E0 - 243 Computer Architecture Assignment - 2

Mandar Bhalerao - mandarmanoj@iisc.ac.in
Kunal Pansare - kunaldinesh@iisc.ac.in

November 26, 2023

1 Introduction

We are required to optimize the problem of "Dilated Convolution(DC)" which involves an input matrix and a kernel matrix. The output of the dilated convolution is stored in the output matrix. The dimensions of these matrix are:-
Input:
1. Input Matrix: Input_Row x Input_Column
2. Kernel Matrix: Kernel_Row x Kernel_Column
Output:
Output Matrix: (Input_Row-Kernel_Row+1) x (Input_Column-Kernel_Column+1)
We will implement and optimize dilated convolution for a GPU using CUDA

2 Theory

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use NVIDIA GPUs (Graphics Processing Units) for general-purpose processing (not just graphics) to accelerate computationally intensive tasks. CUDA enables the creation of parallel programs that can execute on NVIDIA GPUs, harnessing the massive parallel processing power of these devices.

In CUDA programming, the concepts of threads, block size, and grid size are essential for organizing and managing parallel computations on the GPU.
a) Threads:A thread is the smallest unit of execution in a CUDA program. Each thread represents an individual execution flow.
b) Blocks: A block is a group of threads that can be executed together on a multiprocessor of the GPU.
c) Grids: A grid is a collection of blocks. It represents the entire set of threads that are launched for a kernel execution.

Functions of CUDA used are:
a) cudaMalloc(): In CUDA programming, cudaMalloc is a function used for allocating memory on the GPU (Graphics Processing Unit). It is part of the CUDA Runtime API and is often used to allocate device memory for data that will be processed on the GPU.

b) cudaMemcpy(): cudaMemcpy is a function in the CUDA programming environment that is used to copy data between different memory spaces, such as between the host (CPU) and the device (GPU) or between different regions of device memory. It is part of the CUDA Runtime API.

c) cudaFree():cudaFree is a function in the CUDA programming environment that is used to free memory that was previously allocated on the GPU using cudaMalloc or related memory allocation functions. It is part of the CUDA Runtime API.

3 Performance Analysis

a. Improvement in CUDA(%) :- The metric calculates the percentage reduction in execution time achieved by the CUDA compared to a reference execution. This metric quantifies the efficiency gain achieved by CUDA, expressing it as a percentage improvement relative to the reference execution time.

Improvement(%)=(1-(CUDA Execution Time/Reference Execution Time))*100

b. Speedup in CUDA :- Speedup is a performance metric that quantifies the improvement in the execution time of a program or algorithm when compared to a reference or baseline execution. It is typically expressed as the ratio of the reference execution time to the improved execution time. The speedup is calculated using the formula:

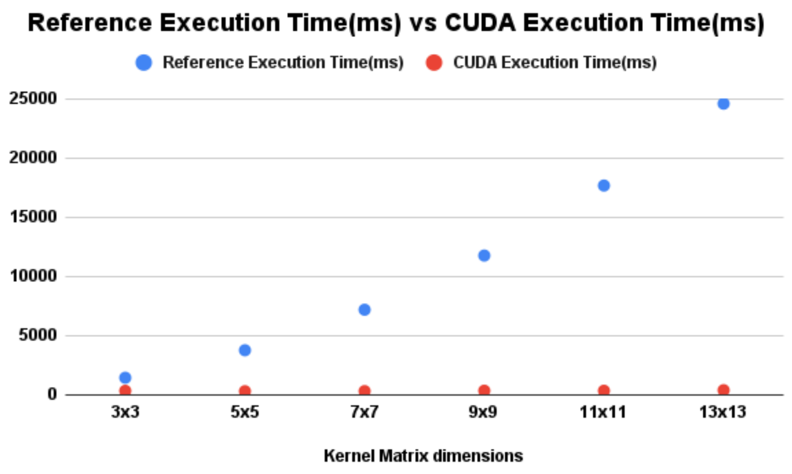
Speedup=Reference Execution Time/CUDA Ececution Time

Sr. No	Kernel Matrix dimensions	Reference Execution Time(ms)	CUDA Execution Time(ms)	Imp in CUDA(%)	Speedup(CUDA)
1	3x3	1448.3	344.8	76.18	4.19
2	5x5	3771.4	310.7	91.76	12.13
3	7x7	7193.2	318.8	95.56	22.55
4	9x9	11768.5	355.4	96.97	33.10
5	11x11	17678.5	351.5	98.01	50.29
6	13x13	24592.7	398.7	98.37	61.68
7	64x64	573759	1105.81	99.80	518.85

Table 1: Input Matrix with dimensions 4096x4096

Sr. No	Kernel Matrix dimensions	Reference Execution Time(ms)	CUDA Execution Time(ms)	Imp in CUDA(%)	Speedup(CUDA)
1	3x3	5856.7	683.2	88.33	8.57
2	5x5	15122.8	755.2	95.00	20.02
3	7x7	29028.9	749.2	97.41	38.74
4	9x9	47549.5	762.5	98.39	62.35
5	11x11	71320.6	808.1	98.86	88.24
6	13x13	99510.5	942.9	99.05	105.53
7	64x64	2338100	3893.7	99.83	600.47

Table 2: Input Matrix with dimensions 8192x8192



Generalized behaviour of reference vs CUDA

4 Conclusion

We delved into the implementation and optimization of dilated convolution using CUDA, a parallel computing platform. The primary objective was to harness the power of GPU parallelism to expedite dilated convolution operations, which are prevalent in modern deep learning architectures.

Our CUDA implementation showcased significant performance improvements compared to a single-threaded CPU approach. The parallel nature of CUDA allowed us to exploit the massive parallelism inherent in GPUs, resulting in faster computation times for dilated convolution across various kernel sizes.