HPCA Programming Assignment 2023-2024
Optimizing Performance of Dilated Convolution

ASAD ALI (23104)
asadali@iisc.ac.in

SPANDAN NAGARE (23072)
spandanrn@iisc.ac.in

Part A

[I] Optimize single-threaded DC (CPU)
[II] Implement and optimize multi-threaded DC (CPU)

Introduction

Dilated Convolution (DC) is a variant of the convolution operation with wide-ranging applications in signal processing, image analysis, and deep learning. As practical applications often involve large matrices, it becomes imperative to address the performance considerations of Dilated Convolution, particularly as matrix sizes increase.

Our initial task involves enhancing the performance of a single-threaded, unoptimized implementation of dilated convolution. Subsequently, our focus shifts to the implementation and optimization of a multi-threaded version of DC. This comprehensive report delves into the intricate details of the optimization endeavors undertaken for both single and multi-threaded DC implementations on CPU architectures.

The intricacies of dilated convolution lie in its ability to expand the receptive field without a proportional surge in parameters. This efficiency is particularly crucial in resource-constrained scenarios, making optimization paramount. The report will elaborate on the strategies employed, such as loop unrolling, vectorization, and cache-aware optimizations, to bolster the efficiency of dilated convolution as matrices grow in size.

The report navigates through the nuances of optimizing dilated convolution, ensuring a thorough exploration of both single and multi-threaded implementations on CPU architectures.

HW/ SW specifications:

The machine that we have used to carry out the experiments has the specifications shown in the following table 1.1.

CPU	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
Memory	16 GB DDR4
OS	Windows 11 Home Single Language; Version: 10.0.22621 N/A Build 22621
Cache	320 KB L1, 5 MB L2, 12 MB L3

[I] Optimize single-threaded DC (CPU)

In this section, we enhance the performance of the single-threaded implementation by identifying and addressing performance bottlenecks in the code through the utilization of hardware performance counters, such as those available through tools like perf. Subsequently, we optimize the source code based on the insights gained from these performance measurements.

1.1 Performance Analysis

To conduct a comprehensive performance analysis, we have been supplied with varying sizes of both kernel matrices and input matrices. Leveraging the perf tool, we systematically examined diverse hardware performance counter values corresponding to specific combinations of input and kernel matrices.

1.2 Bottleneck Identification

To identify the bottleneck in the single-threaded DC execution, we employed perf to record various miss events such as Cache load, Cache store, TLB miss, and LLC miss. Upon analysis, we noted that computation contributes significantly to the execution time. Consequently, our focus now shifts to mitigating computation overhead through optimization techniques.

1.3 Optimization strategies

We are performing the following optimization strategies to optimize our LLD cache misses for DC:

1.3.1 Loop Interchange

Loop interchange stands as a loop transformation method employed in compiler optimization to boost program performance by altering the nesting order of loops. The fundamental concept involves interchanging the positions of two nested loops, thereby augmenting data locality, and optimizing cache performance.

In the single-threaded DC provided, there are four nested loops denoted as ABCD. Consequently, there are a total of 24 permutations of these loops. Table 1.4 provides the execution times for all 24 loop permutation combinations.

Input	**4096**	**4096**	**4096**	**4096**	**4096**	**8192**	**8192**
Kernel	**3**	**5**	**7**	**11**	**13**	**3**	**5**
Permutation							
Reference	1053.53	2727.051	5296.456	13503.75	18552.03	4156.202	11944.58
ABCD	1243.1	3298.695	6347.44	15166.45	21161.4	4955.39	13570.7
ABDC	1241.61	3204.395	6332.605	15397.35	21309.55	4955.755	13235.3
ACBD	1259.07	3200.3	6155.55	15056.4	20965.3	4937.345	13225.1
ACDB	1222.3	3113.805	6017.61	14655.65	21010.85	4665.275	13011
ADBC	1271.875	3236.285	6312.98	15248.8	21439.3	4942.385	13363.85
ADCB	1217.945	3121.12	6301.635	14857.8	20506.8	4698.135	13184.8
BACD	2235.875	4170.94	7354.575	16308.95	22006.2	9371.65	17957.25
BADC	2353.88	4439.385	7599.4	16907.1	23235	9400.13	18658.85
BCAD	3674.61	7084.015	11649.45	23792.5	31458.3	16439.55	33103.45
BCDA	4419.665	11712.5	22685.85	55615.5	78385.85	18319.55	50796.9
BDAC	4012.965	7414.11	12274	25137.95	33977.35	16338.1	31852

BDCA	4329.265	11621.55	22649.95	55543.8	77571.4	18390.3	50361.65
CABD	1334.58	3223.955	6292.3	15046.25	21469.25	4959.99	13592.4
CADB	1321.77	3166.96	6084.74	14834.4	20424.5	4860.225	13147.35
CBAD	3834.34	7206.53	11954.45	24754.9	31763.85	16555.05	33285.4
CBDA	4692.36	11961.25	23023.5	61271.6	78926.95	18598.25	51234.75
CDAB	1303.885	3277.98	6354.985	16269.05	21513.15	4948.36	13937.35
CDBA	4589.175	13083.2	24389.65	60849.35	83800.8	18995.1	54642.35
DABC	1309.755	3367.66	6198.345	15718.15	21253	5003.03	15490
DACB	1192.46	3207.255	6049.615	14846.35	20650	5017.11	14507.6
DBAC	3929.885	7803.245	12439.4	25888.65	33854.4	16482.3	32249.15
DBCA	4371.375	12045.5	23082.6	57150.1	78506.55	18685.6	51119.35
DCAB	1228.345	3284.06	6360.31	15605.9	21495.5	4961.485	13303.75
DCBA	4527.035	12559.55	24410.7	60239.5	86017.35	19129.7	53268.3

As we can see permutation of loops is not optimizing further, we try using other techniques.

1.3.2 Using Loop Invariant

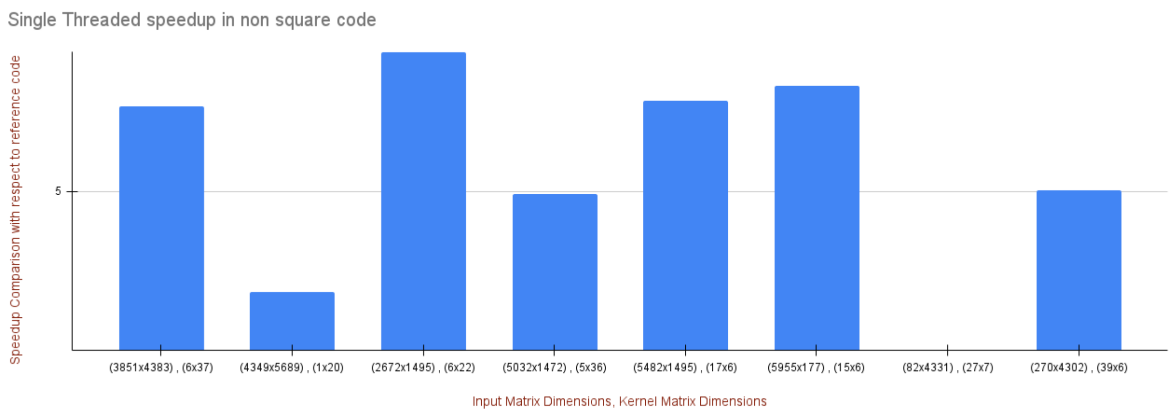
Loop Invariant Code Motion (LICM) optimizes code by identifying and relocating invariant computations outside loops. In our context, the application of LICM minimized redundant computations, resulting in a **1.5-speedup** for "non-vectorized single-thread" execution. This enhancement improved the efficiency of loop execution and overall program performance.

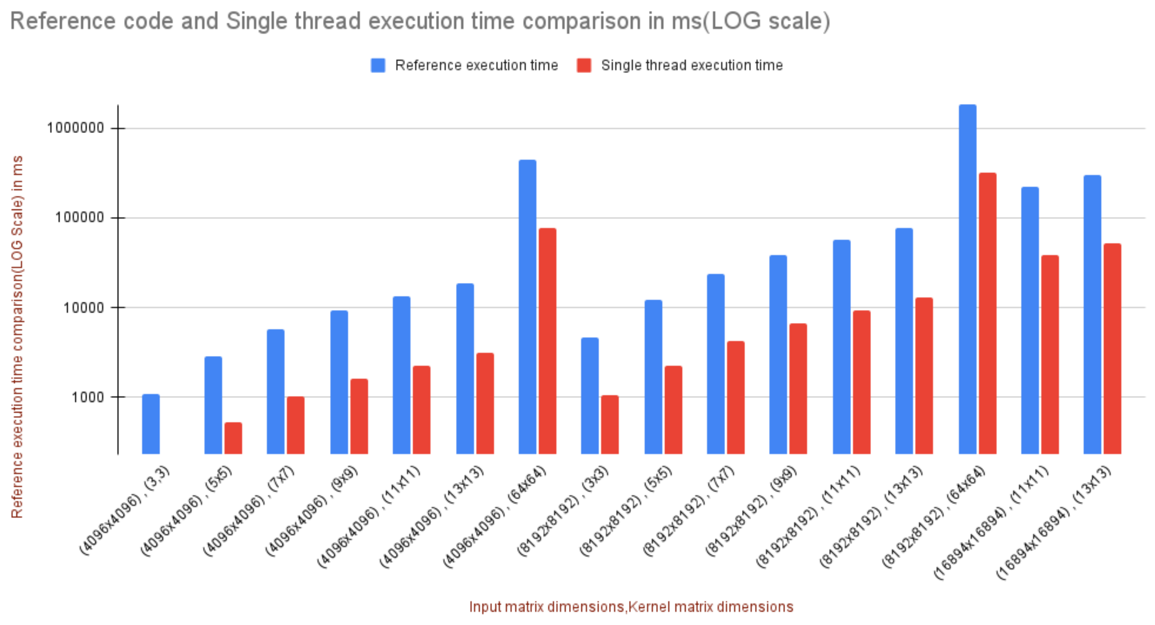
1.3.3 Using SIMD

SIMD, or Single Instruction, Multiple Data, is a computing paradigm designed for enhanced performance by concurrently executing a single instruction on multiple data elements, rather than sequentially processing each element. In our specific computational environment, the incorporation of the AVX2 extension introduces SIMD instructions that operate on 256-bit registers, capable of handling 32 bytes of data. This architectural feature taps into parallelism, enabling simultaneous processing of multiple data elements to expedite computations.

To safeguard the correctness of our optimization strategy, we take special care to address edge cases when the input matrix is not a multiple of 8. This meticulous handling ensures the precise implementation of our SIMD-based approach, maintaining the integrity of the overall optimization process.

1.4 Results





[II] Implement and optimize multi-threaded DC (CPU)

In this section, we've developed a multi-threaded iteration of the DC algorithm. Our objective is to transition from the single-threaded implementation of DC to a multithreaded approach. The aim is to optimize the code and elevate the overall performance of the DC implementation. To assess its scalability, we conducted tests by adjusting the number of threads. Our focus was on optimizing the implementation for maximum scalability, employing "pthreads" for the multi-threaded approach.

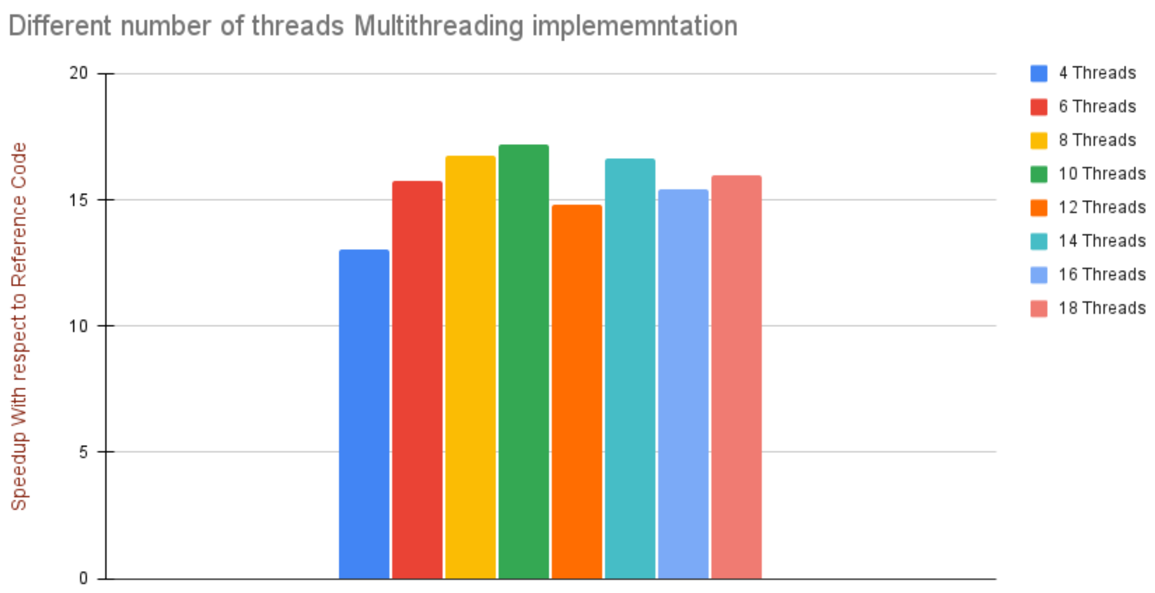
2.1 Performance Analysis

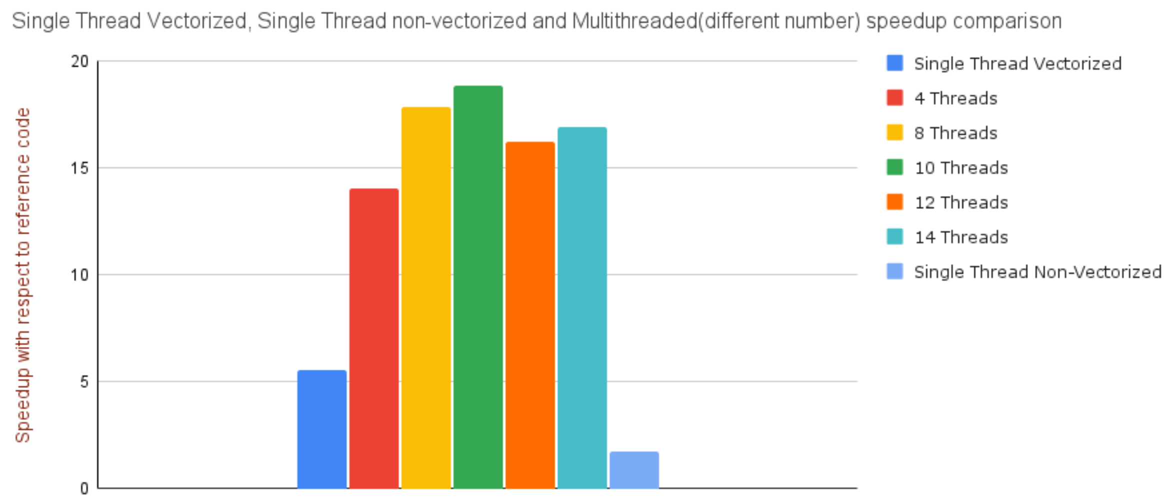
To conduct performance analysis, we were furnished with various sizes of kernel matrices and input matrices. Utilizing perf, we scrutinized hardware performance counter values for specific selected input and kernel matrices.

Following the findings in Part [I], where SIMD yielded approximately 5.3 in performance, we proceeded to transform the SIMD code from single-threaded to multi-threaded. The execution times of the multithreaded code, employing varying numbers of threads, are presented in next section

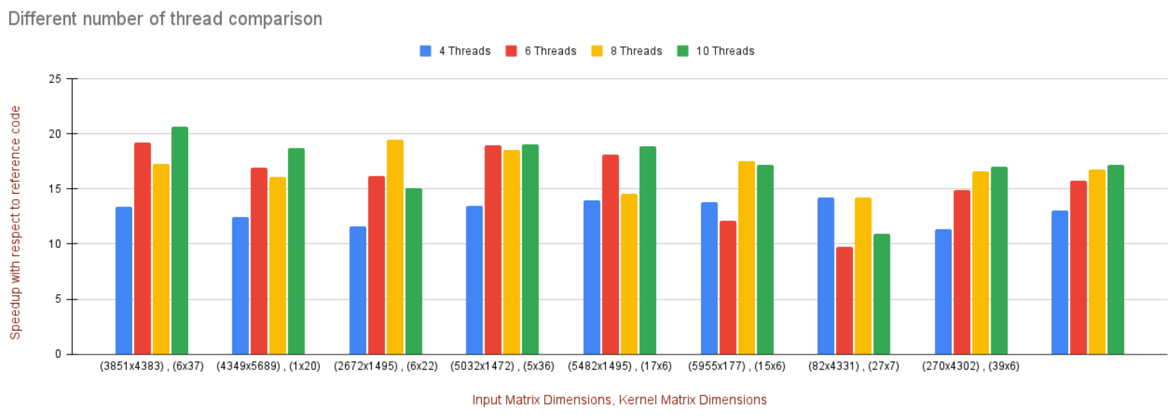
Additionally, results section we'll show the execution times of the multi-threaded, optimized combinations of input and kernel matrices.

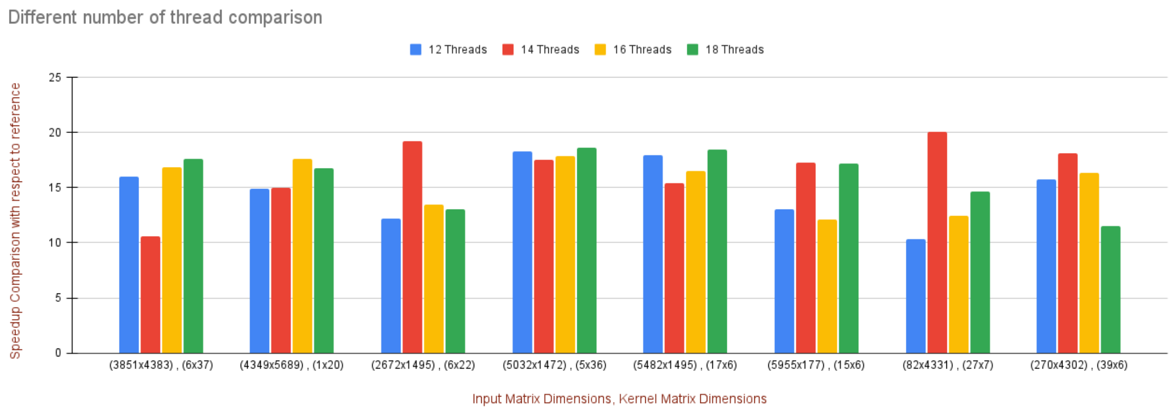
1.4.2 Results



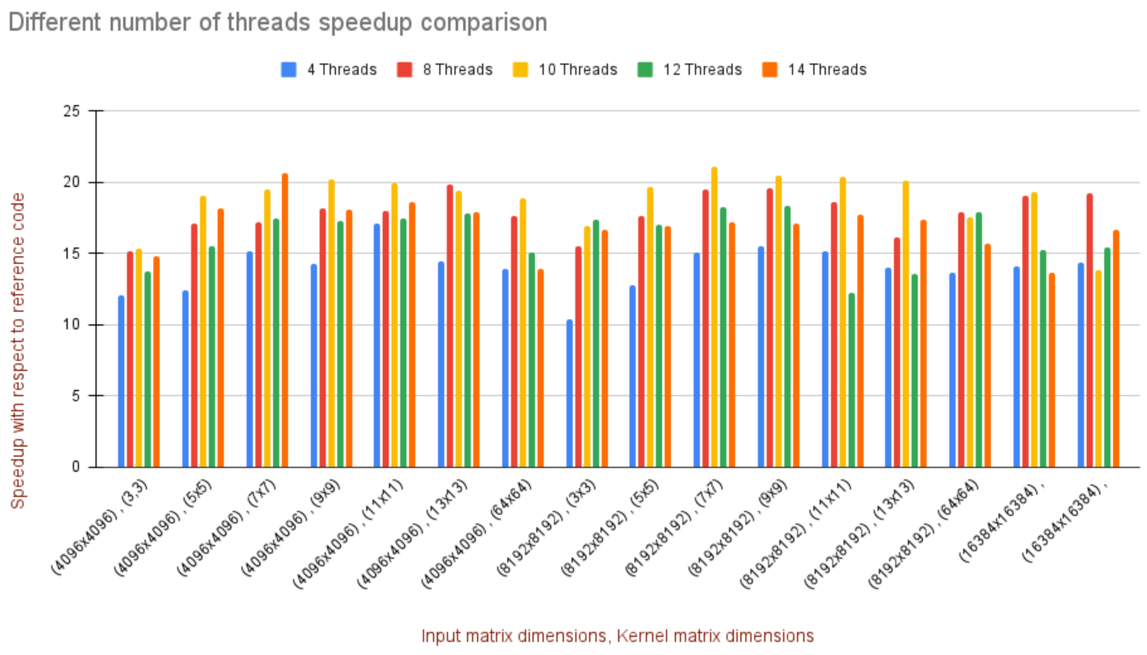


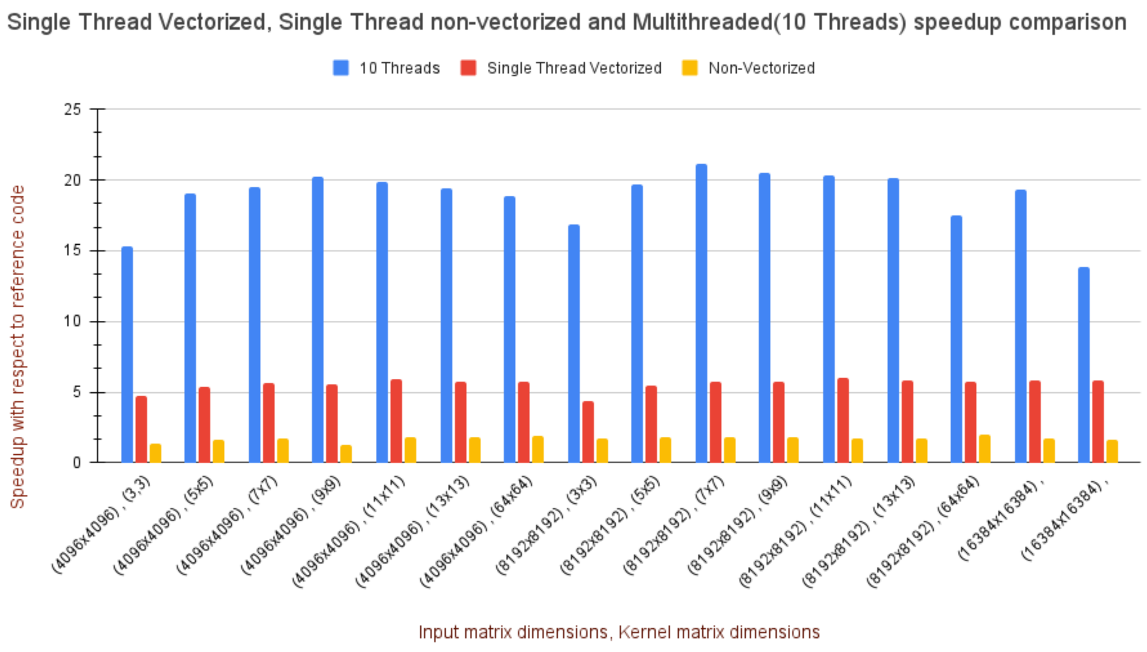
For Non-square input matrix and Non-square Kernel matrix

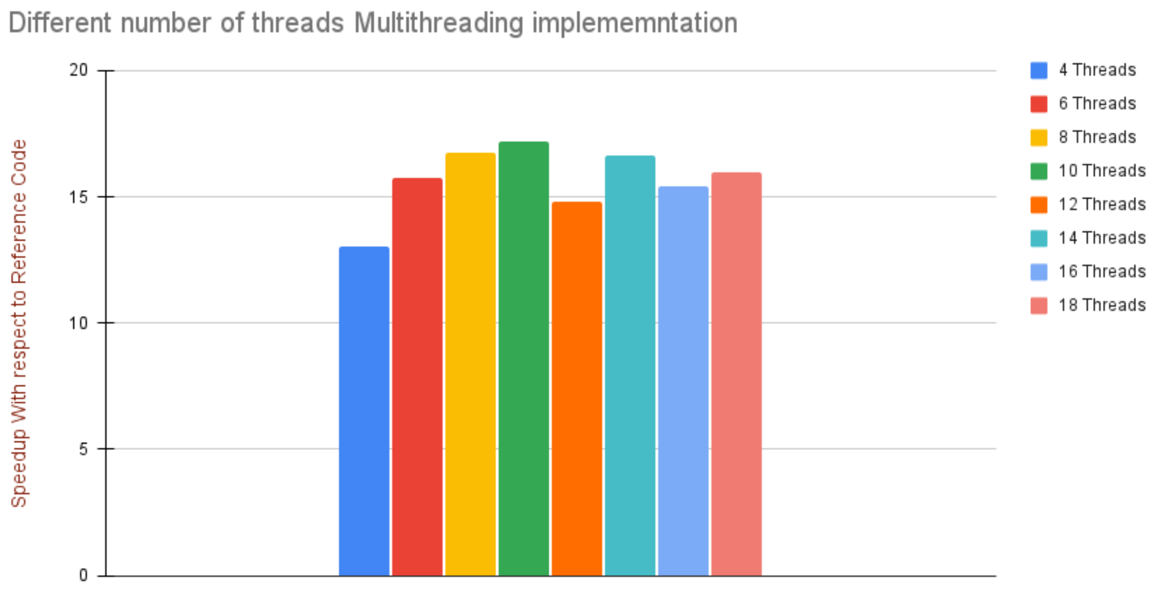




For Square input matrix and Square Kernel matrix


Different number of threads speedup comparison
4 Threads
8 Threads
10 Threads
12 Threads
14 Threads
Speedup with respect to reference code
0
5
10
15
20
25
(4096x4096) , (3,3)
(4096x4096) , (5x5)
(4096x4096) , (7x7)
(4096x4096) , (9x9)
(4096x4096) , (11x11)
(4096x4096) , (13x13)
(4096x4096) , (64x64)
(8192x8192) , (3x3)
(8192x8192) , (5x5)
(8192x8192) , (7x7)
(8192x8192) , (9x9)
(8192x8192) , (11x11)
(8192x8192) , (13x13)
(8192x8192) , (64x64)
(16384x16384) ,
(16384x16384) ,
Input matrix dimensions, Kernel matrix dimensions


Single Thread Vectorized, Single Thread non-vectorized and Multithreaded(10 Threads) speedup comparison
10 Threads
Single Thread Vectorized
Non-Vectorized
Speedup with respect to reference code
0
5
10
15
20
25
(4096x4096) , (3,3)
(4096x4096) , (5x5)
(4096x4096) , (7x7)
(4096x4096) , (9x9)
(4096x4096) , (11x11)
(4096x4096) , (13x13)
(4096x4096) , (64x64)
(8192x8192) , (3x3)
(8192x8192) , (5x5)
(8192x8192) , (7x7)
(8192x8192) , (9x9)
(8192x8192) , (11x11)
(8192x8192) , (13x13)
(8192x8192) , (64x64)
(16384x16384) ,
(16384x16384) ,
Input matrix dimensions, Kernel matrix dimensions



Conclusion

As we can observe from the results, multithreading and SIMD both codes implementation works better for larger matrices compared to smaller input or kernel matrices. Also SIMD gave us overage **5.3** speedup and multithreading implementation gave us speedup of **17** approximately. We got best performance in multithreading code when we launched **10 threads** in our system. Hence we've got obtained performance significantly better than reference program