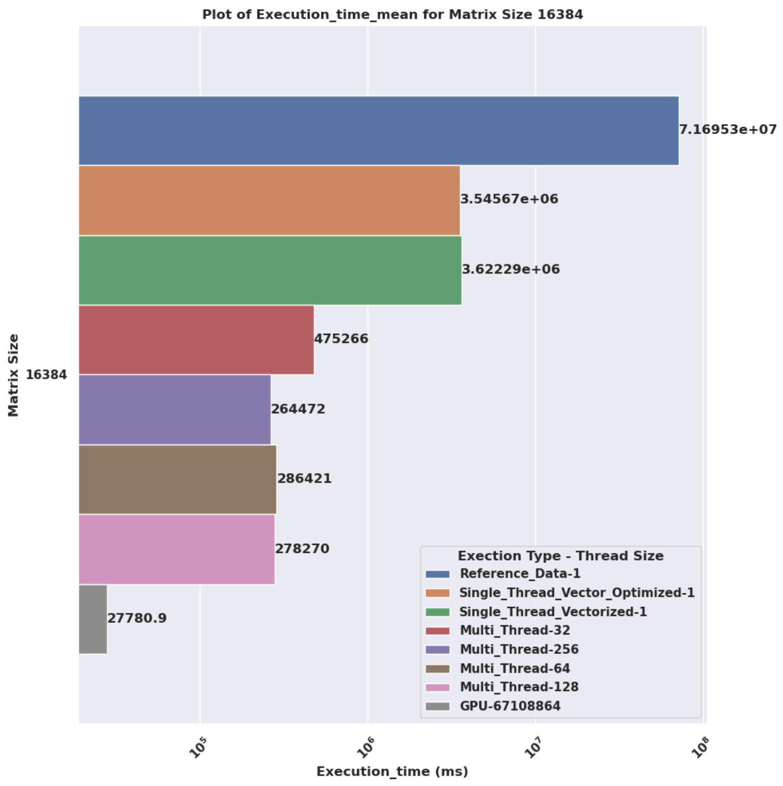


++ We can see all operations except transpose have a hit rate of above 90%.

Concluding remarks :-







It can be observed that with respect to reference execution time the GPU is ~512x, ~1268x, ~2590x faster for matrix sizes of 4096, 8192 and 16384 respectively. It can be observed that the higher the matrix sizes the better its to use GPU because of the number of parallel threads that it can support.

This also supports our previous claim of high parallelism in workload made in multi-threading.

References

“Advanced Vector Extensions.” n.d. Wikipedia. Accessed November 30, 2022.
https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

“Cache Oblivious Approach.” n.d. Lecture 8: The Cache Oblivious Approach. Accessed November 29, 2022.
http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture08.pdf.

“CUDA GPU Tranpose.” n.d. GitHub. Accessed November 29, 2022.
https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/.

“Intel® Intrinsics Guide.” n.d. Intel. Accessed November 29, 2022.
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

“Tutorial on SIMD vectorisation to speed up brute force.” n.d. Codeforces. Accessed November 29, 2022.
https://codeforces.com/blog/entry/98594.