E0 243 High Performance Computer Architecture
Assignment 2 – Part B

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

November 26, 2023

Part B
Implementing and Optimizing Performance of Dilated Convolution (DC) in CUDA

1. Introduction

In this segment of the report, we conduct Dilated Convolution (DC) through NVIDIA's CUDA (Compute Unified Device Architecture). In contrast to the preceding section where we focused on single-thread and multi-thread optimization, our current emphasis is on increasing the performance in NVIDIA GPUs.

CUDA is a powerful platform that empowers developers to tap into the parallel processing capabilities of NVIDIA GPUs. It has become a standard for GPU computing and has played a significant role in advancing the field of parallel programming for high-performance computing.

2. Machine Specifications

CPU	9th 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
Graphics	NVIDIA GTX 1050 Mobile
GPU Memory	4GB DDR 5
CUDA Cores	640

Table 1.1: Machine Specifications

3. Performance Analysis

The DC algorithm has been performed using CUDA, which utilizes the GPU threads to concurrently execute multiple elements of the matrices. We have

Number of threads per block =1024

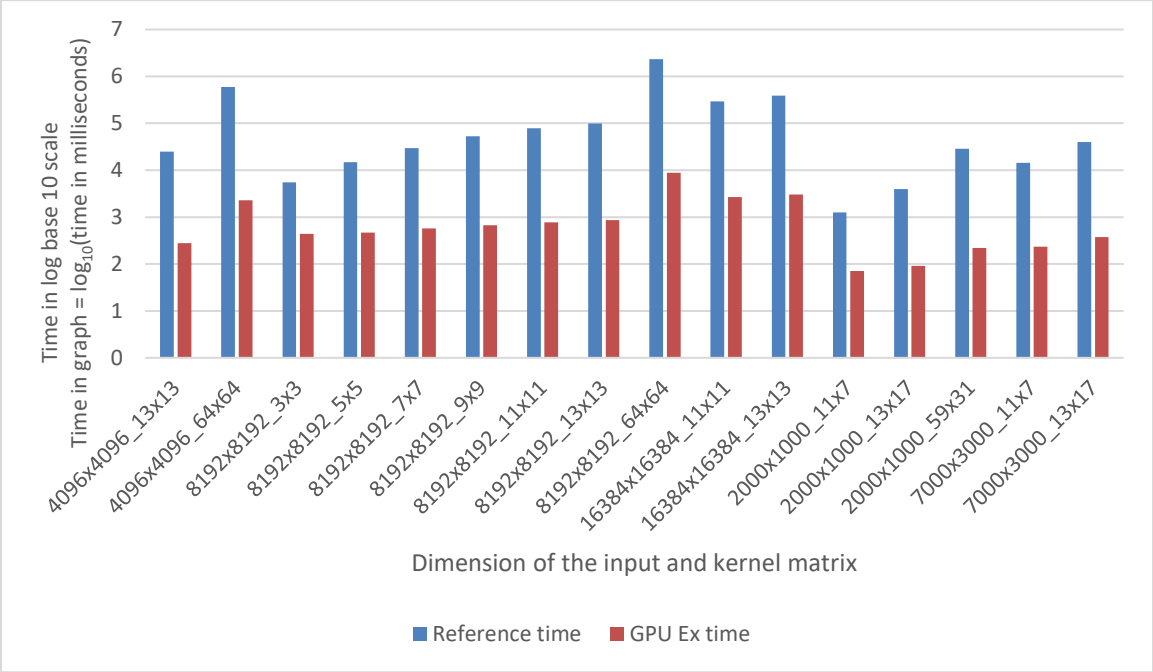
Also, since GPU executes threads in groups of 32 parallel threads, called warps, we are trying to implement blocks of sizes in the multiple of 32. If we do not use the size in the multiple of 32, it may lead to partial warp execution leading to some threads being idle within a warp and will impact the overall efficiency of the GPU. We tried with different thread block sizes multiple of 32, but we got the best results when thread block size was 32x32.

Table 2.1 shows the execution time of CUDA code with block of 32x32.

Dimension Input_kernel	Reference time (in ms)	GPU Ex time (in ms)
4096x4096_3x3	1287.06	154.057
4096x4096_5x5	3495.15	165.233
4096x4096_7x7	6453.44	193.633
4096x4096_9x9	11337.9	198.652
4096x4096_11x11	17448.3	215.645
4096x4096_13x13	24752.6	278.559
4096x4096_64x64	590290	2277.23
8192x8192_3x3	5533.76	435.998
8192x8192_5x5	14766.6	467.474
8192x8192_7x7	29485.9	573.77
8192x8192_9x9	53187.3	670.878
8192x8192_11x11	78380.8	767.513
8192x8192_13x13	99218.7	868.728
8192x8192_64x64	2339310	8835.31
16384x16384_11x11	292608	2662.64
16384x16384_13x13	390965	3046.86
1000x500_3x5	63.155	61.263
1000x500_5x7	153.12	57.267
1000x500_7x9	274.002	58.298
1000x500_9x13	514.729	62.438
1000x500_11x7	306.397	56.919
1000x500_13x17	839.573	60.442
1000x500_59x31	6431.98	90.031
2000x1000_11x7	1259.01	70.545
2000x1000_13x17	3930.8	91.243
2000x1000_59x31	28854.9	219.888
7000x3000_11x7	14275.5	234.813
7000x3000_13x17	39828.2	375.727

Table 1.2: GPU execution time and reference time comparison
Here data in column1, 4096x4096_3x3(say) represents that 4096x4096 is the dimension of input matrix and 3x3 is the dimension of kernel matrix

For few of the rows in the above table (table 1.2) the following graph 1.1 shows the Execution time vs combination of Input and kernel matrix, where execution time is on log base 10 of actual time in milliseconds. The blue bar shows the reference execution time and red bar shows the GPU execution time.



Graph 1.1: GPU execution time and reference time comparison

4. Results

From the graph 1.1 and the table 1.2, we can see that the speedup we got for the smaller input is approx. 10x and the speedup in case of larger input is approx. 300x. This may be because for the smaller input matrix the overhead due to copying the input matrix and kernel matrix from host to device is significantly larger than the actual execution time but in case of larger input and kernel matrix the execution time is larger than the overhead of copying input and kernel matrix so in this case, the execution time hides the overhead time which yields the better speedup for larger input matrix.

5. Conclusion

As we saw in the Analysis and Results section that the optimization of the Dilated Convolution algorithm in CUDA implementation, we have achieved significant improvement in performance with respect to the reference of single threaded CPU execution. In the optimization process, we experimented with the different thread block sizes and identified the most efficient block size for our DC implementation. There can be further improvement in the performance of CUDA implementation using shared memory, as shared memory access is much faster than the global memory access. Our results show the potential of GPUs that how it can be used to speedup DC computations. The speedup we achieved can be beneficial for the real word applications where we require fast processing of high-resolution images or signals.