E0243: High Performance Computer Architecture
Assignment 2 (Part B)

Yash Patel
SR - 22759
yashj@iisc.ac.in

Anjali Chauhan
SR - 22703
anjalic@iisc.ac.in

November 26, 2023

Implementing Dilated Convolution for a GPU using CUDA

Introduction

Convolution is a basic operation in deep learning and image processing, and for many applications, performance optimization is essential. The idea is to increase the efficiency of convolution operations by taking advantage of parallelism and a GPU's computing capacity.

CUDA

NVIDIA created the parallel computing platform and programming model known as CUDA (Compute Unified Device Architecture). By shifting computationally demanding activities to the GPU, CUDA gives users a parallel programming model and a set of tools to speed up programmes. It comes with libraries, runtime components, and a specialised C,C++ like programming language called CUDA C for creating parallel programmes. Through the use of CUDA, developers can take advantage of the GPUs' parallel processing capacity, which speeds up and improves the efficiency of activities in a variety of industries, including scientific computing, machine learning, and image processing.

Processor Specification

We used a system with the given specifications to execute our code and perform optimization.

CPU	AMD Ryzen 7 6800H
CPU Memory	16 GB
GPU	NVIDIA GeForce RTX 3050
GPU Memory	4 GB
CUDA Version	12.2

Table 1: Processor Specification Details

Implementation Details:

1. **CUDA Kernel Implementation:** The "convolutionKernel" CUDA kernel serves as the implementation's central component. This kernel uses a 2D kernel to compute the convolution of a 2D input matrix by utilising GPU parallelism. Each thread is effectively mapped to a distinct output element through the usage of thread indices, and the convolution result is accumulated by the kernel through iteratively going over the input and kernel matrices. The block and grid sizes are set up to maximise parallel processing. The two-dimensional grid (gridDim) and block (blockDim) work together to efficiently divide the burden among the GPU's threads so that they can compute the convolution simultaneously.

2. **Memory Management:** A crucial component of GPU programming is memory management, and the implementation manages data transmission between the host and the GPU device in an acceptable manner. To allocate memory for input, kernel, and output data, one can use CUDA methods like cudaMalloc which facilitates the dynamic allocation of memory on the GPU. Data transport between the host and device memory is made more efficient with the usage of cudaMemcpy.

3. **Performance Considerations:** A number of factors, such as shared memory usage, algorithmic efficiency overall, and grid and block dimensions, affect the speed of the GPU-accelerated convolution. In order to maximise parallelism and minimise idle GPU resources, the 8x8 block dimensions that were selected and the grid dimensions computation are adjusted to the size of the task.

4. **Thread Launch and Synchronisation:** The gpuThread function serves as the GPU thread launcher. It starts the convolution kernel, sets up the grid and block sizes according to the size of the output matrix, and then copies the outcomes back to the host. The proper order of these actions is ensured by the synchronisation between the host and device.

Analysis

We have tested our CUDA implementation code on three different input matrices, and we have determined the execution times of the reference code and the CUDA implementation for different kernel sizes for each input matrix. The GPU execution time and reference code execution time performance comparison graphs are shown below.

We have not shown the results of computation with input matrix size is 16384 x16384 and kernel size is 64x64 as showing that in graph will make it hard to visualise other results.

- For a 4096x4096 input, CUDA implementation shows substantial speedup, ranging from 4.8x for a 3x3 kernel to an impressive 388.533x for a 64x64 kernel. This emphasizes the significant advantage of GPU parallelism for convolution tasks, especially with larger kernel sizes.

- The trend continues for an 8192x8192 input, with speedups ranging from 8.0429x to 429.004x. Larger input sizes magnify the benefits of GPU parallel processing, showcasing its scalability and efficiency across various kernel dimensions.

- In the case of a massive 16384x16384 input, CUDA maintains its effectiveness, achieving speedups from 9.42559x to 142.964x. This underscores the GPU's prowess in handling extensive computational workloads for convolution, demonstrating its suitability for large-scale data processing.

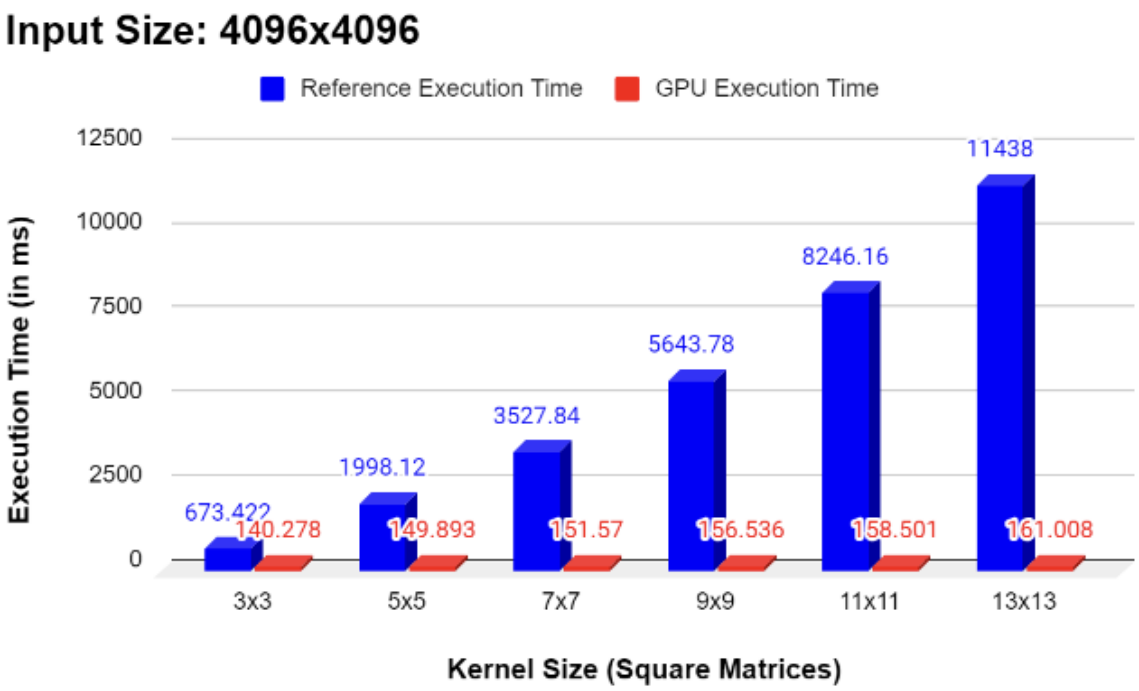

Figure 1: Input Matrix size: 4096 x 4096

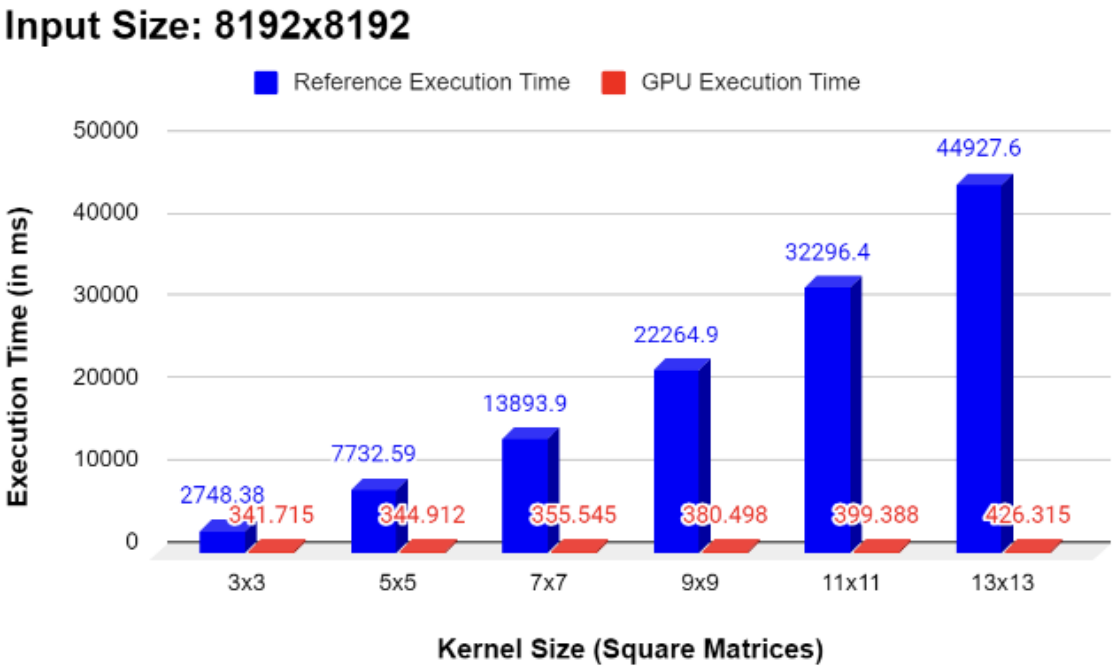

Figure 2: Input Matrix size: 8192 x 8192

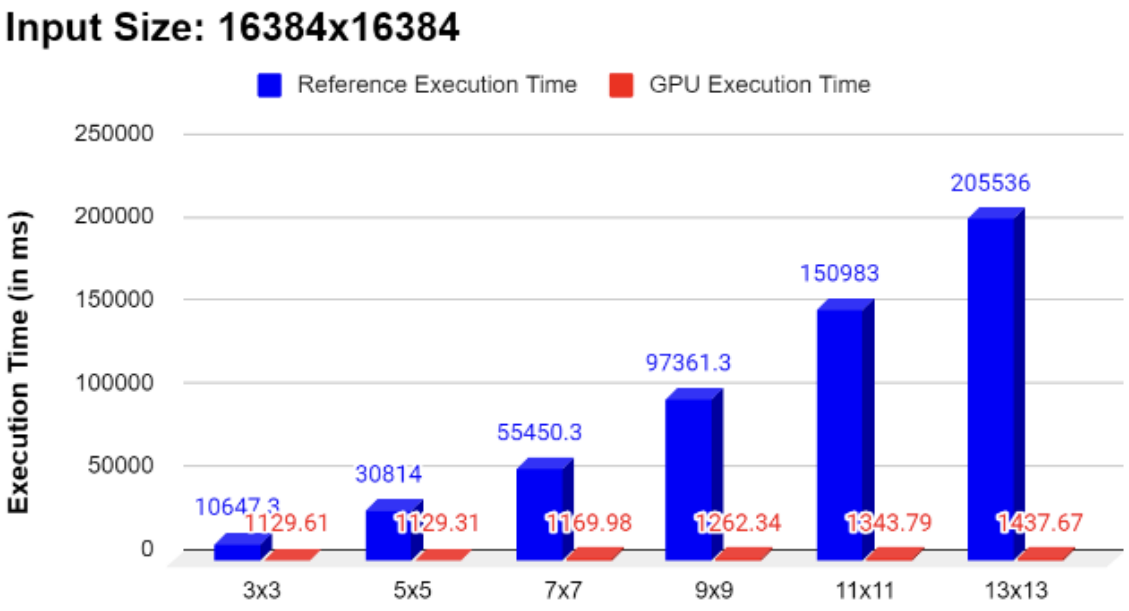

Figure 3: Input Matrix size: 16384 x 16384

- The GPU execution times are significantly lower than their CPU counterparts, indicating the superiority of parallel processing in CUDA for convolution tasks. This efficiency becomes more pronounced as the input size and kernel dimensions increase.
- The observed speedups align with expectations, as GPUs excel in parallelizing repetitive and computationally intensive tasks, such as convolution operations.
- The results highlight the suitability of CUDA for accelerating convolutional operations across varying input sizes and kernel dimensions, making it a compelling choice for tasks demanding high computational throughput.
- Larger kernel sizes showcase higher speedups, emphasizing the GPU's efficiency in handling complex convolutional operations.
- The consistent performance gains affirm the role of GPU acceleration in significantly enhancing the overall efficiency of convolutional operations for large-scale data processing.
- The GPU execution times remain relatively stable, reinforcing the reliability and predictability of CUDA performance across diverse input scenarios.
- These findings underscore the practical benefits of leveraging GPU resources for convolution tasks, offering substantial speedup and efficiency gains compared to traditional CPU implementations.

Conclusion

We have run our single threaded optimised code , multithreaded code with 8 threads and CUDA implementation of reference code for three different input sizes and for each input size we have computed speed up of all three implementations compared to reference code execution time for

different sizes of kernel matrices. Below are the graph representing comparison of speed up of all three implementation:

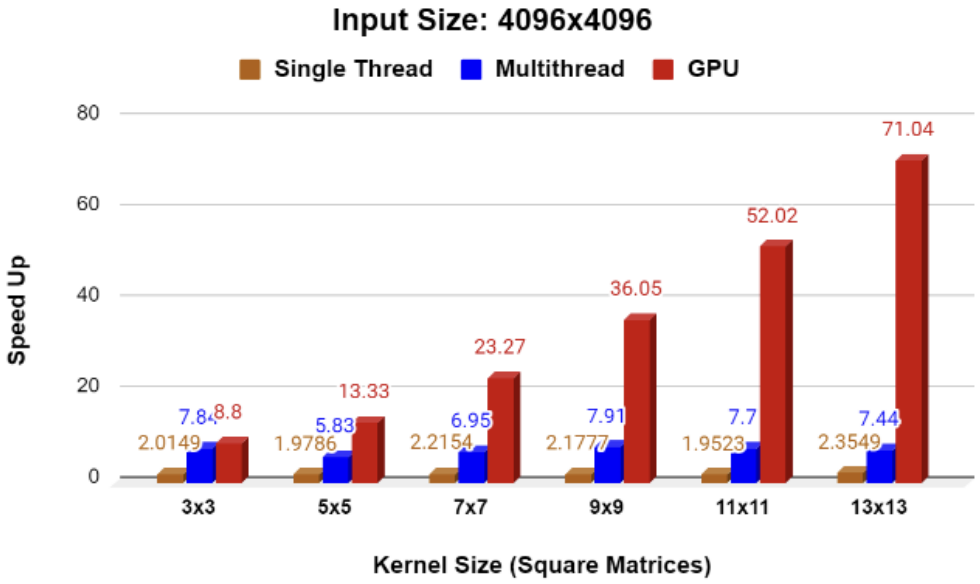

Figure 4: Speed up comparison with input matrix size: 4096 x 4096

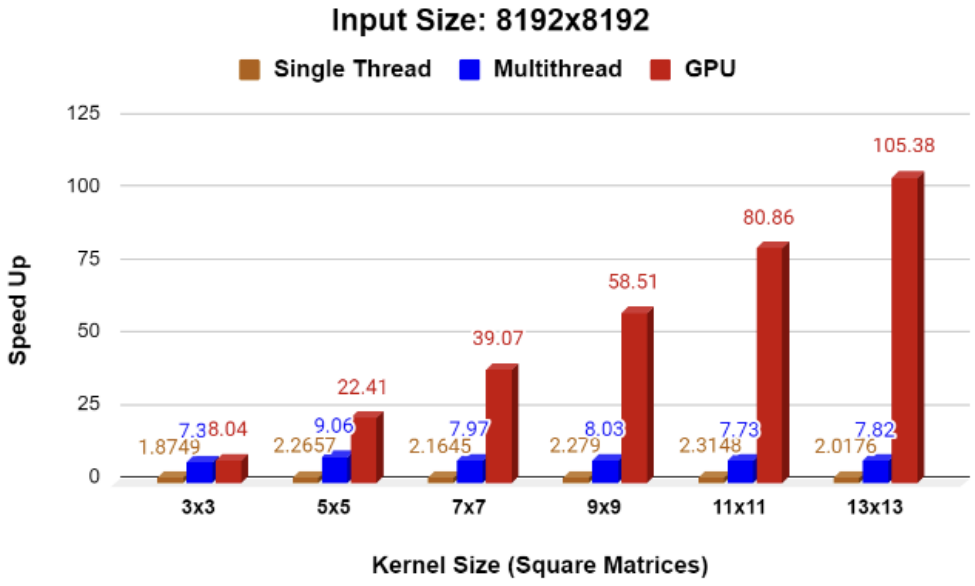

Figure 5: Speed up comparison with input matrix size: 8192 x 8192

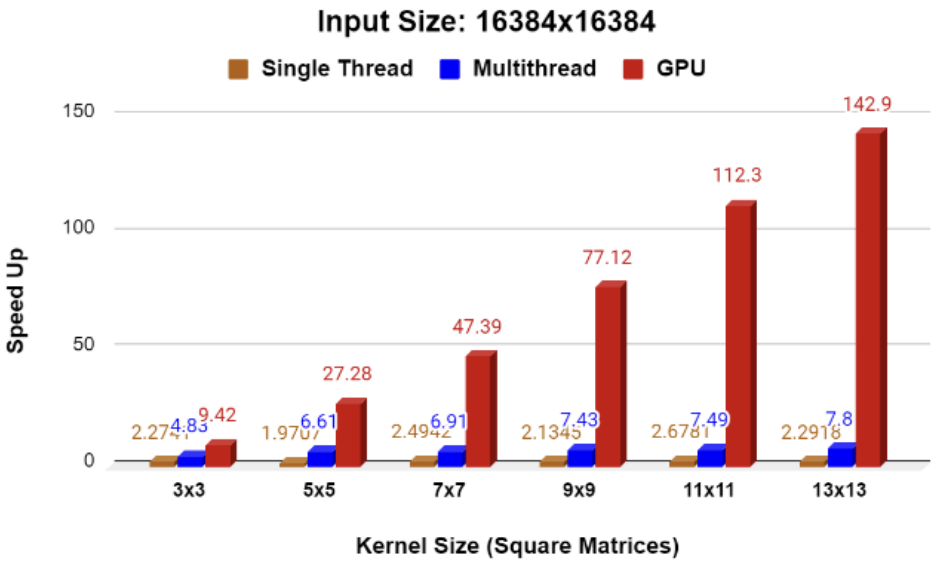

Figure 6: Speed up comparison with input matrix size: 16384 x 16384

The provided observations reveal the superior performance of GPU acceleration over both single-threaded and multithreaded CPU implementations for convolution operations. Across various input sizes and kernel dimensions, the GPU consistently achieves remarkable speedups, showcasing its prowess in parallel processing. Larger input matrices and kernel sizes exhibit higher speedups, emphasising the scalability and efficiency of GPU acceleration for handling complex convolution tasks. While multithreading improves performance compared to single-threading, the GPU outperforms both, particularly in scenarios involving extensive computational loads. The trend underscores the reliability and efficiency of GPU acceleration for convolutional operations, making it a compelling choice for high-performance computing applications. Overall, the observations highlight the substantial advantages of leveraging GPU resources to achieve significant speedup and efficiency gains in convolution tasks compared to traditional CPU implementations.