Performance Optimization of Diagonal Matrix Multiplication

Aman Choudhary (*amanc@iisc.ac.in*)

1. Problem Statement

Given two matrices A and B, we need to compute their **Diagonal Matrix Multiplication** (DMM) efficiently. DMM can be visualised as follows:

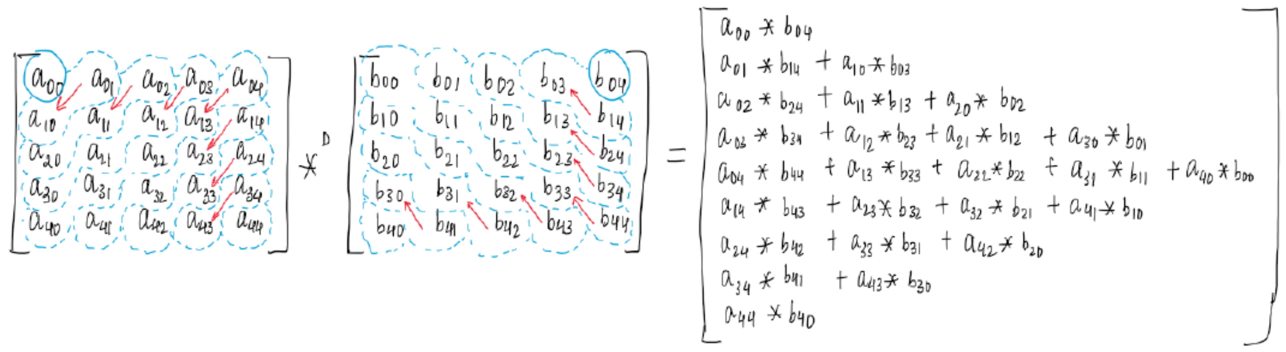

a00 * b04
a01 * b14 + a10 * b03
a02 * b24 + a11 * b13 + a20 * b02
a03 * b34 + a12 * b23 + a21 * b12 + a30 * b01
a04 * b44 + a13 * b33 + a22 * b22 + a31 * b11 + a40 * b00
a14 * b43 + a23 * b32 + a32 * b21 + a41 * b10
a24 * b42 + a33 * b31 + a42 * b20
a34 * b41 + a43 * b30
a44 * b40

Figure 1: *Diagonal Matrix Multiplication*

2. Single Threaded

2.1 First Optimization

- We observe that in order to compute DMM, we need to perform $N \times N$ multiplications. Each element in A is multiplied to a unique element in B exactly once. That is, there exists a one-to-one mapping between elements of A and B.

- On following the arrows drawn in A (Fig 1), it is clear that each subsequent access to elements in A, happens to a different row. Since, an array is stored in **row-major** fashion for C++ programs, this access pattern will result in **poor spatial locality**.

- As pointed out above, we are working with data which is spread out spatially in memory. Hence, we would need more blocks to be present in the data cache at any given point. However, the cache is a limited resource, and it can only support a certain number of cache blocks per program. Soon, memory accesses made by our program would start causing **conflict misses** in the data cache, and pay due penalty.

- Our first optimization is exactly to mitigate this problem. From Fig 2, we observe that the terms of *Output* array can be re-written in a different fashion. Instead of going over elements diagonally, we can achieve the same result, if we instead traverse row-wise for A and column-wise for B.

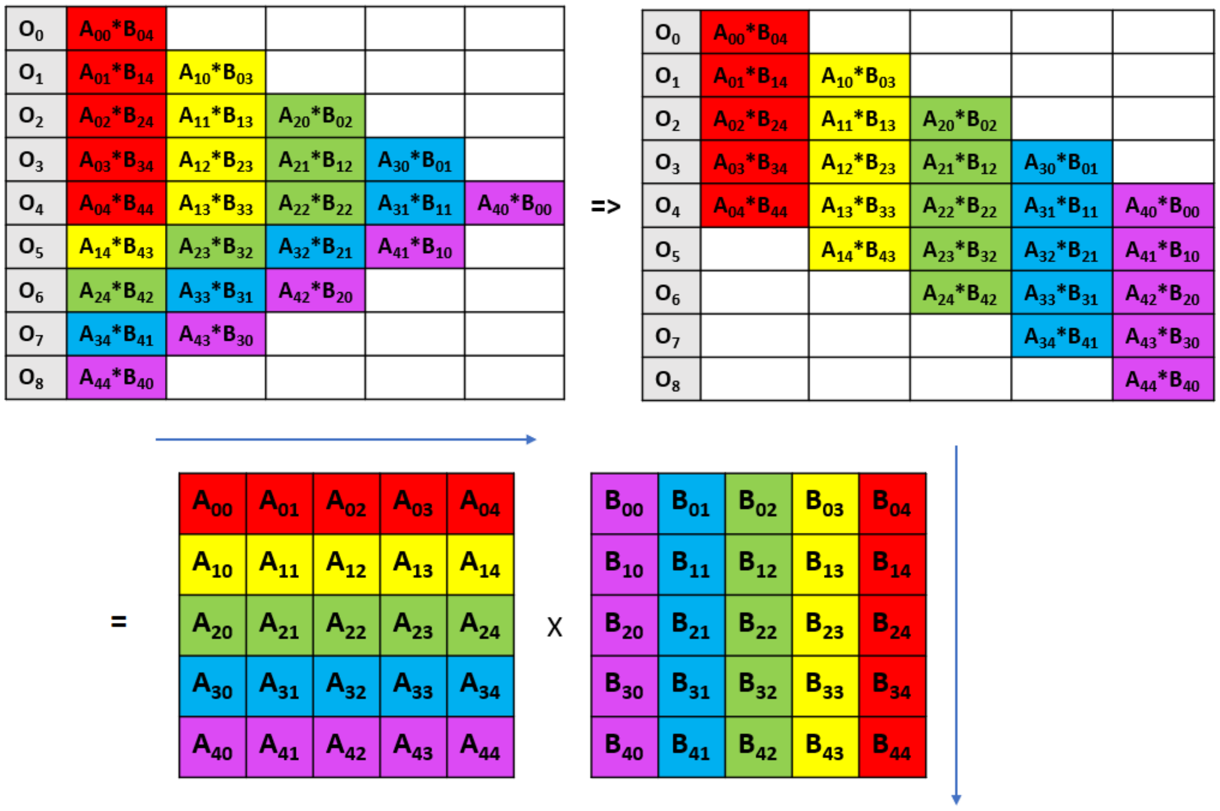

O0 A00*B04
O1 A01*B14 A10*B03
O2 A02*B24 A11*B13 A20*B02
O3 A03*B34 A12*B23 A21*B12 A30*B01
O4 A04*B44 A13*B33 A22*B22 A31*B11 A40*B00
O5 A14*B43 A23*B32 A32*B21 A41*B10
O6 A24*B42 A33*B31 A42*B20
O7 A34*B41 A43*B30
O8 A44*B40
=>
O0 A00*B04
O1 A01*B14 A10*B03
O2 A02*B24 A11*B13 A20*B02
O3 A03*B34 A12*B23 A21*B12 A30*B01
O4 A04*B44 A13*B33 A22*B22 A31*B11 A40*B00
O5 A14*B43 A23*B32 A32*B21 A41*B10
O6 A24*B42 A33*B31 A42*B20
O7 A34*B41 A43*B30
O8 A44*B40
=
A00 A01 A02 A03 A04
A10 A11 A12 A13 A14
A20 A21 A22 A23 A24
A30 A31 A32 A33 A34
A40 A41 A42 A43 A44
X
B00 B01 B02 B03 B04
B10 B11 B12 B13 B14
B20 B21 B22 B23 B24
B30 B31 B32 B33 B34
B40 B41 B42 B43 B44

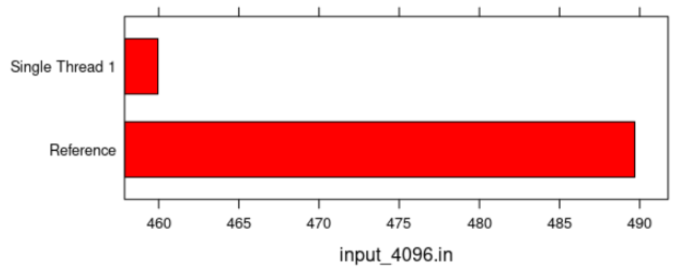
Figure 2: *Rewriting Output Array*

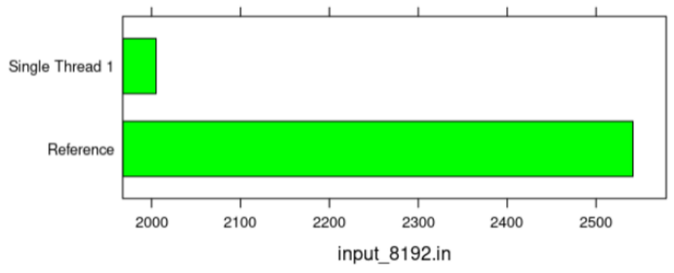
- In the new approach, we have a better access pattern for A. Keeping aside the compulsory misses, we would always hit in the data cache for elements of A. We thus, expect a drop in the execution time, which is validated by the results below.
- A minor optimization was added, where multiplications involving index variables inside loop were replaced with equivalent addition instructions, as they take lesser number of cycles. This was done as the index variables are updated in every iteration.

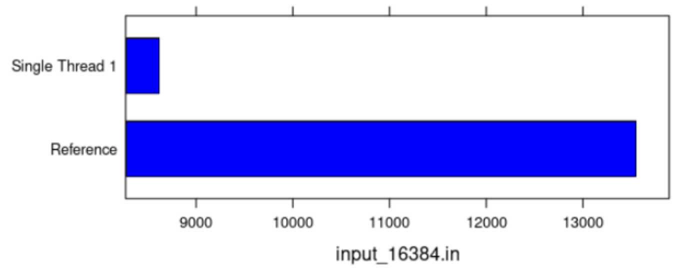
2.2 Results

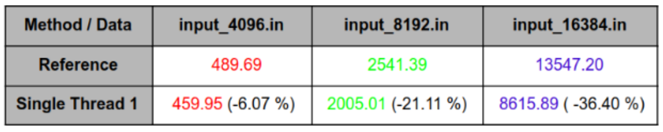
- From Fig 3, we notice that although there is only a slight drop (6.07 %) in execution time for $input4096.in$, the improvement for the other two data sets (21.11 % and 36.40 % respectively) is much better. [**Formula for drop:** (Optimized Time - Reference Time)/Reference Time]
- Using $perf$, the hardware performance counters were also monitored for the given program. Fig 4 reports the significant reduction in L1 and L2 cache misses, corresponding to our first optimization over Single Threaded program.

NOTE: All measurements for execution times/ hardware performance counters were calculated as mean over 5 different runs. The experiments were performed on host with the following configuration: **CPU** - Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, **OS** - Ubuntu 20.04.1 LTS, **RAM** - 4GB





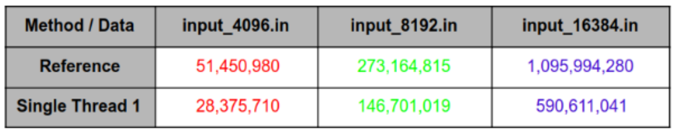




Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	489.69	2541.39	13547.20
Single Thread 1	459.95 (-6.07 %)	2005.01 (-21.11 %)	8615.89 (-36.40 %)

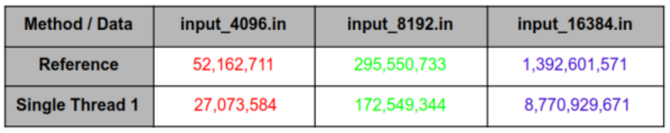
Comparison of Execution Time

Figure 3: *Comparison of Single Threaded-1 Execution time (in milliseconds)*



Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	51,450,980	273,164,815	1,095,994,280
Single Thread 1	28,375,710	146,701,019	590,611,041

Comparison of L1 Cache Misses



Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	52,162,711	295,550,733	1,392,601,571
Single Thread 1	27,073,584	172,549,344	8,770,929,671

Comparison of L2 Cache Misses

Figure 4: *Comparison of Data Cache Misses*

2.3 Second Optimization

- While we have improved the access pattern for A, it is still poor for B because it is being accessed in **column-major**. Consider a sample case in Fig 5, where $N = 8$, and cache line size is $16B$ (4 integers).
- When first 4 elements of 1^{st} row of A are multiplied with top 4 elements of $N - 1^{th}$ column of B, we incur initial compulsory misses (shown in grey rectangular boxes, Fig 5: 1(a)).
- For the bottom 4 elements of $N - 1^{th}$ column in B, 4 new cache blocks are fetched which replace the previous 4 cache blocks which were brought earlier (Fig 5: 1(b)). We assume that these happened due to conflict misses in the cache. In fact, they are bound to happen for large matrices such as ours.
- Now, when we start multiplication of 2^{nd} row of A with top 4 elements of $N - 2^{th}$ column of B, we miss for the same 4 cache blocks, which just got replaced above (Fig 5: 1(c)). As a matter of fact, we are incurring a cache miss for accessing **every element** of B, and this needs to be taken care of.
- Our second optimization performs multiplication over a small batch, instead of the entire matrix in one go. It repeats the process for different batches, to cover the entire matrix. In

Fig 5:2(a,b,c,d), we we assume a batch of 4×4 for our sample example. After paying the price of unavoidable compulsory misses, we make sure that we **completely use** the elements which have been brought in the cache.

- Using this optimisation, we improve the locality of reference of B, and ensure that we are missing for every cache block only once, and not for every element as before.

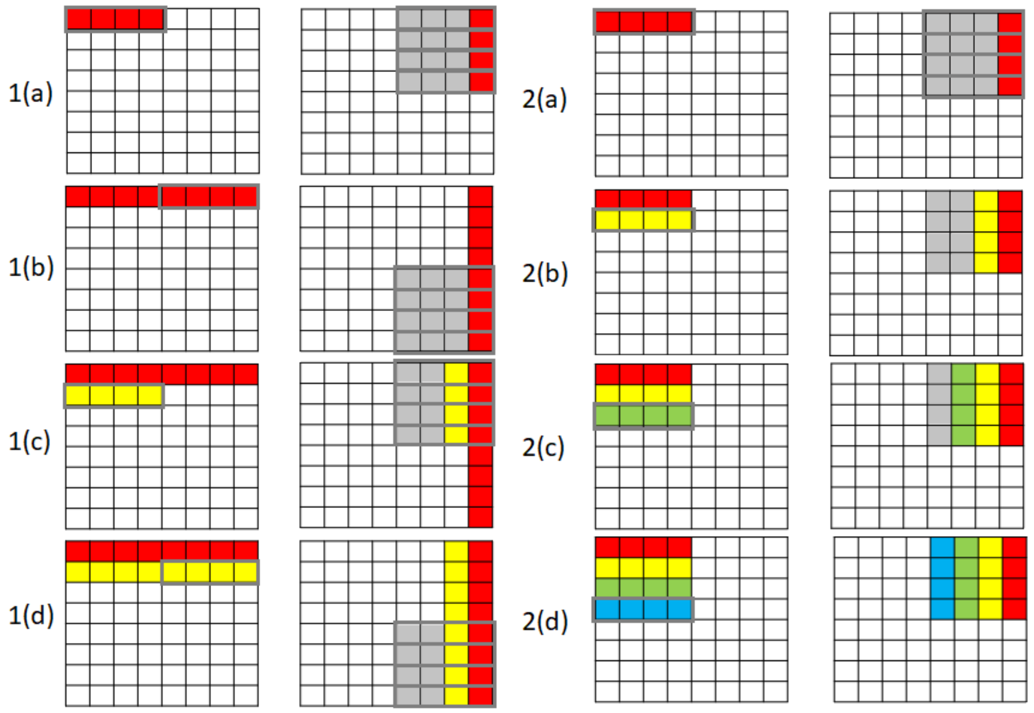


Figure 5: *Improving locality of B by using batches*

2.4 Results

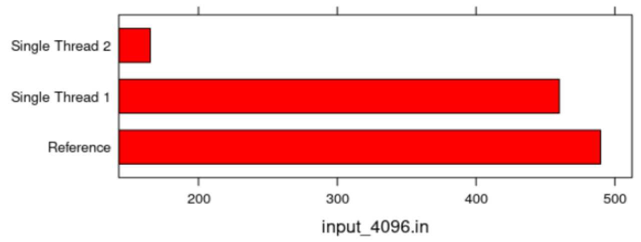
- The results of performing multiplication in small localities showed drastic plunge in the run-time of the single threaded program. A drop of roughly 66 % drop was observed for *input*4096*.in*. The reduction over the larger data-sets was even better (approx. 75 % for both) (Fig 6). Similar drop in data cache misses was noted (Fig 7). Clearly, this optimization showed much greater impact than the first one.

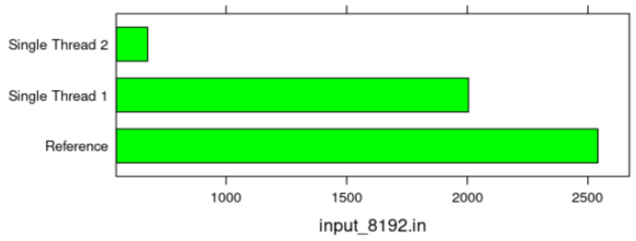
2.5 Choosing Batch Size

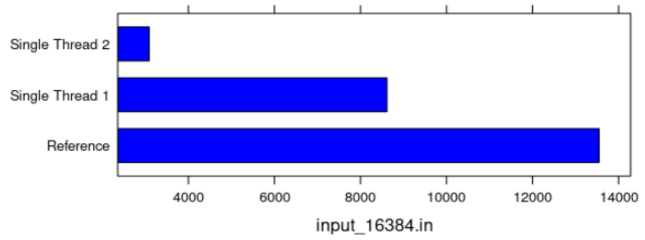
- In order to select the optimal batch size, three different sizes namely (16×16, 32×32, and 64×64 were experimented with. It was found that the program had least run-time when matrix multiplication was computed in batches of size 32×32 (Fig 8).

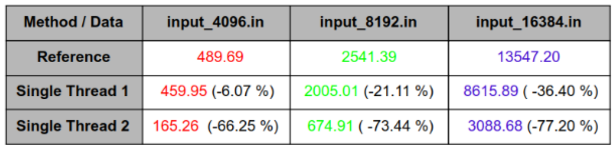
- The results suggest that while increasing batch size from 32 to 64, the spatial locality becomes too large to handle like before, hence the execution time grows accordingly.

- A more interesting change is observed while going from 16 to 32. A single cache block on the native system can store 16 integers. When batch size is 32, two contiguous cache lines are being accessed for every row in the batch. The drop in execution time might be attributed to the perfecting of blocks in a single row, which lie in contiguous locations. The same benefit can't be taken while using batch size of 16, as the multiplication jumps to next row, just after processing the only cache block in that row.





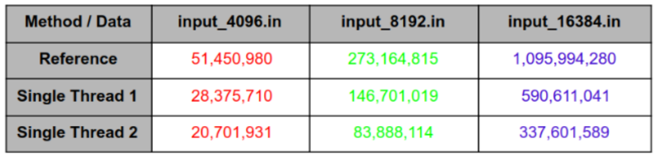




Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	489.69	2541.39	13547.20
Single Thread 1	459.95 (-6.07 %)	2005.01 (-21.11 %)	8615.89 (-36.40 %)
Single Thread 2	165.26 (-66.25 %)	674.91 (-73.44 %)	3088.68 (-77.20 %)

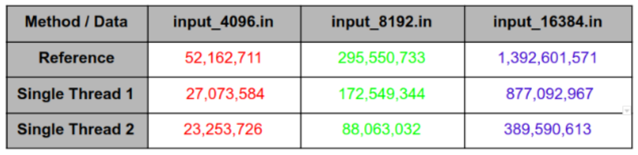
Comparison of Execution Time

Figure 6: *Comparison of Single Threaded-2 Execution time (in milliseconds)*



Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	51,450,980	273,164,815	1,095,994,280
Single Thread 1	28,375,710	146,701,019	590,611,041
Single Thread 2	20,701,931	83,888,114	337,601,589

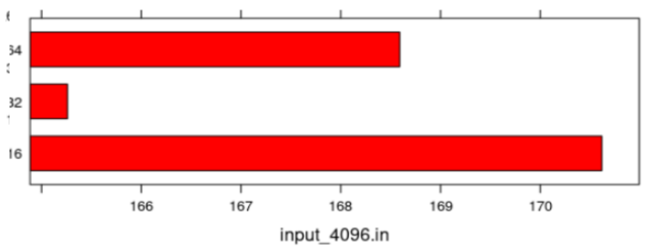
Comparison of L1 Cache Misses

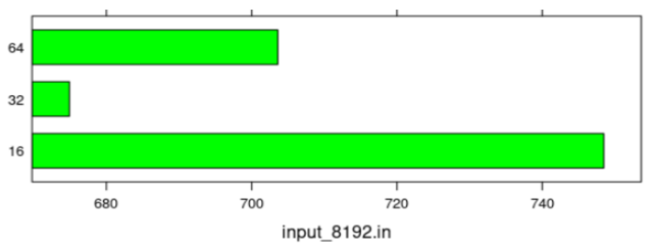


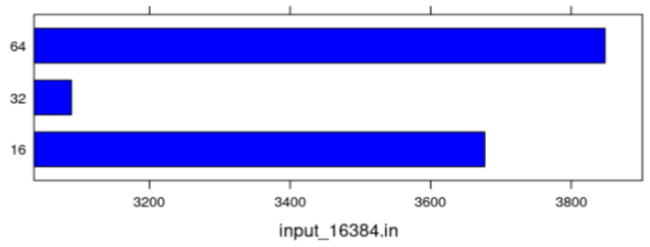
Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	52,162,711	295,550,733	1,392,601,571
Single Thread 1	27,073,584	172,549,344	877,092,967
Single Thread 2	23,253,726	88,063,032	389,590,613

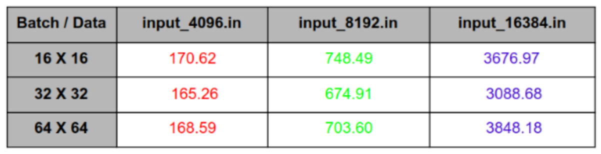
Comparison of L2 Cache Misses

Figure 7: *Comparison of Data Cache Misses*









Batch / Data	input_4096.in	input_8192.in	input_16384.in
16 X 16	170.62	748.49	3676.97
32 X 32	165.26	674.91	3088.68
64 X 64	168.59	703.60	3848.18

Comparison of Execution times for different batch sizes

Figure 8: *Comparison of Single Threaded-2 Execution time for different Batch sizes*

3. Multi Threaded

3.1 Implementation

- The multi-threaded implementation leverages the *pthread* library. It builds from where we left off the single threaded version. We now fix the batch size at 32×32 once and for all. Then, we divide the entire matrix into separate regions. An example is shown in figure below.

- Let us assume we have 4 threads. Accordingly, we have partitioned A into 4 regions, based on number of rows. For B, we do the same division but in a column-wise fashion. Now, thread $T1$ is responsible for multiplying elements in red region in A with corresponding elements in B, and produce the result in the appropriate block in $Output$ array.

- An interesting observation is that each thread works within its own region in A, B and $Output$. There is whatsoever, no sharing of data in between the threads. Each thread can accomplish its part of multiplication independently. Hence, we don't need not any mutex locks. This is a big win, because locking and unlocking the data elements hurts the performance badly.

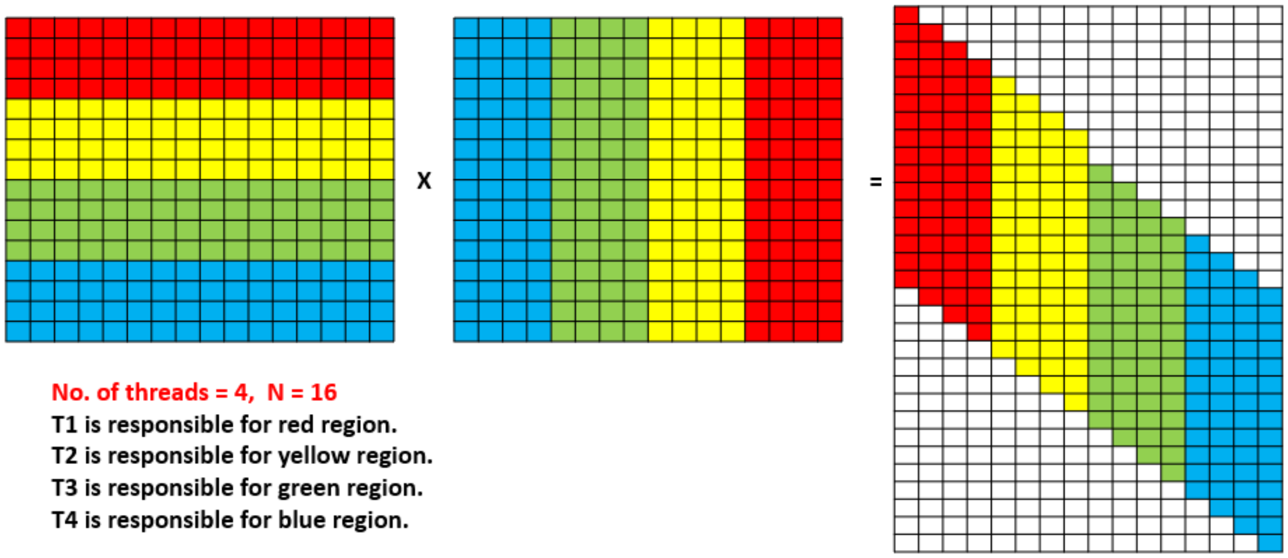

X
=
No. of threads = 4, N = 16
T1 is responsible for red region.
T2 is responsible for yellow region.
T3 is responsible for green region.
T4 is responsible for blue region.

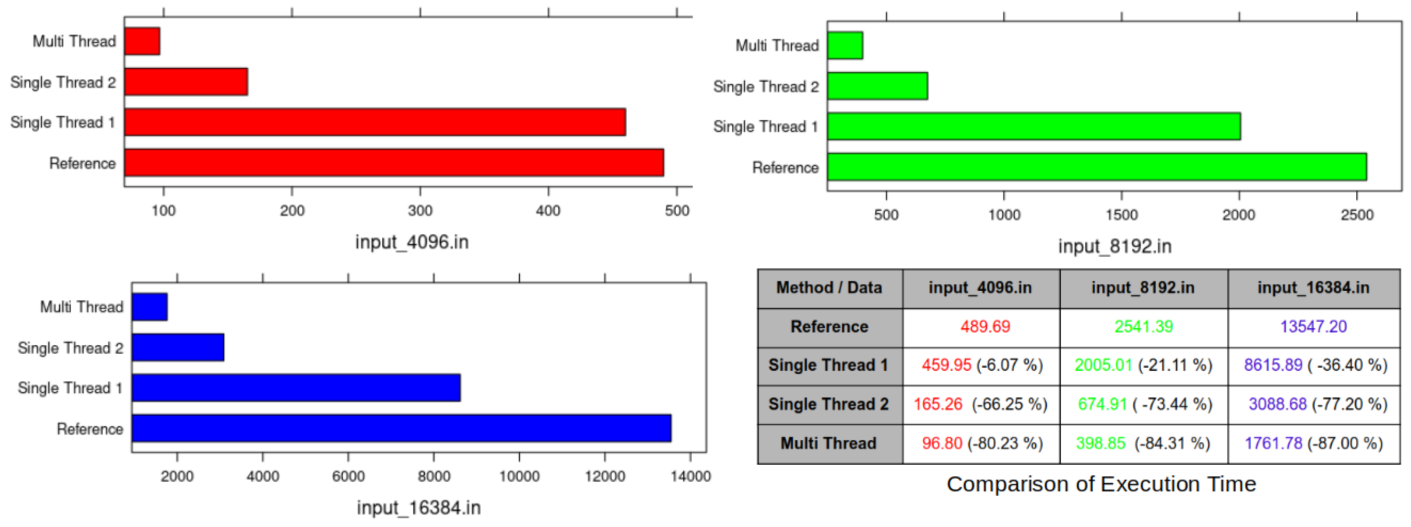
Figure 6: *Implementing DMM with multiple threads*

3.2 Results

As anticipated, the multi-threaded version of DMM displays the best performance compared to all previous versions. We are able to achieve a drop of at least (80 %) across all datasets.

3.3 Choosing number of Threads

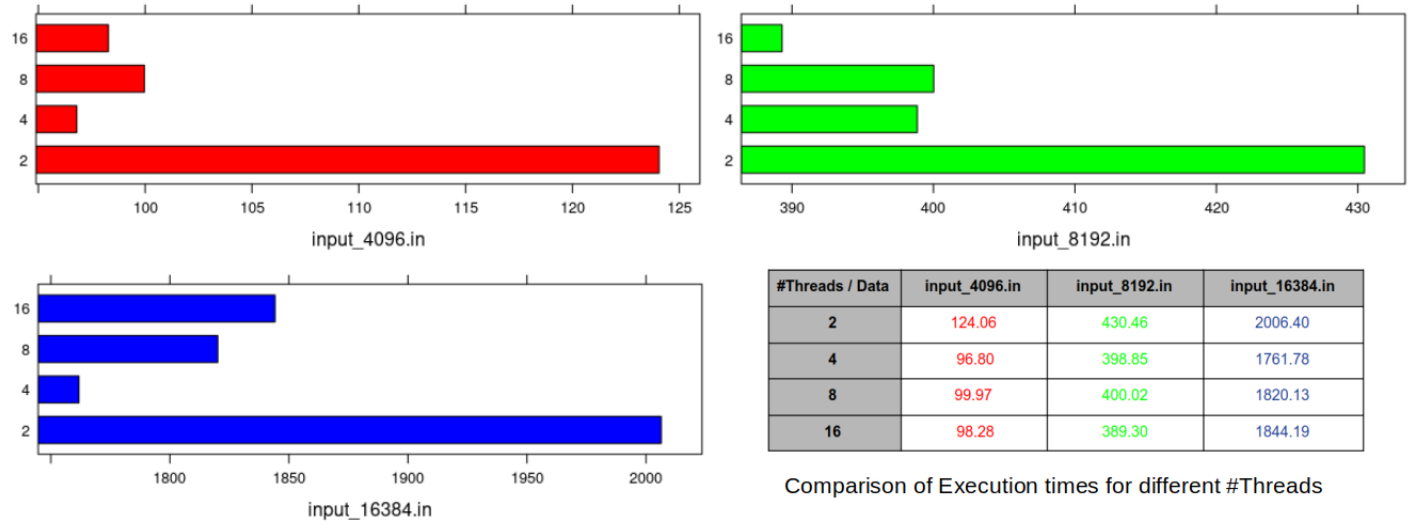
In order to choose the number of threads, we experimented with different number of threads, namely 2, 4, 8, and 16. The host system has following configuration: **Socket(s):** 1, **Core(s) per socket:** 2, **Thread(s) per core:** 2, **CPU(s):** 4 . This means that the system has 4 logical CPU cores, which explains the fact that we achieved the optimal run-time when 4 threads were used. While significant speedup was gained on increasing the no. of threads from 2 to 4, however, increasing the number of threads beyond 4 did not scale well for the host system.



Method / Data	input_4096.in	input_8192.in	input_16384.in
Reference	489.69	2541.39	13547.20
Single Thread 1	459.95 (-6.07 %)	2005.01 (-21.11 %)	8615.89 (-36.40 %)
Single Thread 2	165.26 (-66.25 %)	674.91 (-73.44 %)	3088.68 (-77.20 %)
Multi Thread	96.80 (-80.23 %)	398.85 (-84.31 %)	1761.78 (-87.00 %)

Comparison of Execution Time

Figure 7: *Comparison of Multi Threaded Execution time (in milliseconds)*



#Threads / Data	input_4096.in	input_8192.in	input_16384.in
2	124.06	430.46	2006.40
4	96.80	398.85	1761.78
8	99.97	400.02	1820.13
16	98.28	389.30	1844.19

Comparison of Execution times for different #Threads

Figure 8: *Comparison of Multi-Threaded Execution time for different no. of Threads*

4. Conclusion

We conclude that **improving the spatial locality of reference** was pivotal to improving the performance of the single threaded implementation. A key observation was that the impact of any given optimization was more pronounced in the larger data sets, as compared to the smaller ones. Apart from this, **using multiple threads**, also enabled us to achieve significant amount of speedup by leveraging parallel computation. However, there lies scope for further optimization of the program. This is where **CUDA** would prove to be useful. It would help us to speed up the parallelizable part of the computation even further, by harnessing the power of GPU.