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 B

Implement and optimize DC in CUDA (GPU)

1.1 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 undertaking revolves around developing and optimizing a version of dilated convolution specifically tailored for GPU utilization, employing CUDA rather than single or multithreading. This detailed report delves into the nuanced aspects of the optimization efforts directed at GPU-accelerated 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.

1.2 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	AMD Ryzen 5 4600H with Radeon Graphics, 3.00 GHz
Memory	8 GB DDR4
OS	Ubuntu 22.04.3 LTS, Kernel: 6.2.0-36-generic
Cache	384 KB L1, 3 MB L2, 8 MB L3

Table 1.1

1.3 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. We succinctly presents the execution time for the single-threaded implementation across 16 different combinations of these matrices.

For performance analysis, 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 selected input and kernel matrices.

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.

1.4 Bottleneck Identification

In order to pinpoint the bottleneck in the single-threaded execution of DC, we utilized perf to record a range of miss events. The miss events considered include Cache load, Cache store, TLB miss, and LLC miss.

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.

1.5 Optimization strategies

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

1.5.1 Loop Interchange

Loop interchange is a loop transformation technique used in compiler optimization to improve the performance of programs by changing the nesting order of loops. The basic idea is to interchange the order of two nested loops to enhance data locality and improve cache performance.

In the given single-threaded DC we have 4 nested loops (ABCD let), with that we can have 24 permutations of loops. Table below shows the execution time of all 24 permutation of loops.

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 some other strategies.

1.5.2 Loop Unrolling

Loop unrolling is a compiler optimization technique that involves unfolding the loop to expose more opportunities for parallelization.

1.5.3 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.

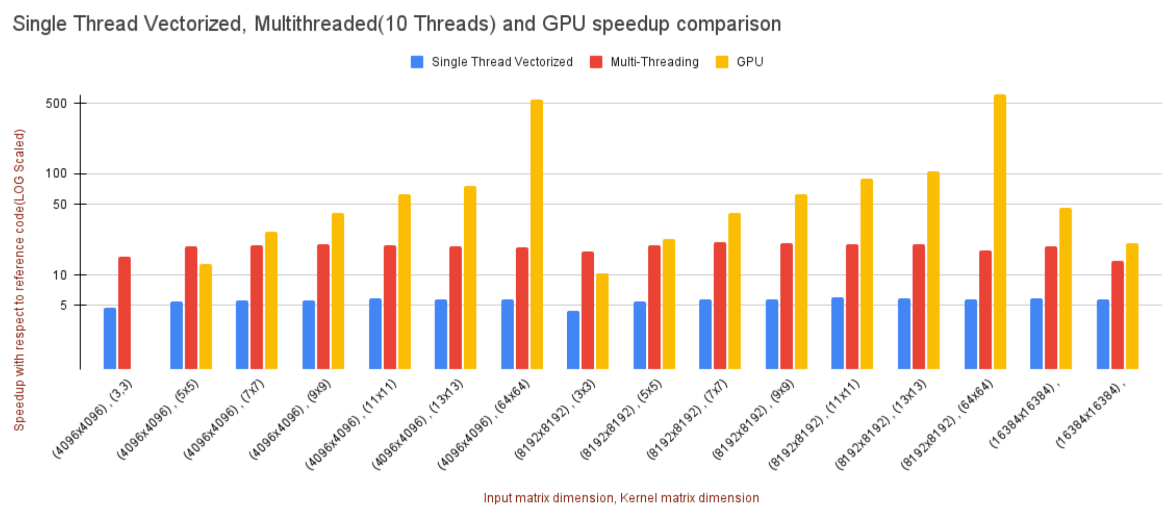
1.5.4 Using SIMD

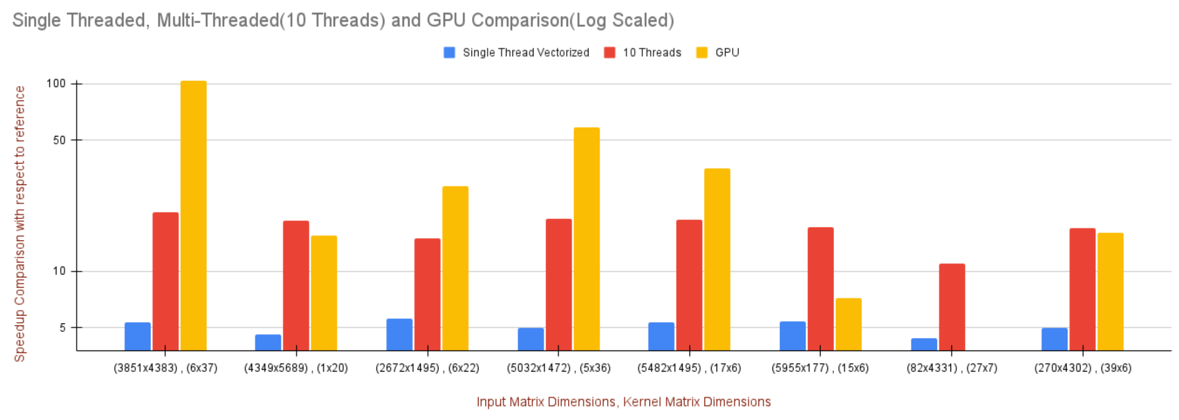
SIMD stand for Single Instruction, Multiple Data. The basic idea behind using SIMD is to use single instruction to perform the same operation on a batch of data elements simultaneously, rather than processing each element sequentially.

The system we are working on has AVX2 extension which provides a set of instructions to perform SIMD which operates on 256-bit registers, which can hold 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 shows

1.6 Results





1.7 Conclusion

As we can observe from the resultsGPUcodes implementation works better for larger matrices compared to smaller input or kernel matrices. GPU gave us overage **150x** speedup approximately. We got the best performance in multithreading code when we launched **32x32 thread block size** in our system. Hence, we've got obtained performance significantly better than reference program.