Assignment 3

High-performance computer Architecture

M.V.S. Ajay | Sr No: 20877

Gulshan Kumar Gabel | Sr No: 21130

GitHub link: https://github.com/gkgabel/HPCA-course-assignment-2022

Experimental Setup:

Machine Specification

For the single threaded and the multi-threaded executions, we have used a system with Intel x86 64-bit ISA with 6 cores and 2 threads per core. The L1D cache size is 288 KB, L1D cache size is 192 KB, L2 cache size is 3MB AND LL Cache size as 12 MB. For the GPU version of the program, we have used the CSA department's Wells Fargo GPU Server with 128 Cores and 256 hardware threads with AMD EPYC 7713 64-Core Processor.

Statistics Used

We use the following hardware statistics to support our arguments and judge the program's performance. DTLB Load misses, L1D cache misses, LLC misses including loads and stores, and PU time (time with which program is in running state on CPU). All these statistics have been recorded using the perf tool single and multi-threaded program. However, the statistics for Cuda program have been recorded using the Nvidia Nsight analysis tool.

We have also used the speed up to quantify the performance of a version. The speed up is calculated using the following formula.

$$Speed\ Up\ = \frac{Runtime\ of\ optimized\ program}{Runtime\ of\ unoptimized\ program}$$

However, the runtime and other statistics for the multithreaded program have been calculated as:

$$Statistic\ of\ multithreaded\ program\ = \max\ \ (statistic\ recorded\ for\ each\ thread)$$

Also, the runtime for Cuda program considers the runtime for Cuda kernels only.

***

The assignment is divided into two parts. We have presented the results for single-threaded and multi-threaded in a combined form since the multi-threaded program is a further optimization of single threaded program.

Part A

1. Optimized single-threaded implementation of RMM on the CPU.

2. Optimized multi-threaded version of RMM on CPU using pthreads

Our solution optimizes a single-threaded implementation of RMM and then further applies multithreading on this code. In RMM, the arithmetic operations are independent and similar operations are performed on each row and column. Hence, there is an enormous potential for exploiting this independence and the inherent data parallelism.

Here is a list of optimizations (with rationale) that we have used to speed up the single-threaded version of the program.

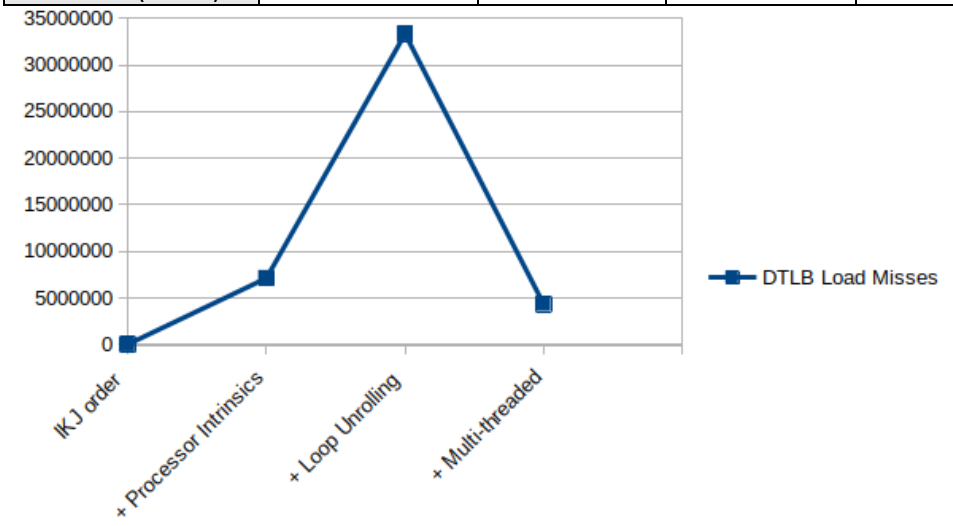
1. We change the loop ordering to ‘’ikj” instead of the naive “ijk” order to take advantage of the cache locality to improve the latency of load and store operations.
2. We used Single instruction multiple data (SIMD) instructions available in the Intel x86 ISA to exploit the data level parallelism present in the code. We used Advanced Vector Extension (AVX) 256-bit integer instructions which can perform eight integer operations simultaneously using 256-bit YMM registers. This introduces a level of data parallelism in the program.
3. Further, for all the programs, we used loop unrolling to exploit the instruction level parallelism (ILP) in the program since the number of branch instructions is less. Branch instruction hinders the out-of-order execution of the loop body, irrespective of the prediction accuracy. Also, this is a significant optimization since we are reducing the number of stores in the memory by reducing the number of iterations. So, we expect a performance increase by unrolling the loops. We also have minimized the number of unnecessary arithmetic computations to save CPU time.
4. In this optimization, we have divided the RMM function among multiple threads to exploit the thread-level parallelism that is given by the presence of multiple cores. For this, we used the pthread library. We have tested the code with a different number of threads. We divide the work of multiplying rows of the input matrix A among different threads equally. So, for each thread, the first loop of multiplication runs its share of rows independently. Here no two threads write to the same cell in matrix C, so we do not have to take care of synchronization among threads.

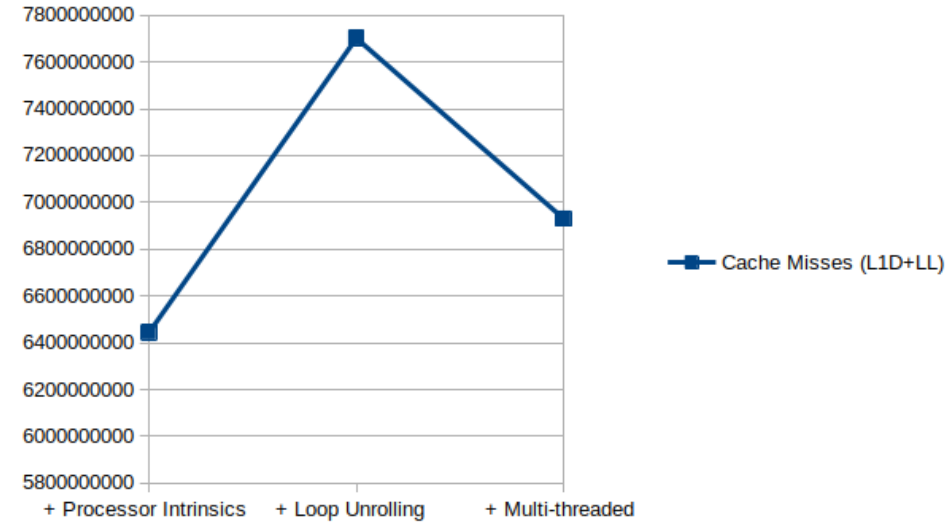
Results and Observations

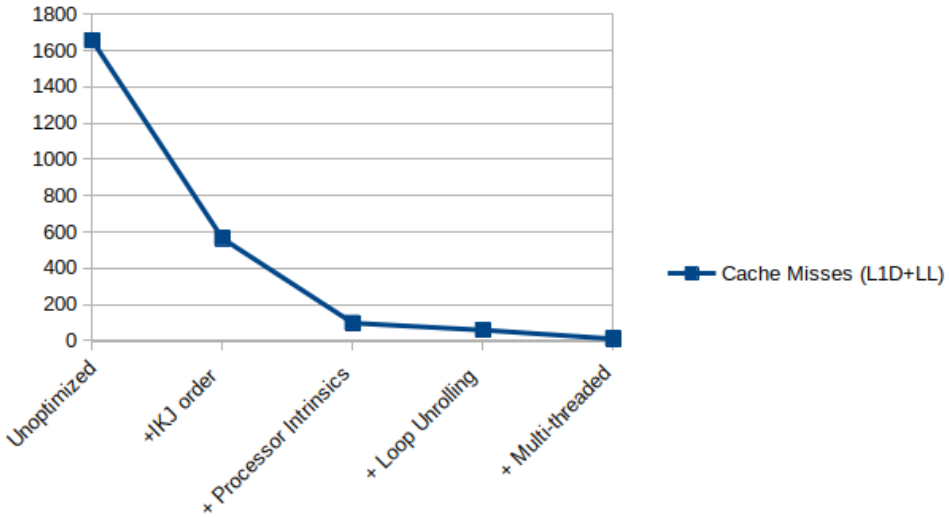
The results from the experiment for after applying each optimization have been tabulated below followed by the set of observations for each experiment.

CPU Statistics for RMM on Matrix size 8192x8192 with different optimizations					
	Unoptimized	IKJ order	IKJ + Processor Intrinsics	IKJ + Processor Intrinsics + Loop Unrolling	Multi-threaded + IKJ + Processor Intrinsics + Loop Unrolling
DTLB Load Misses	48583660019	47634	7143326	33321189	4357188
L1D Misses	143220905080	6441910797	7694486717	6895815210	601266866

LL Load Misses	30645656045	1422562	6992743	34459279	7626725
LL Store Misses	212015	1276	2104	4310	3661
CPU time (in sec)	1655.101	564.314	97.989	59.411	11.444







From the above results, we make the following observations.

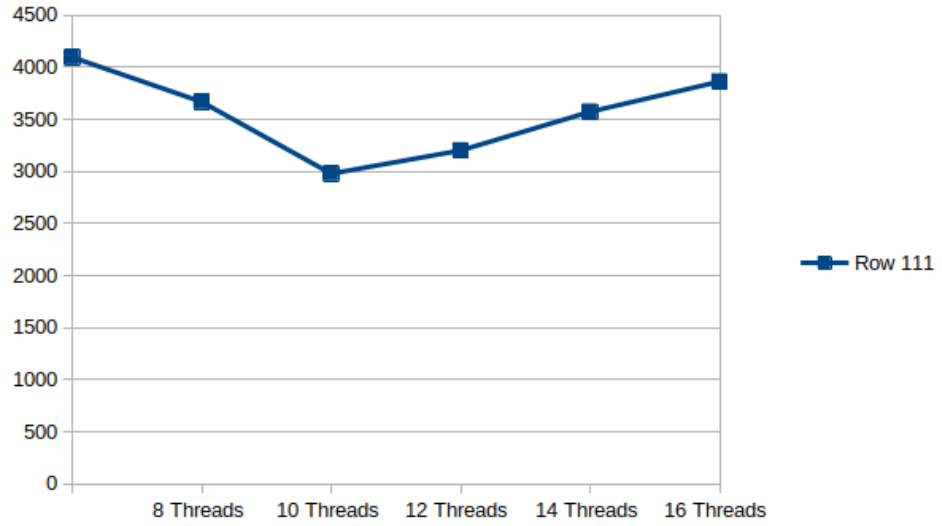
1. The number of DTLB misses decreases for IKJ orders as we go from unoptimized to IKJ order. This creates the best access pattern in terms of locality among all the optimizations. We see that even with processor Intrinsics DTLB misses increase and worsen when we do loop unrolling. This is because when we add processor intrinsic instruction to our program, we load multiple rows at a time. Since the the matrix size is 8192 in the above experiment which means that every row is of size 32K Bytes which is larger than the page size of 4 K Bytes, so there are more TLB misses in this code. Also once we unroll the loop, we further increase the load operations ro each time by 4 fold in our program and hence this leads to more DTLB misses. But we see that the number of DTLB misses reduce for multithreaded programs. Now here since we have a shared load on each core's TLBs we see that DTLB misses reduce per TLB. This in turn, also implies a lesser overhead in page translation in the multithread programs due to better resource utilisation. So we see that the paging overhead is the highest in a fully optimized single-threaded version of our program.
2. From the results, we observe that the unoptimized version has the greatest number of cache misses. However, we see that just changing the order of multiplication reduces the cache misses by a huge proportion. Again, we see a sharp increase in cache misses in case of the addition of processor intrinsic instructions. This is due to the same effect observed for DTLB, which is the accessing of multiple rows in our program for loop these optimizations which are much larger than page size and cache line sizes. Again, we see a decline in the cache miss for the multithreaded program due to load division on each core's cache.
3. Considering the CPU times for each optimization we see that each subsequent optimization reduces the runtime for the code. As expected loop unrolling improves performance by a significant fraction due to the reasons mentioned above. The most significant improvements however are seen when we use processor intrinsic vectorization instruction which decrease the CPU runtime of program by more than 5 times. Further we can see that multithreading again achieves another 5 fold improvement on the single threaded implementation.

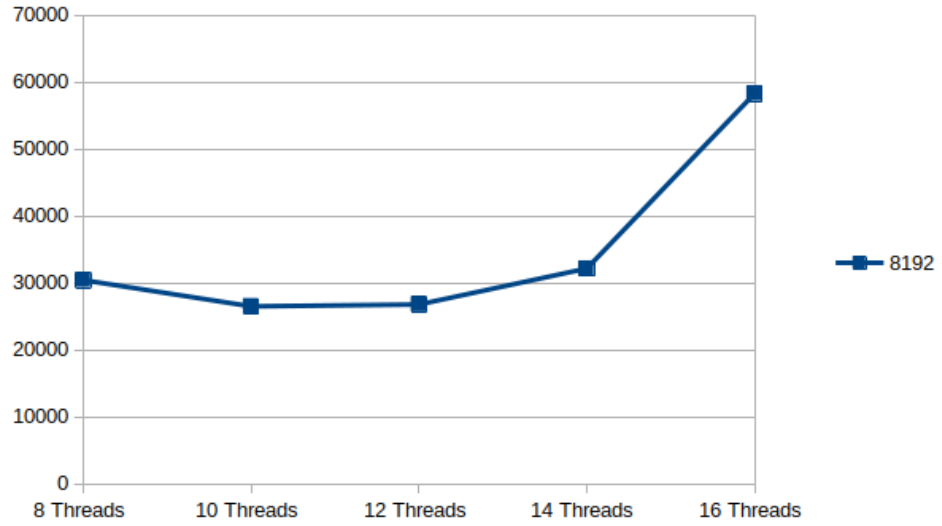
From above results we can see that although our use of AVX intrinsics instruction worsens the address translation overhead as well the cache performance but it fills this performance gap by parallelizing the operations. Similarly, multi-threaded version exploits the thread level parallelism in the program and shares the load of the program to multiple cores. However, there is still an overhead of thread creation involved which becomes less visible for large matrices.

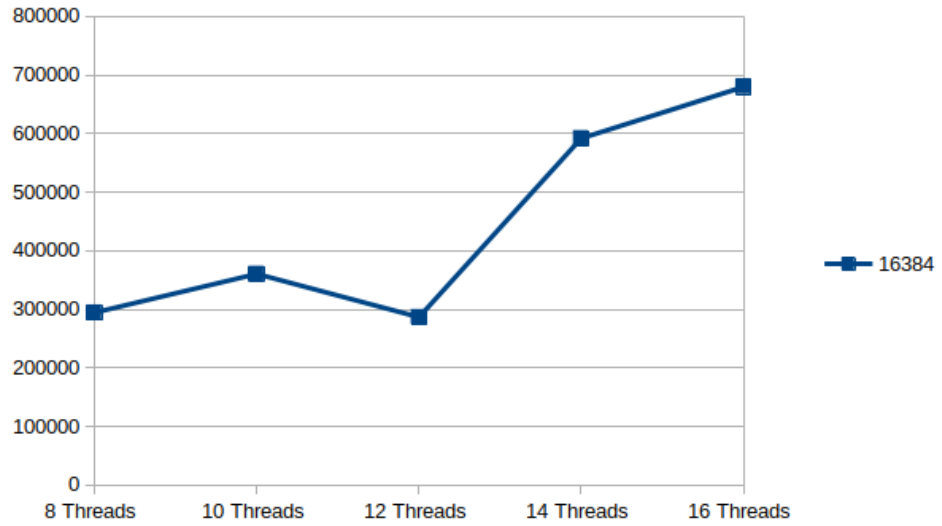
We ran the multi-threaded program for RMM for different thread counts. The results have been mentioned in the following table.

Multithreading time for different threads and varying matrix sizes					
Matrix size	Multithreaded (ms)				
	8 Threads	10 Threads	12 Threads	14 Threads	16 Threads

16	0.511	0.240	0.250	0.390	0.434
4096	3668.3	2979.6	3202.4	3573.6	3863.7
8192	30439.8	26527.7	26805.6	32177.9	58289.8
16384	294140.0	360186.0	286443.0	591580.0	679472.0



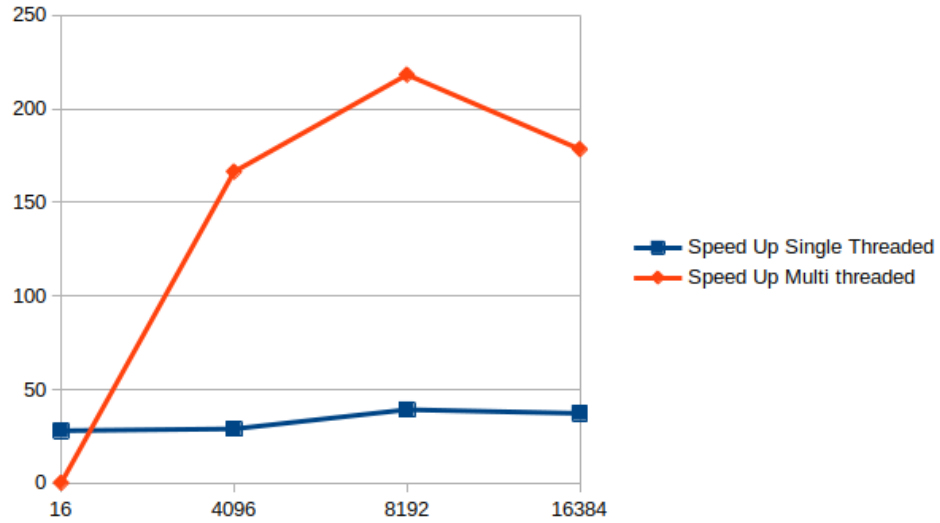




From the above data we can see that we can see that we get the best results when the program is run with thread count of 10 or 12 for smaller matrix but for a matrix of very large size of 16K we see that 12 threads give the best run time. This is expected since our CPU can run 12 parallel threads due to presence of 6 cores which are 2 simultaneous multithreaded. We see that as we increase the number of threads the performance again starts degrading. This may be due to the overhead incurred by the context switches for multiple threads.

The following table indicates our results for final optimized versions of single threaded version and multi-threaded version and compares their speed up with the unoptimized version the program.

Speed up obtained on single-threading and multi-threading on varying matrix sizes					
Matrix size	Unoptimized	Single-threaded		Multi-threaded with 12 threads	
	Time (ms)	Time (ms)	Speed Up	Time (ms)	Speed Up
16	0.028	0.001	28.0	0.250	0.1
4096	533015	18402.7	29.0	3202.4	166.4
8192	5845320	149209	39.2	26805.6	218.2
16384	51069200	1371210	37.3	286443.0	178.4



We observe that the speed up of the optimized program increases with the matrix size. This may be because the effect of overhead incurred by using the intrinsic instruction becomes less pronounced as the number of computations increases. Similarly for large matrix sizes the speed up for multithreaded execution keeps on increasing until 8192. However, we see a small dip in speedup as we move from the 8K matrix to the 16K matrix. This may be due to the increase in cache misses and page translation overhead for larger matrices.

Bottlenecks

We observe that there remain few bottlenecks in the program which limit the scalability of the code.

1. There remains a limit to the speedup which AVX instructions can provide since these instructions can only do a certain number of parallel computations. There remains a large unexploited potential in terms of the thread-level performance of this

program. An issue with the use of these vector intrinsic instructions can be their inflexibility since many instructions require the operands to be aligned in memory.

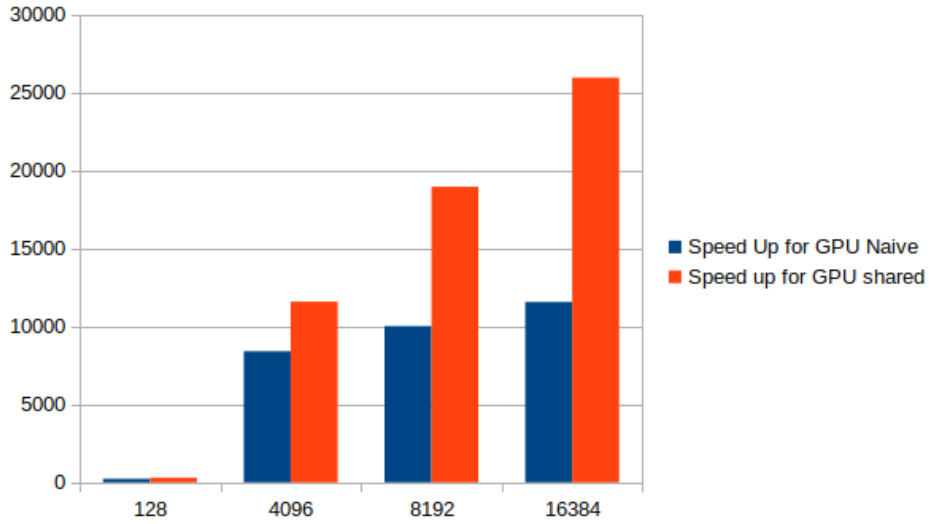
2. While multithreading can speed up the code, it also can incur context switching and thread creation overheads which can limit its performance. Also it becomes very critical to divide the work among thread evenly since the run time of program depends on the slowest threads.

Part B

Optimized multi-threaded version of RMM on CPU using pthreads

In this question we need to improve the performance by using gpus.Each thread is created to compute an output entry ,so there are (N**2)/4 threads.With gpu's we will get more latency per thread,but better throughput compared to cpu.In the cuda programming we will write a kernel which is executed by all threads.In this kernel we will do 4 operations to compute an entry.

Matrix size	Unoptimized (ms)	GPU Naive (ms)	Speed Up for GPU Naive	GPU shared memory (ms)	Speed up for GPU shared
128	18.702	0.081537	229.36	0.063392	295
4096	533015	63.4415	8401.67	45.996	11588.29
8192	5850000	584.2163	10013.41	308.689	18951.11
16384	51100000	4417.244	11568.29	1969.638	25943.85



Matrix size	Speed up for Single-threaded optimized	Speed for multi-threaded optimized	Speed up for GPU shared
128	28.0	0.1	295
4096	29.0	166.4	11588.29
8192	39.2	218.2	18951.11
16384	37.3	178.4	25943.85

Can we further optimize the code:

We can do tiled matrix multiplication using shared memory.We have to sychnronize threads while we are doing these things since we are operating with shared memory.

How we are doing this?

We written 2 kernels such that one kernel is doing tiled matrix multiplication and one kernel is adding all the entries in 2X2 submatrices of big matrix.So,totally here we are using (N**2/4 +N**2) threads.We are using a tile size of 32 for tiled matrix multiplication.Each block consists of 1024 threads.

Why we are getting better?

1)Accessing a shared memory is faster than accessing a DRAM.

2)Tiled matrix multiplication is better than naïve matrix multiplication.

3)Several threads operates on same output entry.

4) More threads are used compared to previous implementations.

Observations:

1.) We are achieving better throughput with G.P.U.,since there are more cpu's.
2.) We are using shared memory which helps us to get better throughput.Shared memory is shared by each thread block.Shared memory is there in L1 cache of core.
3.) If we use sync_threads() it boils down our execution speed ,but it is required to ensure consistency.
4.) Gpus works better only when there is more parallelism in code.