E0 243 Computer Architecture Assignment 02

Kawin M
(kawinm@iisc.ac.in)

November 4, 2021

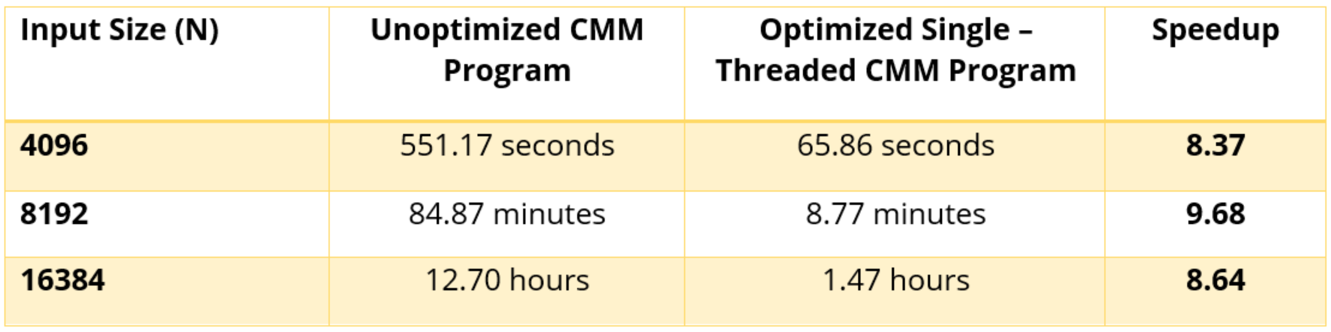
Machine used for evaluation

Processor	Ryzen 5500U
Cores	6 Cores and 12 Threads
Frequency	2.1GHz
L1 Instruction Cache	32KB 8 way
L1 Data Cache	32KB 8 Way
Total L2 Cache	3MB
Total L3 Cache	8MB
RAM	8GB

Part A

Single Threaded Implementation of CMM

Final Result



Input Size (N)	Unoptimized CMM Program	Optimized Single – Threaded CMM Program	Speedup
4096	551.17 seconds	65.86 seconds	**8.37**
8192	84.87 minutes	8.77 minutes	**9.68**
16384	12.70 hours	1.47 hours	**8.64**

Figure 1: Speedup gained by Optimized Single-threaded CMM program compared to Unoptimized CMM Program

Optimization 1: Loop Interchange

Observation: Through performance monitoring, it is observed that the unoptimised CMM program had 8% L1 D-Cache miss rate and 99.76% L1 DTLB miss rate.

Optimization Done: The outer loop traversing the Matrix A row-wise is interchanged with the inner loop traversing the Matrix B columnSet-wise, so as to improve the locality of reference.

Reasoning: The Matrix B is accessed in Column Major Order (CMO), so the inner loop traversing the columnSet in B would contribute to the huge L1 D-Cache misses. In order to reduce that, columnSet is traversed in the outermost loop, whereas the loop traversing the Matrix A in Row Major Order (RMO) is placed as an inner loop to it.

This optimization exploits Temporal locality and it helps in Reducing cache misses in Outermost loop traversing column B because, the same elements in the columnSet are multiplied by each of the rows in Matrix A. Therefore, from the second iteration onwards there are more chances of Cache hit either in the L1 or L2 cache.

Spatial locality also helps here in the inner loop traversing each row in Matrix A and the innermost loop traversing each element in the row. As these elements are accesses sequentially, even the elements in the next row are followed after the end of the current row. Thus, spatial locality helps in reducing the Cache misses further, atleast in the L2 D-Cache.

Runtime and Speedup:

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 1	Speedup
4096	551.17 seconds	545.5 seconds	**1.010**
8192	84.87 minutes	84.71 minutes	**1.002**
16384	12.70 hours	11.84 hours	**1.072**

Result: Reduced,

1. Number of L1 D-Cache load accesses by 1.12%.
2. L1 D-Cache miss rate by 1.25%.
3. DTLB miss rate by 15.50%.

Optimization 2: Loop Fusion

Observation: L1 D-Cache miss rate is still high with 7.9%. In the program, it is found that there are two innermost loops for accessing the elements in a row in the Matrix A, one for performing computation on elements in even positions and other for odd positions in the same row.

Optimization Done: Instead of accessing a single row twice, as these two loops are performing independent operations, these two loops are merged into a single loop with addition of extra variables to keep track of elements in Matrix B.

Reasoning: If we access the consecutive even and odd elements in the same row sequentially, we could decrease the L1 D-Cache miss rate further because of Spatial Locality.

This also decreases the number of branch instructions corresponding to the loops, thus decreases the number of branch mispredictions and the wasted work due to it.

Runtime and Speedup:

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 2	Speedup
4096	551.17 seconds	486.5 seconds	**1.13**
8192	84.87 minutes	62.29 minutes	**1.36**
16384	12.70 hours	9.10 hours	**1.40**

Result: Reduced,

1. Number of instructions committed by 19.38%.
2. Number of L1 D-Cache load accesses by 23.89%.
3. Number of L1 D-Cache load accesses misses by 49.20%.
4. Number of DTLB misses by 49.96%.

all relative to the unoptimised CMM version.

Optimization 3: Merging Computation of Even and Odd rows

Observation: The L1 D-Cache miss rate is still 5.28% and also 5.61% of total committed instructions are branch instructions with 0.05% branch miss rate. In the program, there is an innermost if-else branching statement for performing the computation separately for even and odd rows.

Optimization Done: The computation done by consecutive even and odd rows of Matrix A with a cloumnSet of Matrix B is independent of each other. So, these two computations are merged together. The two innermost loops are merged into single loop so that in one iteration of outer loop, two consecutive rows (one even and one odd) are traversed.

Reasoning: The elements in Matrix B that are multiplied with consecutive even and odd rows of Matrix A, are located very close to each other, so there is a chance that, due to spatial locality, the D-cache misses can be further reduced.

Also, due to temporal locality of accesses in DTLB, as several close elements are accessed. The number of DTLB misses could reduce. This further decreases the number of loop instructions, because now the inner loop traverses two rows at a time instead of one by one. Thus, it decreases the number of times the loop condition is checked and the loop variable is incremented.

Runtime and Speedup:

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 3	Speedup
4096	551.17 seconds	485 seconds	**1.14**
8192	84.87 minutes	61.68 minutes	**1.38**
16384	12.70 hours	8.83 hours	**1.44**

Result: Reduced,

1. Number of instructions committed by 20.42%.
2. Number of branch instructions by 71.09%.
3. Number of L1 D-Cache load accesses by 24.55%.
4. Number of L1 D-Cache load accesses misses by 52.82%.

all relative to the unoptimised CMM version.

Optimization 4: Loop Unrolling

Observation: From the previous optimisation, it is noted that 3.13% of the instructions executed were of loop and branch related instructions (loop condition checking, loop index updating) with 0.10% branch misprediction rate. If we can further reduce these loop related instructions, it is expected that we could achieve a greater speedup.

Optimization Done: The inner loop traversing the rows of Matrix A is unrolled 7 more times.

Reasoning: In a single iteration, this inner loop would traverse and performs computation on 16 consecutive rows of Matrix A. So that loop condition checking and loop index increment is avoided $\frac{7\times N}{8\times 2} = 0.44N$ times.

But this also includes a little overhead because of addition of several local variables to keep track from elements in Matrix B. It is observed that if the loop is unrolled for more than 7 times, there is an increase in execution time.

This unrolling of loop does not affect the final result, because the computation done by each iteration of the inner loop is independent of each other.

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 4	Speedup
4096	551.17 seconds	247.2 seconds	**2.23**
8192	84.87 minutes	36.36 minutes	**2.33**
16384	12.70 hours	5.4 hours	**2.35**

Result: Reduced,

1. Number of instructions committed by 42.41%.
2. Number of branch instructions by 89.08% (Reduced by more than a factor of 2 from previous optimisation).
3. Number of L1 D-Cache load accesses by 40.77%.
4. Number of L1 D-Cache load accesses misses by 6.59% (Number of D-Cache misses almost doubled compared to Optimisation 3).
5. Number of DTLB loads and DTLB misses by around 93.62%.

all relative to the unoptimised CMM version.

Optimization 5: Exploiting L1 D-Cache Prefetching

Observation: The multiplication of an element in Matrix A with an element in columnSet is just 2 positions before the the element that it will multiply in it's next iteration of columnSet.

In the previous HPCA assignment 01, it is observed that, in AMD Zen 2 systems, the L1 D-Cache prefetches the next consecutive 13 blocks along with the requested cache block.

Optimization Done: The inner loop traversing the rows in MAtrix A, performs Multiplication of a row with a two consecutive columnsets in Matrix B in single iteration.

Reasoning: This optimization is based on the Spatial locality concept and exploits prefetching and reduces both the number of the L1 D-Cache misses.

This would also reduces the number of DTLB misses as adjacent elements tend to have same entry in the TLB, thus utilises Temporal locality and hits more frequently.

This would further decrease the number of loop condition checking and variable updation in outer loop by a factor of 2.

As happened in the previous optimization, this also adds little overhead because of addition of local variables. And, this optimization gave optimal result when multiplication is done with 2 columnsets.

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 5	Speedup
4096	551.17 seconds	141 seconds	**3.91**
8192	84.87 minutes	21.06 minutes	**4.03**
16384	12.70 hours	3.08 hours	**3.92**

Result: Reduced,

1. Number of instructions by 1.90 times.
2. Number of branch instructions by 12.78 times.
3. Number of L1 D-Cache load accesses by 1.87 times.
4. Number of DTLB misses by 31.36 times.

all relative to the unoptimised CMM version.

Optimization 6: Loop Unrolling - Row Elements wise

Observation: As the row element multiplications are independent of each other, here also there is an opportunity for unrolling of loop to make use of locality of reference and further reduce the number of branch instructions.

Optimization Done: The innermost loop traversing the elements in the rows of Matrix A is unrolled 1 more time.

Reasoning: Reasoning is same as that of Unrolling in Optimization 4. Here, the optimal results are given when the loop is unrolled once.

Input Size (N)	Unoptimized CMM Program	CMM Program after Optimization 6	Speedup
4096	551.17 seconds	66.8 seconds	**8.25**
8192	84.87 minutes	8.79 minutes	**9.66**
16384	12.70 hours	1.50 hours	**8.47**

Result: Reduced,

1. Number of instructions by 2.19 times.
2. Number of branch instructions by 17.46 times.
3. Number of L1 D-Cache load accesses by 2.58 times.
4. Number of L1 D-Cache misses by 5.27 times.

all relative to the unoptimised CMM version.

Optimization 7: Strength Reduction, Precomputing Values, Breaking chains of Data Dependencies, Exploiting Temporal Locality further

Observation: There are operations such as multiplication by 2 or it's powers, frequently performing the same operations and data dependency between the variables in the program.

Optimization Done:

1. Multiplication by 2 and it's powers are replaced by shift operations.
2. Modulo by 2 is replaced by boolean and operations.
3. Frequently calculated operations such as (N¡¡2) are replaced by a variable, that precomputes this variable before the beginning of the loop.
4. Non-related Local variables in innermost loop are combined into an array.

Reasoning: Replacing of high latency operations with low latency operations, avoiding computation of same value again and again and using array instead of several local variables, because array contains consecutive elements which helps in address translations and also in cache prefetching.

Input Size (N)	Unoptimized CMM Program	Optimized Single – Threaded CMM Program	Speedup
4096	551.17 seconds	65.86 seconds	**8.37**
8192	84.87 minutes	8.77 minutes	**9.68**
16384	12.70 hours	1.47 hours	**8.64**

Result: Reduced,

1. Number of instructions by 2.19 times.
2. Number of branch instructions by 17.5 times.
3. Number of L1 D-Cache misses by 5.32 times.
4. Number of L1 ITLB misses by 23.76 times.
5. IPC was increased from 0.38 in unoptimised version to 1.40 in this optimised version.

all relative to the unoptimised CMM version.

Optimization 8: Vectorization

Observation: The program is compiled with auto optimization feature in GNU compiler and it gave a speedup of 2 times from previous optimization. On inspection of assembly code, all those integer multiplications were replaced by vector operations.

Optimization Done: Integer multiplication operations are replaced by vector operations.

Result: The Execution time increased by a factor of 2 from previous optimization. So, this optimization was not included in the final result.

Multi Threaded Implementation of CMM

Final Result

Input Size (N)	Unoptimized CMM Program	Optimized Multi – Threaded CMM Program	Speedup
4096	551.17 seconds	9.41 seconds	**58.57**
8192	84.87 minutes	1.49 minutes	**56.96**
16384	12.70 hours	0.22 hours	**57.73**

Figure 2: Speedup gained by Optimized Multi-threaded CMM program compared to Unoptimized CMM Program

Implementation

The program idea which was implemented as used in the final optimization of single threaded CMM program is used here, such that, for each iteration in the outer loop (traversing columnset in Matrix B), a thread is created, upto a maximum of 12 threads. And these threads are joined before creation of next set of 12 threads.

Scalability

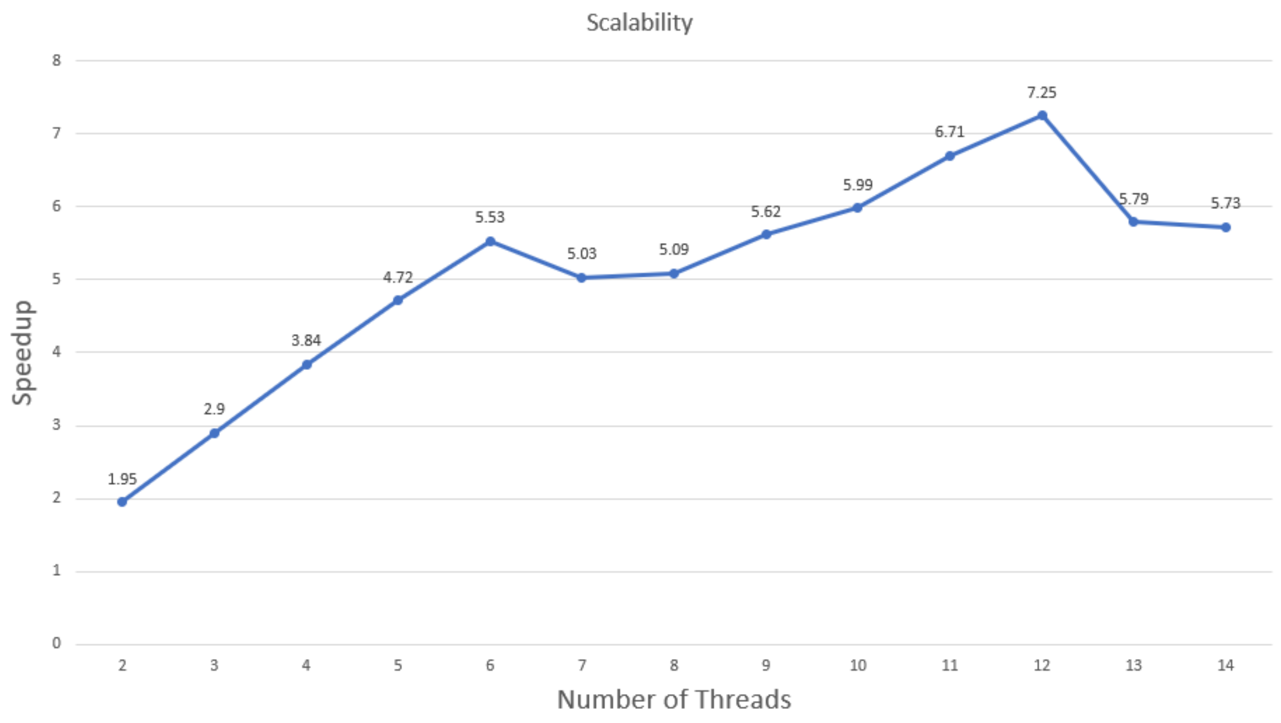


Figure 3: Speedup Achieved with respect to Number of Threads created

Number of Threads	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Runtime - 4096 (in seconds)	66.85	34.23	23.07	17.40	14.17	12.09	13.29	13.13	11.89	11.15	9.95	9.22	11.54	11.66
Speedup		1.95	2.9	3.84	4.72	5.53	5.03	5.09	5.62	5.99	6.71	7.25	5.79	5.73

A maximum of 12 threads was used in the Multi Threaded Implementation of CMM because, the speedup was maximum for 12 threads compared to others.