LAU
الجامعة اللبنانية الأميركية
Lebanese American University

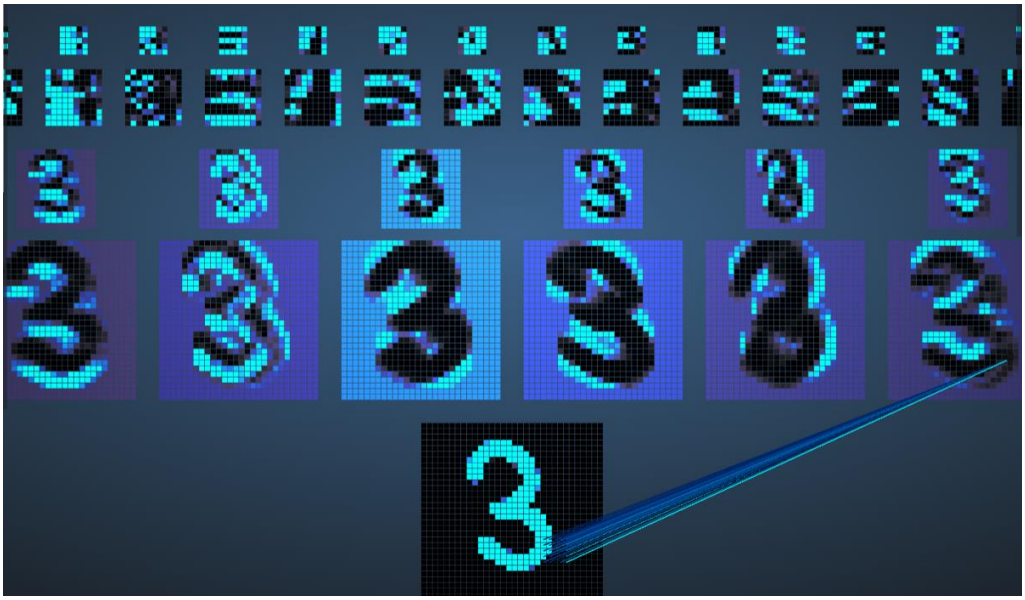
School of
Arts and Sciences

Final Project Report

Parallelizing Convolutional Neural Network (CNN) for image classification

LeNet-5 architecture

Digit Classification



Abdulaziz Al Sayyed 202204622
Ibrahim Mabrouki 202206306
Tala Hareb 202202402

LAU Lebanese American University | School of Arts and Sciences

Overview of CNNs

What is a convolutional neural network?

A *Convolutional Neural Network (CNN)* is a specialized type of deep learning model designed for processing visual and spatial data. It excels at tasks like image recognition, object detection, and video analysis by learning hierarchical features directly from the input data. CNNs consist of key layers: *convolutional layers* for feature extraction, *pooling layers* for dimensionality reduction, and *fully connected layers* for classification. They leverage local connectivity and parameter sharing to efficiently process high-dimensional data while capturing spatial patterns like edges, textures, and shapes. Widely used in fields such as computer vision, autonomous systems, and medical imaging, CNNs have revolutionized how machines interpret visual information but require substantial data and computational resources to perform effectively.

In our study, we focus on *digit recognition*, utilizing CNNs to accurately classify handwritten digits by extracting and analyzing their unique spatial features.

Why the need to paralyze it?

Training deep CNNs on large datasets is highly computing-intensive and time-consuming, primarily due to the sequential nature of the algorithm. It relies on iteratively updating parameters using mini batches, which introduces dependencies that make parallelization challenging.

In our study, we aim to tackle this by parallelizing CNN training using *MPI, OpenMP, and CUDA C*, evaluating each method execution time, performance and accuracy.

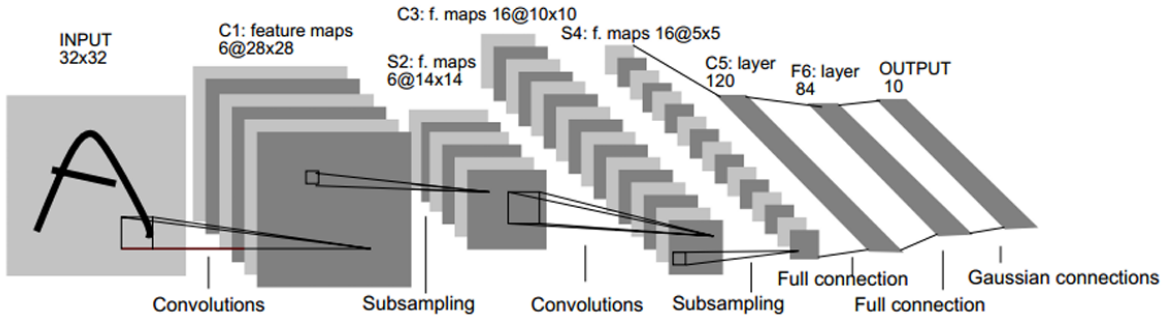
What are the different architectures?

Feature	*LeNet-5*	*AlexNet*	*VGG*	*ResNet*
Year	1998	2012	2014	2015
Depth	7 layers	8 layers	16-19 layers	50-152+ layers
Input Size	32 × 32	227 × 227	224 × 224	224 × 224
Filter Size	5 × 5	11 × 11 (1st layer)	3 × 3	3 × 3
Pooling	Average	Max	Max	Max, Global Avg
Activation	Tanh	ReLU	ReLU	ReLU

In our study, we chose *LeNet-5* for its simplicity allows us to focus on optimizing the core structure with minimal computational overhead. These optimizations serve as a foundation for improving more complex architectures, ensuring broader applicability and impact.

Bonus : Though we used LeNet-5, we will see later that incorporating max pooling and ReLU helped us advance and enhance its performance.

LetNet-5 architecture



Header file

```
#define LENGTH_KERNEL   5

#define LENGTH_FEATURE0 32
#define LENGTH_FEATURE1 (LENGTH_FEATURE0 - LENGTH_KERNEL + 1)
#define LENGTH_FEATURE2 (LENGTH_FEATURE1 >> 1)
#define LENGTH_FEATURE3 (LENGTH_FEATURE2 - LENGTH_KERNEL + 1)
#define LENGTH_FEATURE4 (LENGTH_FEATURE3 >> 1)
#define LENGTH_FEATURE5 (LENGTH_FEATURE4 - LENGTH_KERNEL + 1)

#define INPUT           1
#define LAYER1          6
#define LAYER2          6
#define LAYER3          16
#define LAYER4          16
#define LAYER5          120
#define OUTPUT          10
```

Github source : https://github.com/AbdulazizAlSayyed/LeNet5_CNN.git

Serial Code

1. Convolutional Layer

The convolutional layer applies filters (kernels) to extract features from the input. This includes the forward and backward convolution operations.

Forward Convolution

```
#define CONVOLUTE_VALID(input,output,weight) \
{ \
    FOREACH(o0,GETLENGTH(output)) \
        FOREACH(o1,GETLENGTH(*(output))) \
            FOREACH(w0,GETLENGTH(weight)) \
                FOREACH(w1,GETLENGTH(*(weight))) \
                    (output)[o0][o1] += (input)[o0 + w0][o1 + w1] *
(weight)[w0][w1]; \
}

#define CONVOLUTION_FORWARD(input,output,weight,bias,action) \
{ \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            CONVOLUTE_VALID(input[x], output[y], weight[x][y]); \
    FOREACH(j, GETLENGTH(output)) \
        FOREACH(i, GETCOUNT(output[j])) \
            ((double *)output[j])[i] = action(((double *)output[j])[i] +
bias[j]); \
}
```

Backward Convolution

```
#define CONVOLUTE_FULL(input,output,weight) \
{ \
    FOREACH(i0,GETLENGTH(input)) \
        FOREACH(i1,GETLENGTH(*(input))) \
            FOREACH(w0,GETLENGTH(weight)) \
                FOREACH(w1,GETLENGTH(*(weight))) \
```

```
                    (output)[i0 + w0][i1 + w1] += (input)[i0][i1] *
(weight)[w0][w1]; \
}

#define CONVOLUTION_BACKWARD(input,inerror,outerror,weight,wd,bd,actiongrad) \
{ \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            CONVOLUTE_FULL(outerror[y], inerror[x], weight[x][y]); \
    FOREACH(i, GETCOUNT(inerror)) \
        ((double *)inerror)[i] *= actiongrad(((double *)input)[i]); \
    FOREACH(j, GETLENGTH(outerror)) \
        FOREACH(i, GETCOUNT(outerror[j])) \
            bd[j] += ((double *)outerror[j])[i]; \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            CONVOLUTE_VALID(input[x], wd[x][y], outerror[y]); \
}
```

---

2. Pooling Layer

We used max pooling instead of average pooling in our CNN because it retains the most significant features by selecting the strongest activations, which helps the model focus on key patterns and reduces the risk of losing important information during down sampling.

Forward Pooling

```
#define SUBSAMP_MAX_FORWARD(input,output) \
{ \
    const int len0 = GETLENGTH(*(input)) / GETLENGTH(*(output)); \
    const int len1 = GETLENGTH(**(input)) / GETLENGTH(**(output)); \
    FOREACH(i, GETLENGTH(output)) \
    FOREACH(o0, GETLENGTH(*(output))) \
    FOREACH(o1, GETLENGTH(**(output))) \
    { \
        int x0 = 0, x1 = 0, ismax; \
        FOREACH(l0, len0) \
            FOREACH(l1, len1) \
        { \
            ismax = input[i][o0*len0 + l0][o1*len1 + l1] > input[i][o0*len0 +
x0][o1*len1 + x1]; \
            x0 += ismax * (l0 - x0); \
```

```
            x1 += ismax * (l1 - x1); \
        } \
        output[i][o0][o1] = input[i][o0*len0 + x0][o1*len1 + x1]; \
    } \
}
```

Backward Pooling

```
#define SUBSAMP_MAX_BACKWARD(input,inerror,outerror) \
{ \
    const int len0 = GETLENGTH(*(inerror)) / GETLENGTH(*(outerror)); \
    const int len1 = GETLENGTH(**(inerror)) / GETLENGTH(**(outerror)); \
    FOREACH(i, GETLENGTH(outerror)) \
    FOREACH(o0, GETLENGTH(*(outerror))) \
    FOREACH(o1, GETLENGTH(**(outerror))) \
    { \
        int x0 = 0, x1 = 0, ismax; \
        FOREACH(l0, len0) \
            FOREACH(l1, len1) \
        { \
            ismax = input[i][o0*len0 + l0][o1*len1 + l1] > input[i][o0*len0 + 
x0][o1*len1 + x1]; \
            x0 += ismax * (l0 - x0); \
            x1 += ismax * (l1 - x1); \
        } \
        inerror[i][o0*len0 + x0][o1*len1 + x1] = outerror[i][o0][o1]; \
    } \
}
```

---

3. Fully Connected Layer

The fully connected layer connects the flattened features to the output and computes the final classification scores.

Forward Fully Connected

```
#define DOT_PRODUCT_FORWARD(input,output,weight,bias,action) \
{ \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            ((double *)output)[y] += ((double *)input)[x] * weight[x][y]; \
    FOREACH(j, GETLENGTH(bias)) \
```

```
        ((double *)output)[j] = action(((double *)output)[j] + bias[j]); \
}
```

Backward Fully Connected

```
#define DOT_PRODUCT_BACKWARD(input,inerror,outerror,weight,wd,bd,actiongrad) \
{ \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            ((double *)inerror)[x] += ((double *)outerror)[y] * weight[x][y]; \
    FOREACH(i, GETCOUNT(inerror)) \
        ((double *)inerror)[i] *= actiongrad(((double *)input)[i]); \
    FOREACH(j, GETLENGTH(outerror)) \
        bd[j] += ((double *)outerror)[j]; \
    for (int x = 0; x < GETLENGTH(weight); ++x) \
        for (int y = 0; y < GETLENGTH(*weight); ++y) \
            wd[x][y] += ((double *)input)[x] * ((double *)outerror)[y]; \
}
```

---

3. The Other Complementary functions

Utility Macros and Functions

1. **GETLENGTH**: Computes the number of rows in a multi-dimensional array.
2. **GETCOUNT**: Computes the total number of elements in a 1D array.
3. **FOREACH**: A macro for iterating through array indices.
4. **relu**: Implements the ReLU activation function, returning the positive part of the input.
5. **relugrad**: Computes the gradient of the ReLU activation function.

Input Handling

6. **load_input**: Preprocesses and normalizes the input image, applying mean subtraction and standardization.

Output Handling

7. **softmax**: Computes the softmax probabilities for the output layer to normalize predictions.
8. **load_target**: Loads the correct label and computes the loss/error using softmax.
9. **get_result**: Determines the class label with the highest probability from the output.

Training

10. **TrainBatch**: Performs batch training by aggregating weight updates over multiple inputs and labels.
11. **Train**: Handles single-sample training, updating weights for one input-label pair.

Prediction

12. **Predict**: Uses the forward pass to predict the class of a given input image.

Initialization

13. **Initial**: Initializes weights and biases in the network using random values and proper scaling.

Execution Time Calculation

With average pooling and tanh:

```
6940/10000
Precision is 0.694000
Traing CPU time:134367958
Traing CPU time with second:134.367952s
Test CPU time:8040294
Test CPU time with second:8.040294s
Training time is 134.368543s
Testing time is 8.040310s
```

With max pooling and ReLU :

```
9708/10000
Precision is 0.970800
Traing CPU time:111317
Traing CPU time with second:111.317000s
Test CPU time:5717
Test CPU time with second:5.717000s
```

Max pooling and ReLU perform better because max pooling preserves critical features, while ReLU avoids vanishing gradients, leading to higher precision and faster execution. On the other hand, tanh dilutes important features and suffers from vanishing gradients, *so we optimized our LeNet5 architecture with max pooling and ReLU.*

LAU Lebanese American University | School of Arts and Sciences

OpenMP

How was OpenMP Parallelized?

1. Optimization on Batch Processing

Update: **TrainBatch** function and the training function (from main)

Goal: Training models often involves processing large datasets in smaller "batches" for memory efficiency. Since each batch can be processed independently, parallelizing the batch operations can significantly improve overall training speed.

Process:
Unrolling loops: Reduces the time spent in memory translation by optimizing the way data is fetched. By unrolling the loop to handle more iterations per pass, it reduces the repetitive overhead of loop condition checks.

```
void training(LeNet5 *lenet, image *train_data, uint8 *train_label, int
batch_size, int total_size)
{
    #pragma unroll(10)
    for (int i = 0, percent = 0; i <= total_size - batch_size; i += batch_size)
    {
        TrainBatch(lenet, train_data + i, train_label + i, batch_size);
        if (i * 100 / total_size > percent)
            printf("batchsize:%d\ttrain:%2d%%\n", batch_size, percent = i * 100 /
total_size);
    }
}
```

Parallel for loops: Each batch is independent, making it ideal for parallel processing. This reduces time spent per batch. It splits the computation of each batch among multiple threads.
Delta update handling: During gradient accumulation, multiple threads can try to update the same variable, causing race conditions. critical is generally faster than atomic for complex operations.
Parameter updates: Utilizes a separate parallelization to speed up the parameter updates. Updating model weights is computationally expensive and can be parallelized to speed up training. parallelizing updates reduces bottlenecks during weight adjustments.

```
void TrainBatch(LeNet5 *lenet, image *inputs, uint8 *labels, int batchSize)
{
```

```
    double buffer[GETCOUNT(LeNet5)] = { 0 };
    int i = 0;
    int num =  (sizeof(LeNet5)/sizeof(double));

    #pragma omp parallel for
    for (i = 0; i < batchSize; ++i)
    {
        Feature features = { 0 };
        Feature errors = { 0 };
        LeNet5  deltas = { 0 };
        load_input(&features, inputs[i]);
        forward(lenet, &features, relu);
        load_target(&features, &errors, labels[i]);
        backward(lenet, &deltas, &errors, &features, relugrad);
        #pragma omp critical
        for (int j = 0; j < num; ++j)
            buffer[j] += ((double *)&deltas)[j];
    }
    double k = ALPHA / batchSize;

    #pragma omp parallel for
    FOREACH(i, num)
        ((double *)lenet)[i] += k * buffer[i];
}
```

2. Optimization on Input Data Normalization

Update: **load_input** function

Goal: Normalizing input data (mean-centering and standard deviation scaling) is a critical preprocessing step to stabilize training and ensure faster convergence. This can be computationally heavy, especially with large datasets.

Process:
Parallel for with reduction: The computation of mean and standard deviation requires summing across all input values. Reduction allows multiple threads to compute partial sums and combine results safely.
Zero padding: After normalization, images are padded with a zero boundary to generate 32x32 normalized output data. Many convolutional networks require inputs of fixed dimensions. Padding with zeros maintains the input dimensions without introducing bias.

```
static inline void load_input(Feature *features, image input)
{
```

```
    double (*layer0)[LENGTH_FEATURE0][LENGTH_FEATURE0] = features->input;

    const long sz = sizeof(image) / sizeof(**input);
    double mean = 0, std = 0;

    #pragma omp parallel for reduction(+:mean) reduction(+:std)
    FOREACH(j, sizeof(image) / sizeof(*input))
        FOREACH(k, sizeof(*input) / sizeof(**input))
        {
            mean = mean + input[j][k];
            std  = std + (input[j][k] * input[j][k]);
        }
    mean /= sz;
    std = sqrt(std / sz - mean*mean);

    #pragma omp parallel for
    FOREACH(j, sizeof(image) / sizeof(*input))
        FOREACH(k, sizeof(*input) / sizeof(**input))
        {
            layer0[0][j + PADDING][k + PADDING] = (input[j][k] - mean) / std;
        }
}
```

3. Optimization on Forward Propagation

Update: **forward** function and testing function in main

Goal: Speed up the heavy computations of feature maps, convolution, and max subsampling pooling.

Process:
Parallel for: Each feature map in a layer can be computed independently. Parallelizing this reduces the time needed for convolutions and pooling operations. OpenMP distributes the workload of computing feature maps across threads.

Function optimization: Replaces certain expensive functions (e.g., GETCOUNT) with predefined constants to reduce overhead.

```
static void forward(LeNet5 *lenet, Feature *features, double(*action)(double))
{
    // CONVOLUTION_FORWARD(features->input, features->layer1, lenet->weight0_1,
lenet->bias0_1, action);
```

```
    #pragma omp parallel
for

    for (int x = 0; x < 1; ++x)
        for (int y = 0; y < 6; ++y)
            CONVOLUTE_VALID(features->input[x], features->layer1[y], lenet-
>weight0_1[x][y]);

    #pragma omp parallel
for

    FOREACH(j, 6)
        FOREACH(i, 784)
            ((double *)features->layer1[j])[i] = action(((double *)features-
>layer1[j])[i] + lenet->bias0_1[j]);

    SUBSAMP_MAX_FORWARD(features->layer1, features->layer2);
    #pragma omp parallel for
    for (int x = 0; x < 6; ++x)
        for (int y = 0; y < 16; ++y)
            CONVOLUTE_VALID(features->layer2[x], features->layer3[y], lenet-
>weight2_3[x][y]);

    #pragma omp parallel
for

    FOREACH(j, 16)
        FOREACH(i, 100)
            ((double *)features->layer3[j])[i] = action(((double *)features-
>layer3[j])[i] + lenet->bias2_3[j]);


    SUBSAMP_MAX_FORWARD(features->layer3, features->layer4);
    #pragma omp parallel for
    for (int x = 0; x < 16; ++x)
        for (int y = 0; y < 120; ++y)
            CONVOLUTE_VALID(features->layer4[x], features->layer5[y], lenet-
>weight4_5[x][y]);

    #pragma omp parallel
for

    FOREACH(j, 120)
```

```
        FOREACH(i, 1)
            ((double *)features->layer5[j])[i] = action(((double *)features-
>layer5[j])[i] + lenet->bias4_5[j]);

    DOT_PRODUCT_FORWARD(features->layer5, features->output, lenet->weight5_6,
lenet->bias5_6, action);
}
```

parallel for loop: distributes the testing workload (predicting for each input) across multiple threads using OpenMP.
reduction(+:right): clause accumulates the count of correct predictions (right), ensuring thread-safe updates.

```
int testing(LeNet5 *lenet, image *test_data, uint8 *test_label,int total_size)
{
    int right = 0, percent = 0;

    #pragma omp parallel for reduction(+:right)
    for (int i = 0; i < total_size; ++i)
    {
        uint8 l = test_label[i];
        int p = Predict(lenet, test_data[i], 10);
        right = right + (l == p);
        if (i * 100 / total_size > percent)
            printf("test:%2d%%\n", percent = i * 100 / total_size);
    }
    return right;
}
```

4. Optimization on Softmax

Update : softmax function

Goal: Parallelize the calculation of output scores (10 classes) and error computation since these operations are independent. Optimizes loss computation with independent score updates.

Process:
Parallel for: Speeds up score calculation for each class.

```
static inline void softmax(double input[OUTPUT], double loss[OUTPUT], int label,
int count)
{
    double inner = 0;
```

```
    #pragma omp parallel for reduction(-:inner)
    for (int i = 0; i < count; ++i)
    {
        double res = 0;
        for (int j = 0; j < count; ++j)
        {
             res += exp(input[j] - input[i]);
        }
        loss[i] = 1. / res;
       //#pragma omp atomic : The #pragma omp atomic directive is not necessary
//here for updating inner because the #pragma omp parallel for reduction(-:inner)
//already handles the accumulation safely
        inner = inner - (loss[i] * loss[i]);
    }
    inner += loss[label];

   #pragma omp parallel for
    for (int i = 0; i < count; ++i)
    {
        loss[i] *= (i == label) - loss[i] - inner;
    }
}
```

5. Optimization on Backpropagation

Update: backward function

Goal: Accelerate the backpropagation phase by parallelizing loss computation. Parallelizes gradient computation and memory-coalesced updates.

Process:
Parallel for: Optimizes memory access patterns during the accumulation of gradients, reducing overhead caused by separate accesses.

```
static void backward(LeNet5 *lenet, LeNet5 *deltas, Feature *errors, Feature
*features, double(*actiongrad)(double))
{
    DOT_PRODUCT_BACKWARD(features->layer5, errors->layer5, errors->output, lenet-
>weight5_6, deltas->weight5_6, deltas->bias5_6, actiongrad);

    #pragma omp parallel for
    for(int x = 0; x < GETLENGTH(lenet->weight4_5); ++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight4_5)); ++y)
```

```
            CONVOLUTE_FULL(errors->layer5[y], errors->layer4[x], lenet-
>weight4_5[x][y]);

    #pragma omp parallel for
    FOREACH(i, GETCOUNT(errors-
>layer4))
        ((double *)errors->layer4)[i] *= actiongrad(((double *)features-
>layer4)[i]);

    #pragma omp parallel for
    FOREACH(j, GETLENGTH(errors-
>layer5))
        FOREACH(i, GETCOUNT(errors->layer5[j]))
            deltas->bias4_5[j] += ((double *)errors->layer5[j])[i];

    #pragma omp parallel for
    for (int x = 0; x < GETLENGTH(lenet->weight4_5);
++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight4_5));
++y)
            CONVOLUTE_VALID(features->layer4[x], deltas->weight4_5[x][y], errors-
>layer5[y]);

    SUBSAMP_MAX_BACKWARD(features->layer3, errors->layer3, errors->layer4);

    #pragma omp parallel for
    for(int x = 0; x < GETLENGTH(lenet->weight2_3); ++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight2_3)); ++y)
            CONVOLUTE_FULL(errors->layer3[y], errors->layer2[x], lenet-
>weight2_3[x][y]);

    #pragma omp parallel for
    FOREACH(i, GETCOUNT(errors-
>layer2))
        ((double *)errors->layer2)[i] *= actiongrad(((double *)features-
>layer2)[i]);

    #pragma omp parallel for
    FOREACH(j, GETLENGTH(errors-
>layer3))
        FOREACH(i, GETCOUNT(errors->layer3[j]))
            deltas->bias2_3[j] += ((double *)errors->layer3[j])[i];
```

```
    #pragma omp parallel for
    for (int x = 0; x < GETLENGTH(lenet->weight2_3);
++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight2_3));
++y)
            CONVOLUTE_VALID(features->layer2[x], deltas->weight2_3[x][y], errors-
>layer3[y]);

    SUBSAMP_MAX_BACKWARD(features->layer1, errors->layer1, errors->layer2);

    #pragma omp parallel for
    for(int x = 0; x < GETLENGTH(lenet->weight0_1); ++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight0_1)); ++y)
            CONVOLUTE_FULL(errors->layer1[y], errors->input[x], lenet-
>weight0_1[x][y]);

    #pragma omp parallel for
    FOREACH(i, GETCOUNT(errors->input))
        ((double *)errors->input)[i] *= actiongrad(((double *)features-
>input)[i]);

    #pragma omp parallel for
    FOREACH(j, GETLENGTH(errors-
>layer1))
        FOREACH(i, GETCOUNT(errors->layer1[j]))
            deltas->bias0_1[j] += ((double *)errors->layer1[j])[i];

    #pragma omp parallel for
    for (int x = 0; x < GETLENGTH(lenet->weight0_1);
++x)
        for (int y = 0; y < GETLENGTH(*(lenet->weight0_1));
++y)
            CONVOLUTE_VALID(features->input[x], deltas->weight0_1[x][y], errors-
>layer1[y]);

    SUBSAMP_MAX_BACKWARD(features->layer1, errors->layer1, errors->layer2);
}
```

Execution Time Calculation

With average pooling and tanh:

```
6940/10000
Precision is 0.694000
Training time is 31.419905s
Testing time is 1.840289s
```

With max pooling and reLU:

```
9706/10000
Precision is 0.970600
Training time is 18.777750s
Testing time is 1.166425s
```

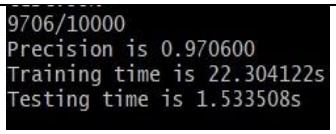
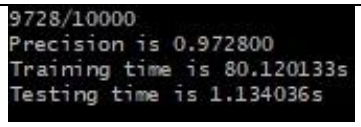
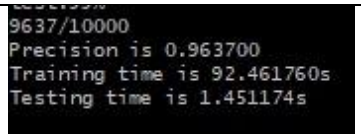
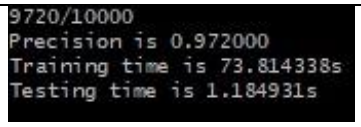
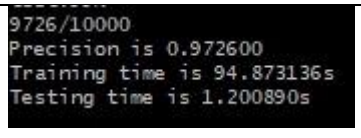
For further analysis and verification, we re-evaluated both techniques on the OMP, and once again, the combination of ReLU and max pooling demonstrated superior performance. As a result, we will be considering this enhanced approach in our analysis.

LAU Lebanese American University | School of Arts and Sciences

OpenMP optimization analysis

A. Calculations

S(p) = Sequential time / Parallel time

Parallelization Technique	Effect on Outcome	Observations	Output	Speed-Up training / Speed-Up testing
Batch Processing (TrainBatch)	Significant speed-up in training phase	Parallelizing batch operations drastically reduced time per batch.		4.99x 3.72x
Input Data Normalization (load_input)	Significant speed-up in testing time. Faster preprocessing with better data consistency	Parallelized mean and std calculation; reduced preprocessing bottlenecks.		1.38x 5.04x
Forward Propagation (forward)	Improvement in feature map computations.	Parallelized convolutions and pooling sped up forward pass.		1.20x 3.93x
Softmax Calculation (softmax)	Improved output and error computation times	Parallelizing score calculations led to faster convergence		1.50x 4.82x
Backpropagation (backward)	Efficient gradient updates and loss propagation	Parallelized gradient computation and memory-coalesced updates reduced overhead.		1.17x 4,76x

Total (while running all the parallel techniques):

Process	Number of Samples	Sequential (s)	OpenMP (s)	Speed Up
Training	60000	111.3170	18.7777	5.92x
Testing	10000	5.7170	1.1664	4.90x

B. Results Discussion

Each parallelization technique targets a specific phase of the training and testing process, contributing uniquely to overall efficiency. Batch processing achieved the highest speed-up in training (4.99x) as it efficiently parallelized relatively lightweight operations, drastically reducing batch time. In contrast, techniques like forward propagation, backpropagation, and softmax calculation deal with computationally heavier operations, such as feature extraction, gradient updates, and score computations, which inherently limit the extent of speed-up achievable. Input data normalization demonstrated significant speed-ups in testing (5.04x) by streamlining preprocessing and reducing I/O overhead, but its impact during training was more modest due to its limited scope in heavier computations. Combining these techniques is crucial, as each resolves specific bottlenecks—whether in preprocessing, forward pass, or backward pass—ensuring that no single phase hinders overall performance. The variability in speed-ups reflects the computational complexity of each phase, with heavier operations showing more incremental improvements despite parallelization.

LAU Lebanese American University | School of Arts and Sciences

Cuda C

How was Cuda C Parallelized?

1. Optimization on testing

Goal: Improve the speed of the testing process, as the function predict operates on every test data independently.

Process: Rewrite the testing function to a CUDA kernel, shared memory is used to store the prediction results for every test data.

Optimization Details:

1. *Large Structures Handling:*
   - Feature and LeNet5 structures are large (72KB and 415KB respectively).
   - Optimization involves moving the sum of deltas out of the CUDA kernel. The GPU processes batches of data, and the CPU aggregates the results.
2. *CUDA Resource Allocation:*
   - Allocates memory on the GPU (cudaMalloc) for the training data, labels, and model weights.
   - Uses device memory for the training batch and partially processes results on the CPU.
3. *Batch Processing Workflow:*
   - Copies training data and labels from host (CPU) to device (GPU) memory (cudaMemcpy).
   - Processes the batch in parallel using CUDA kernels.
   - Aggregates the results on the host after processing each batch.

Code

Memory Allocation:

```
LeNet5 *tgpulenet;
cudaMalloc((void**)&tgpulenet, sizeof(LeNet5)); // Model on GPU
image *gputraindata;
cudaMalloc((void**)&gputraindata, COUNTTRAIN * sizeof(image)); // Training data
on GPU
uint8 *gputrainlabel;
cudaMalloc((void**)&gputrainlabel, COUNTTRAIN * sizeof(uint8)); // Labels on GPU
```

```
LeNet5 *gputrain;
cudaMalloc((void**)&gputrain, sizeof(LeNet5) * batchsize); // Batch processing on
GPU
```

Data Transfer (Host to Device):

```
cudaMemcpy(gputraindata, traindata, COUNTTRAIN * sizeof(image),
cudaMemcpyHostToDevice);
cudaMemcpy(gputrainlabel, trainlabel, COUNTTRAIN * sizeof(uint8),
cudaMemcpyHostToDevice);
```

Batch Processing Loop:

```
for (int i = 0, percent = 0; i < totalsize - batchsize; i += batchsize) {
    cudaMemset(gputrain, 0, sizeof(LeNet5) * batchsize);
    cudaMemcpy(tgpulenet, lenet, sizeof(LeNet5), cudaMemcpyHostToDevice);
    TrainBatch<<<trainBLOCKNUM, trainTHREADNUM>>>(tgpulenet, gputraindata + i,
gputrainlabel + i, batchsize, gputrain);
    cudaMemcpy(s, gputrain, sizeof(LeNet5) * batchsize, cudaMemcpyDeviceToHost);

    // Aggregate results
    for (int mm = 0; mm < batchsize; ++mm) {
        LeNet5 ttt = s[mm];
        FOREACH(j, GETCOUNT(LeNet5)) {
            buffer[j] += ((double *)&ttt)[j];
        }
    }

    // Update model weights
    FOREACH(nn, GETCOUNT(LeNet5)) {
        ((double *)lenet)[nn] += buffer[nn] * ALPHA / batchsize;
    }
}
```

CUDA Kernel for Training Batch:

```
__global__ void TrainBatch(LeNet5 *lenet, image *inputs, uint8 *labels, int
batchSize, LeNet5 *gputrain) {
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    const int num = bid * blockDim.x + tid;
    Feature gpufeatures = 0;
```

```
    Feature gpuerrors = 0;

    // Initialize weights for current batch
    gputrain[num] = 0;

    // Forward and backward pass
    loadinput(&gpufeatures, inputs[num]);
    forward(lenet, &gpufeatures, relu);
    loadtarget(&gpufeatures, &gpuerrors, labels[num]);
    // Further processing for backpropagation would go here...
}
```

2. OPTIMIZATION ON TRAINBATCH

Goal: Speedup the TrainBatch process for it trains a batch of training data in parallel, which is suitable for CUDA application.

Process: Optimization involves moving the sum of deltas out of the CUDA kernel. The GPU processes batches of data, and the CPU aggregates the results.

Optimization Details

1. *Large Structures Handling:*
   - Feature and LeNet5 structures are large (72KB and 415KB respectively).
   - Optimization involves moving the sum of deltas out of the CUDA kernel. The GPU processes batches of data, and the CPU aggregates the results.
2. *CUDA Resource Allocation:*
   - Allocates memory on the GPU (cudaMalloc) for the training data, labels, and model weights.
   - Uses device memory for the training batch and partially processes results on the CPU.
3. *Batch Processing Workflow:*
   - Copies training data and labels from host (CPU) to device (GPU) memory (cudaMemcpy).
   - Processes the batch in parallel using CUDA kernels.
   - Aggregates the results on the host after processing each batch

Code

Memory Allocation:

```
LeNet5 *tgpulenet;
cudaMalloc((void**)&tgpulenet, sizeof(LeNet5)); // Model on GPU
image *gputraindata;
cudaMalloc((void**)&gputraindata, COUNTTRAIN * sizeof(image)); // Training data
on GPU
uint8 *gputrainlabel;
cudaMalloc((void**)&gputrainlabel, COUNTTRAIN * sizeof(uint8)); // Labels on GPU
LeNet5 *gputrain;
cudaMalloc((void**)&gputrain, sizeof(LeNet5) * batchsize); // Batch processing on
GPU
```

Data Transfer (Host to Device):

```
cudaMemcpy(gputraindata, traindata, COUNTTRAIN * sizeof(image),
cudaMemcpyHostToDevice);
cudaMemcpy(gputrainlabel, trainlabel, COUNTTRAIN * sizeof(uint8),
cudaMemcpyHostToDevice);
```

Batch Processing Loop:

```
for (int i = 0, percent = 0; i < totalsize - batchsize; i += batchsize) {
    cudaMemset(gputrain, 0, sizeof(LeNet5) * batchsize);
    cudaMemcpy(tgpulenet, lenet, sizeof(LeNet5), cudaMemcpyHostToDevice);
    TrainBatch<<<trainBLOCKNUM, trainTHREADNUM>>>(tgpulenet, gputraindata + i,
gputrainlabel + i, batchsize, gputrain);
    cudaMemcpy(s, gputrain, sizeof(LeNet5) * batchsize, cudaMemcpyDeviceToHost);

    // Aggregate results
    for (int mm = 0; mm < batchsize; ++mm) {
        LeNet5 ttt = s[mm];
        FOREACH(j, GETCOUNT(LeNet5)) {
            buffer[j] += ((double *)&ttt)[j];
        }
    }
```

```
    // Update model weights
    FOREACH(nn, GETCOUNT(LeNet5)) {
        ((double *)lenet)[nn] += buffer[nn] * ALPHA / batchsize;
    }
}
```

CUDA Kernel for Training Batch:

```
__global__ void TrainBatch(LeNet5 *lenet, image *inputs, uint8 *labels, int
batchSize, LeNet5 *gputrain) {
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;
    const int num = bid * blockDim.x + tid;
    Feature gpufeatures = 0;
    Feature gpuerrors = 0;

    // Initialize weights for current batch
    gputrain[num] = 0;

    // Forward and backward pass
    loadinput(&gpufeatures, inputs[num]);
    forward(lenet, &gpufeatures, relu);
    loadtarget(&gpufeatures, &gpuerrors, labels[num]);
    // Further processing for backpropagation would go here...
}
```

Execution Time Calculation

With Batch size 320:

```
#define trainBLOCK_NUM  2
#define trainTHREAD_NUM 160
#define BATCHSIZE       trainBLOCK_NUM * trainTHREAD_NUM67
```

Output:

```
Precision is 0.968500
REAL Training time is 38.286855s
REAL Testing time is 0.822030s
```

LAU Lebanese American University | School of Arts and Sciences

CUDA C optimization analysis

A. Size Comparison: Performance Differences

Batch Size	Output	Observations
32	Precision is 53073.835900 REAL Training time is 10.436595s REAL Testing time is 0.000009s	Small batch size results in more iterations, higher data transfer overhead, and potential numerical instability. The extremely high precision observed here is likely due to an anomaly, possibly overfitting.
64	Precision is 15157.235100 REAL Training time is 8.068311s REAL Testing time is 0.000012s	With a slightly larger batch size, the training time decreases compared to 32, but the anomaly in precision persists.
320	Precision is 0.968500 REAL Training time is 38.286855s REAL Testing time is 0.822030s	Batch size 300 offers a balance between precision and performance. Training time is longer compared to smaller batch sizes due to the increased computation per batch. Precision is reasonable, and testing time increases slightly.
1024	Precision is 0.726900 REAL Training time is 22.496019s REAL Testing time is 1.560786s	Larger batch size leads to significantly faster training times due to better GPU utilization. However, precision decreases, likely because larger batches smooth gradients, reducing the model's ability to capture finer details.

Conclusion

From the experiments, batch size 300 provides the best balance between precision and speed. Smaller batch sizes like 32 and 64 show unusual precision values, indicating potential issues with data handling or overfitting. Larger batch sizes like 1024 optimize training time but at the cost of lower precision due to fewer weight updates per epoch. For this reasoning we will adapt batch size 320 in our study.

B. Calculations

S(p) = Sequential time / Parallel time

Process	Number of Samples	Sequential (s)	CUDA C (s)	Speed Up
Training	60000	111.3170	38.2865	2.90x
Testing	10000	5.7170	0.8220	6.95x

C. Results Discussion

To achieve better precision, the batch size is kept around 300, which limits the number of threads that can be effectively utilized in CUDA during the *trainBatch process*. As a result, the speedup for the training process, while noticeable at 2.90x, is not as high as expected. This is because CUDA's performance heavily depends on maximizing the utilization of available *GPU threads*. With a relatively small batch size, the workload cannot fully utilize the GPU's parallel processing capabilities.

Additionally, the intermediate data structures and results in the training process are very large, which requires them to be stored in both local and global memory. This, combined with the significant data transfer between the host (CPU) and device (GPU), creates additional overhead that slows down the training process.

On the other hand, the testing process shows a much more impressive speedup of 6.95x. This is because testing involves a simpler and less memory-intensive workload, with fewer data transfers between the host and device. Consequently, CUDA is able to perform the testing process much more efficiently, taking full advantage of the GPU's parallel processing power.

In summary, while the training process is limited by batch size and memory overhead, the testing process demonstrates the full potential of CUDA optimization for lighter workloads with less data transfer.

LAU Lebanese American University | School of Arts and Sciences

MPI

How was MPI Parallelized?

A. TrainBatch:

- Parallelizes the training process by distributing the training data (inputs and labels) across multiple MPI processes using MPI_Scatter.
- Each process computes local updates (deltas) for a portion of the batch, and these updates are accumulated using MPI_Reduce to compute the global deltas.
- The global deltas are applied to update the model on the root process, and the updated model is broadcast to all processes using MPI_Bcast.

```
void TrainBatch(LeNet5 *lenet, image *inputs, uint8 *labels, int batchSize) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int localBatchSize = batchSize / size;

    image *localInputs = (image *)malloc(localBatchSize * sizeof(image));
    uint8 *localLabels = (uint8 *)malloc(localBatchSize * sizeof(uint8));

    MPI_Scatter(inputs, localBatchSize * sizeof(image), MPI_BYTE,
                localInputs, localBatchSize * sizeof(image), MPI_BYTE, 0,
MPI_COMM_WORLD);
    MPI_Scatter(labels, localBatchSize, MPI_BYTE,
                localLabels, localBatchSize, MPI_BYTE, 0, MPI_COMM_WORLD);

    double buffer[GETCOUNT(LeNet5)] = {0};
    for (int i = 0; i < localBatchSize; ++i) {
        Feature features = {0};
        Feature errors = {0};
        LeNet5 deltas = {0};

        load_input(&features, localInputs[i]);
        forward(lenet, &features, relu);
        load_target(&features, &errors, localLabels[i]);
        backward(lenet, &deltas, &errors, &features, relugrad);
```

```
        FOREACH(j, GETCOUNT(LeNet5)) {
            buffer[j] += ((double *)&deltas)[j];
        }
    }

    double globalBuffer[GETCOUNT(LeNet5)] = {0};
    MPI_Reduce(buffer, globalBuffer, GETCOUNT(LeNet5), MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        double k = ALPHA / batchSize;
        FOREACH(i, GETCOUNT(LeNet5)) {
            ((double *)lenet)[i] += k * globalBuffer[i];
        }
    }

    MPI_Bcast(lenet, sizeof(LeNet5), MPI_BYTE, 0, MPI_COMM_WORLD);

    free(localInputs);
    free(localLabels);
}
```

B. PredictBatch:

- Parallelizes the prediction process by distributing the test data (test_data and test_label) across multiple MPI processes using MPI_Scatter.
- Each process computes the number of correct predictions for its local portion of the test data.
- The local counts of correct predictions are accumulated across all processes using MPI_Reduce, and the total number of correct predictions is returned on the root process.

```
int PredictBatch(LeNet5 *lenet, image *test_data, uint8 *test_label, int
total_size) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int localSize = total_size / size;

    image *localTestData = (image *)malloc(localSize * sizeof(image));
    uint8 *localTestLabel = (uint8 *)malloc(localSize * sizeof(uint8));
```

```
    MPI_Scatter(test_data, localSize * sizeof(image), MPI_BYTE,
                localTestData, localSize * sizeof(image), MPI_BYTE, 0,
MPI_COMM_WORLD);
    MPI_Scatter(test_label, localSize, MPI_BYTE,
                localTestLabel, localSize, MPI_BYTE, 0, MPI_COMM_WORLD);

    int localCorrect = 0;
    for (int i = 0; i < localSize; ++i) {
        uint8 predictedLabel = Predict(lenet, localTestData[i], 10);
        if (predictedLabel == localTestLabel[i]) {
            localCorrect++;
        }
    }

    int globalCorrect = 0;
    MPI_Reduce(&localCorrect, &globalCorrect, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    free(localTestData);
    free(localTestLabel);

    if (rank == 0) {
        return globalCorrect;
    }
    return 0;
}
```

C. Parallelized Functions in main:

1. **training Function:**
    - This function internally uses TrainBatch, which parallelizes the training process by distributing the training data and labels across MPI processes.
    - MPI_Scatter is used to distribute the data, MPI_Reduce is used to aggregate updates, and MPI_Bcast ensures the updated model is synchronized across all processes.

```
void training(LeNet5 *lenet, image *train_data, uint8 *train_label, int
batch_size, int total_size) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // No changes needed here; the TrainBatch function handles MPI operations
    for (int i = 0, percent = 0; i <= total_size - batch_size; i += batch_size) {
        TrainBatch(lenet, train_data + i, train_label + i, batch_size);
```

```
        if (rank == 0 && i * 100 / total_size > percent)
            printf("batchsize:%d\ttrain:%2d%%\n", batch_size, percent = i * 100 /
total_size);
    }
}
```

2. **testing Function:**
    - This function internally uses PredictBatch, which distributes the test dataset across MPI processes using MPI_Scatter.
    - Each process computes local predictions, and the results are aggregated using MPI_Reduce.

```
int testing(LeNet5 *lenet, image *test_data, uint8 *test_label, int total_size) {
    int correct = PredictBatch(lenet, test_data, test_label, total_size);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        printf("Total correct predictions: %d\n", correct);
        printf("Accuracy: %.2f%%\n", (correct * 100.0) / total_size);
    }

    return correct;
}
```

3. **Dataset Distribution:**
    - The dataset (train_data, train_label, test_data, and test_label) is broadcasted from the root process to all other processes using MPI_Bcast. This ensures that all processes have access to the complete dataset for their assigned tasks.

```
  if (rank == 0) {
        if (read_data(train_data, train_label, COUNT_TRAIN, FILE_TRAIN_IMAGE,
FILE_TRAIN_LABEL)) {
            printf("ERROR: Dataset not found!\n");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        if (read_data(test_data, test_label, COUNT_TEST, FILE_TEST_IMAGE,
FILE_TEST_LABEL)) {
            printf("ERROR: Dataset not found!\n");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
```

```
    }

    // Broadcast datasets
    MPI_Bcast(train_data, COUNT_TRAIN * sizeof(image), MPI_BYTE, 0,
MPI_COMM_WORLD);
    MPI_Bcast(train_label, COUNT_TRAIN, MPI_BYTE, 0, MPI_COMM_WORLD);
    MPI_Bcast(test_data, COUNT_TEST * sizeof(image), MPI_BYTE, 0,
MPI_COMM_WORLD);
    MPI_Bcast(test_label, COUNT_TEST, MPI_BYTE, 0, MPI_COMM_WORLD);
```

4. **Model Broadcasting:**
    - The LeNet5 model is broadcasted from the root process to all other processes using MPI_Bcast. This ensures that all processes start with the same initial model and stay synchronized after each training batch.

```
LeNet5 *lenet = (LeNet5 *)malloc(sizeof(LeNet5));
    if (rank == 0) {
        if (load(lenet, LENET_FILE)) {
            Initial(lenet);
        }
    }

    // Broadcast the model to all processes
    MPI_Bcast(lenet, sizeof(LeNet5), MPI_BYTE, 0, MPI_COMM_WORLD);
```

Execution Time Calculation

N=8

```
Total correct predictions: 9700
Accuracy: 97.00%
Training CPU time: 12288517
Training CPU time in seconds: 12.288517s
Testing CPU time: 420651
Testing CPU time in seconds: 0.420651s
Training time: 12.316716s
Testing time: 0.420883s
```

Other Calculations

Performance Metrics Table based on varying n (number of processors)

n	Output compiled using gcc	Output compiled using g++	Observation
2	Training time: 22.030156s Testing time: 0.737797s	Training time: 20.224245s Testing time: 0.721309s	Fewer processes were insufficient, leading to longer training and lower accuracy.
4	Training time: 15.183584s Testing time: 0.508331s	Training time: 14.735062s Testing time: 0.527284s	Improved accuracy and reduced training time compared to n=2.
8	Training time: 12.316716s Testing time: 0.420883s	Training time: 12.531597s Testing time: 0.479521s	Optimal configuration; highest accuracy and efficient training/testing times.
10	Training time: 18.619282s Testing time: 0.524559s	Training time: 18.064523s Testing time: 0.449043s	More processes caused overhead, slightly reducing efficiency.
20	Training time: 19.119670s Testing time: 0.445452s	Training time: 19.119670s Testing time: 0.445452s	Excessive overhead with many processes, leading to diminished performance.

Since n =8 yielded the best result, we will use it in our analysis.

S(p) = Sequential time / Parallel time

Process	Number of Samples	Sequential (s)	MPI (s)	Speed Up
Training	60000	111.3170	12.316	9.03x
Testing	10000	5.7170	0.4208	13.58x

Result Discussion:

The MPI implementation achieved significant speedup for both training and testing, with a **9.03x speedup for training** and a **13.58x speedup for testing**, demonstrating the effectiveness of its parallelization strategy. This was accomplished through efficient *workload* distribution using MPI_Scatter, where training and testing data were evenly divided among processes to ensure balanced computational loads. During training, local updates (deltas) were computed by each process and aggregated via MPI_Reduce to form global updates, which were then applied to the model on the root process and *synchronized* across all processes using MPI_Bcast. Similarly, testing leveraged MPI_Scatter to distribute data and MPI_Reduce to consolidate predictions, minimizing overhead while maximizing parallel efficiency. These techniques, combined with the ability to utilize *distributed memory systems*, made MPI particularly adept at handling large-scale datasets and deep learning models, achieving remarkable speedup and consistent performance across all processes.

Bonus:

Even though the code is written in **C**, we observed that **g++** performs slightly faster than **gcc**. This can be attributed to subtle differences in optimization strategies. While both compilers are part of the GNU Compiler Collection and share the same backend, **g++** may apply slightly more aggressive optimizations by default, even when compiling C code. Additionally, **g++** links against runtime libraries that might offer marginal efficiency improvements, such as better handling of memory or system calls. These differences, though minor, highlight that the choice of compiler can impact performance, even for code written in C. Testing with different compilers is crucial to identifying the best option for specific use cases.

The observed results align with the specifications of the machine, as shown in the provided system information. My machine has 8 CPUs and 4 cores, with each core capable of handling 2 threads. This explains why using **n=8** (matching the number of CPUs available) yielded the best performance. At **n=8**, the machine was able to fully utilize all available CPUs without introducing significant overhead or underutilization. Lower values of **n** could not fully utilize the computational resources, while higher values of **n** introduced overhead from context switching and resource contention, leading to diminishing returns in performance.

Conclusion Analysis

This table shows the comparison of performance of MPI, CUDA C, and OpenMP for parallelizing CNNs (Convolutional Neural Networks). It consolidates the execution time and speedup results for training and testing 60,000 and 10,000 samples, respectively.

	Process	*Parallel Time*	*Speed Up*
MPI	**Training**	**17.8024**	**9.03x**
	Testing	**0.7816**	**13.58x**
CUDA C	**Training**	**38.2865**	**2.90x**
	Testing	**0.8220**	**6.95x**
OpenMP	**Training**	**18.7777**	**5.92x**
	Testing	**1.1664**	**4.90x**

From the results, it is evident that MPI achieved the highest speedup for both training (9.03x) and testing (13.58x), outperforming CUDA C and OpenMP. This superior performance can be attributed to MPI's *distributed memory architecture*, which is well-suited for handling large datasets and distributing workloads across multiple nodes. MPI's ability to efficiently balance the workload using MPI_Scatter for data *distribution and synchronize* updates using MPI_Reduce and MPI_Bcast ensures minimal bottlenecks and consistent performance across all processes.

In contrast, CUDA C and OpenMP rely on *shared memory*, which is inherently limited by memory bandwidth and hardware constraints within a single node.

CUDA C, while highly efficient for *GPU-based parallelism*, demonstrated mixed results. It achieved the second-highest testing speedup (6.95x), but its training speedup (2.90x) lagged behind both MPI and OpenMP. These discrepancies can be attributed to *synchronization overhead* and the high costs of memory transfers between the CPU and GPU, which become bottlenecks for larger datasets.

Interestingly, OpenMP outperformed CUDA in training due to the relatively small workload and memory-bound nature of CNN training, which favored CPU-based processing. OpenMP avoids the high overheads of memory transfers and kernel launches inherent to CUDA, making it better suited for this specific task. However, with optimized configurations and larger workloads, CUDA could potentially outperform OpenMP due to its ability to leverage massively parallel GPU cores.

OpenMP, which operates entirely in shared memory, achieved moderate performance for both training (5.92x) and testing (4.90x). However, its scalability was constrained by the number of available CPU cores and thread management overhead.

Future potential enhancement

The results highlight that CNNs, with their computational and memory-intensive operations, benefit significantly from distributed memory systems like MPI. However, the choice of parallelization framework should depend on the available hardware, dataset size, and task requirements. *A hybrid approach combining MPI, CUDA, and OpenMP* could yield even better results by leveraging the strengths of each framework: MPI excels in large-scale distributed training and inter-node synchronization, CUDA is ideal for GPU-accelerated matrix operations and convolutional computations, and OpenMP is well-suited for lightweight multithreading on CPUs for preprocessing and aggregation tasks. While this strategy offers substantial performance gains, it demands careful implementation, coordination, and tuning to minimize overhead and ensure seamless communication between frameworks. When executed effectively, this approach can provide a highly scalable and efficient solution for studying and optimizing CNNs.

References

Lu, H., Jacob, R., & Anderegg, L. (2017). Title of the article. *Journal Name.* http://cucis.ece.northwestern.edu/publications/pdf/LJA17.pdf

U.S. Department of Energy. (2022). *Title of the report* (Report No. 1864734). Office of Scientific and Technical Information. https://www.osti.gov/servlets/purl/1864734

Hinton, G., Srivastava, N., & Krizhevsky, A. (2012). *Improving neural networks by preventing co-adaptation of feature detectors*. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NeurIPS 2012)* (pp. 1–9). https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Duchi, J. C., Ruan, F., & Wainwright, M. J. (2017). *Stochastic methods for composite and weakly convex optimization problems: Convergence rates and optimality*. *SIAM Journal on Optimization, 27*(3), 1122–1146. https://doi.org/10.1137/16M1080173