Performance Optimizing Diagonal Matrix Multiplication (DMM)

Shashank Singh (shashanksing@iisc.ac.in)
Indian Institute of Science, Bangalore, India

Performance Optimization aims to improve program performance by ensuring effective resource utilization. Often, poor performance can be credited to an ineffective memory access pattern. This is optimized by finding hot regions in the program and rewriting these regions to take full advantage of spatial and temporal locality.

1 Introduction

The DMM program in context takes two matrices A and B of size $N \times N$ and produces a target column matrix C of size $2N-1$, such that

$$\overset{A}{\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}} *^D \overset{B}{\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & a_{11} \end{bmatrix}} = \overset{C}{\begin{bmatrix} a_{00} * b_{01} \\ a_{01} * b_{11} + a_{10} * b_{01} \\ a_{11} * b_{00} \end{bmatrix}}$$

The C matrix can be computed in a lot of ways and each way has its own memory access pattern which leads to different performances.
The matrix C can be computed in the following way :

$$C[k] = \sum A[i][j] * B[j][N-i] \text{ such that } i + j = k$$

The formula shows that each element $A[i][j]$ will be multiplied with an element $B[i][j]$ to produce a partial sum for computing $C[i+j]$. All such partial sums can be added to obtain the matrix C.

We will evaluate the performance of different methods and aim to optimize the run time of the program. The metric of speedup is defined as follows :

$$\text{Speedup} = \frac{\text{Run time of reference version}}{\text{Run time of optimized version}}$$

2 About the Experiments

All the experiments were run on the high-end server provided by IISc. The server runs on *Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz* with *192 GB* of RAM and 64 CPUs with 16 Cores per CPU.

All the performance data was collected using **perf** tool and the values shown here are average values of 5 runs. The performance data collected was exclusively due to running of a particular implementation. The calls for warming up caches and TLB and matching of correct results were disabled for this purpose.

The running time and speedup obtained are averaged values of at least 3 runs for every implementation and every input set (4K, 8K, 16K, 32K).

3 Reference (Naive) DMM

The pre-provided reference function calculates the output array by going diagonally along matrix A and matrix B. We will take the run time of this method as the reference for our optimizations for both single threaded and multi threaded approaches.

From *Table 3*, *Table 4*, *Table 5*, it can be observed that there is a huge number of L1, L2 cache and TLB misses. The large numbers suggest that a majority of memory accesses are causing cache misses and tlb misses. This gives us a hint that perhaps we could optimize the memory accesses to ensure more cache and TLB hits.

4 Single Threaded DMM

In the single-threaded optimization, our aim is to reduce the number of cache misses by changing the memory access pattern in a cache friendly way. The following optimizations were tried on matrices of size $4K, 8K, 16K$ and $32K$.

4.1 Version 1 - Row Major Access

This optimization refers to re-implementing the operation such that memory accesses to matrix A are done in row major fashion. Due to the nature of the operation, we can access only one of the matrix in a row-major fashion. However, on implementing this, there was no improvement observed. In fact, the run time became twice (*Table 1*) upon implementing this. The reason of this couldn't be found for sure but this can be attributed to almost all accesses to matrix B being a cache miss due to column wise access.

4.2 Version 2 - Square Sub-Block Access

This version was optimized to exploit cache access for both the matrices A and B. The memory access pattern can be referred from *Figure 1* and *Figure 2*. The core idea is that we divide the matrices into **square sub blocks** (numbered 1 to 9 in the figures) of a smaller dimension. This smaller dimension is to be a small multiple of cache line size. We first complete operations for one sub-block and then move on to another sub-block.

Instead of going for a full row scan on A, in this version we do it for 1 or two cache blocks and then move on to the next row of the sub-block. This enables matrix B to retain its loaded cache line for future use. The first red access in *Figure 1* will cause a cache miss for matrix A as well as for matrix B. Following this the first cache block for the first row for sub-block 1 in both the matrices will be loaded in the cache. Then subsequent two accesses will be a miss for matrix B. But after that, as computation for second row starts in matrix A, all access to sub-block 1 in matrix B will be cache-hits as the block is already present in cache due to previous accesses.

Too large a dimension of sub-block will cause eviction of cache blocks of matrix B before they can be used and hence will increase the runtime. Experimentally, this value was observed to give best results on **2 cache block size** (128 bytes or 32 elements).

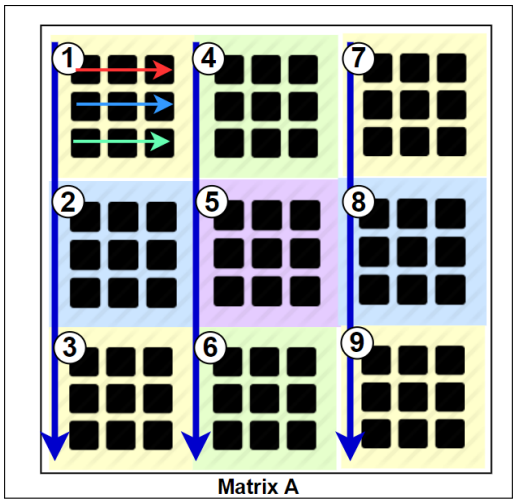

Figure 1: *Version 2 : Memory Access Pattern for Matrix A*

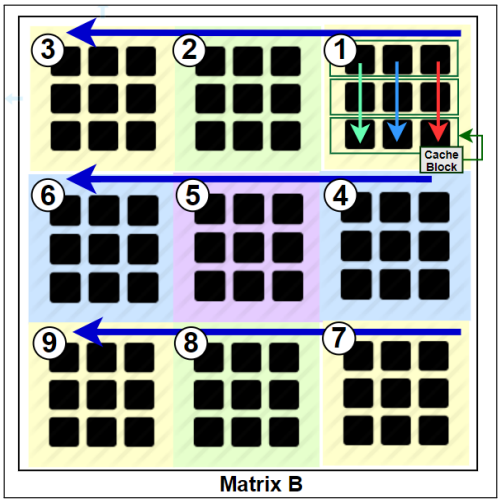

Figure 2: *Version 2 : Memory Access Pattern for Matrix B*

4.3 Version 3: Sub-Blocks - Row Wise

This version is same as Version 2, except for the change that instead of calculating for blocks in a columnar fashion for matrix A, we proceed in a row wise manner (Refer *Figure 3*) and accordingly the direction gets changed for matrix B too (Refer *Figure 4*).

This pattern of memory accesses performs better than Version 2 by a substantial margin. Apparently, this is due to the advantage of hardware prefetching for matrix A while also retaining the cache hits in matrix B from Version 2.

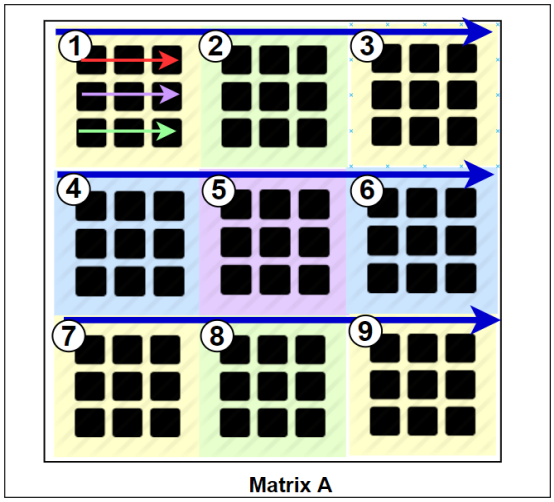

Figure 3: *Version 3 : Memory Access Pattern for Matrix A*

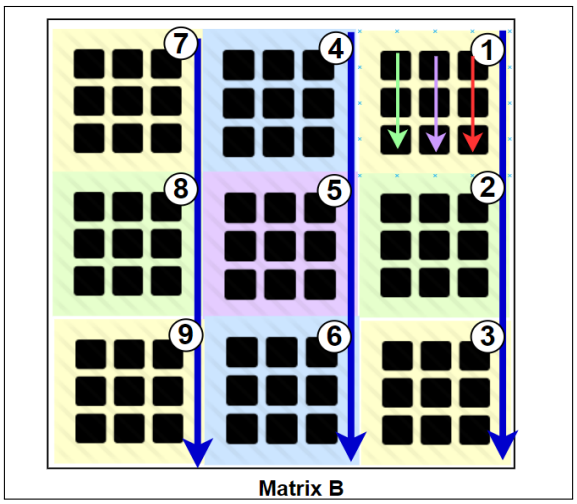

Figure 4: *Version 3 : Memory Access Pattern for Matrix B*

4.4 Performance Comparison

The different versions were analyzed on two metrics of performance - runtime and hardware events (cache misses, tlb misses). Runtime was calculated by evaluating the time difference between timestamps of before and after the procedure was run. Hardware events were collected through *perf* tool and represent an average of 5 runs.

4.4.1 Run Time

Table 1 and *Table 2* contrast the running times and speedup obtained using different approaches with respect to the reference implementation. It can be observed that Version 2 and Version 3 both give significant improvement over the reference. Version 3 performs much better on a large data-set.

	4K	8K	16K	32K
Reference	294.9	1434.2	8003.9	35788.0
Version 1	564.0	2317.4	14035.4	51929.5
Version 2	149.5	727.5	2933.4	14944.5
Version 3	116.3	737.6	3419.1	8928.4

Table 1: *Average running times (in ms) for different versions*

	4K	8K	16K	32K
Version 1	0.523	0.619	0.570	0.689
Version 2	1.973	1.971	2.728	2.395
Version 3	2.535	1.944	2.341	4.008

Table 2: *Average speedup obtained for different versions*

4.4.2 Hardware Events

Table 3, *Table 4* and *Table 5* contain the *L1 Data Cache Misses*, *L2 Cache Misses* and *TLB misses* for the different implementations respectively. It is quite evident from these that *Version 3* outperforms every other implementation in all the 3 metrics. There is almost 66% reduction of *L1 cache misses Version 3*.

	8K		16K		32K	
	Mean	σ	Mean	σ	Mean	σ
Ref	274M	314K	1,106M	0.6M	4.4G	1.0M
V1	149M	269K	602M	1.9M	2.4G	0.1M
V2	94M	43K	378M	0.2M	1.5G	2.2M
V3	86M	37K	348M	1.5M	1.4G	9.8M

Table 3: *Average L1 Data Cache Misses for different versions*

	8K		16K		32K	
	Mean	σ	Mean	σ	Mean	σ
Ref	184M	664K	1,106M	3.0M	6.3G	10.8M
V1	75M	96K	602M	2.0M	3.5G	0.5M
V2	26M	97K	378M	0.2M	0.5G	9.5M
V3	16M	56K	348M	2.5M	0.4G	13.8M

Table 4: *Average L2 Cache Misses for different versions*

	8K		16K		32K	
	Mean	σ	Mean	σ	Mean	σ
Ref	134M	441K	541M	0.9M	2.2G	5.2M
V1	68M	4K	275M	0.1M	1.1G	1.8M
V2	5M	75K	19M	0.2M	0.1G	0.3M
V3	3M	9K	11M	0.1M	0.1G	3.8M

Table 5: *Average D-TLB Misses for different versions*

4.4.3 Overall Result

After considering the speedup data and the performance statistics, it can be prudent to say that *Version 3* is the best optimized implementation for Diagonal Matrix Multiplication. There is further scope of improvement by applying arithmetic optimizations (e.g. replacing multiplications by addition) but these optimizations yield small and oscillating improvements and can be eclipsed by memory related optimizations. *Version 3* has been implemented in the code submitted with this report.

5 Muti-Threaded DMM

The version 3 of Single Threaded Approach can be used as a base for the multi-threaded approach. Since, each element of matrix A is to be multiplied with an element of matrix B which is exclusive to it, there is a clear mutual exclusion on the operands. However, to obtain the final matrix C, many partial sums have to be added, which belong to different threads. This can lead to synchronization issues.

- One solution was to use locking/mutex mechanisms. Using these caused run-time to increase manifolds due to locking/unlocking being in the loop.
- Another solution was to store the partial sums which belong to a thread in a thread-exclusive temporary array. And then when all the threads are finished, the final matrix C is computed by adding these partial sums. This solves the synchronization problem and doesn't incur significant overhead. We select this solution to go ahead.

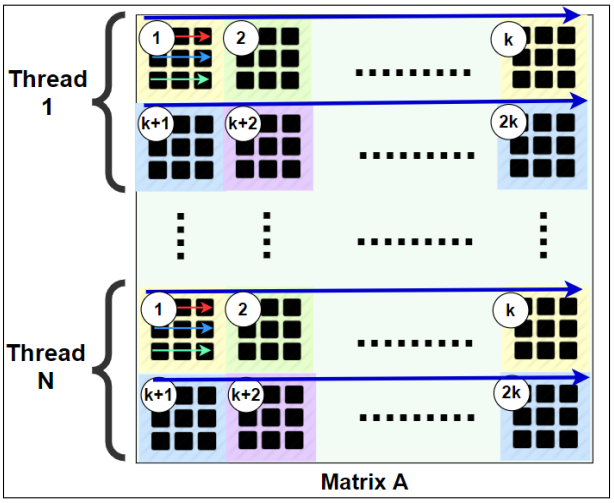

Figure 5: *Thread-wise Memory Access Pattern for Matrix A*

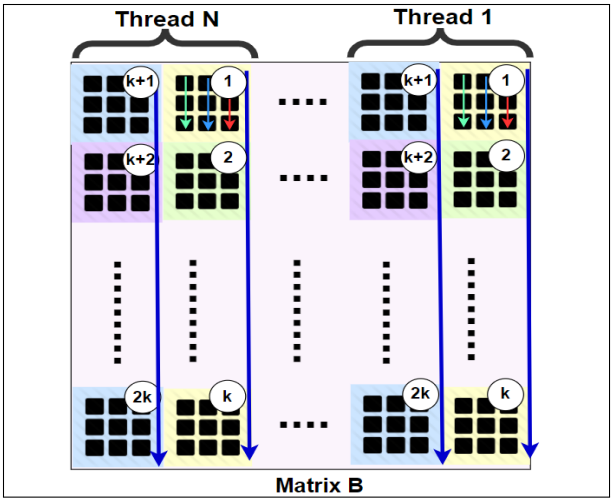

Figure 6: *Thread-wise Memory Access Pattern for Matrix B*

5.1 Performance Analysis

Table 6 and *Table 7* show the trend for run time (in ms) and speedup achieved when number of threads are varied from 2 to 16.

	4K	8K	16K	32K
Reference	298.6	1603.5	7584.9	3162.3
2 Threads	90.4	314.3	1256.9	553.9
4 Threads	59.2	224.6	699.9	327.9
8 Threads	33.3	109.2	489.9	210.8
16 Threads	24.7	89.4	233.1	115.7

Table 6: *Average runtime (in ms) on varying number of threads*

	4K	8K	16K	32K
2 Threads	17.7	5.1	6.0	6.0
4 Threads	27.1	7.1	10.8	11.1
8 Threads	48.1	14.7	15.5	19.8
16 Threads	65.0	17.9	32.5	29.6

Table 7: *Average speedup on varying number of threads*

Scalability : From the comparison chart, it can be observed that there's a consistent rise in speedup as the number of cores are increased. This shows that this multi-threaded implementation is scalable.

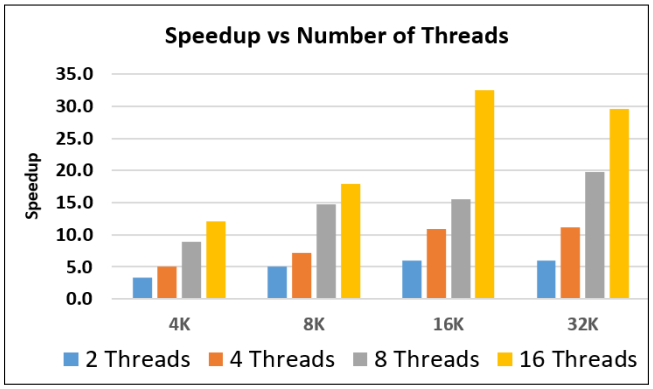

Figure 7: *Speedup comparison across number of threads*

6 Conclusion

The running time and performance of a program depends heavily on the memory access pattern it produces. By optimizing the memory accesses, we can fulfill more requests from caches and hence can have significant improvements in program performance. If the program consists of several independent operations, then we can also take the advantage of multi-threading and distribute the tasks among several threads. Such a multi-threaded implementation, over and above the cache optimizations, can give some really good speedups as is evident from this exercise.

7 References

[1] *Linux Kernel Profiling with perf. Available at https://perf.wiki.kernel.org/index.php/Tutorial*

[2] *Brendan Gregg's blog for perf. Available at http://www.brendangregg.com/perf.html*

[3] *R Documentation at https://www.rdocumentation.org/*