HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY

CAPSTONE PROJECT REPORT:

Convolutional Neural Network Inference (MPI, CUDA)

***Supervised by*:**

Ph.D. Vu Van Thieu

***Presented by*:**

Vũ Hữu An - 20225467

Trần Hữu Đạo - 20220061

Lại Trí Dũng - 20225486

Đàm Quang Đức - 20225483

Tạ Tuấn Hải - 20225442

Nguyễn Trọng Tâm - 20225527

Hanoi - Vietnam

2025

Contents

1 Problem

Convolutional Neural Networks (CNNs) are widely used in modern artificial intelligence applications, particularly in computer vision tasks such as image classification, object detection, and scene recognition. As these networks grow in size and complexity, the computational demands of both training and inference become substantial. Inference—especially the forward pass—is a critical step in deploying CNNs in real-world systems, where high throughput and low latency are often required.

This project leverages three computational strategies to accelerate CNN inference: MPI for distributed processing across multiple CPU or GPU nodes, CUDA for fine-grained parallelism on individual GPUs, and a hybrid MPI+CUDA approach that combines the strengths of both. By distributing workloads across processes with MPI and offloading intensive operations to GPUs using CUDA, the hybrid implementation aims to maximize performance, scalability, and efficiency in large-scale forward inference tasks.

1.1 CNN Inference phase

During the inference phase of a Convolutional Neural Network (CNN), an input tensor is propagated forward through a sequence of layers to produce the final prediction. The architecture typically consists of multiple convolutional layers (each followed by activation and pooling), culminating in fully connected layers after flattening the feature maps.

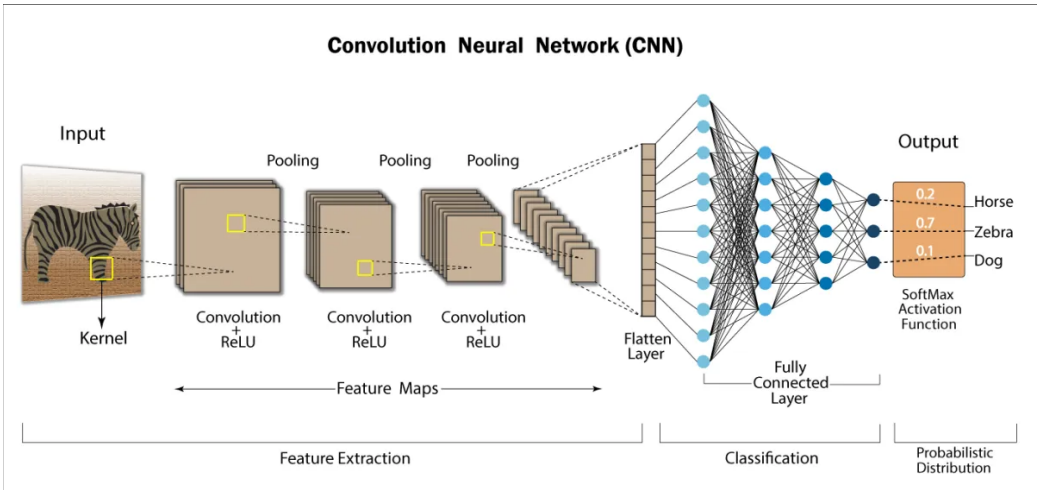

Figure 1: Illustration of CNN inference process

1.1.1 Convolutional Layer

Each convolutional layer applies a set of learnable filters to the input feature maps to extract spatial features. The convolution operation is defined as:

$$Y_{i,j}^{(k)} = \sum_{c=1}^{C} \sum_{m=1}^{K} \sum_{n=1}^{K} X_{i+m,j+n}^{(c)} \cdot W_{m,n}^{(c,k)} + b^{(k)}$$

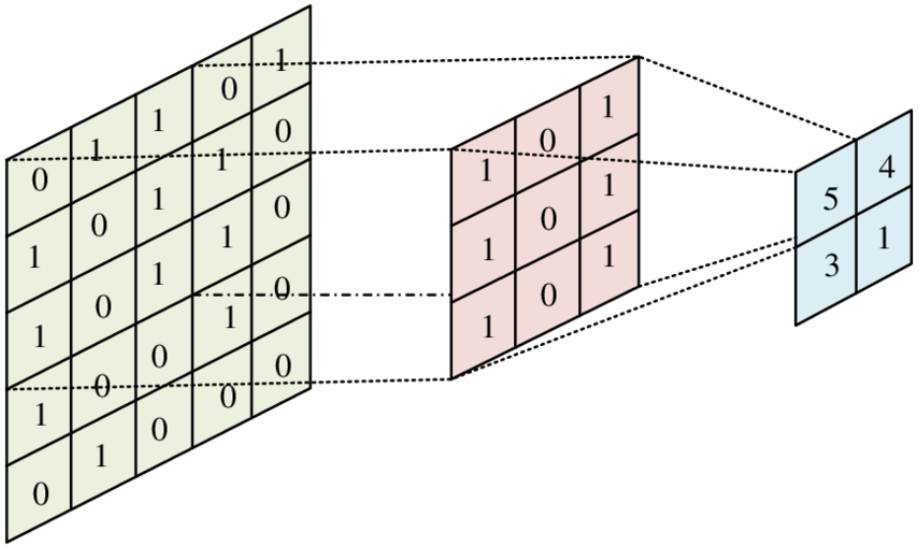

Figure 2: Illustration of Convolution layer

where:

- $X^{(c)}$: the input feature map for channel c,
- $W^{(c,k)}$: the filter weights from input channel c to output channel k,
- $b^{(k)}$: bias for output channel k,
- K: kernel size,
- C: number of input channels,
- F: number of filters (output channels).

1.1.2 Activation Function: ReLU

After convolution, a non-linear activation function is applied. In this project, we use the **ReLU** activation.

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

1.1.3 Max Pooling

Max pooling reduces the spatial size of the feature maps. For a pooling window of size $P \times P$, the maximum value is selected:

$$Y_{i,j} = \max \left\{ X_{i',j'} \mid (i', j') \in \text{window} \right\}$$

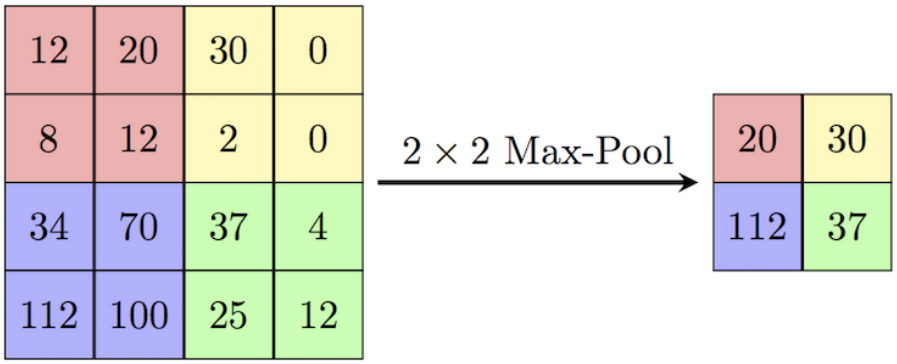

Figure 3: Illustration of Max pooling layer

1.1.4 Layer Stacking

This combination of convolution, activation, and pooling is repeated across multiple layers, allowing the network to learn hierarchical representations from low-level edges to high-level features.

1.1.5 Flattening

After the final convolutional block, the resulting feature maps are flattened into a one-dimensional vector to be used as input to the fully connected layers:

$$\text{Flatten} : R^{C \times H \times W} \rightarrow R^{C \cdot H \cdot W}$$

1.1.6 Fully Connected Layers

The flattened vector passes through one or more dense layers. Each fully connected layer performs:

$$y = f(Wx + b)$$

where f again refers to the ReLU activation, and W, b are the weights and biases of the dense layer.

2 Algorithm (Serial Implementation)

This section details the single-processor algorithm embodied in `src_cnn.c`. The procedure consists of four stages: `conv_forward`, `max_pool_forward`, `flatten`, and `fc_forward`

2.1 Stage 1 – conv_forward

The `conv_forward` algorithm performs a valid 2D convolution over a 3D input tensor using a given convolutional layer. It slides a learnable kernel across each spatial location of the input and applies a ReLU activation to produce the output.

Input:

- `input` – a 3D tensor of shape (C_{in}, H_{in}, W_{in})
- `layer` – convolutional layer parameters:
  - `weights` of shape (C_{out}, C_{in}, K, K)
  - `biases` of shape (C_{out})
  - `kernel_size` $= K$

Output: a 3D tensor of shape $(C_{out}, H_{out}, W_{out})$, where:

$$H_{out} = H_{in} - K + 1, \quad W_{out} = W_{in} - K + 1$$

Procedure: For each output channel c_{out} and each spatial position (i, j) in the output tensor:

- Accumulate the sum of element-wise products between the $K \times K$ kernel slice (for each c_{in}) and the corresponding input patch.

- Add the bias $b_{c_{out}}$.

- Apply the ReLU activation:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

For this implementation, we will use:

- Stride = 1 and no padding (valid convolution)

- Flat array indexing to traverse tensors in memory efficient order

Time Complexity:

$$\mathcal{O}(C_{out} \cdot C_{in} \cdot K^2 \cdot H_{out} \cdot W_{out})$$

Algorithm 1 CONV_FORWARD(input, layer)

$out\_w \leftarrow input.width - layer.kernel\_size + 1$
$out\_h \leftarrow input.height - layer.kernel\_size + 1$
$out\_ch \leftarrow layer.out\_channels$
$in\_ch \leftarrow layer.in\_channels$
$ks \leftarrow layer.kernel\_size$
Allocate $output\_data[out\_w \cdot out\_h \cdot out\_ch]$
for $oc = 0$ to $out\_ch - 1$ **do**
    for $i = 0$ to $out\_h - 1$ **do**
        for $j = 0$ to $out\_w - 1$ **do**
            $sum \leftarrow 0$
            for $ic = 0$ to $in\_ch - 1$ **do**
                for $ki = 0$ to $ks - 1$ **do**
                    for $kj = 0$ to $ks - 1$ **do**
                        $in\_y \leftarrow i + ki$, $in\_x \leftarrow j + kj$
                        $in\_idx \leftarrow ic \cdot input.height \cdot input.width + in\_y \cdot input.width + in\_x$
                        $w\_idx \leftarrow oc \cdot in\_ch \cdot ks^2 + ic \cdot ks^2 + ki \cdot ks + kj$
                        $sum \mathrel{+}= input.data[in\_idx] \cdot layer.weights[w\_idx]$
                    end for
                end for
            end for
            $out\_idx \leftarrow oc \cdot out\_h \cdot out\_w + i \cdot out\_w + j$
            $output\_data[out\_idx] \leftarrow \text{ReLU}(sum + layer.biases[oc])$
        end for
    end for
end for
return Tensor3D$\{out_w, out_h, out_ch, output_data\}$

2.2 Stage 2 - max_pool_forward

The `maxpool_forward` function performs a 2D max pooling operation over the input tensor. It divides each spatial channel into non-overlapping regions of size `pool_size` × `pool_size`, and for each region, it outputs the maximum value. This operation reduces the spatial resolution while retaining the most salient features.

Let:

- $H_{\text{in}}, W_{\text{in}}$ be the height and width of the input
- C be the number of channels
- P be the pooling size
- $H_{\text{out}} = H_{\text{in}}/P$, $W_{\text{out}} = W_{\text{in}}/P$

Time Complexity:

$$\mathcal{O}(C \cdot H_{\text{out}} \cdot W_{\text{out}} \cdot P^2)$$

This accounts for scanning every pooling window in each channel.

Algorithm 2 MAXPOOL_FORWARD(input, pool_size)

$out\_w \leftarrow input.width/pool\_size$
$out\_h \leftarrow input.height/pool\_size$
$out\_ch \leftarrow input.channels$
Allocate $output\_data[out\_w \cdot out\_h \cdot out\_ch]$
for $c = 0$ to $out\_ch - 1$ **do**
    for $i = 0$ to $out\_h - 1$ **do**
        for $j = 0$ to $out\_w - 1$ **do**
            $max \leftarrow -10^9$
            for $pi = 0$ to $pool\_size - 1$ **do**
                for $pj = 0$ to $pool\_size - 1$ **do**
                    $in\_y \leftarrow i \cdot pool\_size + pi$
                    $in\_x \leftarrow j \cdot pool\_size + pj$
                    $in\_idx \leftarrow c \cdot input.height \cdot input.width + in\_y \cdot input.width + in\_x$
                    if $input.data[in\_idx] > max$ **then**
                        $max \leftarrow input.data[in\_idx]$
                    end if
                end for
            end for
            $out\_idx \leftarrow c \cdot out\_h \cdot out\_w + i \cdot out\_w + j$
            $output\_data[out\_idx] \leftarrow max$
        end for
    end for
end for
Free input.data
return Tensor3D$\{out_w, out_h, out_ch, output_data\}$

2.3 Stage 3 - flatten + Fully connected

This stage transforms the spatial 3D feature map into a 1D vector and applies a fully connected (dense) layer with ReLU activation.

Flatten

The `flatten` function converts a 3D tensor of shape (H, W, C) into a 1D vector of size $N = H \cdot W \cdot C$ by storing the data in row-major order.

Time Complexity: $\mathcal{O}(H \cdot W \cdot C)$

Algorithm 3 FLATTEN(input)

```
total ← input.width · input.height · input.channels
Allocate v.data[total]
for i = 0 to total − 1 do
    v.data[i] ← input.data[i]
end for
Free input.data
return Vector{total, v.data}
```

Fully Connected Layer (FC)

The `fc_forward` function performs a matrix-vector multiplication between the input vector and a weight matrix, adds a bias term, and applies the ReLU activation.

Let:

- I be the input size (flattened vector length)
- O be the number of output features

Time Complexity: $\mathcal{O}(O \cdot I)$

Algorithm 4 FC_FORWARD(input, layer)

```
Allocate out.data[layer.out_features]
for i = 0 to layer.out_features − 1 do
    sum ← 0
    for j = 0 to input.size − 1 do
        sum += layer.weights[i · input.size + j] · input.data[j]
    end for
    out.data[i] ← ReLU(sum + layer.biases[i])
end for
Free input.data
return Vector{layer.out_features, out.data}
```

2.4 Final stage - Combining all together into cnn_forward

The `cnn_forward` function performs the full forward pass through the convolutional neural network (CNN), consisting of multiple convolutional layers followed by fully connected layers. The input is passed sequentially through each stage:

1. Copies the input image data into a 3D tensor.
2. Applies a sequence of convolutional layers (optionally followed by pooling).
3. Flattens the resulting 3D tensor into a 1D vector.
4. Applies a sequence of fully connected (dense) layers with ReLU activations.
5. Stores the final scalar output in `cnn.output`.

Let:

- L_c: number of convolutional layers
- L_f: number of fully connected layers
- For each conv layer l: output size $H_{out}^{(l)}, W_{out}^{(l)}, C_{out}^{(l)}$, kernel size $K^{(l)}$, input channels $C_{in}^{(l)}$
- For each FC layer l: input size $I^{(l)}$, output size $O^{(l)}$

Time Complexity:

$$\mathcal{O}\left(\sum_{l=1}^{L_c} C_{out}^{(l)} \cdot C_{in}^{(l)} \cdot (K^{(l)})^2 \cdot H_{out}^{(l)} \cdot W_{out}^{(l)} + \sum_{l=1}^{L_f} O^{(l)} \cdot I^{(l)}\right)$$

Algorithm 5 CNN_FORWARD(cnn)

$total\_input \leftarrow cnn.input\_width \cdot cnn.input\_height \cdot cnn.input\_channels$
Allocate $copy[total\_input]$
for $i = 0$ to $total\_input - 1$ **do**
    $copy[i] \leftarrow cnn.input\_data[i]$
end for
$x \leftarrow$ Tensor3D$\{cnn.input_width, cnn.input_height, cnn.input_channels, copy\}$
for $i = 0$ to $cnn.num\_conv\_layers - 1$ **do**
    $x \leftarrow$ CONV_FORWARD$(x, \&cnn.conv_layers[i])$
end for
$v \leftarrow$ FLATTEN(x)
for $i = 0$ to $cnn.num\_fc\_layers - 1$ **do**
    $v \leftarrow$ FC_FORWARD$(v, \&cnn.fc_layers[i])$
end for
$cnn.output \leftarrow v.data[0]$
Free $v.data$

3 Parallel Design

3.1 MPI-Based Implementation

In the MPI-based design, the CNN forward pass is distributed across multiple processes using a data-parallel approach for each layer type. Each process handles a portion of the computation and communicates partial results using collective MPI operations.

Initialization :

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &NP);
MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
```

3.1.1 Convolutional Layer

The convolution operation is parallelized by distributing the output image **rows** across processes. Each process computes the output values for a subset of rows for all output channels. This requires each process to:

- Access a subset of rows from the input tensor.

```
int out_h = input.height - kernel_size + 1;
int rows_per_proc = out_h / NP;
int rem = out_h % NP;
int my_rows = (Rank < rem) ? rows_per_proc + 1 : rows_per_proc;
```

```
int start_row = Rank * rows_per_proc + (Rank < rem ? Rank : rem);
```

- Perform the full convolution kernel computation over those rows.

```
for (int oc = 0; oc < out_channels; oc++) {
    for (int i = 0; i < my_rows; i++) {
        int global_i = start_row + i;
        for (int j = 0; j < out_w; j++) {
            float sum = 0.0f;
            // inner loops for kernel omitted
            my_output[oc * my_rows * out_w + i * out_w + j] = relu(sum + bias[
                oc]);
        }
    }
}
```

- Store intermediate results locally.

Once local computation is complete, `MPI_Gatherv` is used to collect the partial outputs at rank 0. The final output tensor is constructed only on the root process. This approach avoids data replication and balances workload based on image height.

```
MPI_Gatherv(
    my_output,
    my_rows * out_w * out_ch,
    MPI_FLOAT,
    output_data,
    recvcounts,
    displs,
    MPI_FLOAT,
    0,
    MPI_COMM_WORLD
);
```

3.1.2 Max Pooling Layer

The max pooling operation is parallelized by distributing the **output rows** across MPI processes. Each process is responsible for computing a subset of the output rows across all channels.

- Compute the output tensor dimensions and determine the row range for each process:

```
int out_w = input.width / pool_size;
int out_h = input.height / pool_size;
int out_ch = input.channels;

int rows_per_proc = out_h / NP;
int rem = out_h % NP;
int my_rows = (Rank < rem) ? rows_per_proc + 1 : rows_per_proc;
int start_row = Rank * rows_per_proc + (Rank < rem ? Rank : rem);
```

- For each output pixel, the local process computes the maximum value over a $pool\_size \times pool\_size$ window in the corresponding input region.

```
for (int c = 0; c < out_ch; c++) {
    for (int i = 0; i < my_rows; i++) {
        int global_i = start_row + i;
        for (int j = 0; j < out_w; j++) {
            float max = -1e9;
            for (int pi = 0; pi < pool_size; pi++) {
                for (int pj = 0; pj < pool_size; pj++) {
                    int in_y = global_i * pool_size + pi;
                    int in_x = j * pool_size + pj;
                    int in_idx = c * input.height * input.width + in_y * input.
                        width + in_x;
                    if (input.data[in_idx] > max)
                        max = input.data[in_idx];
                }
            }
            int out_idx = c * my_rows * out_w + i * out_w + j;
            my_output[out_idx] = max;
        }
    }
}
```

- After local computation, each process sends its computed rows to rank 0 using `MPI_Gatherv`.

This design ensures each process computes a balanced workload based on the output height. Input rows that do not fit completely into a pooling window (e.g., when the height is not divisible by `pool_size`) are ignored, following standard CNN behavior.

3.1.3 Fully Connected Layer

The fully connected layer is parallelized by distributing the **output neurons** (i.e., the output vector dimensions) across MPI processes. Each process is responsible for computing a subset of the output values independently.

- Divide the total number of output neurons across processes:

```
int out_features = layer->out_features;
int in_features = input.size;

int outputs_per_proc = out_features / NP;
int rem = out_features % NP;
int my_outputs = (Rank < rem) ? outputs_per_proc + 1 : outputs_per_proc;
int start_idx = Rank * outputs_per_proc + (Rank < rem ? Rank : rem);
```

- Each process computes its assigned output neurons:

```
for (int i = 0; i < my_outputs; i++) {
    int global_i = start_idx + i;
    float sum = 0.0f;
    for (int j = 0; j < in_features; j++) {
        sum += layer->weights[global_i * in_features + j] * input.data[j];
    }
    my_output_data[i] = relu(sum + layer->biases[global_i]);
}
```

- Gather all output segments into a single vector on rank 0 using `MPI_Gatherv`.

3.1.4 Overall Design and Synchronization

All intermediate results are kept only on the root process (rank 0) after each layer. Broadcasts (`MPI_Bcast`) are used to distribute the flattened vector before fully connected layers. This centralized aggregation allows reuse of existing sequential logic and avoids synchronization complexities at the cost of communication overhead.

Time Complexity

Let P be the number of MPI processes.

- **Convolution:** Each process computes $\mathcal{O}\left(\frac{H_{out}}{P} \cdot W_{out} \cdot C_{out} \cdot K^2 \cdot C_{in}\right)$ operations.

- **Max Pooling:** Each process performs $\mathcal{O}\left(\frac{H_{out}}{P} \cdot W_{out} \cdot C\right)$ operations.
- **Fully Connected:** Each process computes $\mathcal{O}\left(\frac{O}{P} \cdot I\right)$ dot products, where I is input size, O is output size.

Communication cost per layer includes gather (`MPI_Gatherv`) and broadcast (`MPI_Bcast`) with time complexity $\mathcal{O}(\log P)$ latency and $\mathcal{O}(n)$ bandwidth per transfer.

3.2 CUDA-Based Implementation

The CUDA-based implementation of CNN inference offloads the computation of convolution, pooling, and fully connected layers onto the GPU. Each layer type is executed by a corresponding CUDA kernel, leveraging parallel thread blocks for efficient execution.

3.2.1 Convolutional Layer

The convolution kernel is executed with a 3D grid of thread blocks where:

- Each thread block covers a 2D spatial tile (output height $\times$ width).
- Each block in the `z`-dimension handles a separate output channel.

The output value at each spatial location is computed by summing over all input channels and kernel elements.

```
int out_c = blockIdx.z;
int out_x = blockIdx.x * blockDim.x + threadIdx.x;
int out_y = blockIdx.y * blockDim.y + threadIdx.y;

if (out_x < out_width && out_y < out_height && out_c < out_channel) {
    float sum = 0.0f;
    for (int in_c = 0; in_c < in_channel; ++in_c) {
        for (int ky = 0; ky < kernel_size; ++ky) {
            for (int kx = 0; kx < kernel_size; ++kx) {
                int in_x = out_x + kx;
                int in_y = out_y + ky;
                int in_idx = in_c * in_height * in_width + in_y * in_width +
                    in_x;
                int w_idx = out_c * in_channel * ks * ks + in_c * ks * ks + ky
                    * ks + kx;
                sum += input[in_idx] * weights[w_idx];
            }
```

```
        }
    }
    int out_idx = out_c * out_height * out_width + out_y * out_width + out_x;
    output[out_idx] = relu_cuda(sum + biases[out_c]);
}
```

This approach ensures each thread independently computes one output value, leading to high concurrency.

3.2.2 Max Pooling Layer

Max pooling is performed in parallel over the spatial grid and channels. Each CUDA thread computes one output value by reducing a $pool\_size \times pool\_size$ window.

```
int c = blockIdx.z;
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;

if (i < out_h && j < out_w && c < in_c) {
    float max_val = -1e9;
    for (int pi = 0; pi < pool_size; ++pi) {
        for (int pj = 0; pj < pool_size; ++pj) {
            int y = i * pool_size + pi;
            int x = j * pool_size + pj;
            int idx = c * in_h * in_w + y * in_w + x;
            max_val = fmaxf(max_val, input[idx]);
        }
    }
    int out_idx = c * out_h * out_w + i * out_w + j;
    output[out_idx] = max_val;
}
```

The use of a 3D grid allows concurrent evaluation of all pooling windows across all channels.

3.2.3 Fully Connected Layer

Fully connected layers are computed entirely on the GPU by performing matrix-vector multiplication followed by a nonlinear activation. While each output element can be computed in a separate thread, in this implementation, the FC layer computation is typically done on the host or in batch mode on the device.

The output element y_i is computed as:

$$y_i = \texttt{relu}\left(\sum_{j=0}^{N-1} W_{ij} \cdot x_j + b_i\right)$$

The input vector is first copied to the GPU memory, multiplied with the weight matrix, and the result is copied back to the host for further processing or final output.

Time Complexity

Let $T_x \times T_y$ be the number of threads per block, and $B_x \times B_y \times B_z$ be the number of blocks launched across the output tensor dimensions. Assume one thread computes one output element whenever possible.

- **Convolution:** For input of size $H \times W \times C_{in}$ and kernel size $K \times K$ producing C_{out} output channels,
  - Total work: $\mathcal{O}(H_{out} \cdot W_{out} \cdot C_{out} \cdot K^2 \cdot C_{in})$
  - Per-thread work (ideal parallelism): $\mathcal{O}(K^2 \cdot C_{in})$
  - Parallel execution time (ideal): $\mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C_{out} \cdot K^2 \cdot C_{in}}{T}\right)$, where $T = T_x \cdot T_y \cdot B_x \cdot B_y \cdot B_z$
- **Max Pooling:** For pooling window size $P \times P$:
  - Total work: $\mathcal{O}(H_{out} \cdot W_{out} \cdot C \cdot P^2)$
  - Per-thread work: $\mathcal{O}(P^2)$
  - Parallel execution time (ideal): $\mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C \cdot P^2}{T}\right)$
- **Flatten:**
  - Total work: $\mathcal{O}(H \cdot W \cdot C)$
  - Per-thread work: $\mathcal{O}(1)$
  - Parallel execution time (ideal): $\mathcal{O}(1)$
- **Fully Connected Layer:** For input dimension I and output size O:
  - Total work: $\mathcal{O}(I \cdot O)$
  - Per-thread work (assuming 1 thread per output): $\mathcal{O}(I)$
  - Parallel execution time (ideal): $\mathcal{O}\left(\frac{I \cdot O}{T}\right)$

The actual performance depends on GPU occupancy, memory coalescing, warp divergence, and shared/global memory usage. While the theoretical parallel time assumes ideal distribution, real-world performance may involve latency due to memory access patterns and launch configuration.

3.3 Hybrid MPI + CUDA Implementation

The hybrid MPI + CUDA implementation combines distributed computing with GPU acceleration to efficiently process multiple images using a convolutional neural network (CNN). This approach distributes input images across multiple MPI processes, each associated with a GPU, and processes them in parallel. Below, we detail the workflow and implementation of the hybrid approach.

3.3.1 Workflow

The hybrid implementation is designed to process multiple input images in parallel across multiple GPUs. The workflow is as follows:

1. The root MPI process initializes the CNN model and generates or loads the input images.
2. The input images are distributed across MPI processes using `MPI_Scatterv`.
3. Each MPI process processes its assigned images using the CUDA-based CNN implementation.
4. The results are gathered back to the root process using `MPI_Gatherv`.

3.3.2 Implementation

The implementation consists of two main functions: `process_image_batch` and `cnn_forward_hybrid`. The former processes a batch of images on a single GPU, while the latter handles the distribution and collection of images and results across MPI processes.

Processing a Batch of Images The `process_image_batch` function processes a batch of images assigned to a single MPI process. Each image is processed using the CUDA-based CNN implementation. The pseudo-code for this function is as follows:

```
function process_image_batch(images, num_images, cnn, results, comm):
    rank = get_rank(comm)
    for each image in images:
```

```
        create a CNN instance for the image
        copy the input data to the CNN instance
        process the image using CUDA-based CNN implementation
        store the result in the results array
```

Distributed Inference The `cnn_forward_hybrid` function handles the distribution of images across MPI processes and the collection of results. It uses `MPI_Scatterv` to distribute the images and `MPI_Gatherv` to gather the results. The pseudo-code for this function is as follows:

```
function cnn_forward_hybrid(all_images, total_images, cnn, all_results, comm):
    rank, size = get_rank_and_size(comm)

    calculate images_per_proc and remainder
    determine my_num_images and my_start_idx based on rank

    allocate memory for local images and results

    if rank == 0:
        calculate recvcounts and displs for scattering images

    scatter images to processes using MPI_Scatterv

    process local batch of images using process_image_batch

    if rank == 0:
        calculate recvcounts and displs for gathering results

    gather results from processes using MPI_Gatherv

    free allocated memory
```

3.3.3 Analysis

Let $T_x \times T_y$ be the number of threads per block, and $B_x \times B_y \times B_z$ be the number of blocks launched across the output tensor dimensions. Assume one thread computes one output element whenever possible.

The hybrid MPI + CUDA implementation achieves efficient parallel processing by distributing the workload across multiple GPUs. The time complexity of the distributed inference is

dominated by the CUDA-based CNN processing. More specifically: which is

- $\mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C_{out} \cdot K^2 \cdot C_{in}}{T}\right)$ for the convolutional layer,
- $\mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C \cdot K^2}{T}\right)$ for the max pooling layer,
- $\mathcal{O}\left(\frac{I \cdot O}{T}\right)$ for the fully connected layer.

Where $T = T_x \cdot T_y \cdot B_x \cdot B_y \cdot B_z$

However, the total computational workload is distributed across P GPUs, where P is the number of MPI processes. This reduces the effective time complexity per GPU to:

$$\mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C_{out} \cdot K^2 \cdot C_{in}}{T \cdot P}\right), \mathcal{O}\left(\frac{H_{out} \cdot W_{out} \cdot C \cdot K^2}{T \cdot P}\right), \mathcal{O}\left(\frac{I \cdot O}{T \cdot P}\right).$$

The communication overhead introduced by MPI includes the scattering of input images and the gathering of results. These operations have a time complexity proportional to the size of the input and output data:

$$\mathcal{O}\left(\frac{\text{Input Size}}{P}\right) \quad \text{and } \mathcal{O}\left(\frac{\text{Output Size}}{P}\right),$$

where the division by P reflects the distribution of data across processes.

In practice, the communication overhead is typically negligible compared to the computational cost on the GPU, especially for large-scale image processing tasks. The hybrid approach achieves near-linear scalability with the number of GPUs, provided the communication overhead remains small relative to the computation time.

This hybrid MPI + CUDA implementation is well-suited for large-scale image processing tasks, where the combination of distributed computing and GPU acceleration provides significant performance benefits. By leveraging P GPUs, the total processing time is reduced approximately by a factor of P, making the system highly efficient for parallel workloads.

4 Results and Performance Analysis

4.1 Benchmark Setup

The benchmark tests were conducted processing a batch of 16 images simultaneously, with each image having the following specifications:

- Input dimensions: RGB images of $512 \times 512 \times 3$
- Network architecture:
  - 12 convolutional layers
  - 5×5 kernel size for all convolutions
  - 2×2 max pooling after each convolution
  - Progressive dimension reduction in fully connected layers: flattened $\rightarrow$ 128 $\rightarrow$ 64 $\rightarrow$ 32 $\rightarrow$ 16 $\rightarrow$ 8 $\rightarrow$ 4 $\rightarrow$ 2 $\rightarrow$ 1
- Weight initialization: $\mathcal{N}(0, 0.25)$ for all layers
- Activation function: ReLU

4.2 Performance Results

Table 1: Performance Comparison of Different Implementation Strategies

Strategy	**Wall Time (s)**	**Speedup**
Serial (Baseline)	3.927	1.000
MPI (8 processes)	3.209	1.223
CUDA (Single GPU)	0.956	4.108
MPI + CUDA (2 GPUs)	0.446	8.805

4.2.1 Analysis of Results

Our experimental results, based on the average processing time per image, reveal several key insights:

1. **Pure MPI Implementation (8 processes)**:
   - Achieves a speedup of 1.223$\times$ over sequential execution
   - The modest improvement suggests overhead from process communication and data distribution

- Limited by CPU-bound nature of computations

2. **CUDA Implementation (Single GPU)**:
   - Demonstrates significant speedup of 4.108×
   - Effectively utilizes GPU's parallel architecture
   - Shows the advantage of GPU's SIMD capabilities for CNN operations
3. **Hybrid MPI + CUDA (2 GPUs)**:
   - Achieves the best performance with 8.805× speedup
   - Nearly linear scaling with the number of GPUs (2× CUDA performance)
   - Effectively combines distributed processing with GPU acceleration

5 Conclusion

Our implementation and analysis of parallel CNN inference using various strategies has yielded significant insights into the effectiveness of different parallelization approaches. The project demonstrates the successful integration of MPI and CUDA technologies to achieve high-performance CNN inference.

Key Achievements:

1. **Comprehensive Implementation**: Successfully developed four distinct versions of CNN inference:
   - Sequential implementation as baseline
   - MPI-based distributed CPU implementation
   - CUDA-based single-GPU implementation
   - Hybrid MPI+CUDA implementation for multi-GPU systems
2. **Scalable Architecture**:
   - Modular design allowing easy switching between implementations
   - Efficient handling of multiple input images
   - Flexible CNN architecture supporting various layer configurations
3. **Performance Optimization**:

- Nearly 9× speedup with hybrid implementation
- Effective load balancing across multiple GPUs
- Minimal communication overhead in distributed processing

Limitations and Challenges:

- MPI implementation shows limited scaling due to communication overhead
- Memory transfer between CPU and GPU remains a bottleneck
- Current implementation requires identical GPU configurations

Future Work:

- **Enhanced Load Balancing**: Implement adaptive work distribution based on device capabilities
- **Memory Optimization**: Explore zero-copy memory and unified memory architectures
- **Network Architecture**: Add support for more complex CNN architectures
- **Multi-Node Support**: Extend to cluster environments with heterogeneous GPU configurations

This project demonstrates that combining MPI and CUDA can achieve significant performance improvements in CNN inference. The hybrid approach effectively leverages both distributed computing and GPU acceleration, providing a robust foundation for high-performance deep learning applications. The modular design and clear performance benefits make this implementation valuable for both research and practical applications in computer vision and deep learning.