Parallelizing Convolutional Neural Networks: A Comparative Study of MPI, OpenMP, and CUDA C Implementations

Mohammad Hijazi Hassan Najjar Ahmad Chawraba

Abstract

This study explores the parallelization of Convolutional Neural Networks (CNNs) using three distinct approaches: Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and CUDA C. We analyze the performance, efficiency, and scalability of each method across various CNN layers, providing insights into their strengths and limitations. Our findings reveal significant speedups, particularly with GPU-based parallelization, while also highlighting challenges in managing communication overhead and optimizing kernel executions.

Contents

1 Introduction

Convolutional Neural Networks (CNNs) are a type of deep learning algorithm primarily designed for analyzing visual data. They are particularly effective at handling grid-structured information, such as images, by using specialized layers to identify and prioritize key features automatically. However, training and deploying CNNs can be computationally demanding, especially with the increasing complexity of models and the rise of high-resolution data. This makes efficient parallelization strategies essential.

One major challenge in training CNNs is their need for large datasets—often consisting of hundreds of thousands of samples—to achieve optimal performance. Consequently, training these networks is highly computationally intensive and can take hours or even days on conventional CPUs. In this project, we aim to leverage the inherent parallelism within CNN layers, exploring various methods to distribute operations across multiple hardware platforms. This approach aims to improve both performance and scalability.

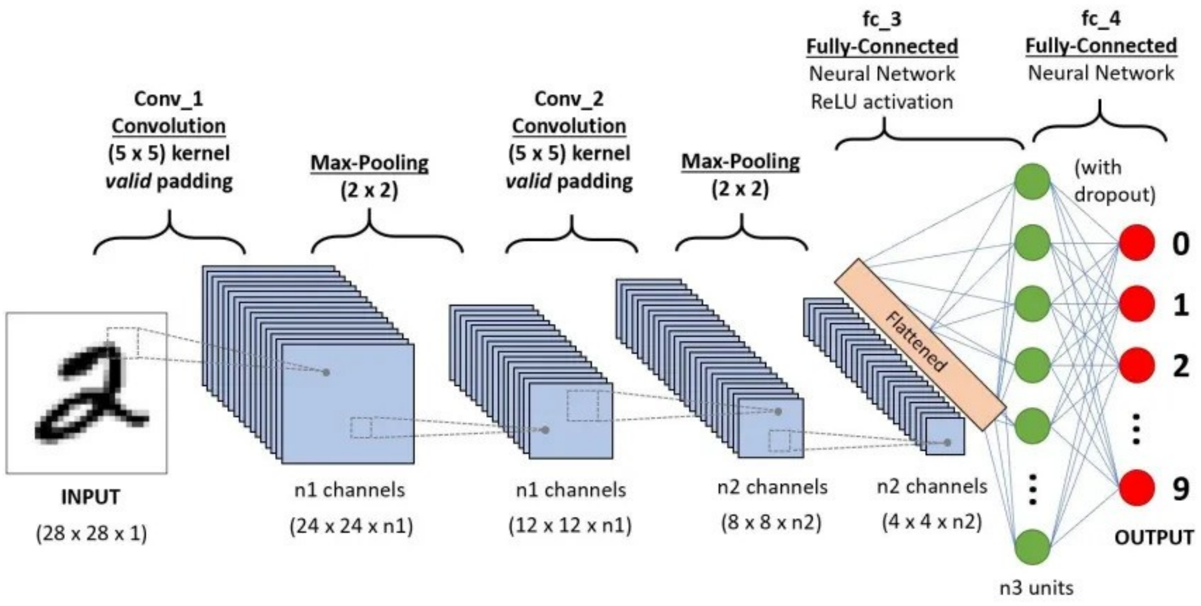


Figure 1: Structure of a Convolutional Neural Network

2 Structure of CNNs

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers:

- Convolutional layer (_c1)
- Pooling layer (_s1)
- Fully-connected layer (_f)

The convolutional layer is the initial component of a convolutional neural network (CNN). Although additional convolutional or pooling layers often follow these, the final stage of the network typically consists of fully connected layers. As data moves through each layer, the network's ability to analyze the image becomes more sophisticated. Early layers detect basic features like edges and colors, while deeper layers progressively identify more complex patterns and larger structures. By the time the data reaches the final layers, the CNN can recognize and classify the complete object in the image.

2.1 Convolutional layer

The convolutional layer is the fundamental component of a CNN. It applies a set of learnable filters to the input image, which are small matrices designed to detect specific features such as edges, colors, or textures. As each filter moves across the image, it computes dot products between the filter values and the input, generating a feature map. This process enables the network to capture spatial hierarchies within the data.

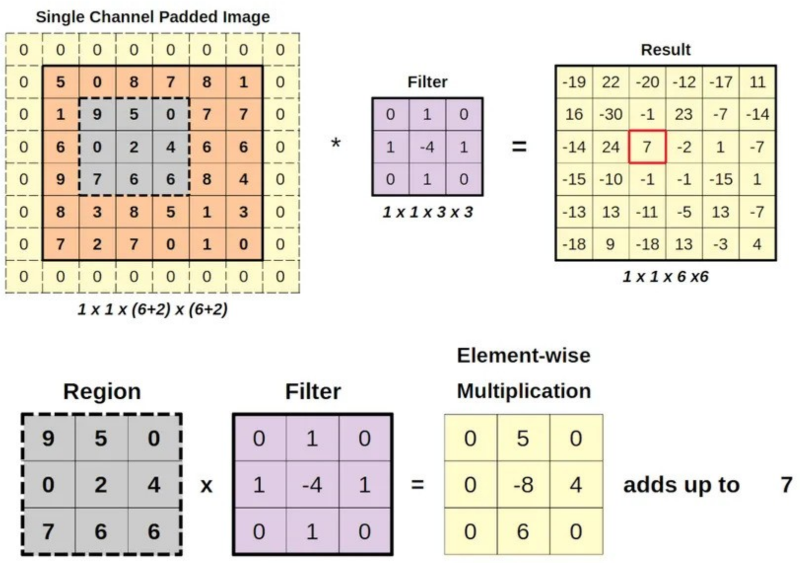


Figure 2: Convolution Operation

2.2 Pooling layer

Pooling layers, or downsampling layers, perform dimensionality reduction by decreasing the number of parameters in the input. Similar to the convolutional layer, the pooling operation involves sweeping a filter across the input, but unlike convolution, this filter does not have any weights. Instead, the kernel uses an aggregation function, such as max or average pooling, to process the values within the receptive field and fill the output array. While pooling layers do result in some information loss, they offer several advantages, including reducing complexity, improving efficiency, and helping to prevent overfitting.

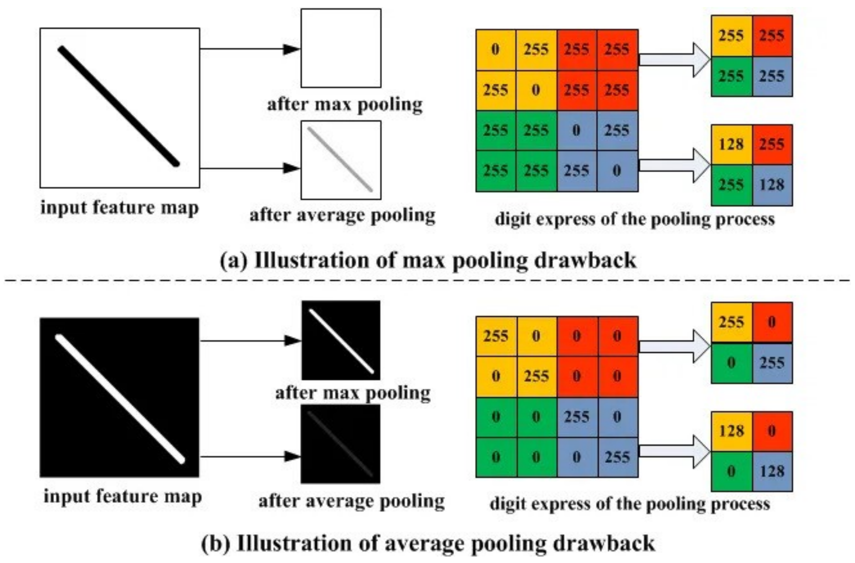

input feature map
after max pooling
after average pooling
digit express of the pooling process
(a) Illustration of max pooling drawback
input feature map
after max pooling
after average pooling
digit express of the pooling process
(b) Illustration of average pooling drawback

Figure 3: Pooling Operation

2.3 Fully-connected (FC) layer

The fully-connected layer consists of neurons that are directly connected to the neurons in the previous and next layers, but not to any neurons within those layers. This structure is similar to how neurons are organized in traditional artificial neural networks (ANNs).

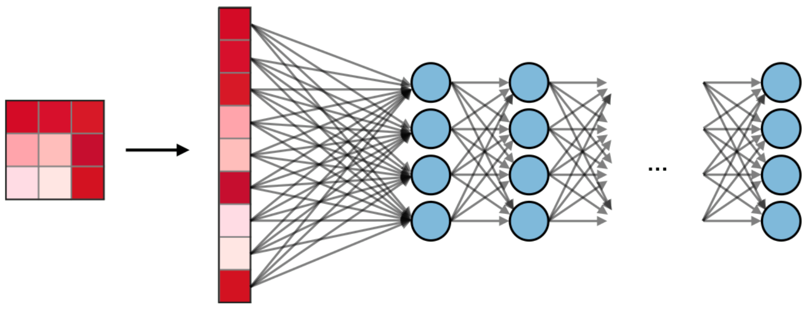


Figure 4: Fully-connected Layer

3 CNN Structure

The CNN architecture includes four main layers: the input layer, which processes the 28x28 pixel images; the convolutional layer, responsible for feature extraction using 6 filters; the pooling layer, which reduces the spatial dimensions of the feature maps; and the fully-connected layer, which converts the extracted features into class predictions for the digits 0-9

4 Parallelization

As per the project guidelines, our project,we used 3 different implementations for parallelization: MPI (Message Passing Interface), OpenMP (Open Multi-Processing), and CUDA C. Each platform is tailored for different hardware and programming needs, allowing us to effectively distribute and manage computational tasks across multiple processing environments. In the following sections, we will explain how we leveraged these three platforms to parallelize the entire process.

4.1 MPI

The parallelization of the convolutional layer in the CNN uses MPI to distribute the computational workload across multiple processes. The input data is divided by rows among the processes, and each process computes the convolution for its assigned rows. Each process performs local computation using a 5x5 filter, storing partial results in a local buffer. Afterward, MPIGather() is used to collect and assemble the results from all processes into the rank 0 process, which holds the full output matrix. This strategy enables parallel processing of the convolution layer, improving computation efficiency across multiple processors.

```
void fp_c1_parallel(const float input[28][28], float preact[6][24][24],
    const float weight[6][5][5], const float bias[6]) {
    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int rows_per_proc = 24 / size;
    int start_row = rank * rows_per_proc;
    int end_row = start_row + rows_per_proc;

    float local_preact[6][rows_per_proc][24] = {0};

    for (int m = 0; m < 6; ++m) {
        for (int x = start_row; x < end_row; ++x) {
            for (int y = 0; y < 24; ++y) {
                float sum = 0.0f;
                for (int i = 0; i < 5; ++i) {
                    for (int j = 0; j < 5; ++j) {
                        sum += input[x + i][y + j] * weight[m][i][j];
                    }
                }
                local_preact[m][x - start_row][y] = sum + bias[m];
            }
        }
    }

    MPI_Gather(local_preact, 6 * rows_per_proc * 24, MPI_FLOAT,
               preact, 6 * rows_per_proc * 24, MPI_FLOAT,
               0, MPI_COMM_WORLD);
}
```

Listing 1: MPI Parallelized Convolutional Function

4.2 OpenMP

The parallelization strategy in the function fpc1 uses OpenMP to speed up the convolution process. The pragma omp parallel for collapse(2) directive is applied to the nested loops iterating over the filter index (m) and the spatial dimensions (x and y). This combines these loops into a single parallelized loop, allowing different sections of the computation to be processed concurrently by multiple threads, which improves performance. This parallelization approach is similar to the one used across different layers of the network, ensuring that each layer's convolutions are handled efficiently in parallel, speeding up the overall execution.

```
void fp_c1(const float input[28][28], float preact[6][24][24], const
    float weight[6][5][5], const float bias[6]) {

     for (int i = 0; i < 6; ++i) {
         for (int j = 0; j < 24; ++j) {
             for (int k = 0; k < 24; ++k) {
                 preact[i][j][k] = 0;
             }
         }
     }

      #pragma omp parallel for collapse(2)
     for (int m = 0; m < 6; ++m) {
         for (int x = 0; x < 24; ++x) {
             for (int y = 0; y < 24; ++y) {
                 float sum = 0.0f;
                 for (int i = 0; i < 5; ++i) {
                     for (int j = 0; j < 5; ++j) {
                         sum += input[x + i][y + j] * weight[m][i][j];
                     }
                 }
                 preact[m][x][y] += sum;
             }
         }
     }


     for (int i = 0; i < 6; ++i) {
         for (int x = 0; x < 24; ++x) {
             for (int y = 0; y < 24; ++y) {
                 preact[i][x][y] += bias[i];
             }
         }
     }
}
```

Listing 2: OpenMP Parallelized Convolutional Function

4.3 CUDA C

4.3.1 Convolutional layer

Forward Propagation ($ForwP\_c1$)

- The kernel is executed using a grid of dim3(6) blocks, with each block representing one output feature map.
- Inside each block, there are dim3(24, 24) threads, each responsible for calculating a single output element in the feature map.
- Each thread computes the dot product between a patch of the input and its corresponding filter weights, then adds the bias term to generate the output element.

```
__global__ void forwP_c1(float input[28][28], float preact[6][24][24],
    float weight[6][5][5], float bias[6]) {
     int m = blockIdx.x;
     int x = threadIdx.x;
     int y = threadIdx.y;

     if (m < 6 && x < 24 && y < 24) {
         float sum = 0.0f;
         for (int i = 0; i < 5; ++i) {
             for (int j = 0; j < 5; ++j) {
                 sum += input[x + i][y + j] * weight[m][i][j];
             }
         }
         preact[m][x][y] = sum + bias[m];
     }
}
```

Listing 3: Forward Propagation Kernel for Convolutional Layer

Activation Function Application (*apply_step_function*)

- **Parallel Execution Strategy:** Similar to *ForwP_c*1
- The kernel is launched with the same setup as $ForwP_c1$, using $configLayer1.blocks$ and $configLayer1.threads$.
- Each thread applies the activation function to its corresponding preactivation value.

```
__global__ void apply_step_function(float (*preact)[24][24], float (*
   output)[24][24], int O)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;

    int x = tx;
    int y = ty;
    int z = bx;

    output[z][y][x] = preact[z][y][x] > 0 ? 1.0f : 0.0f;
}
```

Listing 4: Activation Function Kernel

Output Layer Backpropagation (*backP_output_c*1)

- **Parallel Execution Strategy:** Input Stationarity
- The kernel is launched using a grid of $numBlocks_output_c1$ blocks, where each block processes a segment of the input feature map.
- Each block contains $threadsPerBlock_output_c1$ threads, with each thread responsible for computing the gradient of a specific output element.
- Every thread calculates the gradient by processing its associated input elements and filter weights.
- This approach enhances memory access patterns, ensuring coalesced memory loads and stores for better efficiency.

```
__global__ void backP_output_c1(float d_output[6][24][24], float
    n_weight[1][4][4], float nd_preact[6][6][6]) {
    int c = blockIdx.z;
    int x = blockIdx.y * blockDim.y + threadIdx.y;
    int y = blockIdx.x * blockDim.x + threadIdx.x;

    if (x < 24 && y < 24 && c < 6) {
        float sum = 0.0f;

        for (int i2 = 0; i2 < 4; ++i2) {
            for (int i3 = 0; i3 < 4; ++i3) {
                int preact_x = (x - i2) / 4;
                int preact_y = (y - i3) / 4;

                if (preact_x >= 0 && preact_x < 6 && preact_y >= 0 &&
    preact_y < 6) {
                    sum += n_weight[0][i2][i3] * nd_preact[c][preact_x][
    preact_y];
                }
            }
        }

        d_output[c][x][y] = sum;
    }
}
```

Listing 5: Backpropagation Kernel for Output

Preactivation Backpropagation ($backP\_preact\_c1$)

- The kernel is executed with a grid of $numBlocks_backP_preact_c1$ blocks, where each block is responsible for processing one output feature map.
- Each block contains $threadsPerBlock_backP_preact_c1$ threads, with each thread calculating the gradient of the preactivation value for a specific output element, utilizing the output gradient and the derivative of the activation function.

```
__global__ void backP_preact_c1(float d_preact[6][24][24],  float
    d_output[6][24][24], float preact[6][24][24]) {
    int i = blockIdx.z;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < 6 && j < 24 && k < 24) {
        float s = 1.0f / (1.0f + expf(-preact[i][j][k]));
        float ds = s * (1.0f - s);
        d_preact[i][j][k] = d_output[i][j][k] * ds;
    }
}
```

Listing 6: Backpropagation Kernel for Preactivation

Weight Backpropagation ($backP\_weight\_c1$)

- The kernel is launched with a grid of $numBlocks_weight_c1$ blocks, where each block is responsible for processing a specific filter.

- Each block contains $threadsPerBlock_weight_c1$ threads, with each thread computing the gradient of a particular weight.
- Each thread calculates the gradient for its assigned weight by iterating over the relevant input elements and preactivation gradients.

```
__global__ void backP_weight_c1(float d_weight[6][5][5], float d_preact[6][24][24], float p_output[28][28]) {
    int filter = blockIdx.z;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < 5 && j < 5) {
        float sum = 0.0f;

        int start_x = i;
        int start_y = j;

        for (int x = 0; x < 24; ++x) {
            for (int y = 0; y < 24; ++y) {
                sum += d_preact[filter][x][y] * p_output[x + start_x][y + start_y];
            }
        }

        float normalization_factor = 1.0f / (24.0f * 24.0f);
        d_weight[filter][i][j] = sum * normalization_factor;
    }
}
```

Listing 7: Backpropagation Kernel for Weights

Bias Backpropagation (*backP__bias__c1*)

- The kernel is launched with $blocks_bias_c1$ blocks, where each block processes an individual output feature map.
- Each block consists of $threads_bias_c1$ threads that work together to compute the total sum of preactivation gradients for the assigned feature map.
- This cumulative sum is then utilized to update the bias gradient for that specific feature map.

```
__global__ void backP_bias_c1(float bias[6], float d_preact[6][24][24]) {
    int feature = blockIdx.x;
    int idx = threadIdx.y * blockDim.x + threadIdx.x;
float dt = 1.0E-01f;
    __shared__ float partialSum[256];

    if (idx < 256) {
        partialSum[idx] = 0;
    }
    __syncthreads();

    int stride = blockDim.x * blockDim.y;
    int start = idx;
    for (int i = start; i < 24 * 24; i += stride) {
        int row = i / 24;
        int col = i % 24;
        atomicAdd(&partialSum[idx], d_preact[feature][row][col]);
    }
    __syncthreads();

    if (idx == 0) {
        float sum = 0.0f;
        for (int i = 0; i < blockDim.x * blockDim.y; i++) {
            sum += partialSum[i];
        }
        float d = 24.0f * 24.0f;
        atomicAdd(&bias[feature], dt* sum / d);
    }
}
```

Listing 8: Backpropagation Kernel for Biases

4.3.2 Pooling Layer

Forward Propagation ($ForwP\_s1$)

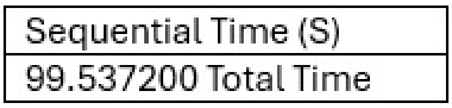
- The kernel is configured with $configSubsample1.blocks$ blocks, with each block handling the computation for one output feature map.
- Inside each block, $configSubsample1.threads$ threads are responsible for processing individual output elements in the feature map.
- Each thread scans a 2×2 region of the input feature map to determine the maximum value, using the specified filter size and stride, and stores the result in the output.
- This approach enhances memory access efficiency by ensuring coalesced memory operations.

```
__global__ void forwP_s1(float input[6][24][24], float output
    [6][12][12], int filter_size, int stride)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int filter = blockIdx.z;

    if (x < 12 && y < 12 && filter < 6) {
        float max_val = -INFINITY;
        for (int i = 0; i < filter_size; ++i) {
            for (int j = 0; j < filter_size; ++j) {
                int xi = x * stride + i;
                int yi = y * stride + j;
                max_val = fmaxf(max_val, input[filter][xi][yi]);
            }
        }
        output[filter][x][y] = max_val;
    }
}
```

Listing 9: Pooling Forward Propagation Kernel

5 Results



Sequential Time (S)
99.537200 Total Time

Table 1: Performance of Entire Network (OpenMP)

Number of threads	Time(ms)	Speed up s(p)	Efficiency
4	39.662612	2.5096	62.4

Table 2: Performance of Entire Network (MPI)

Number of Cores	Time(ms)	Speed up s(p)	Efficiency
2	67.279210	1.4795	72.4
4	51.488621	1.933468	48.3

Table 3: Performance of Entire Network (CUDA)

Number of threads	Time(s)	Speed up s(p)	Efficiency
27,776	3.456210	28.739523	1.0343461 $\times 10^{-3}$

For our parallel implementations, we conducted our tests using Google Colab with a T4 GPU. This environment provided us with the necessary computational power to run our CUDA C implementation efficiently.

Layer	**Time (ms)**
Convolution	93821.421213
Pooling	4987.432453
Fully Connected	2230.743192

Table 4: Sequential Time Results for CNN Implementation per epoch

6 Discussion

This project explored the parallelization of CNNs using MPI, OpenMP, and CUDA C, each of which exhibited unique strengths and challenges. CUDA C provided the most substantial speedup, leveraging the GPU's parallel processing capabilities effectively. However, its efficiency was limited by challenges such as synchronization overhead and kernel optimization, which suggest room for improvement in managing GPU resources.

MPI and OpenMP implementations achieved moderate speedups. MPI benefited from distributing workloads across processes, but communication overhead between nodes limited scalability. OpenMP, while simpler to implement, was constrained by the number of available CPU threads, making it less suitable for highly parallel tasks compared to CUDA.

Overall, the results highlight the importance of selecting the right parallelization approach based on hardware capabilities and workload characteristics. Further optimization, especially for GPU implementations, could unlock even greater performance gains.

7 Conclusion

This project demonstrated the benefits of parallelizing CNN computations using MPI, OpenMP, and CUDA C, each offering distinct trade-offs in performance and ease of implementation. It highlights the importance of leveraging hardware capabilities to meet the computational demands of complex tasks.

Challenges Faced: A major challenge was managing efficient communication in MPI, where overhead from large datasets impacted performance. In CUDA C, optimizing GPU memory management and kernel functions required significant effort. Debugging across the system added complexity, particularly when integrating multiple parallelization strategies.

Future Directions: Future work could explore hybrid strategies to optimize performance. further improvements in scalability are possible. This project provides a foundation for such developments.