**A logo with text and numbers

Description automatically generated**

**Lebanese American University**

**Parallel Programming Course Project**

**Project Title**

Parallelizing LeNet5 CNN

**Submitted by:**

* **Student Name(s)**: [Abdullah Jrad]
* **Student ID(s)**: [202207733]
* **Course Name**: [Parallel Progg/Multic.&Cluster]
* **Instructor’s Name**: [Hamdan Abdellatef]
* **Submission Date**: [5/Dec/2024]

The source codes can be found on github through this link: [CODE](https://github.com/Abedsay/Parallelizing-LeNet-5-CNN)

**Overview of CNNs and LeNet-5**

**What is a Convolutional Neural Network (CNN)?**

Convolutional Neural Networks (CNNs) are deep learning algorithms specifically designed for image processing and classification tasks. CNNs are composed of layers, such as:

* **Convolutional Layers**: Extract spatial features from images using filter kernels.
* **Pooling Layers**: Reduce the spatial dimensions while retaining essential features.
* **Fully Connected Layers**: Map extracted features to the desired output, such as classifications.

These layers work together to identify patterns in images, enabling tasks like digit recognition, object detection, and more.

**What is LeNet-5?**

****

LeNet-5 consists of 7 levels and can process 32-by-32 images that contain hand-written digits.

Among 7 layers, there are 2 convolution layers, 2 subsampling layers, 2 full connection layers and a Gaussian connection layer.

This project aims to accelerate the training of LeNet-5 by parallelizing its computations, focusing on the **convolutional layer** due to its computational intensity.

**Parallelization Strategies:**

**A- MPI:**

**- Design Choices**

1. Distributed Data Parallelism:

Training batches were distributed across multiple processes, with each process independently computing gradients for its subset of data.

1. Synchronization:

After computing gradients, processes used MPI\_Reduce to aggregate gradients and MPI\_Bcast to distribute updated weights.

1. Scalability:

The implementation was tested with 2, 4, 8, 10, and 20 processes to observe performance improvements and communication overhead.

**- Pseudocode**

Below is the pseudocode highlighting the parallelization of the **TrainBatch** function using MPI:

*Sequential Code:*

for each batch in training\_data:

initialize gradients to 0

for each image in batch:

compute forward pass

compute backward pass

accumulate gradients

update weights using gradients

*Parallelized MPI Code:*

distribute training\_data among processes

for each batch in local\_training\_data:

initialize local\_gradients to 0

for each image in local\_batch:

compute forward pass

compute backward pass

accumulate local\_gradients

reduce local\_gradients to global\_gradients (MPI\_Reduce)

if rank == 0:

update weights using global\_gradients

broadcast updated weights to all processes (MPI\_Bcast)

**- Performance Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Processes** | **Training Time (Seconds)** | **Testing Time (Seconds)** | **Precision** |
| 2 | 45.23 | 5.38 | 0.9869 |
| 4 | 25.56 | 5.55 | 0.9878 |
| 8 | 16.67 | 5.79 | 0.9886 |
| 10 | 14.57 | 5.96 | 0.9841 |
| 20 | 9.18 | 6.09 | 0.9864 |

|  |  |  |  |
| --- | --- | --- | --- |
| Sequential | 88.10 | 5.36 | 0.9720 |

**- Discussion**

1. Training Time:

Significant reductions in training time were observed with increasing processes up to 8.

Beyond 8 processes, the improvement diminished.

1. Testing Time:

Testing times increased slightly with more processes, as communication overhead outweighed the benefits for the smaller workload.

1. Precision:

Consistent across all runs, demonstrating correctness of the parallel implementation.

**B- OpenMP:**

**-** **Design Choices**

In the sequential implementation, computations for each image in a batch were performed serially. With OpenMP, these computations were parallelized across multiple threads:

* Forward Pass: Each thread processed one or more images in the batch.
* Backward Pass: Gradients were computed for each image in parallel.

Load Balancing

OpenMP's dynamic scheduling was employed for load balancing, ensuring even distribution of work, especially for large datasets.

**- Pseudocode**

*Convolution Layer Forward Pass (Applied to process multiple images):*

#pragma omp parallel for schedule(dynamic)

for (int i = 0; i < batchSize; ++i) {

for (int f = 0; f < numFilters; ++f) {

perform\_convolution(input[i], filters[f], output[i][f]);

}

}

*Backward Pass (Gradient computation in parallel):*

#pragma omp parallel for schedule(dynamic)

for (int i = 0; i < batchSize; ++i) {

for (int f = 0; f < numFilters; ++f) {

compute\_gradient(input[i], filters[f], gradients[f]);

}

}

*Gradient Accumulation (Critical Section):*

#pragma omp critical

for (int f = 0; f < numFilters; ++f) {

shared\_gradients[f] += local\_gradients[f];

}

**- Performance Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Threads** | **Training Time (Seconds)** | **Testing Time (Seconds)** | **Precision** |
| 2 | 21.92 | 1.27 | 0.9652 |
| 4 | 12.31 | 0.68 | 0.9686 |
| 8 | 8.28 | 0.40 | 0.9702 |
| 10 | 7.18 | 0.35 | 0.9692 |
| 20 | 4.54 | 0.23 | 0.9690 |

|  |  |  |  |
| --- | --- | --- | --- |
| Sequential | 88.10 | 5.36 | 0.9720 |

**- Discussion**

1. Training Time:

The training time with OpenMP dropped drastically to 4.5 seconds (20 threads) compared to 88.1 seconds for the sequential implementation.

The parallelization in OpenMP was effective, as the shared-memory architecture minimized communication overhead.

1. Testing Time:

Testing times differed too with a drop from 5.36 (Sequential) to 0.22 (OpenMP 20 threads).

1. Precision:

Precision remained consistent with the sequential baseline, verifying correctness. The slight drop to 0.9687 was negligible and within acceptable margins.

**C- CUDA:**

**- Design Choices**

* Convolution Layer: The convolution layer was the primary target for parallelization due to its computational intensity. CUDA threads were mapped to pixels in the output feature maps, and each thread was responsible for computing one output pixel by convolving the filter kernel over the input image.
* Fully Connected Layer: The fully connected layer was treated as a matrix-vector multiplication problem. Each CUDA thread was assigned a row of the matrix, and the output was computed by performing a dot product with the input vector.

**- Pseudocode**

*Convolution Layer Forward Pass:*

\_\_global\_\_ void convolution\_forward(float \*input, float \*filters, float \*output, int num\_filters, int image\_width, int image\_height) {

int x = blockIdx.x \* blockDim.x + threadIdx.x; // Output pixel x

int y = blockIdx.y \* blockDim.y + threadIdx.y; // Output pixel y

int filter = blockIdx.z; // Current filter

if (x < image\_width && y < image\_height) {

float sum = 0.0f;

for (int fx = 0; fx < FILTER\_WIDTH; fx++) {

for (int fy = 0; fy < FILTER\_HEIGHT; fy++) {

int ix = x + fx - FILTER\_WIDTH / 2; // Input x

int iy = y + fy - FILTER\_HEIGHT / 2; // Input y

if (ix >= 0 && ix < image\_width && iy >= 0 && iy < image\_height) {

sum += input[iy \* image\_width + ix] \* filters[filter \* FILTER\_WIDTH \* FILTER\_HEIGHT + fy \* FILTER\_WIDTH + fx];

}

}

}

output[filter \* image\_width \* image\_height + y \* image\_width + x] = sum;

}

}

*Fully Connected Layer Forward Pass:*

\_\_global\_\_ void fully\_connected\_forward(float \*weights, float \*input, float \*output, int input\_size, int output\_size) {

int neuron = blockIdx.x \* blockDim.x + threadIdx.x;

if (neuron < output\_size) {

float sum = 0.0f;

for (int i = 0; i < input\_size; i++) {

sum += weights[neuron \* input\_size + i] \* input[i];

}

output[neuron] = sum;

}

}

**- Performance Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Training Time (Seconds)** | **Testing Time (Seconds)** | **Precision** |
| Sequential | 88.10 | 5.36 | 0.9720 |
| CUDA | 17.80 | 0.06 | 0.9620 |

**- Discussion**

1. Training Time:

The training time reduced to 17.8 seconds compared to 88.1 seconds for the sequential implementation.

1. Testing Time:

Testing times were nearly instant at 0.06 seconds, a significant improvement over 5.36 seconds for the sequential implementation.

The testing phase benefited from preloading weights and performing forward passes exclusively.

1. Precision:

Precision remained consistent with the sequential baseline, verifying correctness. The slight drop to 0.9620 was negligible and within acceptable margins.

**D- Performance Evaluation Full Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Processes/Threads** | **Training Time (s)** | **Testing Time (s)** | **Precision** | **Speedup** | **Efficiency (%)** |
| Sequential | 1 | 88.10 | 5.36 | 0.9720 | 1.00 | 100.0 |
| MPI | 2 | 45.24 | 5.38 | 0.9869 | 1.95 | 97.5 |
| MPI | 4 | 25.56 | 5.56 | 0.9878 | 3.45 | 86.3 |
| MPI | 8 | 16.67 | 5.79 | 0.9886 | 5.29 | 66.1 |
| MPI | 10 | 14.57 | 5.96 | 0.9841 | 6.05 | 60.5 |
| MPI | 20 | 9.18 | 6.09 | 0.9864 | 9.59 | 48.0 |
| OpenMP | 2 | 21.92 | 1.27 | 0.9652 | 4.02 | 201.0 |
| OpenMP | 4 | 12.31 | 0.68 | 0.9686 | 7.16 | 179.1 |
| OpenMP | 8 | 8.28 | 0.40 | 0.9702 | 10.64 | 132.9 |
| OpenMP | 10 | 7.18 | 0.35 | 0.9692 | 12.27 | 122.7 |
| OpenMP | 20 | 4.54 | 0.23 | 0.9690 | 19.41 | 97.1 |
| CUDA | - | 17.80 | 0.06 | 0.9620 | 4.95 | - |

**E- Innovation and Advanced Optimizations**

To enhance the CNN, I incorporated MaxPooling and ReLU, which are widely used in modern architectures. MaxPooling reduces computational load by downsampling feature maps, while ReLU introduces non-linearity and prevents the vanishing gradient problem, improving accuracy and efficiency.

I also optimized the compilation process with custom Makefiles. These use the g++ compiler for better C++ optimization which proves faster training and testing times.

**F- Conclusion**

In this project, I worked on three parallel implementations—MPI, OpenMP, and CUDA—for the LeNet-5 CNN and compared them to the sequential version. OpenMP gave the best speedup and efficiency because it uses shared memory, making it faster for this setup. MPI worked well up to 8 processes but slowed down after that due to the time it takes for processes to communicate. CUDA had the fastest testing time by using GPU parallelism, but its precision was slightly lower than the other methods. Overall, the parallel implementations significantly improved training and testing times. To improve further, I could try combining MPI with OpenMP or CUDA to reduce bottlenecks. I could also focus on optimizing CUDA for better precision or work on larger and more advanced CNN models like ResNet or VGG for better performance.

**Reference:** Sequential LeNet-5 CNN: https://github.com/fan-wenjie/LeNet-5