E0 243 High Performance Computer Architecture
Assignment 2 – Part A

Submitted by
Siddharth Baudh (22593), siddharthbau@iisc.ac.in
Dileep Patel (22683), dileeppatel@iisc.ac.in

November 26, 2023

Part A

Optimizing Performance of Dilated Convolution (DC) - Single-threaded and multi-threaded

1. Introduction

Dilated convolution is a variant of convolution operation which offers many applications in various domains like signal processing, image analysis and deep learning etc. Often the size of the matrices is very large when we talk about Dilated Convolution in practical applications so it is important to talk about its performance as the matrix size becomes larger.

We are given the single-threaded unoptimized implementation of dilated convolution and we our primary objective is to optimize its performance as the first activity, and then implement and optimize a multi-threaded version of DC. This report includes the details about various optimization efforts carried out on the single and multi-threaded implementations of DC on CPU architectures along with its implement details.

2. Machine specifications:

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

CPU	9^{th} Gen Intel (R) Core (TM) i5-9300H, 2.40 GHz
Memory	8 GB DDR4
OS	Ubuntu 22.04.3 LTS, Kernel: 6.2.0-36-generic
Cache	Figure 1.1
CPU Cores	4

Table 1.1: Machine Specifications

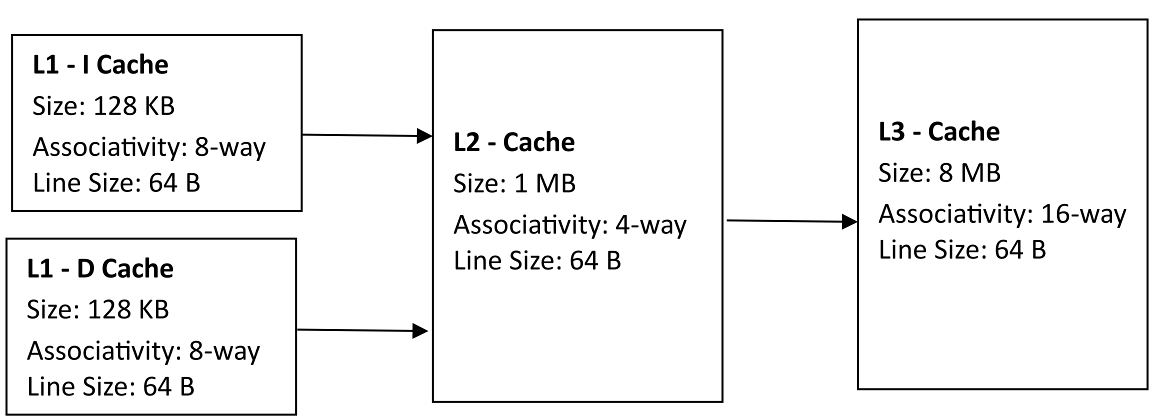


Figure 1.1

3. PART A – I: Optimizing Single-Threaded DC (CPU)

In this section, we are given an unoptimized single-threaded implementation of DC. The goal is to optimize the code and enhance the overall performance of single-threaded DC implementation.

We have been provided with different sizes of kernel matrices and input matrices. We have used perf to examine different hardware performance counter values for particular selected input and kernel matrices.

3.1 Bottleneck Identification

To identify the bottleneck of single threaded execution of DC, we recorded various miss events using perf. We have considered the following miss events in an interval of 100 milliseconds:

- LLC-store-misses
- LLC-load-misses
- L1-icache-load misses
- L1-dcache-load-misses
- Cache misses
- Branch misses

Graph 1.1 shows the Miss events vs Values graph, where on the y-axis we have represented the miss events and, on the x-axis, we have their values occurred in an interval of 100 milliseconds.

Graph 1.1

From the above attached graph, we can see that the bottleneck is L1-dcache load misses for our unoptimized DC implementation. So, we will try to optimize load misses due to L1-dcache.

3.2 Optimization strategies

We are performing the following optimization strategies to optimize L1 D cache misses for given DC implementation:

3.2.1 Loop Interchange

Loop interchange is a loop transformation technique used to improve the performance of programs by changing the nesting order of loops. It is often done to ensure that the elements of a multi-dimensional array are accessed in the order in which they are present in memory, improving locality of reference.

In the given single-threaded DC implementation we have 4 nested loops (let ABCD), with that we can have 24 permutations of loops.

Table 1.2 shows the execution time (in ms) of all 24 permutation of loops with respect to the Reference execution time (in ms). 4096 in input represents we have input matrix of size 4096x4096 and 3 in kernel represents we have kernel matrix of size 3x3.

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

Table 1.2: Different permutations of loop interchange

As we can see interchange of loops is not giving us any performance improvement so, we try some other optimization strategies.

3.2.2 Loop Unrolling

Loop unrolling is a compiler optimization technique that involves unfolding the loop to expose more opportunities for parallelization. We have implemented loop unrolling, but we did not achieve any significant improvement in the performance so we tried next optimization strategy which is blocking.

3.2.3 Blocking

Blocking, is a loop optimization technique used to improve cache locality and reduce cache misses. It divides the computation of a loop into smaller, more manageable blocks that fit into the cache, allowing for better data reuse. We have performed blocking in our code with block sizes 16x16, 32x32 and 64x64 but we did not get any performance improvement using blocking.

3.2.4 SIMD (Vector Processing)

SIMD stand for Single Instruction, Multiple Data. The basic idea behind using SIMD is to use single instruction to perform the same operation on a batch of data elements simultaneously, rather than processing each element sequentially.

The computational architecture of our machine has AVX2 extension, a specialized instruction set designed for SIMD (Single Instruction, Multiple Data) operations. The AVX2 extension operates on 256-bit registers, each capable of accommodating 32 bytes of data.

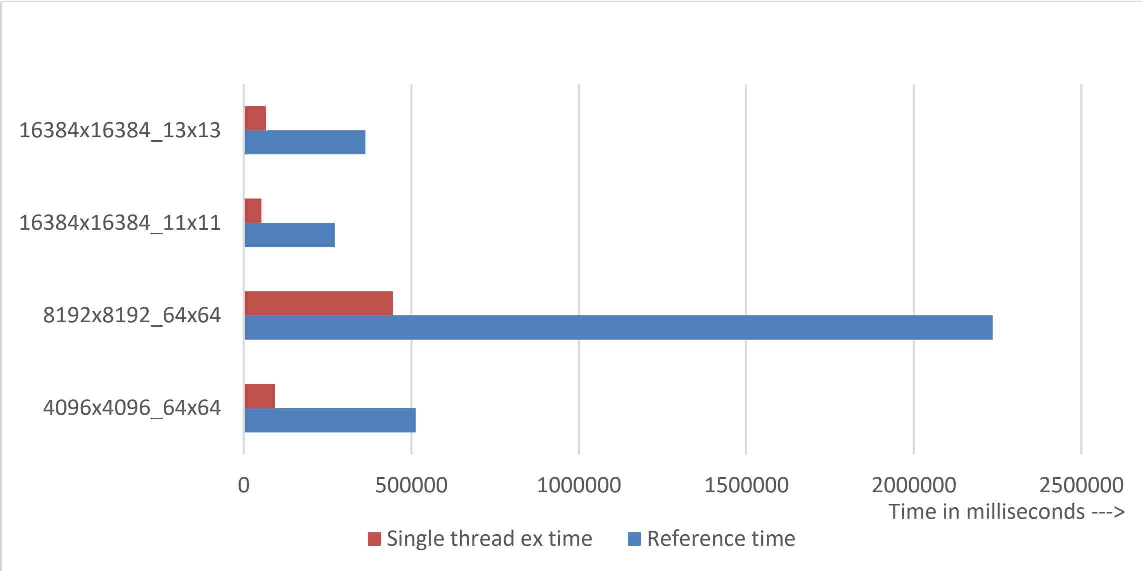
We have padded the input matrix with 8 columns at the end of the matrix. The reason for doing padding is because without padding we will have to handle many if else cases. Now we will only need to pick columns of 8 from output matrix (because our 256-bit AVX2 register we can process 8 integers of 32 bit each). Table 1.3 shows the comparison between reference and optimized SIMD code execution.

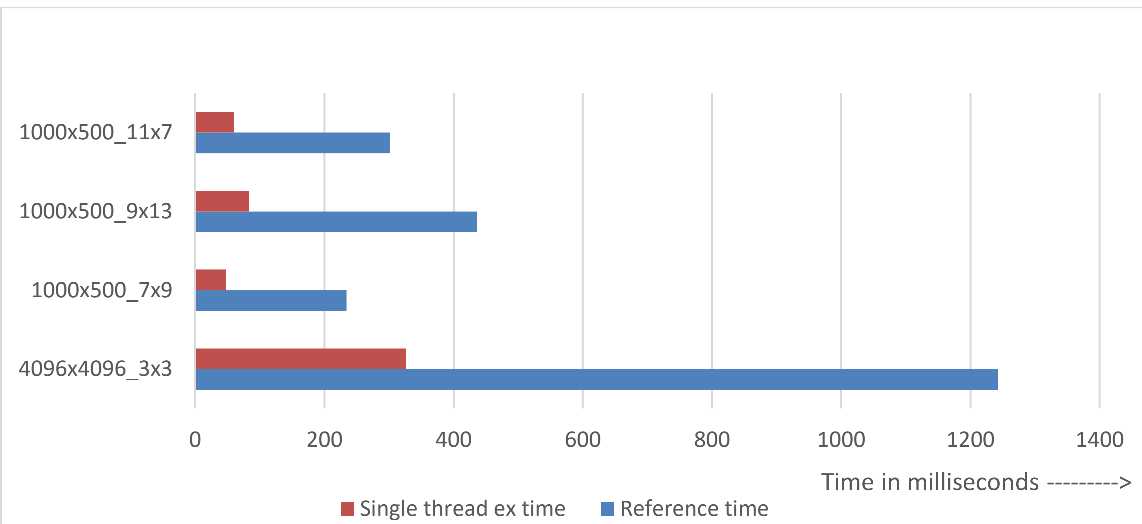
Here in the first column data, for example in 4096x4096_3x3 represents that 4096x4096 is the dimension of input matrix and 3x3 is the dimension of kernel matrix.

Dimensions Input (Kernel)	Reference time (in ms)	Single thread ex time (in ms)
4096x4096_3x3	1242.89	325.727
4096x4096_5x5	3496.31	798.652
4096x4096_7x7	6632.59	1354.26
4096x4096_9x9	11293.8	2137.93
4096x4096_11x11	15526.7	2966.12
4096x4096_13x13	21413.2	4126.47
8192x8192_5x5	12979.9	2811.2
8192x8192_7x7	26613.9	5347.2
8192x8192_9x9	47186.8	9560.85
8192x8192_11x11	75508.7	13487.9
8192x8192_13x13	96091.9	18590.5
8192x8192_64x64	2234770	445330
16384x16384_11x11	271371	52408.4
1000x500_7x9	234.143	47.746
1000x500_9x13	436.348	83.649
1000x500_11x7	301.371	60.062
1000x500_13x17	801.266	153.625
1000x500_59x31	6070.1	1146.71
2000x1000_11x7	1185.76	234.242
7000x3000_11x7	12840	2527.04
7000x3000_13x17	36475.1	6794.55

Table 1.3: Single thread execution time with respect to reference time using SIMD

Following graph 1.2 shows the comparison between few of the reference and optimized SIMD code execution.





Graph 1.2: Comparison between reference execution time and Single thread execution time

Here for example in 4096x4096_3x3, 4096x4096 is the dimension of input matrix and 3x3 is the dimension of kernel matrix

3.3 Results

In this part we found out that using SIMD gives us the speed up between 3.9x and 5.5x with respect to reference. The speedup 3.9x is on smaller input matrices and speedup of 5.5x on larger input matrices.

4. PART A – II: Implementing and Optimizing Multi-Threaded DC (CPU)

In this section, our goal is to implement and optimize the code given in single threaded implementation of DC to multithread and enhance the overall performance of DC implementation.

4.1 Performance Analysis

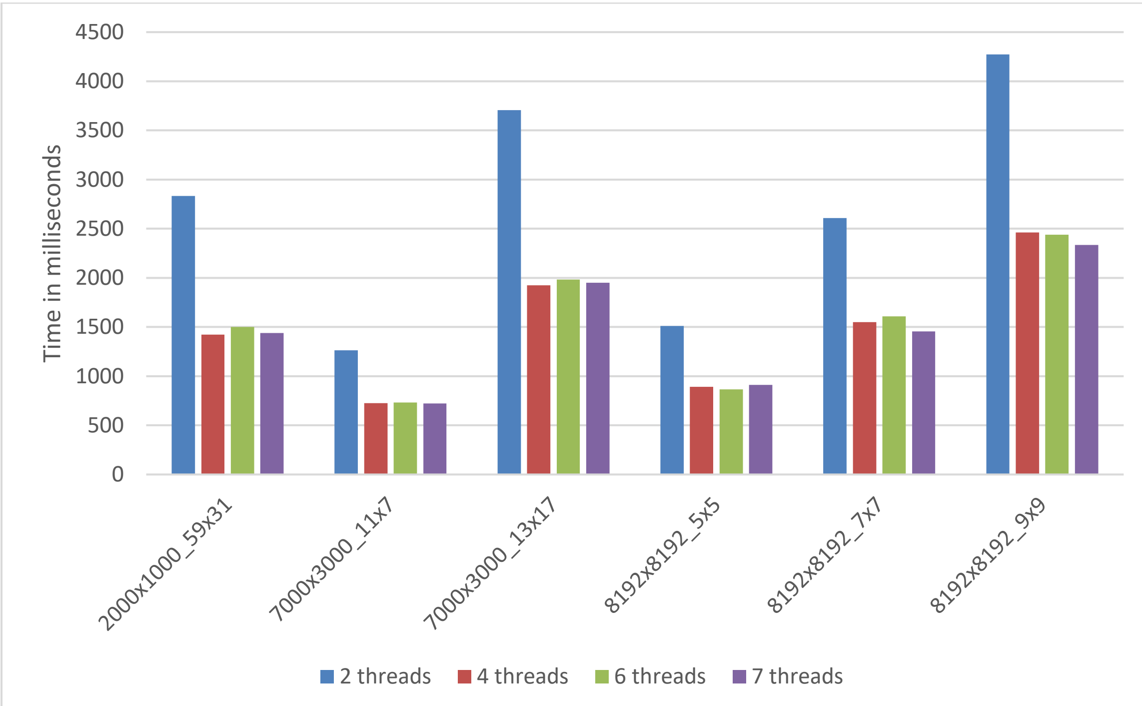
For performance analysis, we have been provided with different sizes of kernel matrices and input matrices. From part 1A we found out that using SIMD gives approx. 4x performance, so we implemented the same SIMD code in multi-thread as well. Table 1.4 shows the execution time of multithreaded code using different number of threads. Here reference time is the execution time of single threaded unoptimized implementation of DC.

		Number of threads			
Dimensions (Input_kernel)	**Reference (in ms)**	**2**	**4**	**6**	**7**
4096x4096_3x3	1242.89	175.297	105.634	120.67	102.633
4096x4096_5x5	3496.31	372.171	202.054	254.34	201.566
4096x4096_7x7	6632.59	652.109	346.325	409.462	347.125
4096x4096_9x9	11293.8	1033.11	539.662	612.992	573.504
4096x4096_11x11	15526.7	1509.42	774.279	891.862	838.037
4096x4096_13x13	21413.2	2141.49	1062.18	1185.95	1173.46
4096x4096_64x64	512481	48892.5	27541	28154.1	26933.5
8192x8192_3x3	5468.78	745.125	449.122	442.196	440.79
8192x8192_5x5	12979.9	1510.47	892.583	865.038	912.292
8192x8192_7x7	26613.9	2607.34	1551.45	1607.29	1453.96
8192x8192_9x9	47186.8	4273.86	2463.4	2438.08	2336.39
8192x8192_11x11	75508.7	6462.85	3531.31	3303.33	3435.04
8192x8192_13x13	96091.9	8684.42	4897.55	4631.78	4718.22
8192x8192_64x64	2234770	224682	119275	124948	115572
16384x16384_11x11	271371	27968	14718.4	15922.6	14607.8
16384x16384_13x13	362436	37659.3	20765.1	20401.9	19603.9
1000x500_3x5	59.184	7.685	4.654	5.421	4.305
1000x500_5x7	132.441	14.73	7.83	9.249	8.393
1000x500_7x9	234.143	24.775	14.464	15.792	13.172
1000x500_9x13	436.348	43.821	23.163	28.004	23.325
1000x500_11x7	301.371	30.296	17.633	18.629	16.524
1000x500_13x17	801.266	84.465	41.514	49.159	42.366
1000x500_59x31	6070.1	681.63	345.28	358.724	350.185
2000x1000_11x7	1185.76	123.304	73.181	78.306	66.939
2000x1000_13x17	3265.3	348.335	182.603	215.366	178.679
2000x1000_59x31	26404.4	2833.25	1423.58	1502.08	1438.72
7000x3000_11x7	12840	1263.88	725.563	731.856	723.955
7000x3000_13x17	36475.1	3707.94	1923.43	1981.69	1949.28

Table 1.4: Multi-thread execution time with different threads

Here first column data 4096x4096_3x3 represents that 4096x4096 is the dimension of input matrix and 3x3 is the dimension of kernel matrix.

For few of the rows of the above table (table 1.4), the following graph 1.3 shows the execution time of different threads and their respective execution time in milliseconds. On y-axis, execution time is shown in milliseconds, on the x-axis we have size combination of input and kernel matrices (Ex. 2000x1000_59x31 represents dimension of input matrix = 2000x1000 and kernel matrix = 59x31) and the bar shows execution time in ms w.r.t to different number of threads.



Graph 1.3: Multi thread execution time with different threads

As we can see in the given graph that using 7 threads (purple bar) gives the maximum speed up in the multithreaded implementation of DC.

4.2 Results

In multithreading we found out that using 7 threads gives us the speed up between 14x to 19x in sufficiently large input sizes with respect to reference of unoptimized single thread execution time.

5. Conclusion

In summarizing our findings, during the performance analysis using 'perf,' we identified that the bottleneck in the code was attributable to data cache load misses in the L1-d cache. To address this issue, our initial attempt involved loop interchange, but it yielded no optimization benefits. Subsequently, we explored loop unrolling, resulting in a notable speedup of 1.5 times compared to the reference implementation. Seeking further improvements, we experimented with blocking techniques, but the observed enhancement was negligible.

Upon delving into SIMD (Single Instruction, Multiple Data) optimization, we discovered that our system supported the AVX2 extension, using SIMD capabilities. While implementing SIMD to optimize the code, we encountered a challenge where it processed data in 256-bit chunks, equivalent to 8 integers. This presented an issue when dealing with matrices whose column dimensions were not multiples of 8, so we had to use the multiple conditional statements.

To overcome this challenge, we addressed it by padding the matrices with an additional 8 columns, copied from the first 8 columns of each matrix. As a result, we achieved a substantial speedup of approx. 5.5 times in single thread and 18 times in multi thread with respect to reference time (unoptimized). So all in all we saw that the optimization techniques implemented in this part significantly improved the overall performance of the Dilated Convolution on the CPU.