Programming assignment: Part 1 Performance optimizing Diagonal matrix multiplication (DMM)

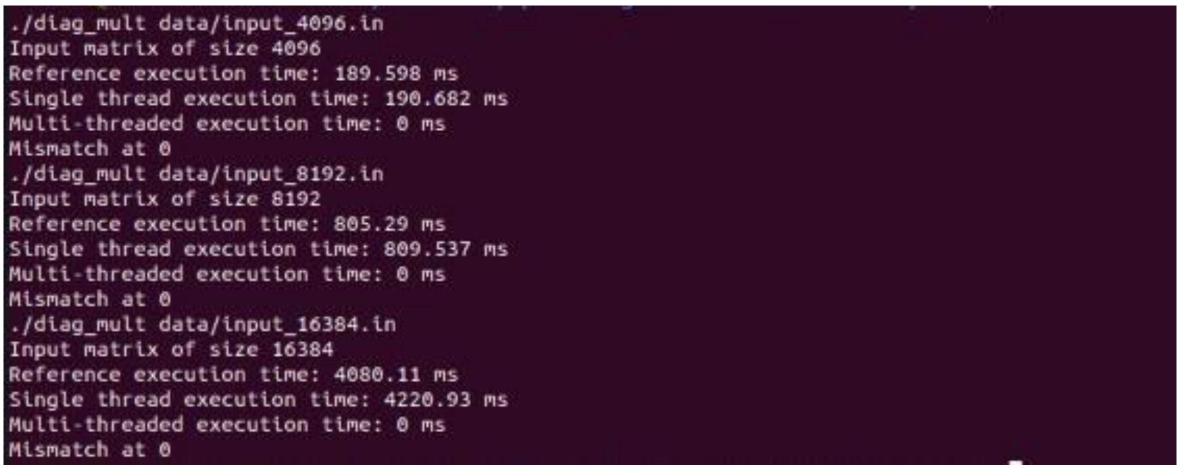
Submitted by
Deepanshu
17981

Introduction

Given a unoptimized single-threaded program "diagonal matrix multiplication (DMM)" of two n*n matrices, we have implemented (1) Optimized single threaded code by analysing relevant hardware performance counter values, and (2) Multithreaded code using pthreads to parallelize the program for better efficiency.

[PartA-I] Optimize single-threaded DMM (CPU)

For the given code, execution time and various performance counters can be found using make run and other perf commands.



```
./diag_mult data/input_4096.in
Input matrix of size 4096
Reference execution time: 189.598 ms
Single thread execution time: 190.682 ms
Multi-threaded execution time: 0 ms
Mismatch at 0
./diag_mult data/input_8192.in
Input matrix of size 8192
Reference execution time: 805.29 ms
Single thread execution time: 809.537 ms
Multi-threaded execution time: 0 ms
Mismatch at 0
./diag_mult data/input_16384.in
Input matrix of size 16384
Reference execution time: 4080.11 ms
Single thread execution time: 4220.93 ms
Multi-threaded execution time: 0 ms
Mismatch at 0
```

By analysing the given code, it was found that most of the time was taken for accessing the matrices elements. One can observe that in the given code elements are accessed randomly from the matrices, i.e., spatial locality property is not exploited (Data available in the cache because of spatial locality property is not used). By accessing data randomly, already available data from the cache will be evicted without getting computed and later they will be loaded again for computation which results in increased execution time.

Command used to examine perf counters-

perf stat -d ./diag_mult data/file_name.in

Events/input	**4096.in**	**8192.in**	**16384.in**
Cycles	8,74,62,63,763	35,29,09,01,493	1,51,76,00,13,330
Instructions	19,57,01,61,098	78,12,06,90,306	3,12,58,33,27,275
Branches	3,75,82,47,501	14,98,65,84,311	59,96,03,19,489
Branch-misses	1,54,78,761	6,13,36,593	24,60,01,648
L1-dcache-loads	6,87,86,52,027	27,50,13,05,708	1,09,98,45,50,037
L1-dcache-load-misses	16,14,33,103	84,62,55,783	3,40,31,67,120
LLC- loads	10,20,32,188	43,94,48,205	2,29,73,50,269
LLC-load-misses	43,50,739	1,67,40,962	11,99,70,879

Optimizing the code

Code can be optimized by exploiting the spatial locality property. While accessing one element from the matrix, whole block in which that element is present will be placed in cache. Now the code should be written in such a way that already available data in the cache should be used. To do this, data from the matrices should be accessed in the blocks of appropriate size. This concept is called blocking. Whole matrix can be divided into blocks and then for a block(submatrix) the program will complete its all computation first before going to the next block.

We are using the block size of 16 in our program, as cache line can contain 16 elements at most. Therefore, whenever any element is accessed, all 16 elements of the block from which data is accessed are loaded into the memory.

Comparing the optimized code with given code

We can examine both codes with respect to their execution times and various perf counters.

Speedup can be calculated as-

Speedup = Reference execution time/Single thread execution time

Events/input	4096.in	8192.in	16384.in
Reference execution time(ms)	192.195	1012.74	4130.18
Single thread execution time(ms)	105.181	427.373	1782.91
Speedup	1.8272	2.3696	2.3165

We can observe the speedup, single threaded optimized program is performing significantly efficient as compared to given unoptimized program

Events/input	4096.in	8192.in	16384.in
Cycles	8,33,61,83,399	35,18,24,06,123	1,41,73,21,66,255
Instructions	19,52,72,45,721	78,20,94,90,227	3,11,97,74,83,199
Branches	3,73,76,07,435	14,97,80,90,942	59,81,73,46,987
Branch-misses	1,51,62,585	6,05,48,733	24,12,41,768
L1-dcache-loads	6,83,32,42,609	27,33,23,20,809	1,09,19,52,87,561
L1-dcache-load-misses	12,70,73,751	65,74,93,521	2,59,65,11,508
LLC- loads	8,31,75,199	36,61,44,448	1,80,02,26,723
LLC-load-misses	39,82,740	1,47,92,020	10,51,74,824

By comparing given events/counters for optimized and unoptimized program we can see improvement. As the optimized code was trying to use already available data in cache as compared to unoptimized code, which was accessing data randomly, because of that we can observe that **L1-dcache-load-misses** are significantly less in the optimized code.

[Part A-II] Implement and optimize multi-threaded DMM (CPU)

In this section we have implemented multithreading on our optimized program. Using multithreading, we can run the independent parts of our code in parallel using different thread for each part. This will result in faster execution of the program and hence better efficiency.

Threads have their own overhead, therefore if we implement too many threads in the program it will result in more execution time and hence poor performance. To find ideal number of threads to be used in the program, we have analysed the program by considering different number of threads (8,16,32,64,128,256). Following are the results of execution on different number of threads-

On input_4096.in

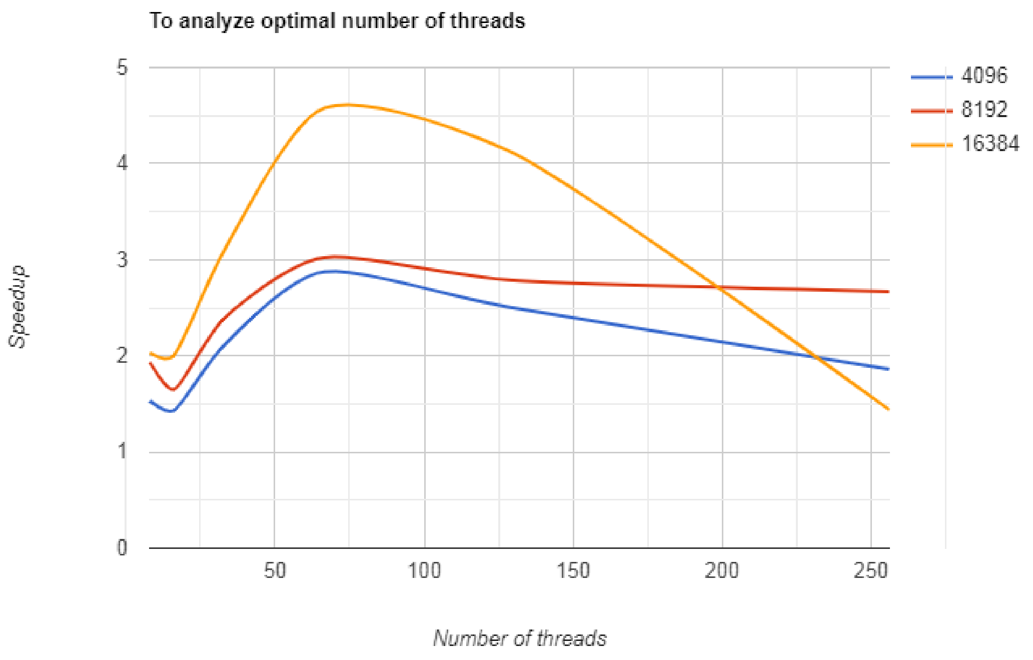
Number of threads	Multithreaded execution time	Reference execution time	Speedup
8	129.854	198.594	1.53
16	131.459	189.236	1.43
32	90.262	188.339	2.08
64	68.334	195.421	**2.86**
128	76.71	192.77	2.51
256	109.379	204.253	1.86

On input_8192.in

Number of threads	Multithreaded execution time	Reference execution time	Speedup
8	498.374	957.245	1.93
16	504.793	837.117	1.65
32	358.465	847.441	2.36
64	281.834	848.812	**3.01**
128	298.382	834.518	2.79
256	311.304	832.423	2.67

On input_16384.in

Number of threads	Multithreaded execution time	Reference execution time	Speedup
8	2109.56	4287.66	2.03
16	2118.72	4246.13	2.00
32	1316.08	4482.77	3.04
64	1032.14	4695.03	**4.54**
128	1201.317	4982.72	4.14
256	2944.86	4255.62	1.44



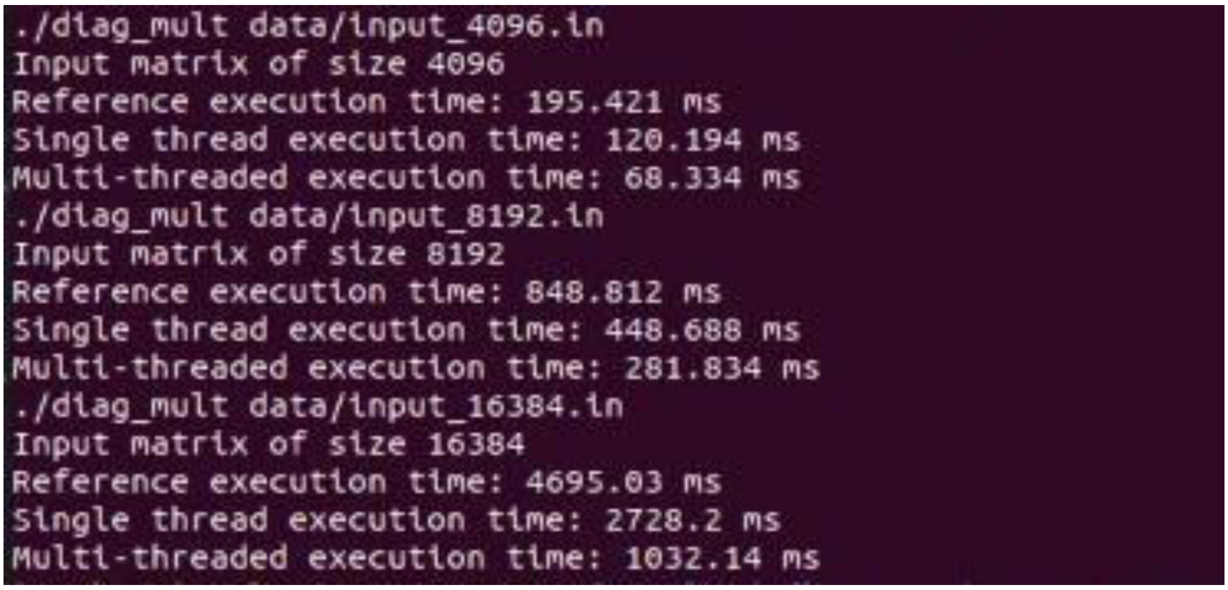
From the above results we can observe that speedup increases by increasing number of threads, but after reaching at a limit, it again decreases. This is because of the thread overheads as mentioned earlier. When there are too many threads, then every thread has too little work and most of the time is consumed in managing threads.

From our observations, 64 is optimal value of number of threads.

Conclusion

We optimized the single threaded program by using blocking. Blocks of size 16 are used to exploit already available data in the cache. Finally, we implemented multithreading code to parallelize the program. By analysing on the different number of threads, we found that for 64 number of threads the program performs best.

Following is the output of the optimized program



```
./diag_mult data/input_4096.in
Input matrix of size 4096
Reference execution time: 195.421 ms
Single thread execution time: 120.194 ms
Multi-threaded execution time: 68.334 ms
./diag_mult data/input_8192.in
Input matrix of size 8192
Reference execution time: 848.812 ms
Single thread execution time: 448.688 ms
Multi-threaded execution time: 281.834 ms
./diag_mult data/input_16384.in
Input matrix of size 16384
Reference execution time: 4695.03 ms
Single thread execution time: 2728.2 ms
Multi-threaded execution time: 1032.14 ms
```