Project 4 Report for CSC4005 Parallel Computing

Jingquan Li (119010148)

December 13, 2024

Contents

1 Introduction

In this project, I explored how parallel computing techniques can be applied to accelerate machine learning algorithms. Specifically, I worked on handwritten digit recognition using a neural network trained on the MNIST dataset. The project gave me an opportunity to implement and optimize multi-layer perceptrons (MLPs) while gaining hands-on experience with OpenAcc, a programming model for GPU-accelerated computing.

I started by implementing a sequential version of the MLP to better understand the fundamental operations involved in forward and backward propagation. This served as my baseline for measuring the improvements achieved through parallelization. Then OpenAcc was used to optimize the computations in two ways: first with kernel-based acceleration and then with a fusion-based approach. My main focus was to compare their execution times and analyze their efficiency.

Through this project, I learned how parallel computing can significantly improve the performance of machine learning tasks. I also deepened my understanding of how to effectively utilize hardware resources to optimize computational workflows. In this report, I will describe my implementation process, share my performance analysis, and reflect on what I learned throughout the project.

2 Compilation and Execution

To compile and execute the project, I followed the steps outlined below. The project was developed and tested in a Linux environment with GPU support to ensure compatibility with OpenAcc.

2.1 Compilation Steps

First, I navigated to the project directory and created a `build` folder for compiling the code. The commands I used are as follows:

```
cd /path/to/project4
mkdir build && cd build
cmake ..
make
```

The `cmake` command generates the necessary build files and `make` compiles the project into an executable. If compilation failed due to container or environment issues, I switched to manual compilation using `gcc` and `pgc++` with the appropriate optimization flags.

2.2 Execution Steps

Once the compilation succeeded, the program was executed using the provided shell script. To submit the program to the cluster, I ensured that I was in the `project4` directory and used the following command:

```
sbatch ./test.sh
```

This script runs the program and outputs the training results for the MLP under different configurations. The execution times for both sequential and parallel implementations were recorded and analyzed.

3 Design and Implementation

3.1 Sequential Implementation

The sequential implementation of the MLP forms the baseline for evaluating the performance of parallelized versions. In this implementation, I manually handled the forward and backward propagation steps without any parallelization. All computations were executed sequentially on the CPU. In the following, I explain the key components and logic of the sequential implementation.

3.1.1 Forward Propagation

Forward propagation consists of two fully connected layers with activation functions applied after each layer. The key steps include the following.

- **Matrix Multiplication (GEMM):** I implemented a general matrix-matrix multiplication (GEMM) function to compute the linear transformation in each layer. This function iterates over batches, input dimensions, and output dimensions to calculate the matrix product.
- **Adding Bias:** A bias vector is added to the output of each linear transformation using the `add_bias` function.
- **Activation Function:** The ReLU activation function is applied element-wise to introduce non-linearity.
- **Softmax:** The softmax function computes the output probabilities for each class by normalizing the logits. I implemented a stable version of softmax to prevent numerical instability.

3.1.2 Backward Propagation

The backward propagation updates the weights and biases using stochastic gradient descent (SGD). The main steps include the following.

- **Cross-Entropy Loss and Gradient:** I computed the cross-entropy loss and its gradient to measure the error between predictions and true labels.
- **Gradient Computation for Each Layer:** The gradients for the weights, biases, and intermediate activations were calculated using the chain rule. For example, the `relu_grad` function computes the gradient of the ReLU function.
- **Weight and Bias Updates:** Using the gradients, I updated the weights and biases of both layers using the SGD update rule.

3.1.3 Epoch Training

I implemented a `nn_epoch_cpp` function that processes data in mini-batches. For each mini-batch, the function performs forward propagation, computes the loss, performs backpropagation, and updates the parameters. The use of mini-batches ensures efficient memory usage and smooth convergence during training.

3.1.4 Accuracy Calculation

To evaluate the performance of the model, I calculated the accuracy after each epoch. The `argmax` function determines the predicted class for each input, and the `mean_acc` function calculates the proportion of correctly classified samples.

3.1.5 Implementation Challenges

One of the challenges was handling the numerical stability in the softmax and cross-entropy calculations. I resolved this by subtracting the maximum value in each row during the softmax computation and adding a small epsilon value to avoid division by zero or taking the logarithm of zero.

3.2 Kernel-Based OpenAcc Implementation

To improve the performance of the sequential MLP implementation, I utilized OpenAcc directives to parallelize individual operations at the kernel level. This approach focuses on accelerating specific functions such as matrix multiplication, activation functions, and gradient computations by offloading them to the GPU.

3.2.1 Design of Kernel-Based Acceleration

In this method, I wrapped key computational functions like `gemm`, `add_bias`, and `Softmax` with OpenAcc directives. The execution of each function was parallelized by adding `pragma acc` directives to exploit data-level parallelism. The functions were declared with `#pragma acc routine seq` to ensure compatibility with GPU execution.

Key Design Decisions

- **Fine-Grained Parallelism:** I used parallel loops within each kernel function to divide work among GPU threads.
- **Explicit Data Transfers:** I specified `#pragma acc data` regions to manage data movement between the host (CPU) and the device (GPU). This allowed me to explicitly control memory allocations and transfers for improved performance.
- **Flexibility:** By keeping the individual functions of the kernel separate, I ensured that each computational operation could be independently optimized.

3.2.2 Implementation Details

Forward Propagation The forward propagation steps, including matrix multiplication, bias addition, ReLU activation, and softmax normalization, were parallelized.

- `gemm`: Parallelized over the batch size and dimensions of the weight matrices.
- `add_bias`: Each bias addition was parallelized with the batch size and output dimensions.
- `Relu`: Applied in a parallel loop on all elements of the activation matrix.
- `Softmax`: Each row was processed independently to compute normalized probabilities.

Backward Propagation The backpropagation computations, including gradient calculation and parameter updates, were also parallelized:

- `cross_entropy_loss_grad`: Computed gradients of the loss function in parallel for each element.
- `input_grad`: The gradients were backpropagated through the layers using parallel loops.
- `update_weight` and `update_bias`: Updated weights and biases in parallel to improve efficiency.

Data Management To reduce GPU-CPU communication overhead, I used `#pragma acc data` regions to explicitly copy input data, weights, and intermediate results to the GPU. For example:

```
#pragma acc data copyin(array_batch[0:m_b*input_dim], Weight1[0:input_dim*hidden_num]) \
                  create(fc_1_mid[0:m_b*hidden_num])
{
    gemm(array_batch, Weight1, fc_1_mid, m_b, input_dim, hidden_num);
}
```

3.2.3 Challenges and Solutions

One of the challenges in this implementation was managing data transfers efficiently to avoid frequent GPU-CPU communication. By encapsulating multiple operations in a single data region wherever possible, I minimized transfer overhead. Furthermore, numerical stability issues in the softmax and cross-entropy functions were addressed by using techniques such as max value subtraction and the addition of a small `epsilon`.

3.3 Fusion-Based OpenAcc Implementation

To further optimize the performance of the MLP training process, I implemented a fusion-based approach using OpenAcc. Unlike the kernel-based approach, which treats each operation as a separate kernel, the fusion-based approach minimizes data transfer between the CPU and GPU by keeping all required data on the GPU throughout an epoch.

3.3.1 Design of Fusion-Based Acceleration

The fusion-based approach is designed to optimize GPU utilization and reduce communication overhead:

- **Single Data Region:** I wrapped the entire training process of an epoch within a single `#pragma acc data` region. This ensures that data remains on the GPU for the entire epoch, avoiding repeated data transfer.
- **Computation Pipelines:** I fused the forward and backward computations within the same data region. This allows seamless reuse of intermediate results stored in GPU memory.
- **Reduced Overhead:** By reducing the number of `copyin`, `copyout`, and `present` directives, the fusion-based approach eliminates much of the communication overhead observed in the kernel-based method.

3.3.2 Implementation Details

Data Region Management In this approach, all training data, weights, biases, and intermediate buffers are allocated on the GPU at the beginning of each epoch. For example:

```
#pragma acc data copyin(input_array[0:input_num*input_dim], label_array[0:input_num], \
                        Weight1[0:input_dim*hidden_num], Weight2[0:hidden_num*class_num], \
                        bias1[0:hidden_num], bias2[0:class_num]) \
                 create(fc_1_mid[0:batch_num*hidden_num], relu_1_out[0:batch_num*hidden_num], \
                        softmax_out[0:batch_num*class_num], hidden_grad[0:batch_num*hidden_num])
```

Forward Propagation The forward propagation consists of two fully connected layers, ReLU activation, and softmax normalization. Each step is executed directly on the GPU using `#pragma acc parallel loop`. For example:

- `gemm`: Computes matrix multiplication for each fully connected layer.
- `add_bias` and `Relu`: Adds bias and applies the activation function in parallel.
- `Softmax`: Normalizes the output logits to probabilities.

Backward Propagation Gradients for the loss function, weights, and biases are computed and updated directly on the GPU. Functions such as `cross_entropy_loss_grad`, `input_grad`, and `update_weight` were parallelized to utilize GPU threads.

Testing During the testing phase, intermediate results are stored on the GPU for efficient inference. The prediction process also benefits from the same optimization strategy used during training.

3.3.3 Challenges and Solutions

- **Data Management:** Managing multiple buffers for intermediate results on the GPU required careful planning to avoid memory conflicts. This was resolved by preallocating buffers using the `create` directive.
- **Debugging:** Debugging was more complex in the fusion-based approach due to the reduced CPU-GPU communication. I added intermediate checks on the GPU data for validation.
- **Numerical Stability:** As in the kernel-based approach, numerical stability issues were addressed in the softmax and loss functions by using the subtraction of the max value and adding a small epsilon.

4 Experimental Results and Analysis

4.1 Training Time Comparison

The training times for the three implementations (Sequential, OpenACC Kernel, and OpenACC Fusion) were recorded over 10 epochs. Table 1 summarizes the results, and Figure 1 visualizes the trends.

Table 1: Training Time per Epoch for Different Implementations

Epoch	Sequential (ms)	OpenACC Kernel (ms)	OpenACC Fusion (ms)
1	46943	9105	5283
2	46938	8100	5205
3	46938	8112	5230
4	46918	8125	5262
5	46915	8160	5282
6	46917	8190	5308
7	46928	8185	5319
8	46942	8190	5326
9	46928	8200	5340
10	46927	8234	5359

4.2 Analysis of Results

The sequential implementation has a consistent training time of approximately 46,900 ms per epoch. This is significantly reduced in the OpenACC Kernel implementation, which achieves training times between 8,100 and 9,100 ms per epoch. The OpenACC Fusion implementation demonstrates further optimization, with training times ranging from 5,200 to 5,400 ms per epoch.

Figure 1 illustrates the reduction in training time achieved by the two parallel implementations. The OpenACC Fusion method exhibits the most substantial improvement due to the elimination of frequent GPU-CPU communication and better utilization of GPU resources.

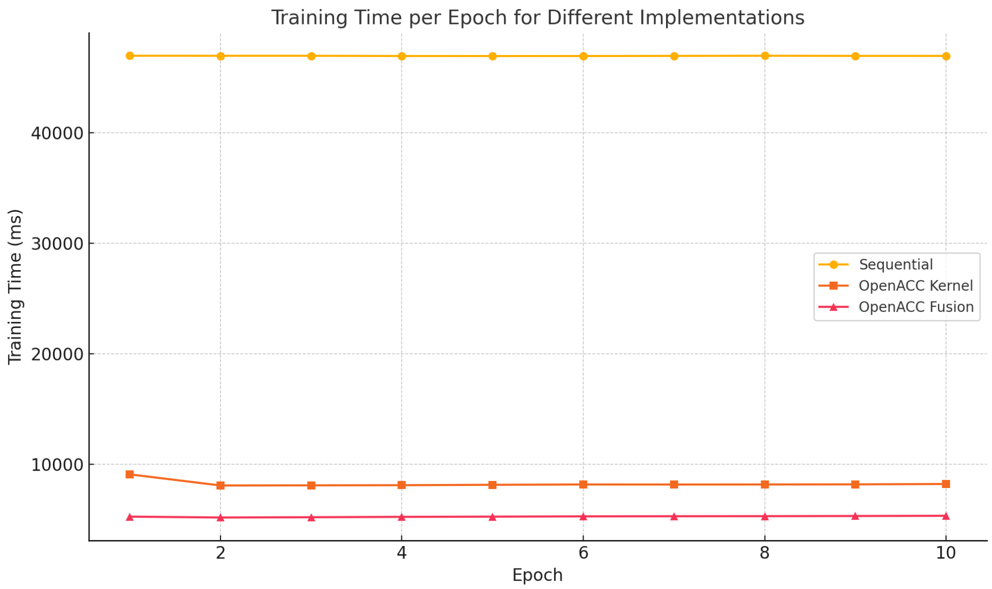

Figure 1: Training Time per Epoch for Different Implementations

4.3 Performance Insights

- **Sequential Implementation:** The high training time is due to the lack of parallelization, with all computations performed on the CPU.
- **OpenACC Kernel Implementation:** Although it significantly reduces training time, frequent data

transfers between the CPU and GPU introduce some overhead.

- **OpenACC Fusion Implementation:** By keeping data on the GPU throughout an epoch, the fusion-based approach minimizes communication overhead and achieves the best performance.

The results demonstrate the effectiveness of parallel computing in accelerating neural network training, with the OpenACC Fusion implementation offering the most efficient solution.

5 Optimizations Applied to Improve Performance

This section details the optimizations applied to the implementation of the three tasks—sequential, OpenACC kernel-based, and OpenACC fusion-based—highlighting how each step was designed to improve performance.

5.1 Sequential Implementation

The sequential implementation served as a baseline for the performance comparison. Although no parallelization was applied, the following optimizations were included.

- **Memory Management:** Buffers for intermediate results, such as activations and gradients, were pre-allocated to avoid frequent memory allocations during training.
- **Efficient Matrix Operations:** Functions like `gemm` (matrix multiplication) and `add_bias` were implemented with loop optimizations to ensure minimal overhead in computation.
- **Numerical Stability:** Functions such as `Softmax` and `cross_entropy_loss` were designed to avoid numerical underflow or overflow by subtracting the maximum value from logits and adding a small epsilon (`1e-20`).

5.2 OpenACC Kernel-Based Optimization

The kernel-based implementation introduced GPU parallelization using OpenACC directives. Key optimizations included:

- **Parallelized Operations:** Loops within critical functions such as `gemm`, `add_bias`, and `Relu` were parallelized using `#pragma acc parallel loop`.
- **Data Transfer Management:** Data was explicitly copied between the CPU and GPU using `#pragma acc data copyin` and `#pragma acc data copyout`, ensuring only the necessary data was transferred to minimize overhead.
- **Fine-Grained Kernels:** Each function was treated as a separate kernel, allowing flexibility in debugging and tuning. For example, backpropagation gradients and weight updates were handled independently.
- **Thread Optimization:** Nested loops were collapsed using the `collapse` directive, enabling efficient utilization of GPU threads for multi-dimensional operations.

5.3 OpenACC Fusion-Based Optimization

The fusion-based implementation further improved performance by reducing GPU-CPU communication overhead. The following optimizations were applied:

- **Single Data Region:** A single `#pragma acc data` region was defined for the entire training epoch, ensuring that all data (weights, biases, activations, gradients, etc.) remained on the GPU for the duration of the epoch.
- **Fused Computation Pipelines:** Multiple operations, such as forward propagation, loss calculation, and backpropagation, were fused into a single data region, eliminating the need for intermediate data transfers.
- **Memory Allocation on GPU:** Intermediate buffers for activations, gradients, and losses were pre-allocated on the GPU using the `create` directive to reduce memory management overhead.
- **Minimized Overhead:** By handling data transfer at the epoch level, communication overhead was significantly reduced compared to the kernel-based implementation, where data was transferred for each operation.

5.4 Performance Comparison

Each optimization level progressively reduced the training time:

- The kernel-based implementation achieved a significant speed-up over the sequential version by leveraging GPU parallelization.
- The fusion-based implementation further optimized performance by reducing communication overhead and better utilizing GPU resources, achieving the fastest training times across all epochs.

These optimizations demonstrate the importance of leveraging both parallelization and efficient data management techniques to achieve substantial performance gains in machine learning workloads.

6 Conclusion

In this project, I explored and implemented multiple strategies to optimize the training process of a two-layer neural network for handwritten digit recognition using the MNIST dataset. Starting from a sequential implementation, I progressively enhanced the performance through GPU-based parallelization using OpenACC, employing both kernel-based and fusion-based approaches.

The sequential implementation provided a baseline for comparison, highlighting the limitations of CPU-based computation for machine learning tasks. By introducing OpenACC kernel-based optimization, I achieved a significant reduction in training time through fine-grained parallelization of key operations such as matrix multiplication and activation functions. Finally, fusion-based optimization further improved performance by minimizing GPU-CPU communication overhead, resulting in the fastest execution times among all implementations.

This project highlighted the importance of parallel computing in the acceleration of machine learning workloads. The results demonstrated that a careful balance of computational optimization and memory management can lead to substantial performance improvements. In addition, it emphasized the need to adapt optimization strategies based on underlying hardware to fully utilize available resources.

Through this work, I gained valuable insight into the challenges and benefits of parallel programming and the potential of GPU acceleration in real-world applications. The techniques learned and applied here are applicable not only to neural networks, but also to a wide range of high-performance computing tasks.