# Optimizing Performance Of Dilated Convolution

Anishka Ratnawat
*Computer Science and Automation*
*IISc, Bangalore, India*
*anishkar@iisc.ac.in*

Snigdha Shekhar
*Computer Science and Automation*
*IISc, Bangalore, India*
*snigdhas@iisc.ac.in*

**Abstract**

**A Graphics Processing Unit (GPU) is a specialized processor designed to accelerate graphics rendering. However, due to their highly parallel architecture, GPUs have found applications beyond graphics and are widely used for parallel processing tasks. In the context of parallel processing, a GPU is a massively parallel processor that can efficiently perform thousands of concurrent operations.**

**The challenge for the GPU programmer is not simply getting good performance on the GPU, but also in coordinating the scheduling of computation on the system processor and the GPU and the transfer of data between system memory and GPU memory. GPUs have virtually every type of parallelism that can be captured by the programming environment: multithreading, MIMD, SIMD, and even instruction-level.**

**NVIDIA decided to develop a C-like language and programming environment that would improve the productivity of GPU programmers by attacking both the challenges of heterogeneous computing and of multifaceted parallelism. The name of their system is CUDA, for Compute Unified Device Architecture. CUDA produces C/C++ for the system processor (host) and a C and C++ dialect for the GPU (device, thus the D in CUDA).**

**NVIDIA decided that the unifying theme of all these forms of parallelism is the CUDA Thread. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can group thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism**

## I. Introduction

Graphics Processing Units (GPUs) optimize code execution through parallelism and highly specialized architectures tailored for specific types of computations. It uses various styles of parallelism to optimize code including SIMD, MIMD, thread-level parallelism. GPUs also rely on multithreading within a single multithreaded SIMD Processor to hide memory latency. However, efficient code for both vector architectures and GPUs requires programmers to think in groups of SIMD operations.

A Grid is the code that runs on a GPU that consists of a set of Thread Blocks. A Thread Block is assigned to a Streaming Multiprocessor (SM) that executes that code by the Thread Block Scheduler. The programmer tells the Thread Block Scheduler, which is implemented in hardware, how many Thread Blocks to run. It is similar to a vector processor, but it has many parallel functional units instead of a few that are deeply pipelined, as in a vector processor. SIMD Processors are full processors with separate PCs and are programmed using threads. The GPU hardware then contains a collection of multithreaded SMs that execute a Grid of Thread Blocks (bodies of the vectorized loop); that is, a GPU is a multiprocessor composed of multithreaded SIMD Processors.

A GPU can have from one to several dozen multithreaded SIMD Processors. For example, the Pascal P100 system has 56, while the smaller chips may have as few as one or two. To provide transparent scalability across models of GPUs with a differing number of multithreaded SIMD Processors, the Thread Block Scheduler assigns Thread Blocks (bodies of a vectorized loop) to multithreaded SIMD Processors.

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a thread of SIMD instructions. It is a traditional thread that exclusively contains SIMD instructions. These threads of SIMD instructions have their own PCs, and they run on a multithreaded SIMD Processor. The SIMD Thread Scheduler knows which threads of SIMD instructions are ready to run and then sends them off to a dispatch unit to be run on the multithreaded SIMD Processor. Thus GPU hardware has two levels of hardware schedulers: (1) the Thread Block Scheduler that assigns Thread Blocks (bodies of vectorized loops) to multithreaded SIMD Processors and (2) the SIMD Thread Scheduler within a SIMD Processor, which schedules when threads of SIMD instructions should run. Because the thread consists of SIMD instructions, the SIMD Processor must have parallel functional units to perform the operation which are called SIMD lanes. [1]

## II. Methodology

### A. System Configuration

We used Google Colab to implement DC for a GPU using CUDA. The configuration of the system on Colab is as follows:

| Model name: | Intel(R) Xeon(R) CPU @ 2.00GHz |
|---|---|
| GPU Name | NVIDIA Tesla T4 |
| CUDA Version | 12.0 |
| System RAM | 12.7 GB |

TABLE I: Processor Configuration

### B. GPU Otimization Strategy

The code is structured as a CUDA kernel, which allows it to be executed in parallel on multiple threads. The gpuThreadKernel function is designed to be executed by many threads in parallel.The kernel is organized into a grid of thread blocks (gridSize) and each block contains multiple threads (blockSize). This structure allows for efficient organization and utilization of GPU resources.The code allocates device memory (cudaMalloc) for input, kernel, and output matrices on the GPU. Data is then transferred from the host (CPU) to the device (GPU) using cudaMemcpy. Thread indices are calculated using blockIdx and threadIdx to determine the position of each thread in the grid. This indexing is used to map threads to specific elements of the input and output matrices. The block and grid dimensions (blockSize and gridSize) are chosen to maximize GPU occupancy, ensuring that a sufficient number of threads are active to fully utilize the GPU resources. This can improve overall execution time by reducing idle time.

### C. Code

Following is the code used to optimize the Dilated Convolution using GPU:

```
#include <cuda_runtime.h>

// CUDA kernel for the computation
__global__ void gpuThreadKernel(int input_row, int input_col,
                                int *input, int kernel_row, int kernel_col,
                                int *kernel, int output_row, int output_col,
                                long long unsigned int *output) {
    // Calculate thread indices
    int output_i = blockIdx.y * blockDim.y + threadIdx.y;
    int output_j = blockIdx.x * blockDim.x + threadIdx.x;
    int dilation = 2;
    int input_i, input_j;
    long long unsigned int partial_sum;
    // Check if thread is within the output dimensions
    if (output_i < output_row && output_j < output_col) {

        int output_row_skip = output_i * output_col;
        partial_sum = 0;
        input_i = output_i;

        for (int kernel_i = 0; kernel_i < kernel_row; ++kernel_i) {
            int kernel_row_skip = kernel_i * kernel_col;
            int input_row_skip = input_i * input_col;
            input_j = output_j;

            for (int kernel_j = 0; kernel_j < kernel_col; ++kernel_j) {

                partial_sum = partial_sum + input[input_row_skip + input_j]
                * kernel[kernel_row_skip + kernel_j];
```

```
                input_j = input_j + dilation;
                if(input_j >= input_col)
                    input_j = input_j % input_col;
            }

            input_i = input_i + dilation;
            if(input_i >= input_row)
                    input_i = input_i % input_row;
        }

        output[output_row_skip + output_j] = partial_sum;
    }
}


// Wrapper function to call the CUDA kernel
void gpuThread(int input_row, int input_col, int *input,
               int kernel_row, int kernel_col, int *kernel,
               int output_row, int output_col, long long unsigned int *output) {

    // Define block and grid dimensions
    dim3 blockSize(16,16);
    dim3 gridSize((output_col + blockSize.x - 1) / blockSize.x,
                  (output_row + blockSize.y - 1) / blockSize.y);

    // Allocate device memory
    int *device_input, *device_kernel;
    long long unsigned int *device_output;

    cudaMalloc((void **)&device_input, input_row * input_col * sizeof(int));
    cudaMalloc((void **)&device_kernel, kernel_row * kernel_col * sizeof(int));
    cudaMalloc((void **)&device_output,
    output_row * output_col * sizeof(long long unsigned int));

    // Copy input and kernel data to device
    cudaMemcpy(device_input, input, input_row * input_col *
    sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(device_kernel, kernel, kernel_row * kernel_col
    * sizeof(int), cudaMemcpyHostToDevice);

    // Launch the kernel
    gpuThreadKernel<<<gridSize, blockSize>>>(input_row, input_col, device_input,
                                             kernel_row, kernel_col, device_kernel,
                                             output_row, output_col, device_output);

    // Copy the result back to the host
    cudaMemcpy(output, device_output, output_row * output_col
    * sizeof(long long unsigned int), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(device_input);
    cudaFree(device_kernel);
    cudaFree(device_output);
}
```

### D. Implementation Details

The above code is implemented as follows:-

- Grid and Block Dimensions:
  - blockSize is set to a 2D block of size (16, 16). Each block contains 16 X 16 threads. A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension.
  - gridSize is calculated based on the output dimensions, ensuring that there are enough blocks to cover the entire output matrix. [2]
- Device Memory:
  - **cudaMalloc**: It is used to allocate device memory. The size of the allocated memory is determined by the dimensions of the matrices and their data types. Allocates memory on the GPU for the input matrix (device_input), kernel matrix (device_kernel), and output matrix (device_output).
  - **cudaMemcpy**: It is used to copy the input matrix and kernel matrix from the host (CPU) to the device (GPU). Then it is again used to copy the result (output matrix) from the device back to the host.
  - **cudaFree**: It frees the allocated device memory to avoid memory leaks.

## III. Observation

After implementing the dilated convolution for a GPU using CUDA, we have observed that the GPU execution time is much lower than the reference execution time. As we increase the size of the input matrix, the speedup also increases, as seen in TABLE II and TABLE III. It can also be seen in Fig. 1 through the line graph. Similarly, as we increase the size of the kernel matrix, the speedup also increases. As observed in TABLE II the size of the kernel matrix was 64x64, so the speedup was greater as compared to TABLE III where the kernel matrix is 13x13.

Secondly, we also observed that the performance improvement for a small input matrix such as 128x128 is negative, i.e., the GPU execution time is greater than the Reference execution time. This can be reasoned by the fact that when we have a small matrix size, the GPU capabilities may not be fully utilized, and the overhead of transferring data between the CPU and GPU, as well as managing the parallel processing, can overshadow the optimization performed by the GPU. Hence, it can incur more run time than the reference execution time.

| Input Matrix | Reference execution Time | GPU Execution Time | Speedup | Performance Improvement |
|---|---|---|---|---|
| 128x128 | 172.855 ms | 175.781 ms | 0.9833542874 | -1.6 |
| 1024x1024 | 44085.9 ms | 238.489 ms | 184.8550667 | 99.45903566 |
| 2048x2048 | 188580 ms | 535.822 ms | 351.9452355 | 99.71586488 |
| 4096x4096 | 793646 ms | 779.016 ms | 1018.780102 | 99.90184339 |
| 8193x8192 | 3.22652e+06 ms | 1790.51 ms | 1.80E+03 | 9.99E+01 |

TABLE II: Speedup and Performance of Different Input Matrix At kernal Matrix of 64x64

| Input Matrix | Reference execution Time | GPU Execution Time | Speedup | Performance Improvement |
|---|---|---|---|---|
| 1024x1024 | 1813.85 ms | 109.902 ms | 16.50424924 | 93.94095432 |
| 2048x2048 | 7230.41 ms | 313.968 ms | 23.02913036 | 95.65767363 |
| 4096x4096 | 33208.3 ms | 320.72 ms | 103.5429658 | 99.03421735 |
| 8192x8192 | 133592 ms | 624.366 ms | 213.9642453 | 99.53263219 |
| 16384x16384 | 512916 ms | 2021.49 ms | 253.7316534 | 99.60588283 |

TABLE III: Speedup and Performance of Different Input Matrix At kernal Matrix of 13x13

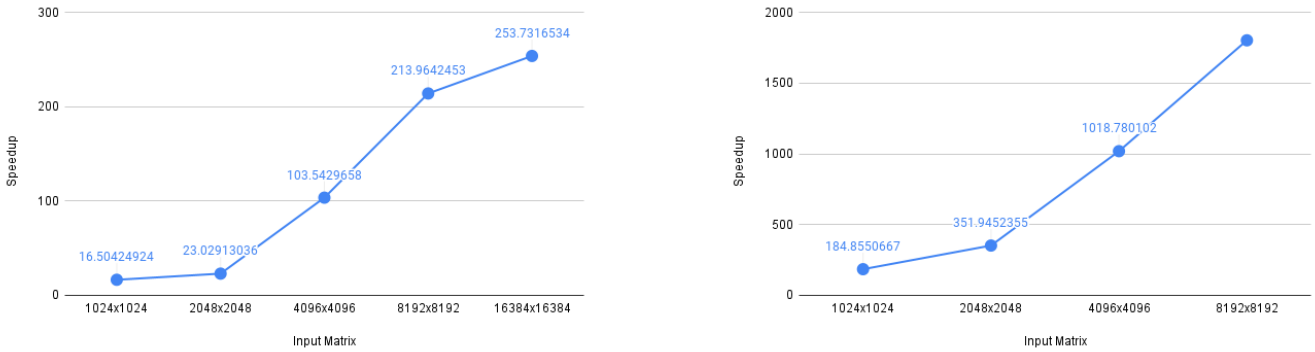


Fig. 1: Relation between Speed and input Matrix for 64x64 and 13x13 respectively

## IV. Conclusion

In this report, we have implemented the program for Dilated Convolution on a GPU using CUDA. We recorded the speedup obtained using the GPU on various input and kernel matrix sizes. Our GPU implementation of 2D convolution serves as a testament to the power of parallel processing in accelerating computationally intensive tasks. The achieved performance improvements have implications for a wide range of applications, from image processing to deep learning. By leveraging the parallel capabilities of GPUs, we contribute to the ongoing efforts to enhance the efficiency of convolution operations, opening avenues for improved performance in diverse computational domains.

## References

[1] Hennessy, J. L., Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach. India: Elsevier Science.
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html