HPCA Programming Assignment 2021-2022-PART A
Optimizing performance of Checkered Matrix Multiplication

Submitted by Mayank Sati

5 November 2021

1 Introduction

Hardware Specification : Intel Core i7 7500U Processor 2 Core / 4 Threads, base frequency 2.70GHz and Max Turbo Frequency 3.50 GHz. Physical Memory: 8GB. L1 Cache Size 48 KB, 12-way Set Associative. L2 Cache size 512KB 8-way Set Associative. L3 Cache size 8MB, 16-way Set Associative.
Most of the Checkered Matrix Multiplication (CMM) run time analysis is done by keeping the base input set N as 2048.
The structure of the report is as follows:
Section 2 discusses about the Checkered Matrix Multiplication and how the bottlenecks were identified and optimisations were made. Section 3 shows the data and speedup obtained by applying optimisations on single threaded code. Section 4 discussed about multi threaded CMM and the scalability of the same with varying number of threads.

2 Optimisation in CMM

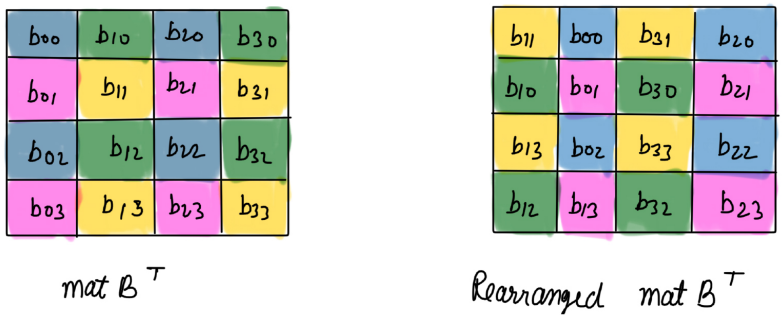
A normal Matrix Multiplication A*B accesses the A matrix's i^{th} row and B matrix's j^{th} column in order to compute the i,j cell of output matrix making its time complexity proportional to $O(n^3)$. The 'checkered' multiplication in CMM gets more complex as it accesses 2 rows of A and 2 columns of B in a checkered pattern in order to compute i and j cell of output.
As N gets large, since j is accessed column wise and matrix is stored as a row major order, each different access for j can be present in different cache line and hence making the pipeline do significant amount of work just to load the cache lines into the cache hierarchy making it a performance bottleneck (backed by the data as well).
A simple way to reduce the accesses would be to transpose the B matrix (matB) for a normal multiplication, but in CMM, this can lead to inconsistent result in output matrix so I did some modifications in matB to ensure that the heavy duty $O(n^3)$ matrix multiplication accesses both matrix row wise (hence reducing the cache accesses to different blocks on every iteration).
First, I simply computed the transpose of matB. Then in order to maintain the correct result, I applied some changes in transposed matrix $matB^T$ as:
For a 4*4 block in $matB^T$, i.e cell entries [i,j], [i,j+1], [i+1,j], [i+1,j+1] of matB where i and j are even, I did the following rearrangement:

- Shift matB[i,j] to matB[i,j+1]
- Shift matB[i+1,j+1] to matB[i,j]
- Shift matB[i,j+1] to matB[i+1,j]
- Shift matB[i+1,j] to matB[i+1,j+1]

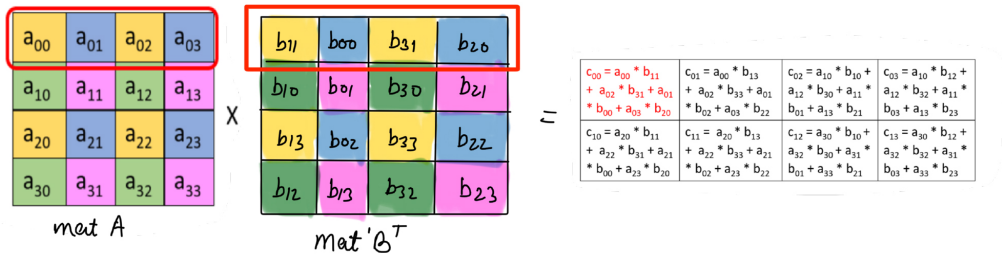
The code that does the above rearrangement is given as:

```
for(int i=0;i<N;i+=2)
{
    for(int j=0;j<N;j+=2)
    {
        t=matB[i*N+j+1];
        matB[i*N+j+1]=matB[i*N+j];
        matB[i*N+j]=matB[(i+1)*(N)+j+1];
        matB[(i+1)*(N)+j+1]=matB[(i+1)*(N)+j];
        matB[(i+1)*(N)+j]=t;
    }
}
```

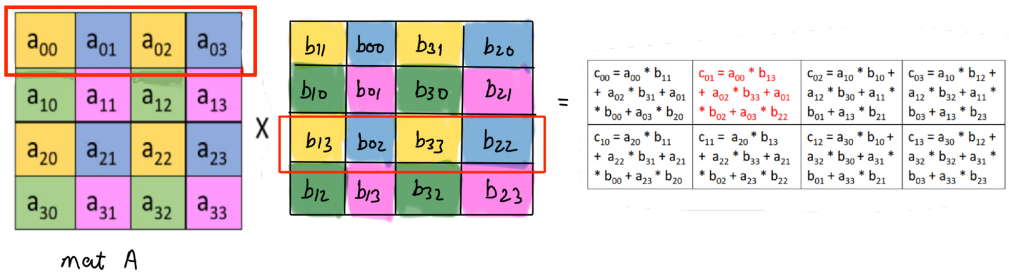
The rearranged matB^T compared to original matB^T will look like:



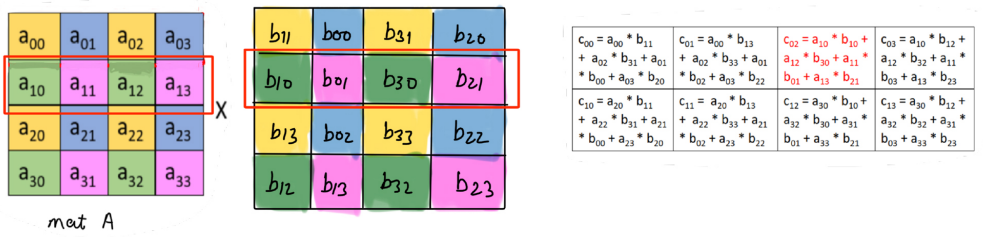
Now, if we do CMM to get output matrix, we will be accessing both the matA and rearranged matB^T in row major order which is optimal way to do matrix multiplication. CMM will now look like:



Output Matrix entry [0,0]



Output Matrix entry [0,1]



Output Matrix entry [0,2]

And so on..

By applying this transformation, CMM can now happen row major wise without any inconsistency in the data. Using this re-arrangement accounts for extra $O(n^2)$ computation, but the trade off is that the heavy duty $O(n^3)$ multiplication is done optimally and we will later see that this transformation greatly increases the speedup of the program with respect to naiive CMM.
Here is the Code for Matrix Multiplication:

```
  //for 1st half
   for(int i=0;i<N;i+=2)//row of matA
   {
       for(int j=0;j<N;j+=2)//row of rearranged matB
       {
           output[(i/2)*N+j/2]=0;
           for(int k=0;k<N;k++)
           {
               output[(i/2)*N+j/2]+=matA[i*N+k]*matB[j*N+k];
           }
       }
   }

//for 2nd half
   for(int i=1;i<N;i+=2)//row of matA
   {
       for(int j=1;j<N;j+=2)//row of rearranged matB
       {
           output[(i/2)*N+N/2+j/2]=0;
           for(int k=0;k<N;k++)
           {
               output[(i/2)*N+N/2+j/2]+=matA[i*N+k]*matB[j*N+k];
           }
       }
   }
```

2.1 Bottlenecks in Naiive CMM:

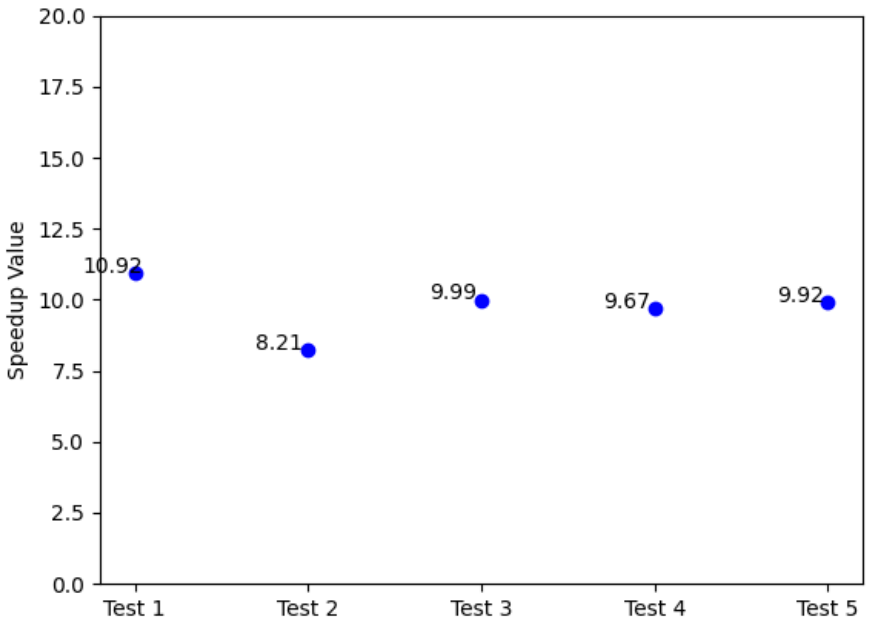
As stated earlier, in naiive CMM, matB is accessed column wise and could lead to a cache miss for every column access, making it a heavily memory bound task.
For N=2048, L1-dcache-load-misses = 12.12% of all L1-dcache accesses
For N=4096, L1-dcache-load-misses = 16.31% of all L1-dcache accesses

3 [PartA-I] Single-Threaded CMM Report:

As discussed earlier, I re-arranged matB in order to perform the heavy duty $O(n^3)$ multiplication so that elements are accessed row major wise. Apart from that, I did loop unrolling 16 times in order to reduce the branch calculation and further optimise our code. Performance Analysis for Optimised Single Threaded code against un-optimised code for N=2048 is given as:

	Unoptimised CMM	**Optimised CMM**
Cycles	5,78,16,08,78,570	3,91,93,15,38,718
Cycles(Relative)	100%	67.78%
IPC	0.54	0.83
IPC(Relative)	100%	153.7%
L1-D$-load-misses (of all L1D$ accesses)	12.12%	8.80%
L1-D$-load-misses(Relative)	100%	72.60%
Execution Time(ms)	95036.4	8699.02
Execution Time(Relative)	100%	9.15%

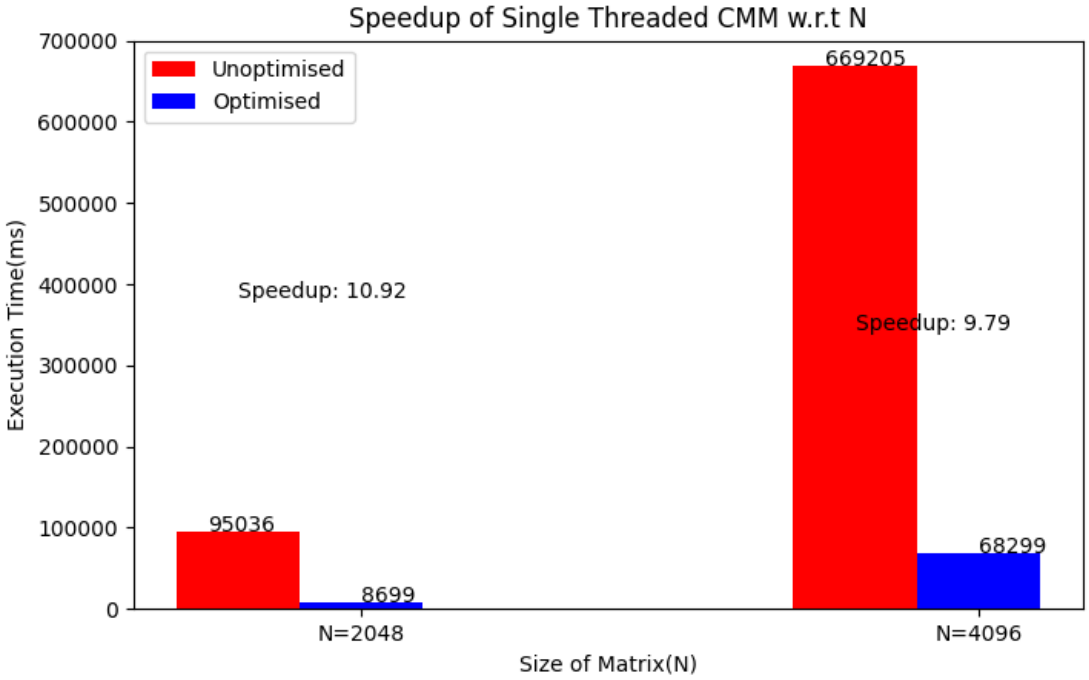
Speedup = Reference execution time/Single threaded execution time.
For N=2048, speedup measured on 5 runs is reported as:



Average Speedup = 9.742

Coming to performance bottlenecks, the optimised code reduces L1-D cache load misses by 27.4% for N=2048 which is the root cause of getting an average speedup of 9.74.

Comparision of speedup for N=2048 and N=4096 is given as:



NOTE: The speedup obtained (and all the other stats, in general) is aggregated value as the main.cpp program runs reference code 2 times prior to executing the single threaded and multithreaded code, hence the perf command will count the data for the whole program and will not give the exact analysis (but is comparable) hence the speedup for N=2048 will be more than 9.742. I decided not to modify the main.cpp (stated clearly not to modify), and provide report for aggregated perf statistics.

4 [PartA-II] Multi-Threaded CMM (CPU)

Single threaded CMM can be further optimised by scheduling independent tasks to different threads for execution. I analysed the parts of optimised single threaded CMM code that could be scheduled in parallel and varied the number of threads in order to attain maximum speedup.

4.1 Identifying Parts of the Code that can run in Parallel

I calculated transpose of matrix matB in a single thread as its outer and inner loops are independent and could lead to synchronisation problem later if implemented in a mltithreaded environment. So I focused on parallelising the rearrangement of $matB^T$ and the Checkered Matrix Multiplication itself.
The rearrangement code for $matB^T$ (described earlier) can be executed independently in parallel by only changing the outer loop each thread executes.
Likewise, CMM can also run in parallel by executing different set of disjoint outermost loop values to different threads.

4.2 Multi-Threaded CMM (CPU) Report:

Here, I analysed the speedup of CNN by varying the number of threads and tried to obtain the optimal number of threads that would maximise the speedup. The following data is obtained by keeping N=2048:

		Optimised CMM			
	Un-Optimised CMM	**Threads=2**	**Threads=4**	**Threads=8**	**Threads=16**
Cycles	5,78,16,08,78,570	4,11,94,17,72,863	4,62,34,60,27,029	4,62,34,77,51,217	4,59,98,92,45,559
Cycles(Relative)	100%	71.25%	79.97%	79.97 %	79.56%
IPC	0.54	1.06	0.94	0.94	0.95
IPC(Relative)	100%	196.29%	174.07%	174.07%	175.92%
L1-D$-load-misses (of all L1D$ accesses)	12.12%	6.95%	6.96%	6.98%	6.99%
L1-D$-load-misses(Relative)	100%	57.34%	57.42%	57.59%	57.67%
Execution Time(ms)	95036.4	5114.74	4297.43	4279.78	4333.4
Execution Time(Relative)	100%	5.38%	4.52%	4.50%	4.55%
Context Switches	2,005	1,728	9,814	6,729	6,980
Context Switches(Relative)	100%	86.18%	489.42%	335.61%	348.13%
CPU Migrations*	96	142	160	134	143
CPU Migrations (Relative)	100%	147.9%	166.6%	139.59%	148.9%

*CPU migrations reports the number of times the process has migrated to a new CPU.

Speedup of Multithreaded CMM on different number of threads is shown in following figures:

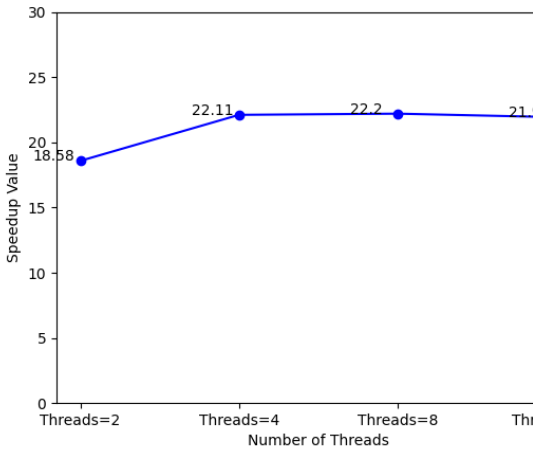


Figure 1: Speedup w.r.t Unoptimised CMM

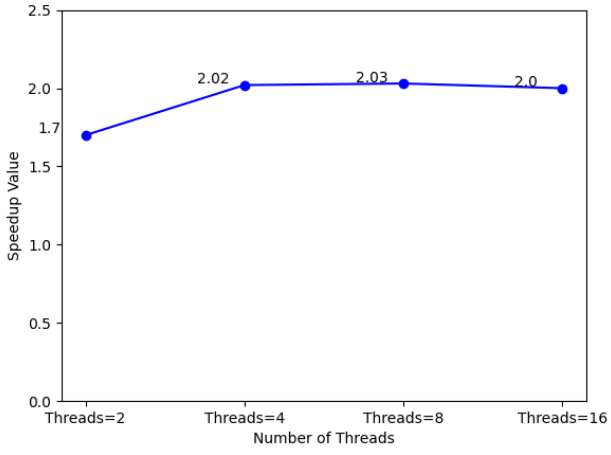


Figure 2: Speedup w.r.t Optimised Single Threaded CMM

The maximum speedup is obtained when the number of threads is 8. The trend of speedup increasing by 2 as threads are doubles fails quickly and speedup comes to a steady value of 2 when threads are 4 or more. The reason behind this is that now, other bottlenecks start to arise which were not present in single threaded CMM like CPU migrations and context switches which seems to dominate as number threads are increased above 8.

5 Conclusion

In spite of CMM being heavily 'scattered' matrix multiplication, the report shows how a simple modification can optimise CMM to obtain speedup of around 10 in a single threaded program. This speedup almost doubles to around 22 when we exploit the concept of multithreading. Although the CMM has a running time complexity proportional to $O(n^3)$, but on applying some optimisations, a significant amount of time can be saved to get a lower running time of the problem.

6 References

[1]https://www.brendangregg.com/perf.html
[2]https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
[3]https://www.softprayog.in/programming/posix-threads-programming-in-c
[4]https://matplotlib.org/stable/users/index.html