Optimizing Performance of Checkered Matrix Multiplication(CMM)

Arman Gupta (armangupta@iisc.ac.in),

Indian Institute of Science, Bangalore

1. Abstract

Performance Optimisations enhance the execution of the program by identifying the bottlenecks and ensuring proper and efficient use of resources. These bottlenecks often arise from the inefficient memory accesses which leads to lots of cache and tlb misses. In the following sections, we will analyse and will come up efficient optimisation to deal with these bottlenecks.

2. Introduction

Checkered Matrix Multiplication (CMM) is a variant of matrix multiplication. We are given 2 matrices A and B of size N * N where N= 2^k, k is a natural number and k >= 2. We need to produce the output matrix of size (N/2) * N. Following is the explanation of the algorithm:

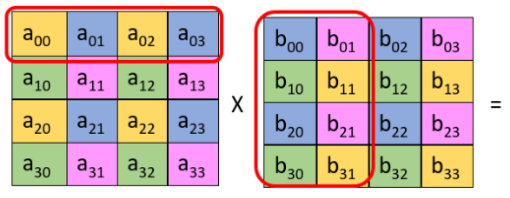
Let us define even and odd positions in matrices A and B. In matrix A, we traverse the elements row-wise, and an element Aij is at even position if j is even and odd otherwise. In matrix B, we traverse the elements column-wise, and an element Bij is even if i is even, odd otherwise.

Let us define columnset in matrix B as a pair of consecutive columns, e.g., 0 th and 1 st columns of B form the 0th columnset, 2nd and 3rd columns of B form the 1st columnset and so on.

We follow a pattern to multiply the matrices as described below. Note that a row in matrix A is multiplied with a columnset in B.

Step 1: Let us take the 0 th row of matrix A (say rowA0) and multiply it with 0 th columnset of matrix B (= {0 th column of B, 1 st column of B}) to get C 00 .

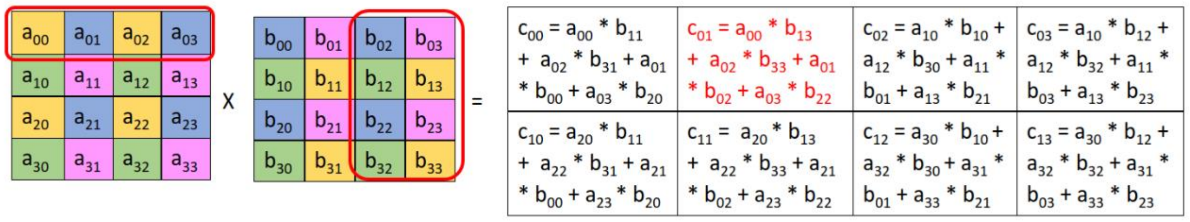
The even elements rowA1 are multiplied with odd elements of 1 st column in the columnset (marked by yellow in figure). And the odd elements of rowA1 are multiplied with even elements of 0 th column in the columnset (marked by blue in figure).


a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
X
b_{00} b_{01} b_{02} b_{03}
b_{10} b_{11} b_{12} b_{13}
b_{20} b_{21} b_{22} b_{23}
b_{30} b_{31} b_{32} b_{33}
=

$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

Step 2: Next, we shift the columnset, it now consists of column 2 and 3. Let us take 0 th row of matrix A (rowA0) and multiply it the 1 st columnset of matrix B to get C 01 .

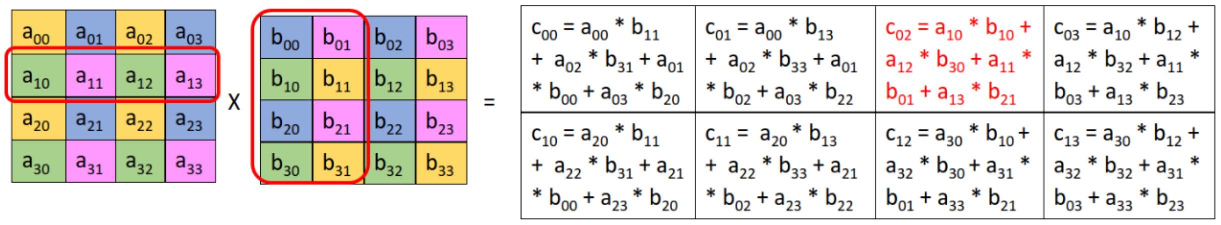
Similar to previous step, the even elements rowA0 are multiplied with odd elements of 1 st column in the columnset, and the odd elements of rowA0 are multiplied with even elements of 0 th column in the columnset.



$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

Note that each row contributes to only half row elements in matrix C, now to obtain the other half elements in this row, we move to the 1 st row in matrix A (say rowA1)

Step 3: We again start from the first columnset i.e., columnset = {0 th column of B, 1 st column of B}. We multiply the 1 st row of matrix A (rowA1) with this columnset.
However, this time, the even elements rowA1 are multiplied with odd elements of 0 th column in the columnset (marked by green in figure). And the odd elements of rowA1 are multiplied with even elements of 1 st column in the columnset (marked by pink in figure).



$c_{00} = a_{00} * b_{11} + a_{02} * b_{31} + a_{01} * b_{00} + a_{03} * b_{20}$	$c_{01} = a_{00} * b_{13} + a_{02} * b_{33} + a_{01} * b_{02} + a_{03} * b_{22}$	$c_{02} = a_{10} * b_{10} + a_{12} * b_{30} + a_{11} * b_{01} + a_{13} * b_{21}$	$c_{03} = a_{10} * b_{12} + a_{12} * b_{32} + a_{11} * b_{03} + a_{13} * b_{23}$
$c_{10} = a_{20} * b_{11} + a_{22} * b_{31} + a_{21} * b_{00} + a_{23} * b_{20}$	$c_{11} = a_{20} * b_{13} + a_{22} * b_{33} + a_{21} * b_{02} + a_{23} * b_{22}$	$c_{12} = a_{30} * b_{10} + a_{32} * b_{30} + a_{31} * b_{01} + a_{33} * b_{21}$	$c_{13} = a_{30} * b_{12} + a_{32} * b_{32} + a_{31} * b_{03} + a_{33} * b_{23}$

Please note that, every row odd numbered row (o-based indexing), follows the same rule as A1. Whereas, every even numbered row, follows the same rule as rowA0 mentioned previously.
Also, the product of the i th row of matrix A and the j th column of matrix B will be placed in the cell C[floor(i/2)][j] if i is even and in C[floor(i/2)][j+N/2] if i is odd.

3. Processor, Tools and Techniques

The codes are run on Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz having 4 physical cores with 2 threads each. The performance analysis is done using the perf tool. Each of the code are run separately 5 times for each Input sets and the data shown here in this report is average of 5 runs. For collecting the performance analysis data, we have disabled the caches and tlbs warmup and checking of the correctness of result. We have used Input set of size 512, 1024,2048 and 4096 for our analysis.

4. Reference Checkered Matrix Multiplication (Unoptimized Single Thread)

We are considering the code provided to us as reference code. Upon running the perf tool on the reference code for input set of size 512, 1K, 2K and 4K, we found out the following bottlenecks:

Input Size	**512**		**1K**		**2K**		**4K**	
Bottleneck	**Mean**	**S.D**	**Mean**	**S.D**	**Mean**	**S.D**	**Mean**	**S.D**
L1 dcache Load misses	58.2M	0.68M	531.4M	1M	6.85G	5.77M	73.3G	15.17M
LLC loads	72.6M	1.52M	543.3M	6.92M	4.35G	6.82M	35.4G	300.15M
D-TLB load misses	345	58	5686	1361	4.3G	2.5M	34.37G	3.19 M

Table 1: Bottlenecks in the Reference Code

Now we see the bottlenecks in the code, and we need to optimise the code in order to reduce the above performance issues. We will see the two optimisation techniques to deal with the above issues.

5. Single Threaded Checkered Matrix Multiplication

In the single threaded code optimisation, our goal is to reduce the above-mentioned bottlenecks and improves the execution time of the reference code. We are suggesting two optimised code-

1. **Transposing the matrix A and traversing the matrix B is Row Major Order:**
   In this optimised code, we have transposed the matrix A and formed the new matrix A' and then multiplied matrix A' with matrix B by accessing both the array in Row Major Order. Using this technique, each element of the output matrix is updated simultaneously. Consider the following Matrix A' and Matrix B :

a00	a10	a20	a30
a01	a11	a21	a31
a02	a12	a22	a32
a03	a13	a23	a33

b00	b01	b02	b03
b10	b11	b12	b13
b20	b21	b22	b23
b30	b31	b32	b33

Now, we will update the output matrix C in the following way:

(i) We will take row 0 of A' and multiplied with the relevant element of row 1 in matrix B such that it satisfies semantics of CMM i.e. elements at even indices of row 0 of A' are multiplied with elements at odd indices of row 1 of B and elements at odd indices of row 0 of A' are multiplied with elements at even indices of row 1 of B. After performing this step, we get C matrix as

a00 * b11	a00 * b13	a10 * b10	a10 * b12
a20 * b11	a20 *b13	a30 * b10	a30 * b12

(ii) Now we will take row 1 of A' and multiplied with the relevant element of row 0 in matrix B i.e. elements at even indices of row 1 of A' are multiplied with elements at even indices of row 0 of B and elements at odd indices of row 1 of A' are multiplied with

elements at odd indices of row 0 of B and these terms are added to respective elements in output matrix C. After performing this
step, we get C matrix as

a00 * b11 + a01* b00	a00 * b13 + a01 * b02	a10 * b10 +a11*b01	a10 * b12 +a11*b03
a20 * b11 + a21 *b00	a20 *b13 +a21 *b02	a30 * b10 + a31*b01	a30 * b12 +a31*b03

(iii) Now we will take row 2 of A' and multiplied with the relevant element of row 3 in matrix B similar to what we did in step (i) and add those terms to respective elements of output matrix C. Now, the matrix C is

a00 * b11 + a01* b00 + a02 * b31	a00 * b13 + a01 * b02 + a02 * b33	a10*b10+a11*b01+a12 * b30	a10 * b12 +a11*b03+ a12 * b32
a20 * b11 + a21 *b00 +a22 * b31	a20 *b13 +a21 *b02+ a22*b33	a30 * b10 + a31*b01+ a32 * b30	a30 * b12 +a31*b03+ a32 * b32

(iv) Now we will take the last row 3 of A' and multiplied with the relevant element of row 2 in matrix B like what we did in step (ii) and add those terms to respective elements output matrix C. Now, the matrix C is

a00 * b11 + a01* b00 + a02 * b31 + a03 * b20	a00 * b13 + a01 * b02 + a02*b33 + a03 * b22	a10 * b10 +a11*b01 +a12*b30+a13 * b21	a10 * b12 +a11*b03+ a12 * b32 + a13 * b23
a20 * b11 + a21 *b00 + a22 * b31 + a23 * b20	a20 *b13 +a21 *b02+ a22 * b33 + a23* b22	a30 * b10 + a31*b01 + a32*b30+a33* b21	a30 * b12 +a31*b03+ a32 * b32 + a33 * b23

Note that when the row number x of A' is even we are considering the (x+1)th row of B and when row number x of A' is odd we are considering the (x-1)th row of B.
Thus we can see that we get the expected output matrix but here all the elements of the output matrix are updated simultaneously.

2. **Transposing the matrix B :**
In this optimised code, we have transposed the matrix B as the Matrix B is being accessed in the Column major order in the unoptimized code but the C++ language specifications state that arrays are laid out in memory in a row-major order i.e., the elements of the first row are laid out consecutively in memory, followed by the elements of the second row, and so on. As a result, in order to maximize performance in C++, the code should access multi-dimensional arrays using a row-major order. Now elements which are accessed in the column major order in matrix B, will now be accessed in row major order in transpose of matrix B. This code is similar to reference code, we have just changed the order of access of the matrix B.

6. Performance Comparison

The two suggested optimised codes will be compared with the reference code based on their execution time and performance counters like L1 Dcache Load misses, LLC loads and D-TLB load misses.

Input Size	**512**		**1K**		**2K**		**4K**	
Code	Mean	S.D	Mean	S.D	Mean	S.D	Mean	S.D
Reference Code	58.2M	0.68M	531.4M	1M	6.85G	5.77M	73.3G	15.17M
Optimised Code 1	3.69M	0.02M	33.94M	0.26M	278.3 M	0.675M	2.32G	42.5M
Optimised Code 2	7.26M	0.21M	67.3M	0.58M	550.7M	1.75M	4.8G	38.5M

Table 2: Comparison using L1 Dcache Load Misses

Input Size	**512**		**1K**		**2K**		**4K**	
Code	Mean	S.D	Mean	S.D	Mean	S.D	Mean	S.D
Reference Code	72.6M	1.52M	543.3M	6.92M	4.35G	6.82M	35.4G	300.15M
Optimised Code 1	1.21M	0.20M	5.64M	0.176M	9.62M	0.25M	71.9M	5.07M
Optimised Code 2	1.38M	0.06M	5.3M	0.91M	14.38M	0.67M	121.9M	13.17M

Table 3: Comparison using number of LLC load requests

Input Size	**512**		**1K**		**2K**		**4K**	
Code	Mean	S.D	Mean	S.D	Mean	S.D	Mean	S.D
Reference Code	345	58	5686	1361	4.3G	2.5M	34.37G	3.19 M
Optimised Code 1	471	55	14389	268	5.52M	0.76M	27M	4M
Optimised Code 2	492	85.6	15312	321	11.82M	0.806	85.16M	0.67M

Table 4: Comparison using DTLB load misses

Input Size	**512**	**1K**	**2K**	**4k**
Code	**Average**	**Average**	**Average**	**Average**
Reference Code	256.475	2017.3	103884.4	954188.25
Optimised Code 1	179.37	1402.8	10785.76	78916.86
Optimised Code 2	157.5	1233.85	10436.28	83714.02

Table 5: Average Execution time(ms)

From above tables 2,3,4 and 5, we can observe that both optimised codes are performing better than the reference code by a very large margin. We can also observe that Optimised Code 1 seems to be more efficient than the Optimised Code 2 but still the running time of both code seems comparable for the above inputs because the minor page faults in case of Optimized Code 1 is more than that in case of Optimised Code 2 and based on our observation, Optimised Code 1 seems to perform well for larger input set of size 4K, 8k and 16k but the difference between execution time is not much significant.

Input Size	**512**		**1K**		**2K**		**4K**	
Code	Mean	S.D	Mean	S.D	Mean	S.D	Mean	S.D
Optimised Code 1	1157.2	1.09	4227	1.1	16514.7	0.9	65665.7	0.3
Optimised Code 2	1029.4	0.9	3717.2	1.64	14468.8	0.8	57475.6	0.54

Table 6 : Comparison using Minor Page Fault

We have implemented the optimised code 2 for the submission of this assignment.

7. Multi-Threaded Checkered Matrix Multiplication

For implementing the Multi-Threaded CMM, we have used the Optimised Code 2. We can observe that calculating the elements of output array is mutually exclusive operation hence we can calculate the elements in parallel using threads and then combine the result to form the single output array. We have implemented the multithreaded code in the following way

(a) Divide the Matrix A into subsection of rows according to thread id and then that thread will cover the multiplication of that subsection of A with the matrix B and store the partial result in output matrix C.

(b) All the results from each thread will be combined into single output matrix.

Performance Analysis

Input Size	**512**	**1K**	**2K**	**4k**
Code	**Average**	**Average**	**Average**	**Average**
Reference	256.475	2017.3	103884.4	954188.25
1 thread	157.5	1233.85	10436.28	83714.02
2 threads	83.6022	630.408	5605.93	44626.3
4 threads	42.688	375.29	3152.56	24805.76
8 threads	39.554	362.73	2948.24	23130.72

Table 7 : Average Execution time(ms)

Let’s test the scalability of the Multithreaded CMM. First, we will calculate the speedup of each code with respect to given Reference Code.

Speedup = Execution time of Reference Code / Execution time of the referred Code

We get the following table.

Input Size	**512**	**1K**	**2K**	**4k**
Code	**Average**	**Average**	**Average**	**Average**
1 thread	1.63	1.63	9.95	11.398
2 threads	3.07	3.2	18.53	21.38
4 threads	6.01	5.37	32.95	38.47
8 threads	6.49	5.56	35.23	41.25

Table 8: Speedup of the different code w.r.t Reference Code

From table 8, we can observe that there is approx. 2x speedup going from single thread to 2 threads and similar trend is observed going from 2 threads to 4 threads, but this trend seems to fail while going from 4 threads to 8 threads. The possible reason for this trend failure may be because the machine we are using has 4 physical cores with 2 threads each hence we have total 8 logical cores, but it seems like these extra 4 logical cores are not much helpful here because algorithm is doing better job at keeping CPU busy.
If the processor's execution resources are already well utilized, then there is little to be gained by enabling Intel HT Technology. For instance, code that already can execute four instructions per cycle will not increase performance when running with Intel HT Technology enabled, as the process core can only execute a maximum of four instructions per cycle.
One more point to be noted that as an increasing number of threads or processes share the limited resources of cache capacity and memory bandwidth, the scalability of a threaded application can become constrained. Memory-intensive threaded applications can suffer from memory bandwidth saturation as more threads are introduced. In such cases, the threaded application won’t scale as expected, and performance can be reduced.

8. Conclusion

In majority of applications, the major concern is how to access memory efficiently. The inefficient memory access pattern leads to degraded performance of the application. The performance analysis of the programs helps us identity the bottlenecks. Improving the memory access pattern and other bottlenecks lead to less execution time as we have shown in this experiment. Moreover, if there are independent set of instructions in the program, we can form threads out of these instructions and can run parallelly. Thus, improving the execution time of program even further by utilising the multiple cores.

9. Reference

[1] Perf Tutorial https://perf.wiki.kernel.org/index.php/Tutorial
[2] Performance insights to intel hyper threading technology
http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/
[3] Intel Guide for Developing Multithreaded Applications
http://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications/