# Boosting Convolutional Neural Networks Performance with GPU Acceleration

Final Project Report

Warren Liu, Chris Ma

School of STEM, Computer Science & Software Engineering

University of Washington Bothell

CSS535: High Performance Computing

Dr. Erika Parsons

March 15th, 2023

# Table of Contents

## *Abstract*

Convolutional Neural Networks (CNNs) have become a cornerstone of deep learning in computer vision, driving the need for efficient and powerful solutions. In this project, we investigate the performance of a custom CNN containing only convolutional and pooling layers, implemented on both CPU and GPU platforms. Utilizing OpenCV for image input and output, we compared the performance of the CPU-based implementation with the GPU-accelerated version, optimizing the GPU-based kernel functions using techniques such as shared memory usage, register memory usage, and loop unrolling. Our results demonstrate that GPU acceleration provides significant performance improvements, highlighting the potential benefits of using GPUs for deep learning tasks. However, we also encountered challenges, such as memory alignment issues and the complexity of 3D index calculations, emphasizing the need for careful consideration when implementing and optimizing CNNs on GPU platforms.

## 1. Introduction

Convolutional Neural Networks (CNNs), as shown in Figure 1, have emerged as a dominant technique in the field of deep learning, particularly for computer vision tasks such as image classification, object detection, and segmentation. CNNs are designed to automatically learn complex patterns and features from images by employing multiple layers of convolution and pooling operations, followed by fully connected layers for classification. The success of CNNs can be attributed to their ability to hierarchically learn representations from raw input data, reducing the need for handcrafted feature extraction and engineering.

Despite their impressive performance, training, and inference of CNNs can be computationally expensive, especially for large-scale problems and high-resolution images. This computational burden has led to a growing interest in leveraging the power of Graphics Processing Units (GPUs) to accelerate CNN processing, as GPUs offer massive parallelism and high computational throughput.

In this project, we build a CNN containing only convolutional and pooling layers, focusing on the most computational stress components of the network to gain a deeper understanding of the performance characteristics and potential bottlenecks in CNN processing. Our work compares the performance of a CPU-based implementation with a GPU-accelerated version, utilizing OpenCV for image input and output. We also explore various optimization techniques for the GPU-based kernel functions, such as shared memory usage, register memory usage, and loop unrolling, to further enhance the performance.

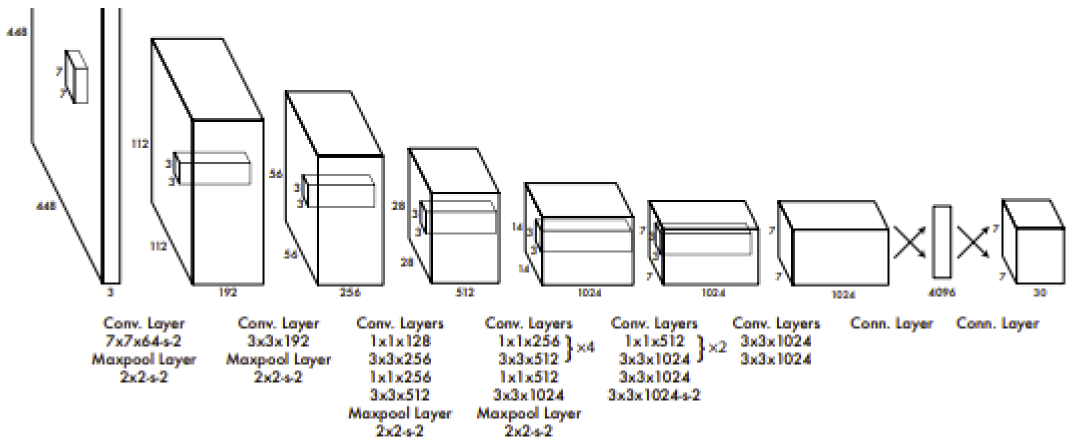


*Figure 1 A Typical CNN [1]*

## 2. Resources and Related Work

In this project, we explore the implementation of Convolutional Neural Networks (CNNs) on GPUs, inspired by the work of Redmon et al. [1] on the popular You Only Look Once (YOLO) object detection system. YOLO has significantly impacted the computer vision field and has undergone several updates, with the current version being YOLOv5 [2]. The convolutional and pooling layers in CNNs require extensive computation, making them a suitable candidate for GPU acceleration.

To understand the details of CNN layers, we referred to resources such as "What are convolutional neural networks?" by IBM [3] and "Beginners Guide to Convolutional Neural Networks" by Sabina Pokhrel [4]. Modern machine learning frameworks like TensorFlow, Caffe, and PyTorch have implemented GPU-accelerated versions of various neural networks, including CNN. Our goal in this project is to apply the GPU and high-performance computing knowledge acquired throughout this course to build a CNN on the GPU from scratch, focusing on the matrix operations involved in convolutional and pooling layers.

## 3. Methodology

### 3.1 Image I/O and Processing

In our project, we employ the OpenCV library to facilitate efficient and convenient input and output image handling. Notably, OpenCV represents images using its own `uchar` data type. For the CPU implementation, accessing `uchar` type pixels is straightforward, as we can utilize the `.at()` function provided by the OpenCV library. However, for the GPU implementation, we face the challenge of being unable to use host-provided functions, such as `.at()`.

To address this issue, we must convert the OpenCV Mat object representing the image into a 2D integer array before passing it to the CUDA kernel function. This conversion ensures compatibility with the GPU implementation and enables seamless processing of the images within the CNN. By adapting the image representation to the GPU's requirements, we can leverage the performance benefits of both CPU and GPU implementations while maintaining the convenience of OpenCV for image handling.

### 3.2 Convolutional Layer

In the convolutional layer, the input image undergoes a transformation to extract its features. This transformation involves convolving the image with a kernel (or filter), as illustrated in Figure 2. Our dataset, sourced from Kaggle and collected by UNMOVED.FINANCE [5], consists of grayscale images with a resolution of 150x150. We employed nine distinct filters capable of extracting various features from the images, including vertical and horizontal lines, left and right diagonal lines, cross lines, X-shaped lines, plus sign lines, and square and diamond shapes. Each filter is a 3x3 matrix. Consequently, the convolutional layer processes the 150x150 images by sliding the 3x3 filters over them, computing the dot product between the filter and the image region, and ultimately generating a 148x148 convolved matrix for each image.

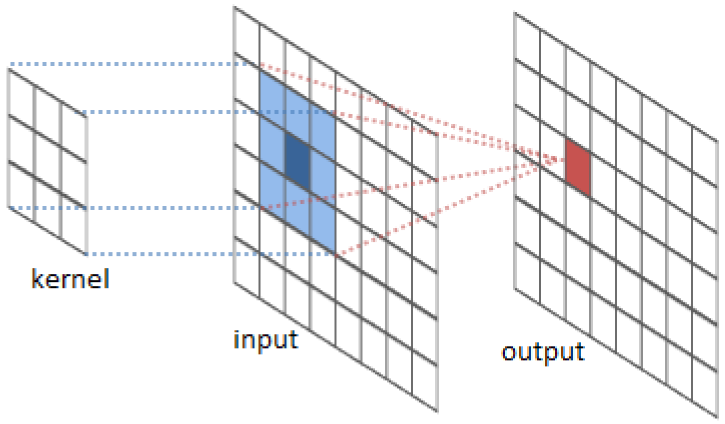


*Figure 2 Convolutional Layer [6]*

### 3.3 Pooling Layer

Similar to the convolutional layer, the pooling layer slides a kernel or filter across the input image. However, unlike the convolutional layer, the pooling layer reduces the number of parameters in the input as shown in Figure 3, which leads to some information loss. On the upside, this layer diminishes complexity and enhances the efficiency of the CNN. Various types of pooling exist, with max pooling and average pooling being the most prevalent methods in convolutional neural networks. In our implementation, we utilized max pooling. By striding a 3x3 filter over each image, we identified the maximum value within the 3x3 region and assigned it to the corresponding index in the output matrix. Consequently, the 148x148 images produced by the convolutional layer are downsized to 49x49 images in the pooling layer.

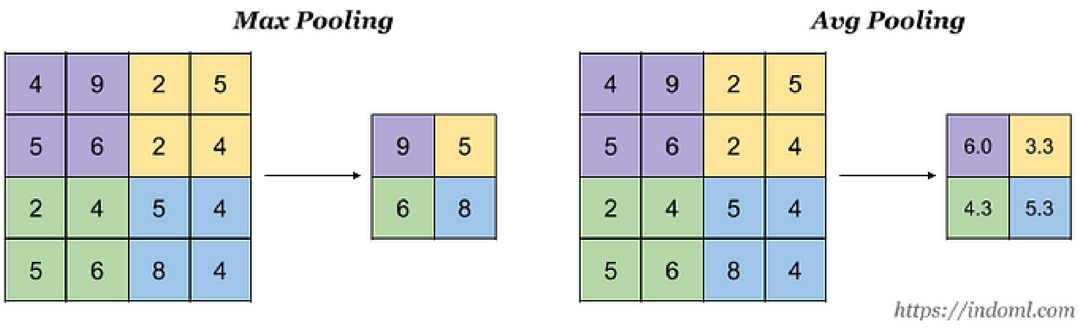


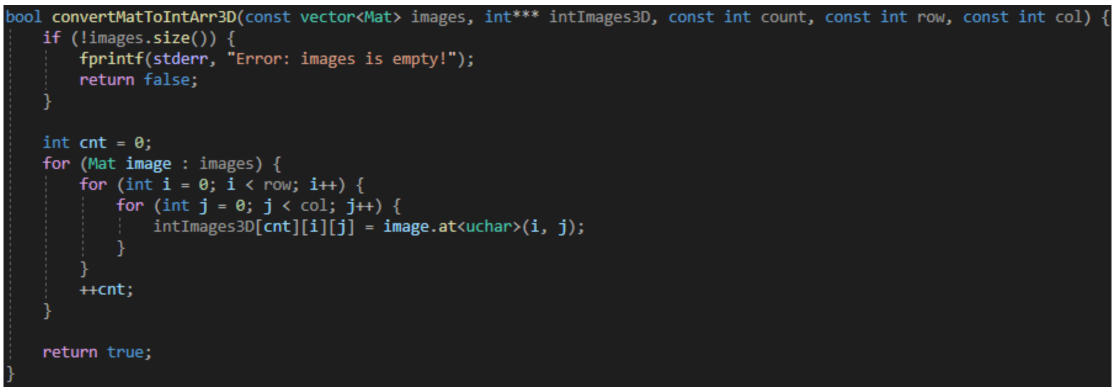
*Figure 3 Pooling Layer [7]*

### 3.4 Data Structure to Store Images

One of the most significant challenges in our project is handling the storage and representation of input images for GPU processing. In the CPU implementation, we can conveniently store all input images in a `vector<Mat>` object. However, as mentioned earlier, we need to convert images into 2D integer arrays for compatibility with CUDA kernel functions. Consequently, to store all images efficiently, we need to transform the `vector<Mat>` object into a 3D integer array as shown in Figure 4. Using a 2D integer array would require running the kernel function for each image separately, leading to a substantial performance decrease.

For faster access to images in the 3D array, we come up with a non-traditional approach for indexing. Instead of using the x and y axes for width and height and the z-axis for stacking different images, we use the x-axis for image indices in the array and the y and z axes for width and height, as shown in Figure 5. However, a new challenge arises: how to copy the 3D array to the device memory, given that we have only studied copying 1D and 2D arrays.

Upon further investigation, we discovered that cudaMalloc3D can be utilized to allocate 3D space on the device. We attempted to copy the 3D array, derived from the `vector<Mat>` object, to the device using cudaMemcpy3D. However, all attempts were unsuccessful. Our analysis revealed that the 3D array passed to the cudaMalloc3D-allocated 3D space must be contiguous in host memory, which is incompatible with the dynamically allocated array used to store images. Since our program is designed to accommodate various parameters, dynamic allocation is essential. As a solution, we decided to flatten the 3D array from the `vector<Mat>` object into a 1D dynamic array as shown in Figure 6, making it compatible with the CUDA kernel.



```
bool convertMatToIntArr3D(const vector<Mat> images, int*** intImages3D, const int count, const int row, const int col) {
    if (!images.size()) {
        fprintf(stderr, "Error: images is empty!");
        return false;
    }

    int cnt = 0;
    for (Mat image : images) {
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                intImages3D[cnt][i][j] = image.at<uchar>(i, j);
            }
        }
        ++cnt;
    }

    return true;
}
```

*Figure 4 Convert vector<Mat> Object to 3D Integer Array*

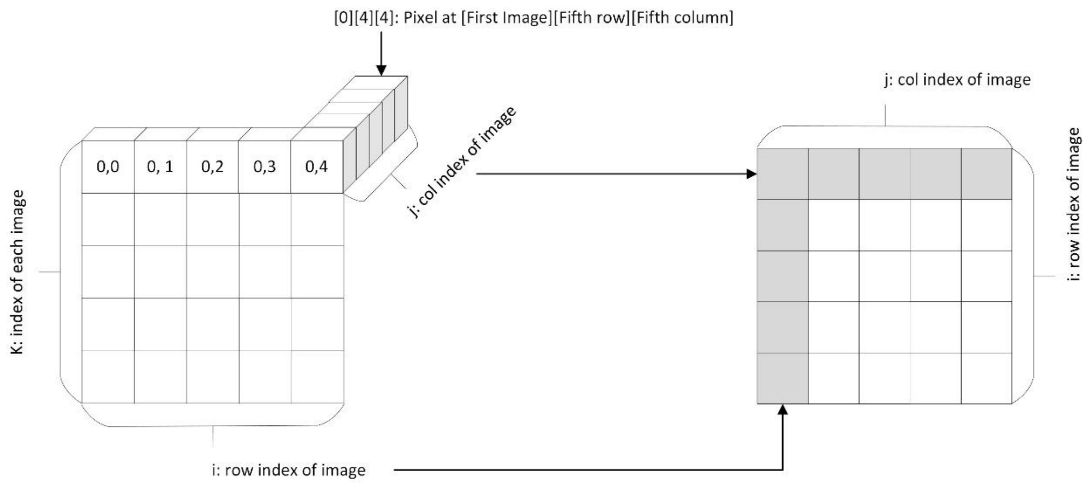
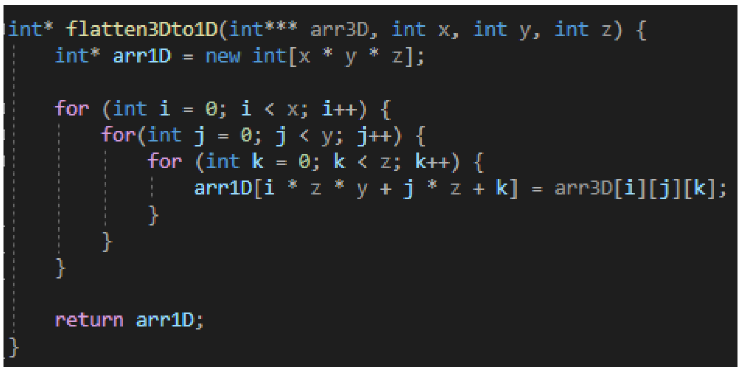


*Figure 5 3D Array to Store Images*



```
int* flatten3Dto1D(int*** arr3D, int x, int y, int z) {
    int* arr1D = new int[x * y * z];

    for (int i = 0; i < x; i++) {
        for(int j = 0; j < y; j++) {
            for (int k = 0; k < z; k++) {
                arr1D[i * z * y + j * z + k] = arr3D[i][j][k];
            }
        }
    }

    return arr1D;
}
```

*Figure 6 Convert 3D Array to 1D Array*

## 3.5 CPU Implementation

In the CPU implementation we iterate all images to process them one by one.

In the convolutional layer, we apply filters to each image tile without padding. Consequently, the dimensions of the image after this layer are slightly reduced. The last (filter size - 1) columns and rows will not be processed by the filter. As a result, the dimensions of the convolved image are the original image dimensions minus the filter size plus 1. After determining the dimensions of the convolved image, we iterate through each pixel. We find the corresponding pixel of the original image, compute its dot product with the corresponding index in the filter, and store the result in the convolved image's pixel. We perform this operation nine times for the nine filters we created, generating nine convolved images for each input image. Figure 7 shows an example of this process, the difference is that the images and filters in the figure have 3 channels but ours only have 1 channel.

We implemented two different algorithms for this function. One iterates through each filter in the filter array using a loop as shown in Figure 8, while the other unrolls the loop and hardcodes the nine filters directly as shown in Figure 9. Notably, the hardcoded version is ten times faster, demonstrating the benefits of loop unrolling. We also use this unrolled version in our benchmarks to compare with the GPU implementation.

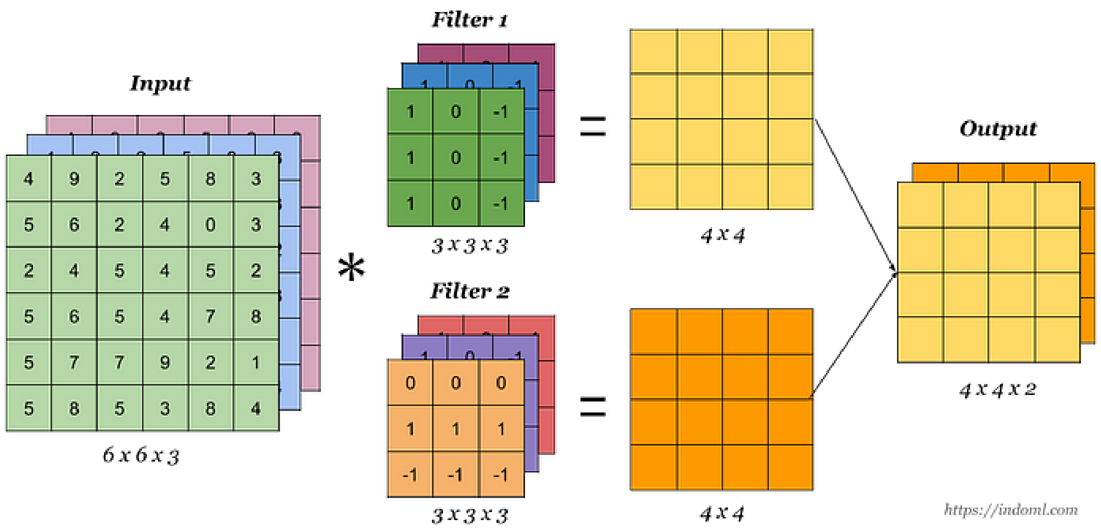
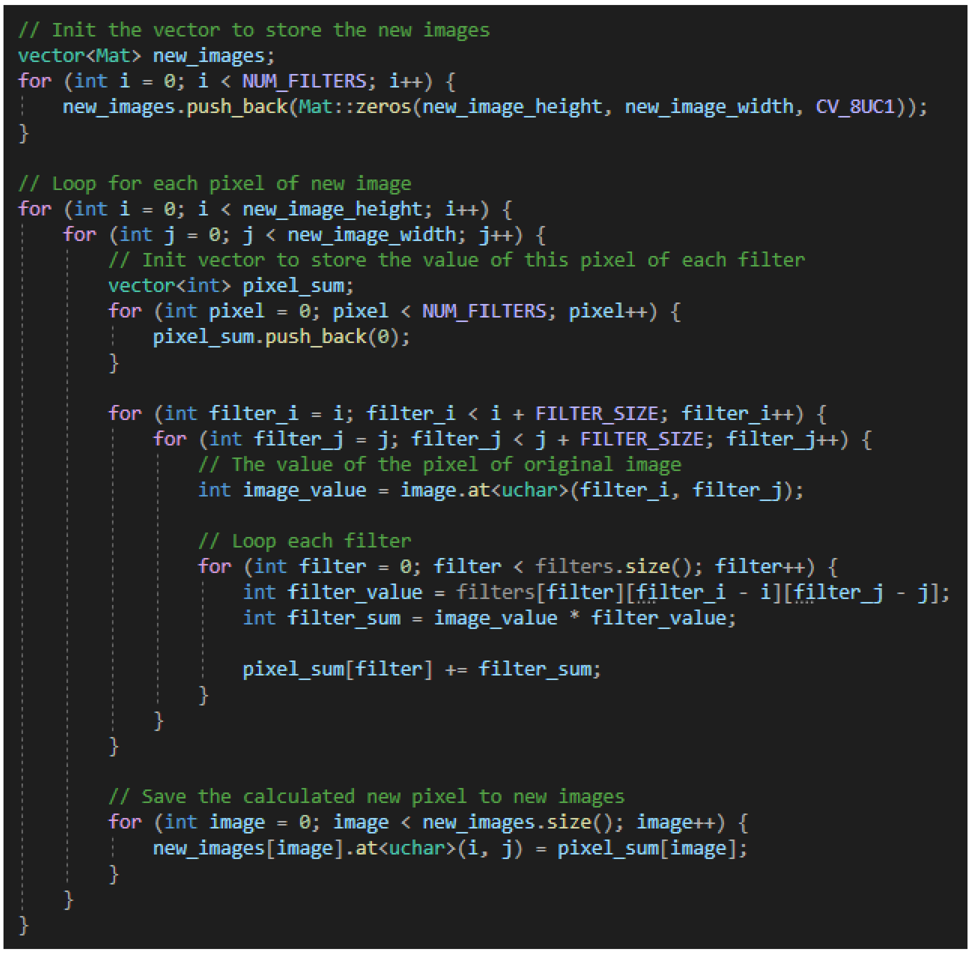
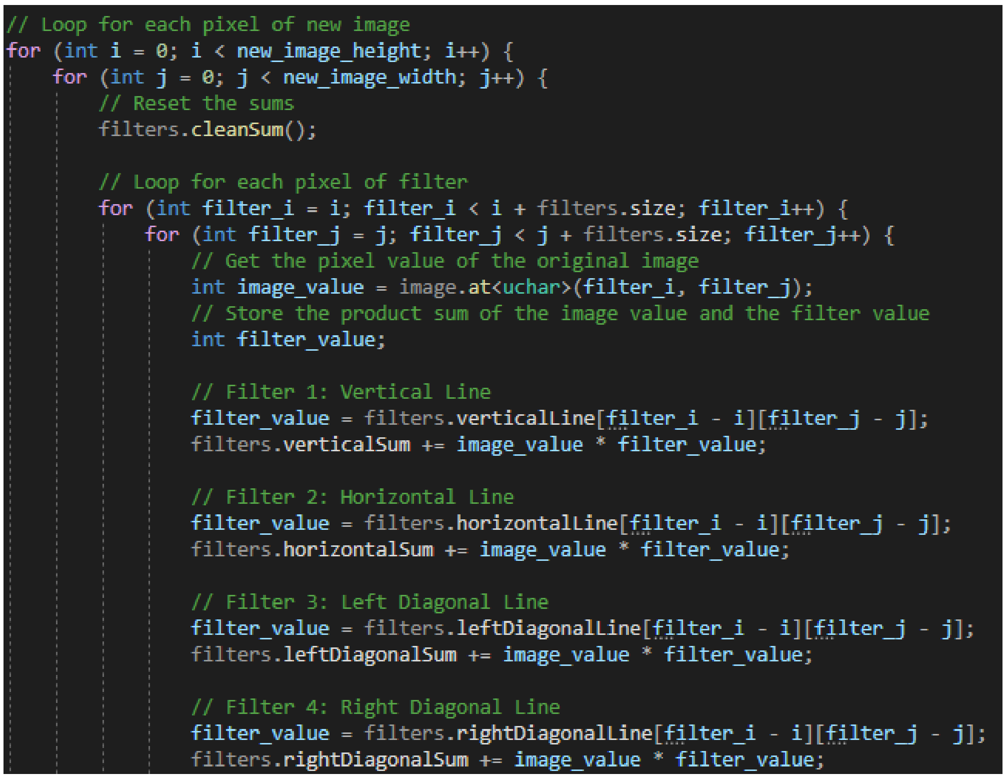


Figure 7 An Example of Convolutional Layer Process [7]



```
// Init the vector to store the new images
vector<Mat> new_images;
for (int i = 0; i < NUM_FILTERS; i++) {
    new_images.push_back(Mat::zeros(new_image_height, new_image_width, CV_8UC1));
}

// Loop for each pixel of new image
for (int i = 0; i < new_image_height; i++) {
    for (int j = 0; j < new_image_width; j++) {
        // Init vector to store the value of this pixel of each filter
        vector<int> pixel_sum;
        for (int pixel = 0; pixel < NUM_FILTERS; pixel++) {
            pixel_sum.push_back(0);
        }

        for (int filter_i = i; filter_i < i + FILTER_SIZE; filter_i++) {
            for (int filter_j = j; filter_j < j + FILTER_SIZE; filter_j++) {
                // The value of the pixel of original image
                int image_value = image.at<uchar>(filter_i, filter_j);

                // Loop each filter
                for (int filter = 0; filter < filters.size(); filter++) {
                    int filter_value = filters[filter][filter_i - i][filter_j - j];
                    int filter_sum = image_value * filter_value;

                    pixel_sum[filter] += filter_sum;
                }
            }
        }

        // Save the calculated new pixel to new images
        for (int image = 0; image < new_images.size(); image++) {
            new_images[image].at<uchar>(i, j) = pixel_sum[image];
        }
    }
}
```

*Figure 8 Convolutional Layer CPU Implementation Uses Loops for Filters*

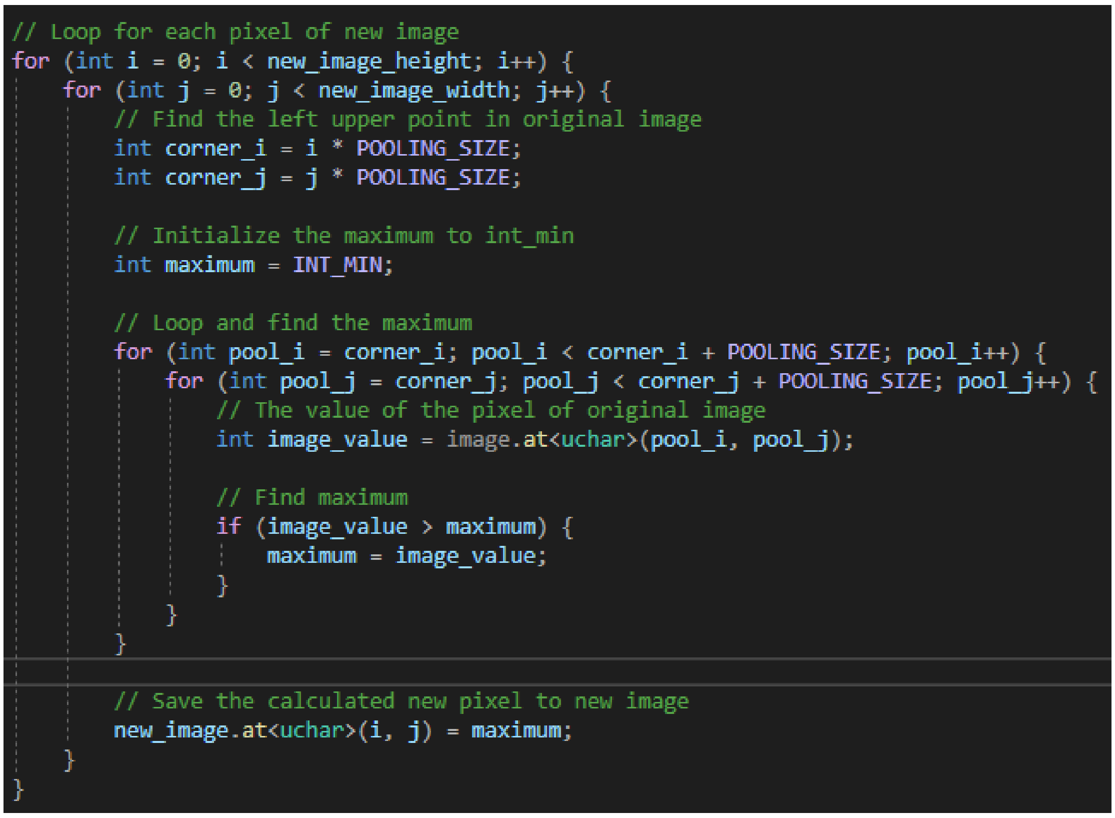


```
// Loop for each pixel of new image
for (int i = 0; i < new_image_height; i++) {
    for (int j = 0; j < new_image_width; j++) {
        // Reset the sums
        filters.cleanSum();

        // Loop for each pixel of filter
        for (int filter_i = i; filter_i < i + filters.size; filter_i++) {
            for (int filter_j = j; filter_j < j + filters.size; filter_j++) {
                // Get the pixel value of the original image
                int image_value = image.at<uchar>(filter_i, filter_j);
                // Store the product sum of the image value and the filter value
                int filter_value;

                // Filter 1: Vertical Line
                filter_value = filters.verticalLine[filter_i - i][filter_j - j];
                filters.verticalSum += image_value * filter_value;

                // Filter 2: Horizontal Line
                filter_value = filters.horizontalLine[filter_i - i][filter_j - j];
                filters.horizontalSum += image_value * filter_value;

                // Filter 3: Left Diagonal Line
                filter_value = filters.leftDiagonalLine[filter_i - i][filter_j - j];
                filters.leftDiagonalSum += image_value * filter_value;

                // Filter 4: Right Diagonal Line
                filter_value = filters.rightDiagonalLine[filter_i - i][filter_j - j];
                filters.rightDiagonalSum += image_value * filter_value;
```

*Figure 9 Convolutional Layer CPU Implementation Uses Loops Unrolling*

Following the convolutional layer, the output is passed to the pooling layer. Similar to the convolutional layer, the pooling layer needs to determine the dimensions of the pooled image. For instance, if the filter size is 3x3, the pooled image dimensions will be reduced by a factor of 3. We do not apply padding and simply discard the boundary pixels in the original image that cannot be processed. Subsequently, we stride the filter across the image, find the maximum value in the filter region, and store it in the corresponding pixel in the pooled image as shown in Figure 10.

Additionally, we store the results of the images output by each layer. By doing so, we can utilize OpenCV to visualize the results and compare them with the GPU implementation's output to validate correctness.



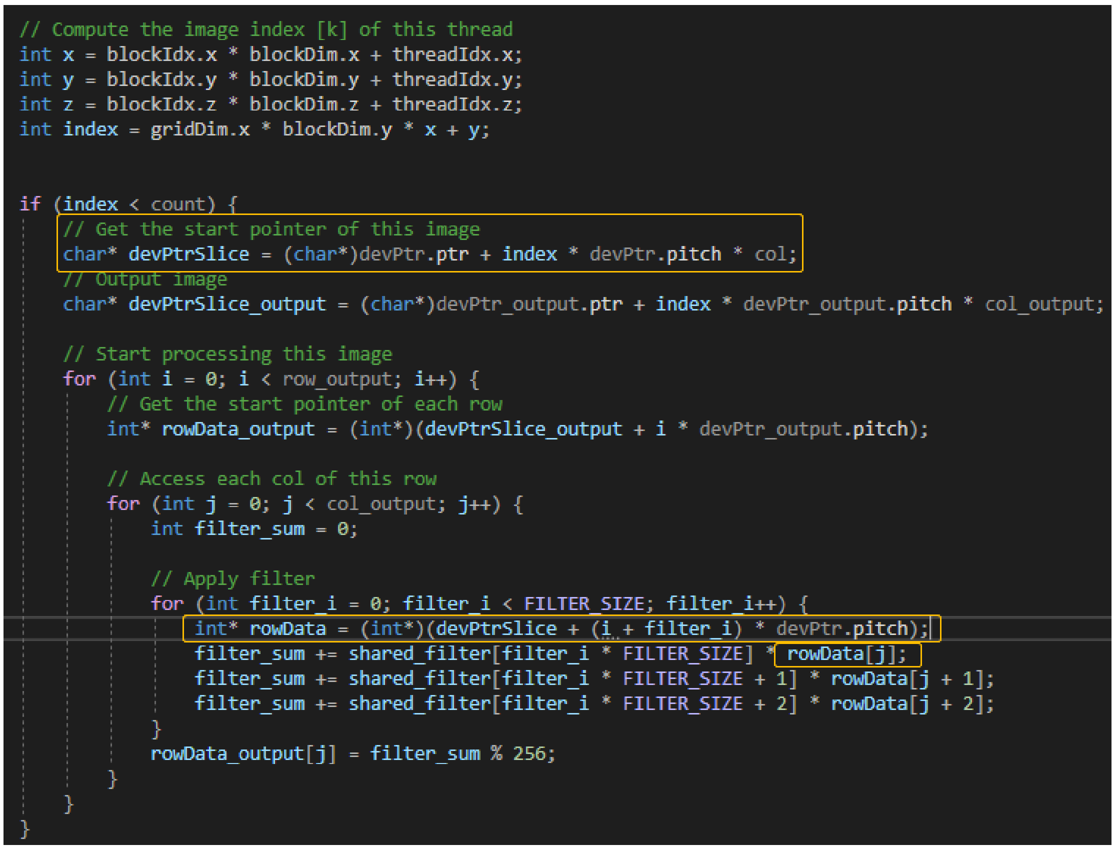
```
// Loop for each pixel of new image
for (int i = 0; i < new_image_height; i++) {
    for (int j = 0; j < new_image_width; j++) {
        // Find the left upper point in original image
        int corner_i = i * POOLING_SIZE;
        int corner_j = j * POOLING_SIZE;

        // Initialize the maximum to int_min
        int maximum = INT_MIN;

        // Loop and find the maximum
        for (int pool_i = corner_i; pool_i < corner_i + POOLING_SIZE; pool_i++) {
            for (int pool_j = corner_j; pool_j < corner_j + POOLING_SIZE; pool_j++) {
                // The value of the pixel of original image
                int image_value = image.at<uchar>(pool_i, pool_j);

                // Find maximum
                if (image_value > maximum) {
                    maximum = image_value;
                }
            }
        }

        // Save the calculated new pixel to new image
        new_image.at<uchar>(i, j) = maximum;
    }
}
```

*Figure 10 Polling Layer CPU Implementation*

## 3.6 GPU Implementation

In the GPU implementation, the core concept for each layer's implementation remains the same. However, instead of processing input images sequentially, we leverage the parallelism of the GPU by assigning one thread to process each image. If the input size exceeds the available threads, some threads will be responsible for processing multiple images. In our case, since we only have tens of thousands of input images and there are many more available threads on the device, we can guarantee that all input images will be processed concurrently. This is expected to result in a significant performance increase compared to the CPU implementation.

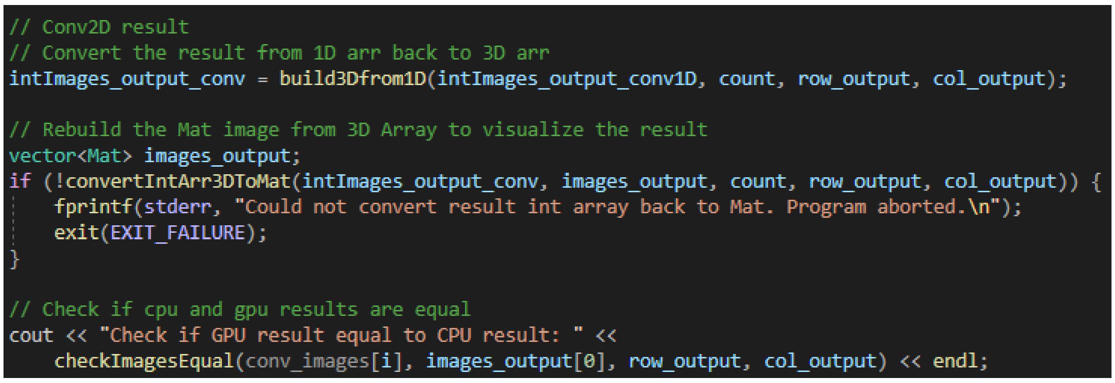
However, as previously mentioned, it is challenging to allocate the 3D space on the device, copy the 1D-represented 3D array to the device memory, and access this 3D space from the kernel function. We spent considerable time figuring out how to copy the 3D space to the device and access it. Ultimately, we successfully used the `cudaPitchedPtr` object to find the pointer pointing to the start of the 3D space, added byte shifts to locate the pointer pointing to each x index (the index of each image), and added byte shifts again to access each y index (the column index of the image). Once we successfully accessed each z index (the actual pixel of each input image), we performed the same core algorithm for each layer. The byte shift operations are demonstrated in Figure 11.



```
// Compute the image index [k] of this thread
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;
int index = gridDim.x * blockDim.y * x + y;


if (index < count) {
    // Get the start pointer of this image
    char* devPtrSlice = (char*)devPtr.ptr + index * devPtr.pitch * col;
    // Output image
    char* devPtrSlice_output = (char*)devPtr_output.ptr + index * devPtr_output.pitch * col_output;

    // Start processing this image
    for (int i = 0; i < row_output; i++) {
        // Get the start pointer of each row
        int* rowData_output = (int*)(devPtrSlice_output + i * devPtr_output.pitch);

        // Access each col of this row
        for (int j = 0; j < col_output; j++) {
            int filter_sum = 0;

            // Apply filter
            for (int filter_i = 0; filter_i < FILTER_SIZE; filter_i++) {
                int* rowData = (int*)(devPtrSlice + (i + filter_i) * devPtr.pitch);
                filter_sum += shared_filter[filter_i * FILTER_SIZE] * rowData[j];
                filter_sum += shared_filter[filter_i * FILTER_SIZE + 1] * rowData[j + 1];
                filter_sum += shared_filter[filter_i * FILTER_SIZE + 2] * rowData[j + 2];
            }
            rowData_output[j] = filter_sum % 256;
        }
    }
}
```

*Figure 11 Use cudaPitchedPtr Object to Access 3D Space in CUDA Kernel*

Upon completing the two layers, we transfer the results from the device back to the host, reconstruct the 3D array from the 1D-represented 3D array, and convert the 3D array into an `OpenCV Mat` Object. As shown in Figure 12, this enables us to compare the results with the stored outputs of the CPU implementations mentioned earlier.



```
// Conv2D result
// Convert the result from 1D arr back to 3D arr
intImages_output_conv = build3Dfrom1D(intImages_output_conv1D, count, row_output, col_output);

// Rebuild the Mat image from 3D Array to visualize the result
vector<Mat> images_output;
if (!convertIntArr3DToMat(intImages_output_conv, images_output, count, row_output, col_output)) {
    fprintf(stderr, "Could not convert result int array back to Mat. Program aborted.\n");
    exit(EXIT_FAILURE);
}

// Check if cpu and gpu results are equal
cout << "Check if GPU result equal to CPU result: " <<
    checkImagesEqual(conv_images[i], images_output[0], row_output, col_output) << endl;
```

*Figure 12 Reconstruct Mat Object From 1D Array and Compare with CPU Implementation Output*

We further optimized the kernel functions in our study. In the convolutional layer, each thread processes one image, and all threads within the same block require the entire filter data. To address this, we utilized shared memory to store the filter, which reduces the time needed to access data from the global memory. Moreover, we employed two loops to iterate through the rows and columns of the filter, and we further unrolled these loops by hardcoding access to the nine filter indices. Additionally, we used register memory to store the partial sums of the dot products and only set the overall sum as the result pixel at the end, instead of immediately adding the partial sum to the result pixel. In the pooling layer, we applied similar strategies, using register memory and unrolling loops for enhanced performance.

## 3.7 Experiment Setup

We initially read the input images from disk and then sequentially executed the CPU and GPU implementations of the convolutional and pooling layers. For the CPU implementations, both the convolutional and pooling layers accept and output OpenCV Mat objects, allowing us to pass the input images directly from the convolutional layer to the pooling layer.

In theory, for the GPU implementations, since the output of the convolutional layer serves as the input for the pooling layer, we would first allocate memory space for the input and output of the convolutional layer, execute its kernel function, and then only allocate space for the output of the pooling layer, as its input is already in the device memory. This approach could save time for copying data and allocating memory space. However, due to time constraints, we implemented the kernel functions for these two layers separately and could not combine them. Consequently, the kernel functions are run sequentially, requiring us to copy the output of the convolutional layer from the device to the host, allocate space for both the input and output of the pooling layer, and then copy them into the device memory.

For the above setup, we conducted benchmarks for the two layers in both implementations based on different input image numbers. Additionally, we tested the performance of the kernel functions against their optimized versions on different input image numbers and used Nsight Compute for profiling. The first experiment is illustrated in Figure 13.

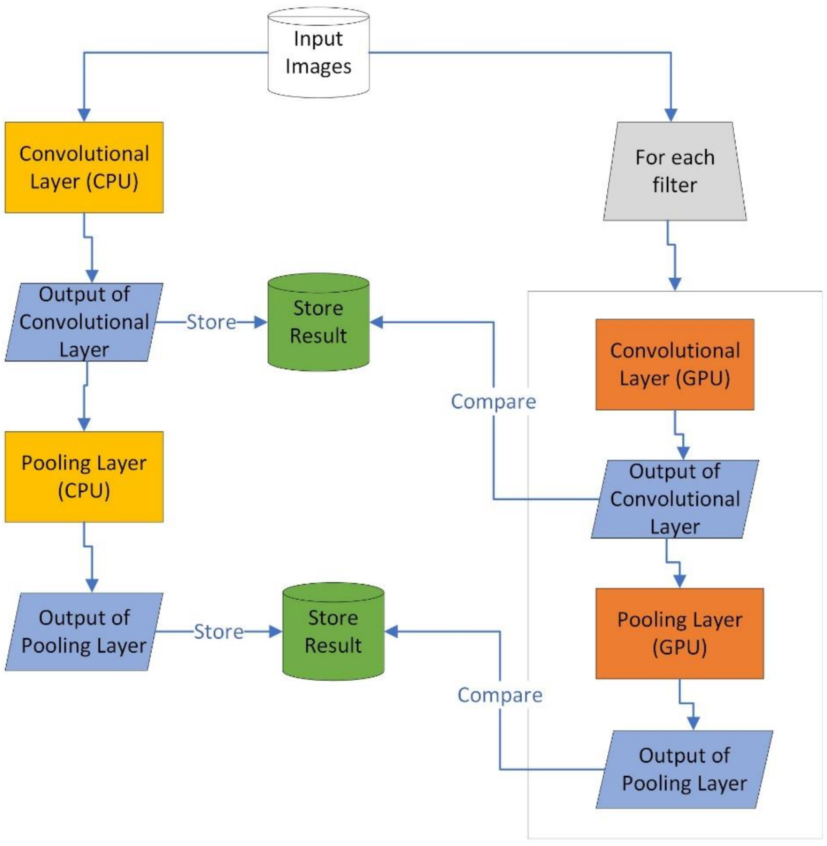


*Figure 13 Experiment Setup*

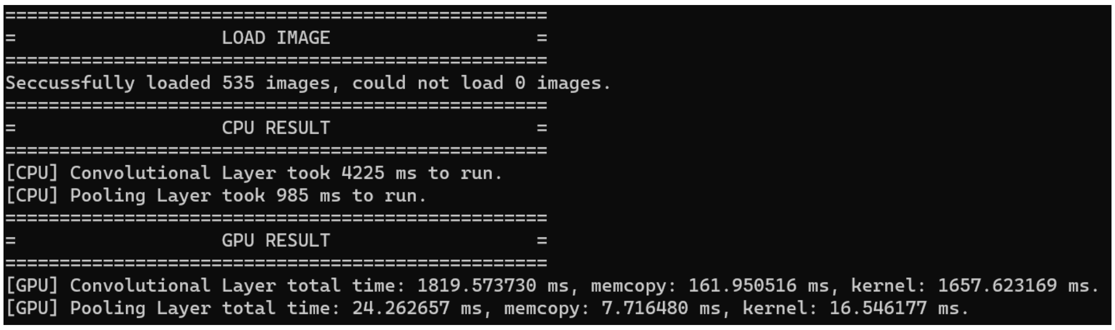
## 4. Experiments and Results

### 4.1 Hardware Specification

| | |
|---|---|
| CPU | Intel® Core™ i7-7800X |
| GPU | GeForce RTX 2070 Super |
| Graphic Processor | TU104 |
| Cores | 3560 |
| Memory Size | 8 GB |
| Memory Type | GDDR6 |
| Bus Width | 256 bits |
| Maximum SMs | 40 |
| Maximum Resident Block/SM | 16 |
| Maximum Thread/Block | 1024 |
| Compute Capability | 7.5 |
| CUDA Version | Release 12, V12.0.76 |

### 4.2 Execution Snapshots

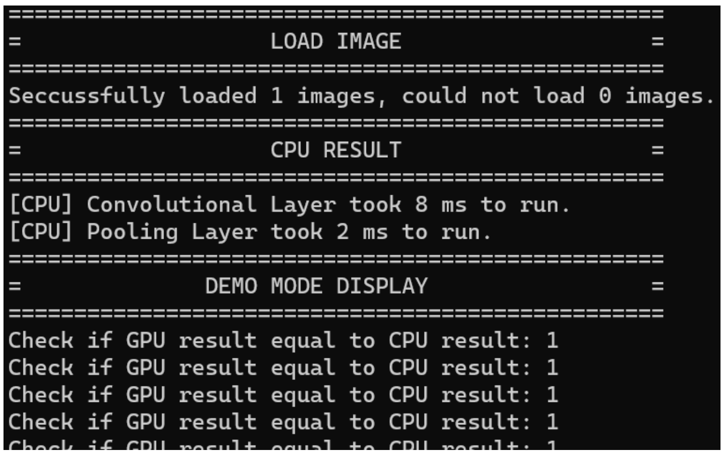
### 4.2.1 Program output



```
==================================================
=                   LOAD IMAGE                   =
==================================================
Seccussfully loaded 535 images, could not load 0 images.
==================================================
=                   CPU RESULT                   =
==================================================
[CPU] Convolutional Layer took 4225 ms to run.
[CPU] Pooling Layer took 985 ms to run.
==================================================
=                   GPU RESULT                   =
==================================================
[GPU] Convolutional Layer total time: 1819.573730 ms, memcopy: 161.950516 ms, kernel: 1657.623169 ms.
[GPU] Pooling Layer total time: 24.262657 ms, memcopy: 7.716480 ms, kernel: 16.546177 ms.
```

*Figure 14 Program Output Snapshot*

### 4.2.2 Demo Output



```
==================================================
=                   LOAD IMAGE                   =
==================================================
Seccussfully loaded 1 images, could not load 0 images.
==================================================
=                   CPU RESULT                   =
==================================================
[CPU] Convolutional Layer took 8 ms to run.
[CPU] Pooling Layer took 2 ms to run.
==================================================
=               DEMO MODE DISPLAY                =
==================================================
Check if GPU result equal to CPU result: 1
Check if GPU result equal to CPU result: 1
Check if GPU result equal to CPU result: 1
Check if GPU result equal to CPU result: 1
Check if GPU result equal to CPU result: 1
```

*Figure 15 Demo Mode Output Snapshot*

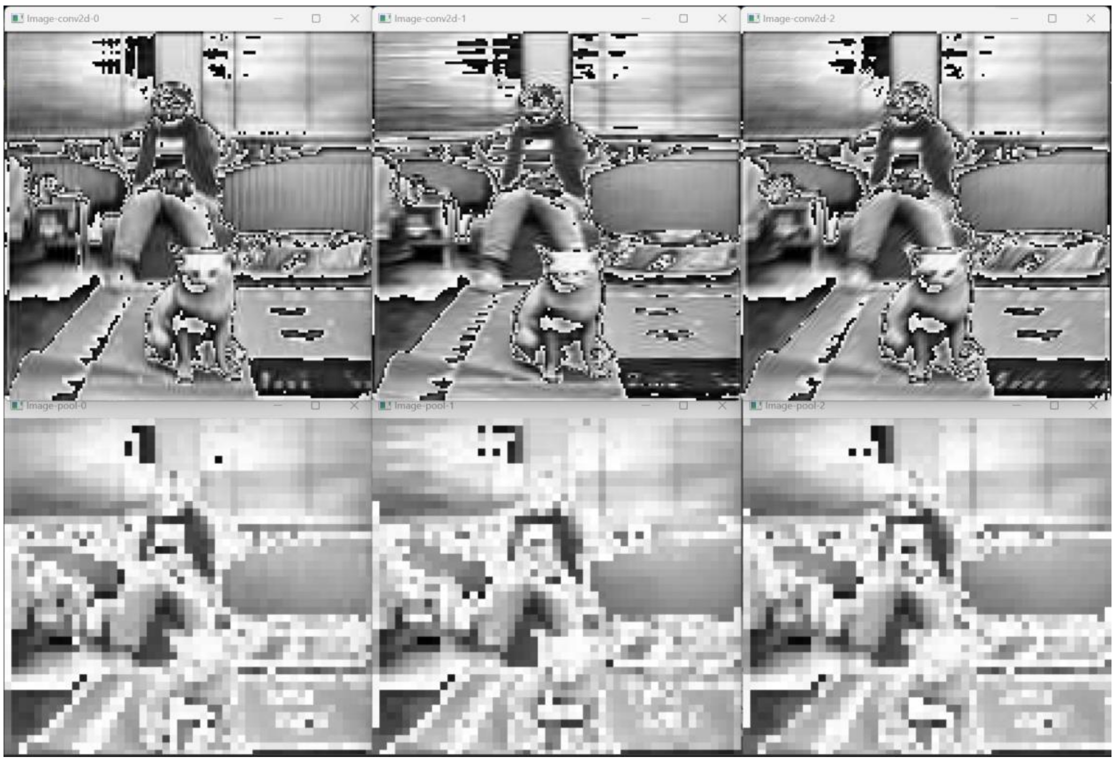


*Figure 16 Demo Mode Displaying Convolved and Pooled Images*

Following the convolutional layer, we observed that the resulting images from different filters detect distinct patterns. After applying max pooling, the image dimensions are reduced, but the patterns remain recognizable. In our demonstration, we compared the GPU results with the CPU results in a single run to ensure the correctness of our implementation.

### 4.3 Experiments on CPU Implementations

#### 4.3.1 Convolutional Layer

| Number of Images | Execution Time(ms) |
|---|---|
| 1 | 32 |
| 500 | 10506 |
| 1000 | 21244 |
| 2500 | 50659 |
| 5000 | 102755 |
| 10000 | 203823 |
| 15060 | 302536 |

*Table 1 Convolutional Layer CPU Implementation Benchmark*

It is evident that as the number of images increases from 500 to 15,060, the execution time of the program exhibits a linear growth pattern with respect to the number of images.

#### 4.3.2 Pooling Layer

| Number of Images | Execution Time(ms) |
|---|---|
| 1 | 3 |
| 500 | 1510 |
| 1000 | 3064 |
| 2500 | 7807 |
| 5000 | 15075 |
| 10000 | 32549 |
| 15060 | 47558 |

*Table 2 Pooling Layer CPU Implementation Benchmark*

The execution time of the pooling layer is considerably shorter than that of the convolutional layer, owing to its lower computational complexity. Additionally, as the number of images increases from 500 to 15,060, the execution time of the program exhibits a linear growth pattern with respect to the number of images.

### 4.4 Experiments on GPU Implementations

#### 4.4.1 Convolutional Layer

| Number of Images | MemCopy Time(ms) | Kernel Time(ms) | Total(ms) |
|---|---|---|---|
| 1 | 2 | 1338 | 1340 |
| 500 | 399 | 1690 | 2090 |
| 1000 | 827 | 1792 | 2619 |
| 2500 | 2151 | 1845 | 3996 |
| 5000 | 4031 | 2056 | 6088 |
| 10000 | 8519 | 4141 | 12660 |

*Table 3 Convolutional Layer GPU Kernel Benchmark*

| Number of Images | OPT MemCopy Time(ms) | OPT Kernel Time(ms) | OPT Total(ms) |
|---|---|---|---|
| 1 | 2 | 1015 | 1017 |
| 500 | 402 | 1321 | 1724 |
| 1000 | 846 | 1624 | 2470 |
| 2500 | 2141 | 1732 | 3874 |
| 5000 | 3959 | 1930 | 5889 |
| 10000 | 8611 | 3823 | 12434 |

*Table 4 Convolutional Layer GPU Kernel Optimized Benchmark*

| | Convolution | OPT Convolution |
|---|---|---|
| **Compute Throughput** | 13.92% | 12.65% |
| **Memory Throughput** | 27.65% | 28.34% |
| **Register/Thread** | 35 | 34 |
| **Global Memory Usage** | 193.38M | 101.78M |
| **Shared Memory Usage** | 0M | 91.6M |
| **L1 Hit Rate** | 14.41% | 12.48% |
| **Occupancy** | 63.8% | 64.9% |
| **Active Warps Per SM** | 30.6 | 31.1 |

*Table 5 Convolutional Layer GPU Different Kernel Profile*

As we only optimized the kernel, it is expected that there would be no difference in the memory copy time between the naive version and the optimized (OPT) version. However, it is worth noting that there is an improvement in kernel execution time in the OPT version, which can be explained by examining the profiling results. After implementing shared memory and loop unrolling techniques, we observe an increase in occupancy and active warps per SM, leading to the enhanced performance observed in the OPT version.

### 4.4.2 Pooling Layer

| Number of Images | MemCopy Time(ms) | Kernel Time(ms) | Total(ms) |
|---|---|---|---|
| 1 | 0.117568 | 23.218912 | 23.336479 |
| 500 | 6.85008 | 30.01856 | 36.868641 |
| 1000 | 12.947487 | 30.086945 | 43.034431 |
| 2500 | 37.310432 | 31.798689 | 69.109123 |
| 5000 | 61.551331 | 33.929089 | 95.480423 |
| 10000 | 132.997223 | 47.117344 | 180.114563 |

*Table 6 Pooling Layer GPU Kernel Benchmark*

| Number of Images | OPT MemCopy Time(ms) | OPT Kernel Time(ms) | OPT Total(ms) |
|---|---|---|---|
| 1 | 0.125376 | 16.557089 | 16.682465 |
| 500 | 6.454624 | 21.556959 | 28.011583 |
| 1000 | 13.633504 | 26.884159 | 40.517662 |
| 2500 | 38.337345 | 27.04096 | 65.378304 |
| 5000 | 62.524323 | 29.518784 | 92.043106 |
| 10000 | 124.763321 | 45.611874 | 170.375198 |

*Table 7 Pooling Layer GPU Kernel Optimized Benchmark*

| | Convolution | OPT Convolution |
|---|---|---|
| **Compute Throughput** | 12.68% | 8.83% |
| **Memory Throughput** | 26.66% | 26.42% |
| **Register/Thread** | 30 | 37 |
| **Global Memory Usage** | 11.3M | 11.3M |
| **Shared Memory Usage** | 0M | 0M |
| **L1 Hit Rate** | 12.47% | 13.77% |
| **Occupancy** | 63.5% | 63.5% |
| **Active Warps Per SM** | 30.5 | 30.5 |

*Table 8 Pooling Layer GPU Different Kernel Profile*

Given that we only optimized the kernel, it is natural that there would be no difference in memory copy time between the naive version and the optimized (OPT) version. However, it is important to note that there is an improvement in kernel execution time in the OPT version, which can be attributed to the profiling results. We observe that after implementing loop unrolling, the number of registers per thread and the L1 hit rate increased, contributing to the enhanced performance seen in the OPT version.

## 4.5 Analysis and Discussion

### 4.5.1 Performance of Convolutional Layer in Each Implementation

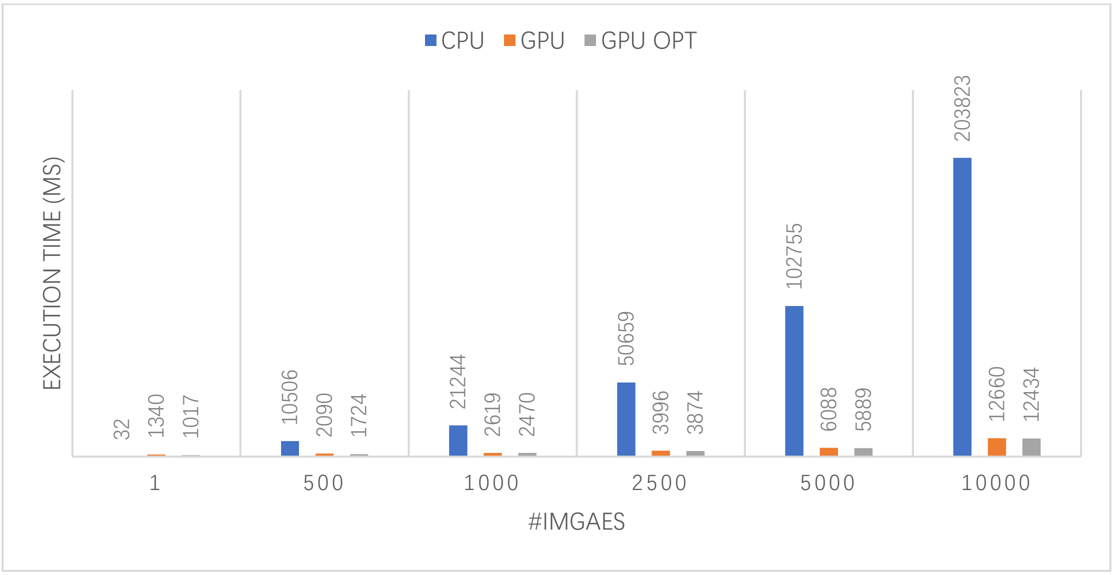


*Figure 17 Convolutional Layer Execution Time in Different Implementations*

| Number of Images | GPU to CPU Improvement Rate |
|---|---|
| 500 | 6.1X |
| 1000 | 8.11X |
| 2500 | 13.08X |
| 5000 | 17.45X |
| 10000 | 16.39X |

*Figure 18 Convolutional Layer Performance Improvement Rate GPU To CPU*

The chart above illustrates the relationship between the total execution time for the convolutional layer and the number of images. As the number of images increases, the

GPU demonstrates better performance than the CPU, with the GPU's execution time exhibiting a more linear increase. This showcases the GPU's superior capabilities in handling larger numbers of images. From the table, it is evident that the improvement rate is approximately 17 times greater when the number of images is large. This further highlights the advantages of utilizing GPU for processing extensive image sets.

### 4.5.2 Convolutional Layer Kernel Optimization

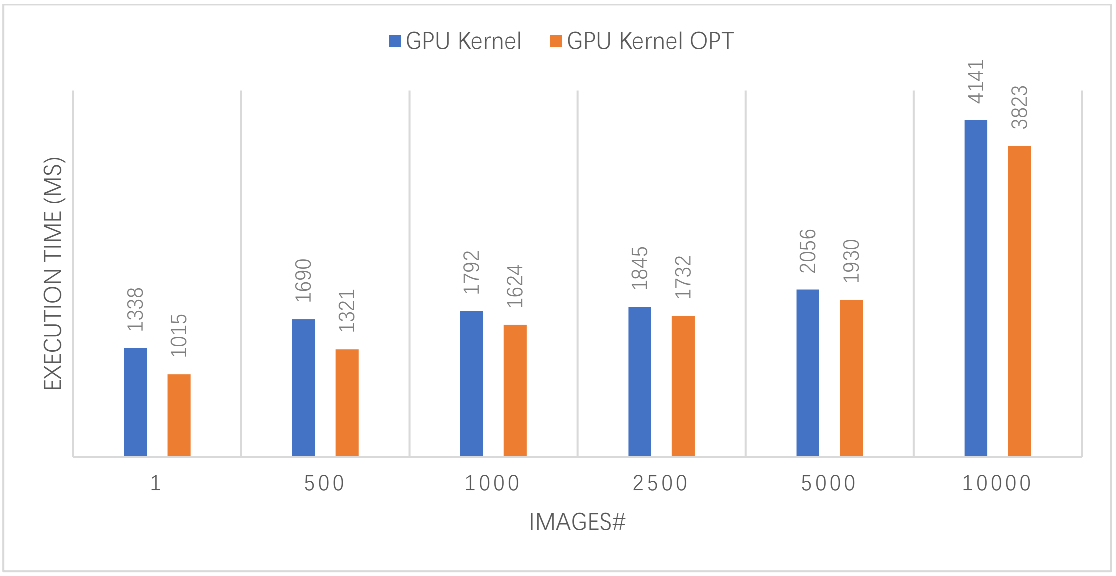


*Figure 19 Convolutional Layer Execution Time in Different Kernel Implementations*

| #Images | OPT to Naïve Improvement Rate |
|---|---|
| 500 | 21.83% |
| 1000 | 9.38% |
| 2500 | 6.12% |
| 5000 | 6.13% |
| 10000 | 7.68% |

*Figure 20 Convolutional Layer Performance Improvement Rate Kernel To Kernel OPT*

Following the optimization of the kernel function using shared memory, register memory, and loop unrolling, we observed a substantial enhancement in performance concerning the kernel execution time. These optimizations contributed to a more efficient and rapid execution of the convolutional layer on the GPU. As the number of images increases, the improvement rate is approximately 7%, indicating that the optimization strategies have a positive impact on the performance for large-scale image processing.

### 4.5.3 Performance of Pooling Layer in Each Implementation

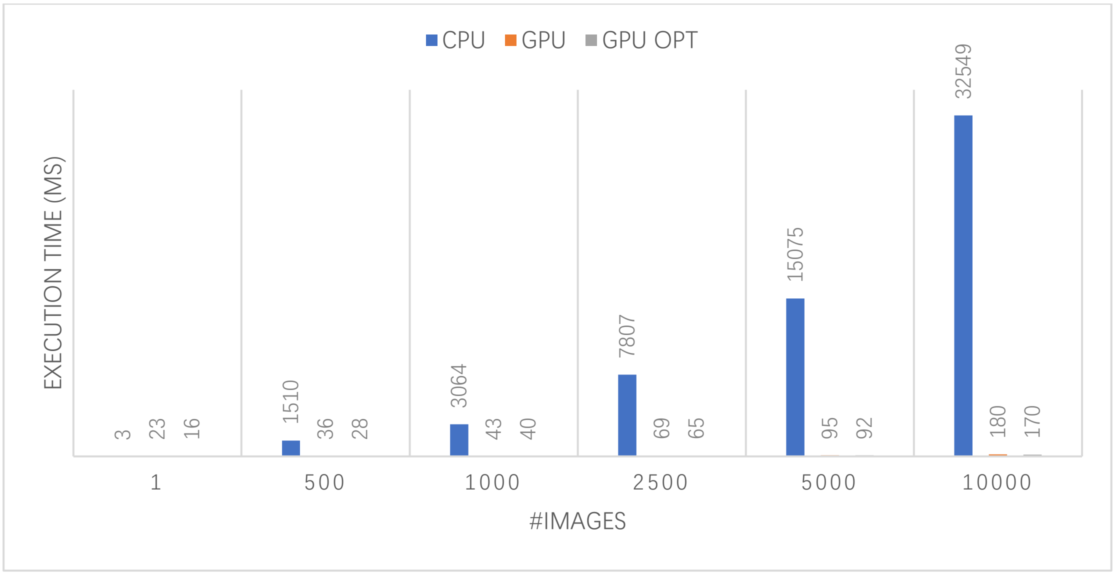


*Figure 21 Pooling Layer Execution Time in Different Implementations*

| **#Images** | **GPU to CPU Improvement Rate** |
|---|---|
| 500 | 53.9X |
| 1000 | 76.6X |
| 2500 | 120.7X |
| 5000 | 163.9X |
| 10000 | 191.5X |

*Figure 22 Pooling Layer Performance Improvement Rate GPU To CPU*

The chart above illustrates the relationship between the total execution time of the pooling layer and the number of images. As the number of images increases, it becomes evident that the GPU outperforms the CPU. From the table, it can be deduced that the improvement rate reaches approximately 190X when dealing with a large number of images, highlighting the GPU's superior performance in handling extensive image datasets.

### 4.5.4 Pooling Layer Kernel Optimization

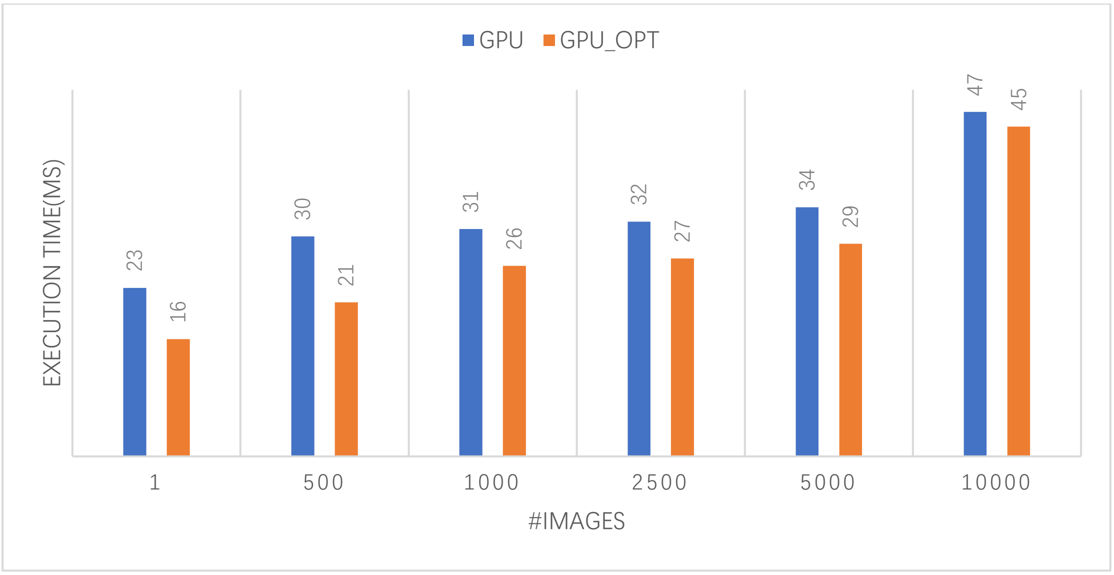


*Figure 23 Pooling Layer Execution Time in Different Kernel Implementations*

| #Images | OPT to Naïve Improvement Rate |
|---|---|
| 500 | 28.2% |
| 1000 | 10.6% |
| 2500 | 15% |
| 5000 | 13% |
| 10000 | 3.2% |

*Figure 24 Pooling Layer Performance Improvement Rate Kernel To Kernel OPT*

Focusing solely on the kernel execution time, the benefits of using loop unrolling become evident. It is clear that the performance of the optimized code surpasses that of the non-optimized version. When the number of images increases, the improvement rate reaches approximately 3%, showcasing the advantages of loop unrolling in enhancing performance.

## 5. Conclusions

### 5.1 Limitations

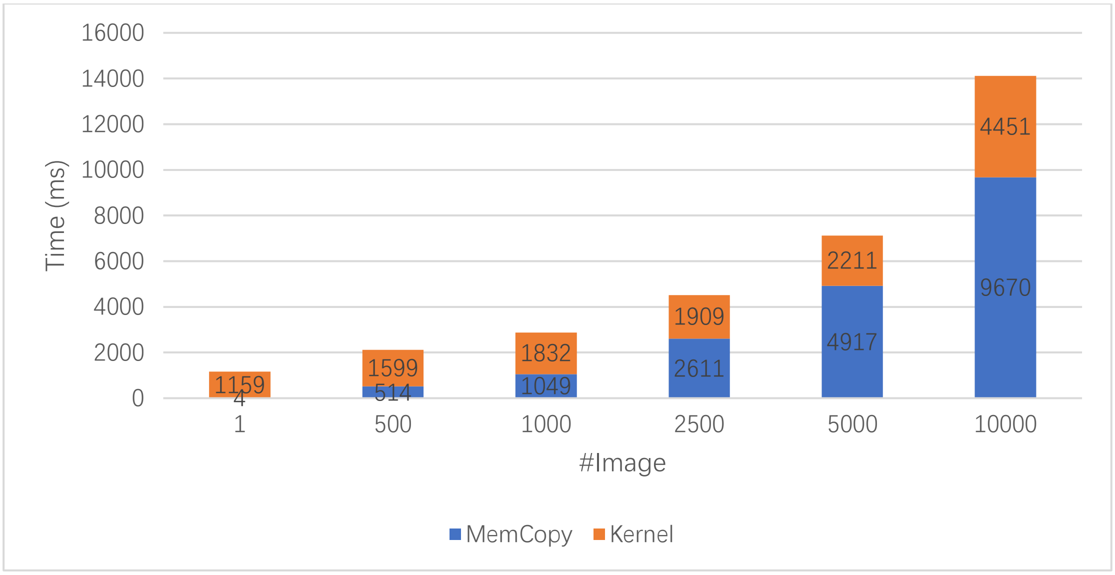


*Figure 25 Limitations of GPU Implementations*

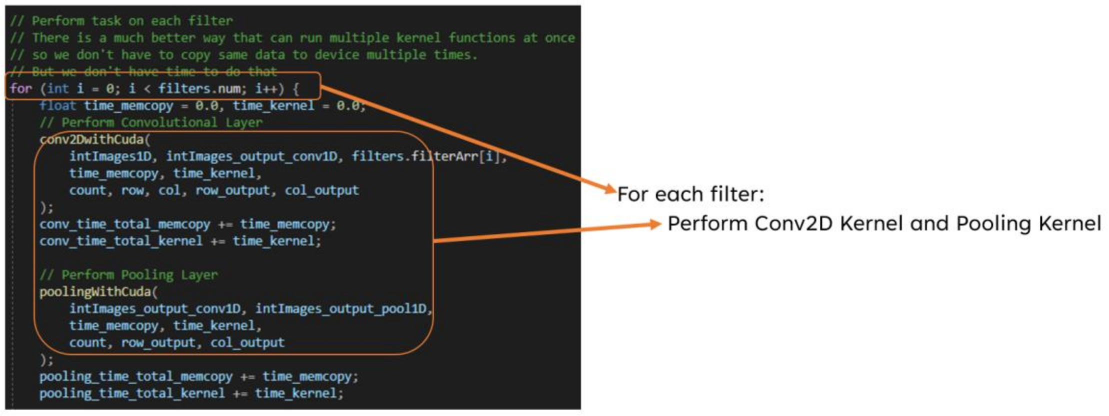
The graph above illustrates a comparison between memory copy time and kernel time, revealing that data transfer overhead has become a bottleneck for the GPU program. The time required for transferring data between the host and the device increases at a much faster pace compared to the actual execution time, impacting overall performance. The root of this issue can be traced back to our design, and this problem will be further elaborated upon in the first subsection of the future work section.

Another limitation pertains to thread usage in the kernel function. As noted in our design, we currently utilize one thread to process one image on the GPU. However, for large image sets, this configuration may not yield optimal results. This issue will be discussed in greater detail in the second subsection of the future work section.

Lastly, our experiments focus solely on comparing the performance between our CPU and GPU implementations. It would be beneficial to include comparisons with existing libraries to gain a more comprehensive understanding of the performance differences.

## 5.2 Future Work

### 5.2.1 Optimize Data Transfer



```
// Perform task on each filter
// There is a much better way that can run multiple kernel functions at once
// so we don't have to copy same data to device multiple times.
// But we don't have time to do that
for (int i = 0; i < filters.num; i++) {
    float time_memcopy = 0.0, time_kernel = 0.0;
    // Perform Convolutional Layer
    conv2DwithCuda(
        intImages1D, intImages_output_conv1D, filters.filterArr[i],
        time_memcopy, time_kernel,
        count, row, col, row_output, col_output
    );
    conv_time_total_memcopy += time_memcopy;
    conv_time_total_kernel += time_kernel;

    // Perform Pooling Layer
    poolingWithCuda(
        intImages_output_conv1D, intImages_output_pool1D,
        time_memcopy, time_kernel,
        count, row_output, col_output
    );
    pooling_time_total_memcopy += time_memcopy;
    pooling_time_total_kernel += time_kernel;
```

*Figure 26 Source of Data Transfer Bottleneck*

From the above code snippet, we can see that in our current implementation, we ran two layers for each filter, which means we had to transfer data to the device every time we used a different filter. This happened because we only realized that we had multiple filters after implementing the CUDA function, and we didn't have time to modify our program. If we could pass all filters and images to the device at once and run multiple kernel functions simultaneously, we believe the data transfer time could be significantly reduced, improving overall performance.

### 5.2.2 Optimize Thread Usage

At present, we employ one thread to process one image on the GPU. However, upon reviewing our device specifications, we discovered that we have a sufficient number of threads to allocate multiple threads to work on each image. As a result, it is feasible to adopt a tiling approach to further decrease kernel execution time. Nonetheless, handling 3D arrays within the kernel poses a significant challenge, and this approach would further complicate the task of determining the indices of the matrix that each thread is responsible for. Consequently, we were unable to implement this algorithm within our current timeframe.

### 5.2.3 Find Libraries for Performance Comparison

We initially intended to utilize other libraries, such as TensorFlow, to implement the same layers and compare the results. However, configuring TensorFlow and using it in C++ proved to be quite challenging, and we were unable to complete this aspect within our current timeframe. Future work should focus on successfully configuring TensorFlow C++ or identifying alternative existing libraries capable of computing convolutional and pooling layers for comparison purposes.

## 5.3 Conclusion

Through various experiments and comparisons, we have demonstrated that utilizing GPU in CNNs can significantly enhance performance. Moreover, by employing techniques such as using register and shared memory, and loop unrolling, we were able to further optimize the kernel. In this project, we built the CNN from scratch without relying on any libraries. While we are satisfied with our research and results, our initial underestimation of the project's complexity led to some issues that arose due to time constraints. As a result, future work is needed to address these challenges and further improve our implementation.

## References

[1] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016.

[2] X. Yu, T. W. Kuan, Y. Zhang and T. Yan, "YOLO v5 for SDSB Distant Tiny Object Detection," in *2022 10th International Conference on Orange Technology (ICOT)*, Shanghai, China, 2022.

[3] IBM, "What are convolutional neural networks?," IBM, [Online]. Available: https://www.ibm.com/topics/convolutional-neural-networks. [Accessed 2023].

[4] S. Pokhrel, "Beginners Guide to Convolutional Neural Networks," Medium, 19 Sep 2019. [Online]. Available: https://towardsdatascience.com/beginners-guide-to-understanding-convolutional-neural-networks-ae9ed58bb17d. [Accessed March 2023].

[5] UNMOVED.FINANCE, "30k Cats and Dogs 150x150 Greyscale," Kaggle, Feb 2023. [Online]. Available: https://www.kaggle.com/datasets/unmoved/30k-cats-and-dogs-150x150-greyscale. [Accessed March 2023].

[6] "Bringing Parallelism to the Web with River Trail," Github, [Online]. Available: http://intellabs.github.io/RiverTrail/tutorial/. [Accessed March 2023].

[7] "Student Notes: Convolutional Neural Networks (CNN) Introduction," indoml, [Online]. Available: https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/. [Accessed March 2023].