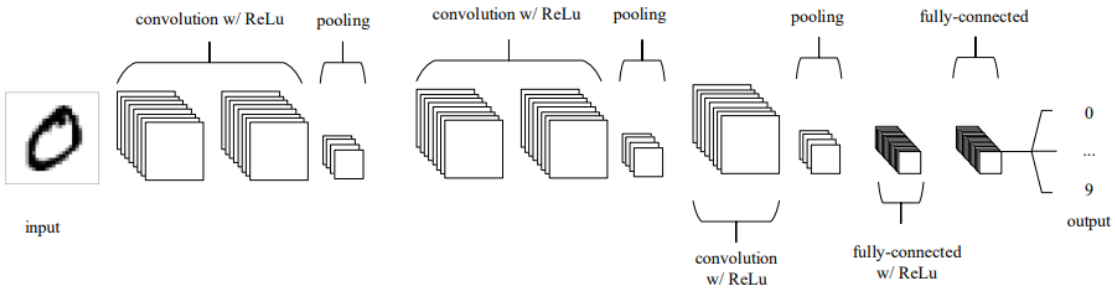
Parallelizing Convolutional Neural Networks

Tamer Kobba, Yousef Idress, Mohammad Yazbek

April 30, 2024

1 Introduction

Convolutional Neural Networks (CNNs) are a class of deep learning algorithms predominantly used for analyzing visual imagery. They excel at processing data with a grid-like topology, such as images, by employing specialized layers to automatically detect and prioritize the most relevant features. As the computational demands of training and deploying CNNs grow, particularly with the advent of high-resolution data and complex model architectures, the need for efficient parallelization strategies becomes crucial. The biggest drawback with using CNNs is training them as they require very large datasets (usually in the order of hundreds of thousands) to converge to their global optima. As a result, they are very computationally intensive and can take many hours and even days to run on traditional CPUs. In this project, we aim to exploit the parallelism in each layer of the CNN setting the stage for a detailed exploration of various methods for parallelizing their operations across multiple hardware platforms to enhance performance and scalability.



2 How do convolutional neural networks work?

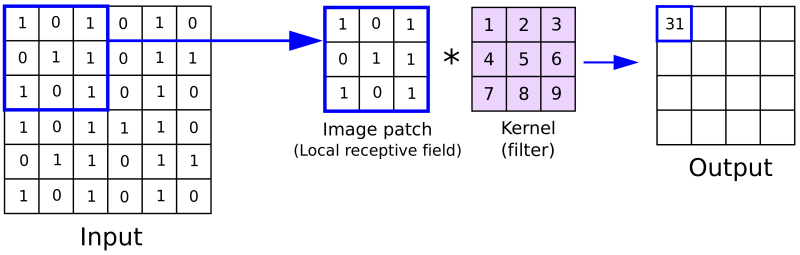
Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

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

The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

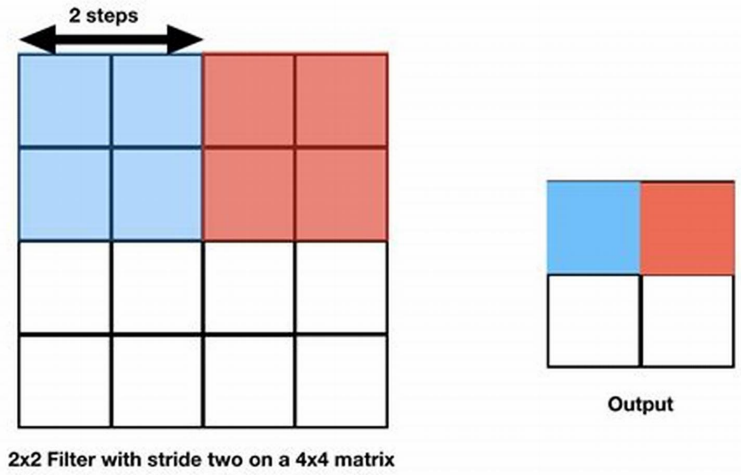
2.1 Convolutional layer

The convolution layer is the core building block of a CNN. It applies a set of learnable filters to the input image, which are essentially small matrices used to detect specific features like edges, colors, or textures. Each filter slides across the image spatially, computing dot products between the filter values and the input, producing a feature map. This process allows the network to capture spatial hierarchies in data.



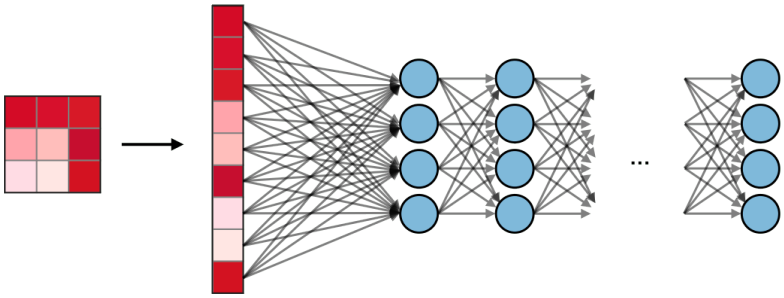
2.2 Pooling layer

Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array.While a lot of information is lost in the pooling layer, it also has a number of benefits to the CNN. They help to reduce complexity, improve efficiency, and limit risk of overfitting.



2.3 Fully-connected (FC) layer

The fully-connected layer contains neurons of which are directly connected to the neurons in the two adjacent layers, without being connected to any layers within them. This is analogous to way that neurons are arranged in traditional forms of ANN



3 CNN Structure

First we're gonna start by breaking down the structure of our sequential Implementation and then how we parallelized each layer.

- Input layer It prepares the 28x28 pixel images to be processed by subsequent layers. Its main role is to handle data input, ensuring the images are correctly formatted and passed along to the next layer.
- Convolutional layer This first convolutional layer plays a crucial role in feature detection. It uses 6 filters, each of size 5x5, to scan through the 28x28 input image. The output from this layer is a set of 6 feature maps, each with a reduced dimension of 24x24.
- Pooling layer We used a unique approach to reduce the spatial dimension of its input. Unlike traditional pooling layers that perform straightforward max or average operations, this layer employs a set of trainable parameters that act similarly to a convolution operation with a 4x4 kernel. The output is a significantly reduced spatial dimension of 6x6, achieved by using a convolution-like operation with a stride of 4, allowing for a dramatic reduction in size. This approach attempts to maintain learnable parameters in a layer that typically performs a fixed function, which lead the CNN to perform much better than a traditional approach when testing
- Fully-connected (FC) layer The final layer of the network transforms the condensed feature representations from the pooling into class predictions for the 10 possible digits (0 through 9) in MNIST. Each unit in this layer is connected to all activations from the previous layer, using these connections to weigh the importance of various features for making final predictions. Bias terms are added here as well, which are adjusted during training to refine the decision boundaries further. This layer is critical as it integrates all learned features into probabilities or classifications that determine the network's output.

4 Parallelization

When Parallelizing our CNN we faced some challenges as parallelizing has some challenges

- Low Arithematic Intensity : The computational results of each layer need to be shared among all threads as the computation of the next layer depends on the results of the previous layer. Similarly, the back-propagation step also requires results from previous layers. As a result, CNNs inherently have a high communication to computation ratio.
- Dependencies : layer l depends on the result of layer l - 1. So there is parallelism within a layer but not between layers.

We selected three powerful and distinct platforms for implementing parallelization in our project: MPI (Message Passing Interface), OpenMP (Open Multi-Processing), and CUDA C. Each platform is uniquely suited to different types of hardware and programming challenges, enabling us to efficiently distribute and manage computational tasks across various processing environments. We will now discuss how parallelized everything using these 3 platforms.

4.1 MPI

We parallelized MPI by using data partitioning using rank and collapsed all nested for loops to better utilize our threads. We used MPI_Gather with two helper functions to collect our results, build and flatten to build our tensor and collapse it into a 1D array when needed in order to pass it through our parameters.

4.2 OpenMP

The main approach to parallelizing the functions involved identifying computationally intensive loops and data parallelism opportunities. OpenMP directives such as parallel for were used to distribute loop iterations across threads, enabling concurrent execution and speeding up computation. Additionally, collapse clauses were utilized to collapse nested loops into a single loop, enhancing parallelism and reducing loop overhead.

For example, in the $fp\_c1function$, which computes the preactivation values for the first convolutional layer, the nested loops responsible for iterating over the input dimensions and computing the convolutional operation were parallelized using the parallel for directive. By distributing the computation of preactivation values across threads, significant speedup was achieved, leveraging the multicore architecture efficiently.

Similarly, in functions like bp_preact_s1 and bp_output_c1, which involve gradient computations and backpropagation, loop-level parallelization was applied using OpenMP pragmas. These sections of the code are computationally intensive and benefit greatly from parallel execution, especially when dealing with large input and weight dimensions.

4.3 CUDA C

4.3.1 Convolutional layer

Forward Propagation (fp_c1)

- **Parallelization Strategy:** Output Stationarity
- The kernel is launched with a grid of dim3(6) blocks, where each block corresponds to one output feature map.
- Within each block, there are dim3$(24, 24)$ threads, one for each output element in the feature map.
- Each thread computes the dot product between the input patch and the corresponding filter weights, and adds the bias to produce the output element.
- This approach allows for efficient memory access patterns and coalesced memory loads/stores.

```
__global__ void fp_c1(float (*input)[28], float (*preact)[24][24], float (*weight)
    [5][5], float *bias)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;

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

    float sum = 0.0f;
    for (int i = 0; i < 28; ++i) {
        for (int j = 0; j < 28; ++j) {
            for (int k = 0; k < 5; ++k) {
                for (int l = 0; l < 5; ++l) {
                    sum += input[i + k][j + l] * weight[z][k][l];
                }
            }
        }
    }
```

```
    preact[z][y][x] = sum + bias[z];
}
```

Applying Activation Function (*apply_step_function*)

- **Parallelization Strategy:** Same as fp_c1
- The kernel is launched with the same configuration as fp_c1, i.e., $configLayer1.blocks$ 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;
}
```

Backpropagation for Output (*bp_output_c1*)

- **Parallelization Strategy:** Input Stationarity
- The kernel is launched with a grid of $numBlocks\_output\_c1$ blocks, where each block processes a tile of the input feature map.
- Within each block, there are $threadsPerBlock\_output\_c1$ threads, each responsible for computing the gradient for one output element.
- Each thread iterates over the corresponding input elements and filter weights to compute the gradient for its output element.
- This approach allows for efficient memory access patterns and coalesced memory loads/stores.

```
__global__ void bp_output_c1(float (*d_output)[24][24], float (*weight)[4][4], float
    (*d_preact)[6][6])
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;
    int z = bz;

    float sum = 0.0f;
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            sum += weight[z][i][j] * d_preact[z][y / 2 + i][x / 2 + j];
        }
    }
    d_output[z][y][x] = sum;
}
```

Backpropagation for Preactivation (*bp_preact_c1*)

- **Parallelization Strategy:** Output Stationarity

- The kernel is launched with a grid of $numBlocks\_bp\_preact\_c1$ blocks, where each block corresponds to one output feature map.
- Within each block, there are $threadsPerBlock\_bp\_preact\_c1$ threads, one for each output element in the feature map.
- Each thread computes the gradient of the preactivation value for its corresponding output element, based on the output gradient and the derivative of the activation function.

```
__global__ void bp_preact_c1(float (*d_preact)[24][24], float (*d_output)[24][24],
    float (*preact)[24][24])
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;
    int z = bz;

    d_preact[z][y][x] = d_output[z][y][x] * (preact[z][y][x] > 0 ? 1.0f : 0.0f);
}
```

Backpropagation for Weights (bp_weight_c1)

- **Parallelization Strategy:** Input Stationarity
- The kernel is launched with a grid of $numBlocks\_weight\_c1$ blocks, where each block corresponds to one filter.
- Within each block, there are $threadsPerBlock\_weight\_c1$ threads, one for each weight in the filter.
- Each thread iterates over the corresponding input elements and preactivation gradients to compute the gradient for its weight.

```
__global__ void bp_weight_c1(float (*d_weight)[5][5], float (*d_preact)[24][24], float
    (*input)[28])
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;

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

    float sum = 0.0f;
    for (int i = 0; i < 24; ++i) {
        for (int j = 0; j < 24; ++j) {
            sum += d_preact[z][i][j] * input[i + x][j + y];
        }
    }
    d_weight[z][y][x] = sum;
}
```

Backpropagation for Biases (bp_bias_c1)

- **Parallelization Strategy:** One Block per Feature Map
- The kernel is launched with $blocks\_bias\_c1$ blocks, where each block corresponds to one output feature map.
- Within each block, there are $threads\_bias\_c1$ threads, which cooperatively compute the sum of the preactivation gradients for the corresponding feature map.

- The sum is then used to update the bias gradient for that feature map.

```
__global__ void bp_bias_c1(float *bias, float (*d_preact)[24][24])
{
    int bx = blockIdx.x;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    __shared__ float sum[16][16];
    sum[ty][tx] = 0.0f;
    __syncthreads();

    for (int i = 0; i < 24; ++i) {
        for (int j = 0; j < 24; ++j) {
            sum[ty][tx] += d_preact[bx][i][j];
        }
    }
    __syncthreads();

    if (tx == 0 && ty == 0) {
        float tmp = 0.0f;
        for (int i = 0; i < 16; ++i) {
            for (int j = 0; j < 16; ++j) {
                tmp += sum[i][j];
            }
        }
        bias[bx] = tmp;
    }
}
```

The code snippet above shows the implementation of the 'bp bias c1' kernel, which is responsible for computing the gradients of the biases with respect to the preactivation values in the convolutional layer.

The parallelization strategy employed here is to launch one block per output feature map, with each block containing multiple threads that cooperatively compute the sum of the preactivation gradients for that feature map. This sum is then used to update the bias gradient for that feature map.

Here's how the kernel works:

1. Each thread within a block initializes its portion of the shared memory array `sum` to zero.
2. The threads then cooperatively compute the sum of the preactivation gradients for their corresponding feature map by iterating over the preactivation gradients and accumulating the values in the shared memory array `sum`.
3. After all threads have finished computing their partial sums, a synchronization barrier is used to ensure that all threads have completed their computations before proceeding to the next step.
4. A single thread (with `threadIdx.x == 0` and `threadIdx.y == 0`) is responsible for summing up all the partial sums stored in the shared memory array `sum` and storing the final sum in the corresponding bias gradient element `bias[bx]`.

.

4.3.2 Pooling layer

Forward Propagation (fp_s1)

- **Parallelization Strategy:** Output Stationarity
- The kernel is launched with a grid of $configSubsample1.blocks$ blocks, where each block corresponds to one output feature map.
- Within each block, there are $configSubsample1.threads$ threads, one for each output element in the feature map.

- Each thread computes the maximum value within a 2×2 window of the input feature map, using the corresponding filter weights and bias.
- This approach allows for efficient memory access patterns and coalesced memory loads.

```
__global__ void fp_s1(float (*input)[24][24], float (*preact)[6][6], float (*weight)
    [4][4], float *bias)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;
    int z = bz;

    float max_val = -INFINITY;
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            float val = input[z][2 * y + i][2 * x + j] * weight[z][i][j];
            max_val = fmax(max_val, val);
        }
    }
    preact[z][y][x] = max_val + bias[z];
}
```

Applying Activation Function (*apply_step_function*)

- **Parallelization Strategy:** Same as fp_s1
- The kernel is launched with the same configuration as fp_s1, i.e., $configSubsample1.blocks$ and $configSubsample1.threads$.
- Each thread applies the activation function to its corresponding preactivation value.

```
__global__ void apply_step_function(float (*preact)[6][6], float (*output)[6][6], int
    O)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int x = bx * blockDim.x + tx;
    int y = by * blockDim.y + ty;
    int z = bz;

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

Backpropagation for Output (*bp_output_s1*)

- **Parallelization Strategy:** Input Stationarity
- The kernel is launched with a grid of 5 blocks, where each block processes a tile of the input feature map.
- Within each block, there are $(216 + 5 - 1)/5$ threads, each responsible for computing the gradient for one output element.
- Each thread iterates over the corresponding input elements and filter weights to compute the gradient for its output element.

- This approach allows for efficient memory access patterns and coalesced memory loads/stores.

```
__global__ void bp_output_s1(float (*d_output)[6][6], float (*weight)[6][6][6], float
    (*d_preact)[6][6])
{
    int tx = threadIdx.x;
    int bx = blockIdx.x;

    int x = bx * blockDim.x + tx;
    int y, z;

    float sum = 0.0f;
    for (z = 0; z < 6; ++z) {
        for (y = 0; y < 6; ++y) {
            sum += weight[z][y][x] * d_preact[z][y][x];
        }
    }
    d_output[0][0][x] = sum;
}
```

4.3.3 FC layer

Forward Propagation (fp_f)

- **Parallelization Strategy:** Output Stationarity
- The kernel is launched with a grid of $configFullyConnected.blocks$ blocks, where each block corresponds to one output neuron.
- Within each block, there are $configFullyConnected.threads$ threads, which collaboratively compute the dot product between the input and the corresponding weight vector.
- Each thread computes a partial sum of the dot product, and the results are accumulated in shared memory.
- The final dot product is then computed and the bias is added to produce the output.
- This approach allows for efficient memory access patterns and parallelism within each output neuron.

```
__global__ void fp_f(float (*input)[6][6], float *preact, float (*weight)[6][6][6],
    float *bias)
{
    int tx = threadIdx.x;
    int bx = blockIdx.x;

    __shared__ float sum[256];
    sum[tx] = 0.0f;
    __syncthreads();

    for (int i = 0; i < 6; ++i) {
        for (int j = 0; j < 6; ++j) {
            for (int k = 0; k < 6; ++k) {
                sum[tx] += input[bx][i][j] * weight[bx][i][j][k];
            }
        }
    }
    __syncthreads();

    float tmp = 0.0f;
    for (int i = 0; i < 256; ++i) {
        tmp += sum[i];
    }
    preact[bx] = tmp + bias[bx];
}
```

Applying Activation Function ($apply\_step\_function$)

- **Parallelization Strategy:** One Block, One Thread per Output Neuron
- The kernel is launched with one block and 10 threads, where each thread applies the activation function to one output neuron.
- This approach is suitable for the small number of output neurons in the fully connected layer.

```
__global__ void apply_step_function(float *preact, float *output, int 0)
{
    int tx = threadIdx.x;
    output[tx] = preact[tx] > 0 ? 1.0f : 0.0f;
}
```

Backpropagation (bp_f)

- **Parallelization Strategy:** Output Stationarity
- The kernel is launched with a grid of $gridSize$ blocks, where each block corresponds to one output neuron.
- Within each block, there are $blockSize$ threads, which collaboratively compute the gradients of the weights and biases.
- Each thread computes a partial sum of the gradients, and the results are accumulated in shared memory.
- The final gradients are then computed and stored in global memory.
- This approach allows for efficient memory access patterns and parallelism within each output neuron.

```
__global__ void bp_f(float (*d_weight)[6][6][6], float *d_bias, float *d_preact, float
    (*input)[6][6])
{
    int tx = threadIdx.x;
    int bx = blockIdx.x;

    __shared__ float sum_w[256];
    sum_w[tx] = 0.0f;
    __syncthreads();

    for (int i = 0; i < 6; ++i) {
        for (int j = 0; j < 6; ++j) {
            for (int k = 0; k < 6; ++k) {
                sum_w[tx] += d_preact[bx] * input[bx][i][j];
                d_weight[bx][i][j][k] = sum_w[tx];
            }
        }
    }
    __syncthreads();

    float tmp = 0.0f;
    for (int i = 0; i < 256; ++i) {
        tmp += d_preact[bx] * sum_w[i];
    }
    d_bias[bx] = tmp;
}
```

5 Parallel Techniques

- **Data Partitioning:** The computations in the convolutional layers and the fully connected layer are partitioned across different blocks and threads based on the output elements (feature maps or neurons). Each thread or block is responsible for computing a subset of the output elements, effectively partitioning the data across the GPU's processing units.

  Example: In the forward propagation kernel fp_c1, the computation of each output feature map is assigned to a separate block, effectively partitioning the data across blocks.

- **Divide-and-Conquer:** Several kernels employ a divide-and-conquer approach, where the computations are divided into smaller sub-problems and assigned to different threads or blocks, which can work independently on their respective sub-problems. The final result is then obtained by combining the partial results from the threads or blocks.

  Examples: - In the fp_f kernel (forward propagation of the fully connected layer), the dot product computation for each output neuron is divided among multiple threads within a block, with each thread computing a partial sum, which is then accumulated in shared memory to obtain the final dot product. - In the bp_bias_c1 kernel (backpropagation of biases for the convolutional layer), the computation of the sum of preactivation gradients for each feature map is divided among multiple threads within a block, using shared memory to accumulate the partial sums.

- **Embarrassingly Parallel Computations:** Some computations in the CNN implementation can be considered embarrassingly parallel, where the sub-problems can be computed independently without any data dependencies or communication between threads or blocks.

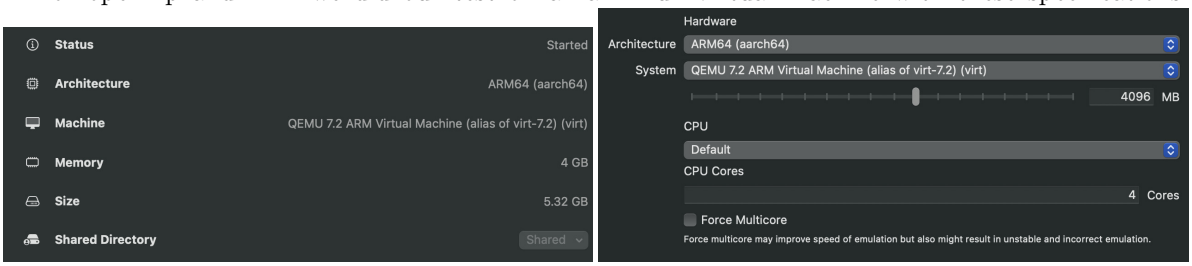
  Example: The application of the activation function (e.g., $apply\_step\_function$) can be considered an embarrassingly parallel computation, as each thread can independently apply the activation function to its corresponding preactivation value without any data dependencies.

6 Results

Sequential time	Time (s)
Total Time	102.317095

Table 1: Sequential Time Results for CNN Implementation per epoch

for openmp and MPI we did our test on a kali linux virtual machine with these specifications



6.1 MPI

Table 2: Speedup and Efficiency for Different Number of Cores

Num Cores	Speedup	Efficiency (%)
2	1.52734	0.76367
4	1.01832	0.3545

6.2 OpenMP

Table 3: Speedup and Efficiency for Different Number of Threads

Num Threads	Speedup	Efficiency (%)
2	2.72047	1.36
4	2.62118	0.655295
6	2.158702	0.359783
8	Machine would crash	Machine would crash

6.3 CUDA C

We ran all of our test using the t4 GPU on Google Colab I begin by testing each individual layer I do

Layer	Time (ms)
Convolution	97699.512123
Pooling	5127.317961
Fully Connected	2129.152015

Table 4: Sequential Time Results for CNN Implementation per epoch

this by modifying the CNN to only have one of these 3 layers and recording the time for each :

6.3.1 Convolutional layer

Table 5: Performance

Number of threads	Time(ms)	Speed up s(p)	Efficiency
3456	90.173	1083.467	0.3135

6.3.2 Pooling layer

Table 6: Performance

Number of threads	Time(ms)	Speed up s(p)	Efficiency
3456	5.1927	986.73	0.285512

6.3.3 FC layer

Table 7: Performance

Number of threads	Time(ms)	Speed up s(p)	Efficiency
5120	0.386624	5507.03529	1.07559283

6.3.4 Entire network

Table 8: Performance

Number of threads	Time(ms)	Speed up s(p)	Efficiency
27,776	2996.9857	34.14	1.229118664 x$10^{-}3$

Our experimental results demonstrated a significant speedup of 34.14 when parallelizing the CNN training with CUDA C compared to a sequential implementation. However, upon closer analysis, we observed that the parallelized approach suffered from overhead due to the simultaneous launch of a large number of kernels. This overhead led to suboptimal performance, particularly when compared to testing each layer individually. The overhead introduced by kernel launching overshadowed the benefits of parallel execution, resulting in a lower overall performance.The observed overhead in the parallelized CNN training can be attributed to several factors, including kernel launch latency, memory transfers between the host and device, and synchronization overhead. The simultaneous execution of multiple kernels exacerbated these overheads, leading to diminished performance gains

7 Discussion

7.1 MPI Implementation

The Message Passing Interface (MPI) implementation showcases the utilization of data partitioning but suffers from limited scalability. The speedup and efficiency results for varying core counts indicate some performance gains, albeit with significant drops in efficiency as more cores are utilized. This reflects the high communication overhead of MPI, which may not be ideal for tasks where results from each layer need extensive sharing across threads, as evidenced by the diminishing returns on speedup when increasing the core count.

7.2 OpenMP Implementation

OpenMP provides an easier method to apply parallelization directives to computationally heavy loops. However, the results also show that increasing the number of threads beyond a certain threshold leads to diminishing returns, and even system instability. This suggests that while OpenMP is suitable for shared-memory systems, it struggles with the overhead of thread management and synchronization, particularly when thread counts exceed the optimal balance.

7.3 CUDA C Implementation

The CUDA C implementation highlights the significant potential of GPU-based parallel processing for CNNs. Detailed performance analysis for individual layers (convolutional, pooling, and fully-connected) illustrates substantial speedups, particularly due to the fine-grained parallelism achievable with CUDA. However, when scaling to the full network, some performance inefficiencies due to kernel launch overhead and synchronization were observed, suggesting areas for further optimization.

8 Conclusion

This study of parallelizing CNN training using MPI, OpenMP, and CUDA C demonstrates varied results influenced by the characteristics of each platform and the specific computational demands of neural networks. MPI is constrained by high communication costs, OpenMP by thread management overhead past a certain parallelism level, and CUDA C shows robust performance in handling intensive operations with the need for better overhead management in full network training scenarios.

Future work should consider hybrid approaches that combine these technologies to offset their individual limitations. Further optimization of CUDA C implementations could also enhance performance, particularly in managing kernel execution and minimizing synchronization overhead. Exploring new architectures and parallel processing frameworks may provide additional improvements in performance and efficiency, crucial for advancing CNN applications. LINK TO THE GITHUB REPO

References

- Parallel medical image registration model based on convolution neural network and Transformer.
- Communication-Efficient Parallelization Strategy for Deep Convolutional Neural Network Training
- Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication
- Hwu, W.-m. W., Kirk, D. B., El Hajj, I. (2023). Programming massively parallel processors: A hands-on approach (3rd ed.). Morgan Kaufmann.