Parallelized Optimization on Machine Learning Algorithm

CHONG-YAN CHEN, National Yang Ming Chiao Tung University, Taiwan
LU-YING LIN, National Yang Ming Chiao Tung University, Taiwan
LI-LUN LIN, National Yang Ming Chiao Tung University, Taiwan

This project mainly aims to accelerate the forward and backward process of a neural network with some common parallel programming techniques.

In the experiment, we tried to parallelize the process on both CPU and NVIDIA-GPU in order to speed up the algorithms. For CPU, we utilized the easy-to-use programming model OpenMP to build a multithreaded solution. As for GPU, we chose CUDA for its platform-specific optimization and the resulting performance.

The result of our experiment shows that parallelism on the forward and backward process do play an important role in accelerating neural networks.

ACM Reference Format:
Chong-Yan Chen, Lu-Ying Lin, and Li-Lun Lin. 2023. Parallelized Optimization on Machine Learning Algorithm. 1, 1 (January 2023), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Machine learning is a powerful tool for understanding and predicting complex phenomena, but the computational demands of training machine learning models can be high. This can be especially challenging when working with large datasets or when training models that require a lot of computation, such as deep neural networks. In this project, our goal was to accelerate machine learning algorithms to make them more practical and efficient.

To understand the potential for accelerating machine learning algorithms, it's important to consider the history and evolution of computing systems. The development of multi-processing and parallel programming techniques has played a crucial role in improving the performance and capabilities of modern computers.

When computers were first introduced, the speed of the computation mostly depends on the clock rate of the processor. In recent decades, continuing to raise the clock rate would have caused computing systems to overheat. As a result, the focus of modern computer systems has shifted to multi-processing in order to improve computation capability. In most high-performance computer clusters, parallel programming is necessary to fully utilize the strength of such multi-processor systems.

To speed up machine learning and take advantage of multi-processor systems, we implemented a simple deep learning library and conducted experiments on a simple 2-layer neural network using the famous MNIST dataset. We then evaluated the performance of the parellel programming technique we used in the implementation on different batch sizes, and hidden dimensions.

Authors' addresses: Chong-Yan Chen, National Yang Ming Chiao Tung University, Taiwan; Lu-Ying Lin, National Yang Ming Chiao Tung University, Taiwan; Li-Lun Lin, National Yang Ming Chiao Tung University, Taiwan.

Manuscript submitted to ACM

In the previous works, there are many popular DL frameworks that utilize the power of multi-core CPU and GP-GPU to accelerate the algorithms. We are interested in how these frameworks implement their algorithms and the performance of their implementations. Therefore, we compare our implementation with their implementation.

In this report, we will first provide a brief overview of some popular machine learning frameworks. Next, we will describe the methods we used to implement and evaluate our techniques, including the datasets and models we used. We will then present the results of our experiments and discuss their implications for the practicality and efficiency of machine learning algorithms. Finally, we will conclude with a summary of the project and suggestions for future work.

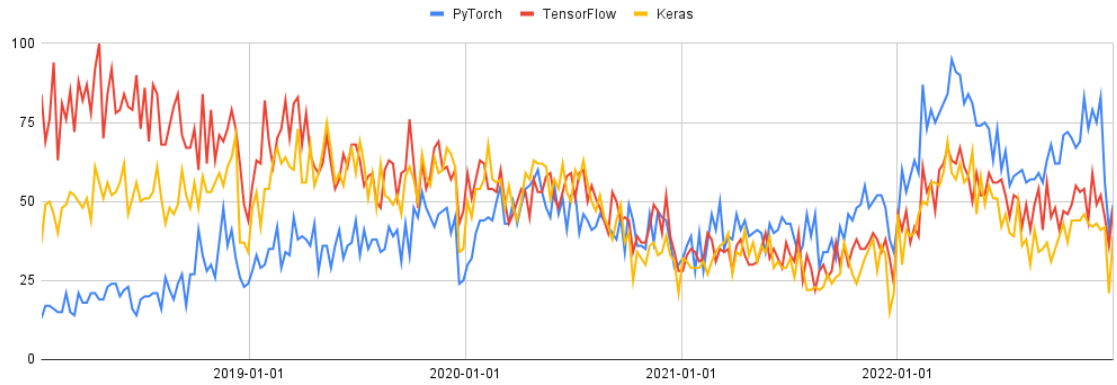

Fig. 1. Search trends of popular machine learning frameworks

There are many popular machine learning frameworks that have been widely adopted in the industry and academia. PyTorch is a flexible and intuitive platform for developing machine learning models, known for its dynamic computation graph and support for distributed training. It has been used in numerous research papers and industry projects. TensorFlow is an open-source platform for machine learning and deep learning applications, offering a wide range of tools and libraries for building and training machine learning models. It has been used in numerous research projects and real-world applications, such as image and speech recognition. Keras is a high-level neural network library that runs on top of TensorFlow and allows for easy and fast prototyping of deep learning models. It is a popular choice for researchers and practitioners due to its simplicity and ease of use.

2 PROPOSED SOLUTION

In our implementation, we focused on improving the efficiency of the forward and the backward process of a neural network, which are essential for making predictions and updating the model's parameters during training. The forward process involves transforming input data into output data through a series of computations, while the backward process computes the gradient of the loss produced by the loss function. This gradient is used to optimize the neural network using first-order optimization methods, such as **stochastic gradient descent (SGD)**, which is employed in our implementation.

In the forward process, the input data is transformed through a series of linear and nonlinear operations, such as matrix multiplication and element-wise function application. As the input data is processed in the neural network, the data needed for calculating the gradient in the backward process are stored in the memory buffer of each layer.

These data are accumulated as batches of data and passed through until the `zero_grad` function is invoked to clear the memory buffer.

In our implementation, there are three different modules, including a linear module, a sigmoid module, and a mean squared error (MSE) module.

- The linear module takes an input matrix x with dimension of $n \times m$ matrix where n is the batch size and m is the input dimension of the layer. It is multiplies the input data with a weight matrix w and added a bias β:

$$\hat{x} = x \times w + \beta$$

- The sigmoid module applies a element-wise sigmoid function to the input data:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- The MSE module computes the mean of the squared error for each of the m component. This module computes the loss between the output of the model and the desire output (usually the ground truth).

$$\text{loss}(x, y) = \frac{1}{m} \sum_{i=1}^{m} (x_i - y_i)^2$$

In the backward process, we use back-propagation [2] to compute the gradient of the loss function with respect to the model's trainable parameters. This algorithm utilizes the chain rule of calculus and computes the gradient from the last layer to the first layer.

For example, in our experiments, we implemented a simple 2-layer neural network with 2 linear modules, 2 sigmoid modules as activation functions, and a MSE module for the loss function. The final loss function can be expressed as:

$$\text{loss}(\text{sigmoid}(\text{sigmoid}(x \times w_1 + \beta_1) \times w_2 + \beta_2), y)$$

Where the x represents the input, y represents the ground truth, and w_1, w_2, β_1, β_2 are trainable parameters in the model. The gradient of the trainable variables can be calculated with the following equation:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \times \frac{\partial p}{\partial w}$$

where $\frac{\partial L}{\partial p}$ is the partial derivative back-propagated from the next layer and $\frac{\partial p}{\partial w}$ is the partial derivative of the current layer and the next layer. This process is propagated layer by layer until the gradient of the loss function with respect to all of the model's trainable parameters has been computed.

In order to provide a clear performance comparison between sequential and threaded version of our implementation without spending lots of time to run the experiment with single thread, we construct a simple neural network (NN) model in C++ (instead of Python, which is a popular language for machine learning). We implement the model in three different versions: single-threaded, multi-threaded and GPU.

The single-threaded version serves as a performance baseline for comparison, while the multi-threaded and GPU versions allow us to explore the relation between the improvement and the use of parallelization.

2.1 Multi-threaded Version - OpenMP

The first parallelized version we made in our experiment is the multi-threaded CPU version. Since we were aware that threaded CPUs might not be the most effective solution for accelerating machine learning algorithms, we chose to use

OpenMP as our development framework so that we can benefit from not only the easier developing experience but also the speed-up of parallel programming.

In this version, we applied parallelism using the simple `#pragma omp parallel for` directive in OpenMP on the forward, backward, and update process of the neural network. The performance of our NN model saw a significant improvement in execution time, demonstrating the power of parallel programming in accelerating machine learning algorithms. By utilizing multiple CPU cores simultaneously, we were able to significantly reduce the time it took to train our NN model on the dataset. The use of parallel programming techniques allowed us to leverage more computational resources of the system and achieve faster training times, making the machine learning algorithm more practical and efficient.

2.2 GPU Version - CUDA

For parallelizing the computation of our NN on the GPU, which is also the most adopted approach in real world applications, we used the NVIDIA-GPU specific programming platform, CUDA, to write the program and take advantage of its platform-specific optimizations for improved performance. Although there is a library implemented by NVIDIA called `cuDNN` that provides primitives for deep neural networks, we chose not to use it in order to explore the potential for optimization on our own.

To facilitate ease of memory management and optimize the use of resources, we transferred all the data that is needed to the GPU's global memory. In addition, we employed several parallel programming techniques such as padding and tiling to further optimize our implementation.

With all the optimization applied, we can see an obvious improvement in the performance of our program with respect to the multi-threaded version. The result also demonstrate why parallelzing machine learning algorithms with GPU is a common practice nowadays.

3 EXPERIMENTAL METHODOLOGY

In our experiment, we used a machine with an i7-13700K processor, 16GB of RAM, and a GeForce RTX 3090 GPU. The machine was running Ubuntu 22.04.1 LTS and had GCC 11.3.0 as the compiler and CUDA 11.8 installed. To ensure stable performance on the hybrid-core architecture, we limited the machine to using 8 cores.

3.1 Input Set - MNIST

MNIST is a well-known dataset in the machine learning community that consists of 60,000 labeled images for training and 10,000 images for testing. These images are grayscale and 28×28 pixels in size, with black represented by a value of 0 and white represented by a value of 255. The goal of the MNIST dataset is to correctly classify handwritten digits as one of the 10 digits. It is commonly used as a benchmark for evaluating the performance of machine learning algorithms.

In our work, we used a neural network to predict the handwritten digits in MNIST and compared the loss value and accuracy of our results to those obtained using the NN in the Torch library. We also measured the execution time of each method to assess the performance of our parallelization methods and techniques.

3.2 Experiment

We trained and evaluated the performance of our NN model on the MNIST dataset using single-threaded, OpenMP, and CUDA implementations. As mentioned earlier, the input data is represented as a $n \times m$ matrix, where n is the batch size and m is the input dimension of the layer. The batch size determines the number of samples processed at a time in

our model, while the hidden dimension, which is the input dimension of the hidden layer, determines the number of neurons in the hidden layers of the neural network.

The shape of the input matrix, which is determined by the batch size and hidden dimension, plays a key role in determining the degree of parallelism in the model. This is because the fundamental operations of neural networks, such as matrix multiplications and element-wise function applications, are highly amenable to parallelization because the nature of the independence.

As a result, we focused on these two important parameters: batch size and hidden dimension. The batch size plays a crucial role in determining the level of parallelism, while the hidden dimension affects not only the level of parallelism but also computational complexity. By adjusting these parameters, we were able to study the impact of parallelism on our model.

- Batch Size: To investigate the effect of batch size on the level of parallelism, we run the program with batch size of 2^k for $k = 0..15$ and measured the execution time to determine whether the parallelization was effective.
- Hidden Dimension: To evaluate the influence of the hidden dimension on the level of parallelism, we tested hidden dimensions ranging from 100 to 1000 and measured the execution time of our program. However, the hidden dimension also affects computation complexity, which we will discuss in more detail later.

4 EXPERIMENTAL RESULT

4.1 Single-threaded and OpenMP

In this section, we will compare the performance of the single-threaded version of our neural network and the parallelized version implemented with openMP. To further discuss the relation between the shape of input data and the degree of parallelism, we also conducted experiments with different batch sizes and different hidden dimension in the neural network.

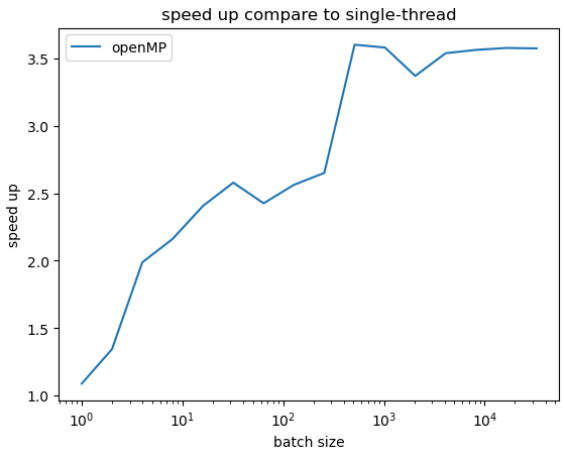

Fig. 2. Speedup of OpenMP (batch size)

In the result of the experiment, we conducted our experiment with a fixed hidden dimension 300 and can see that the degree of the parallelism increases as the batch size of the input data increases. This result shows that to fully utilize

the strength of our 8-core processor setting, larger batch size is required. In our 8-core processor experiment, we can see that the speed up of the openMP implementation increases as the batch size increases.

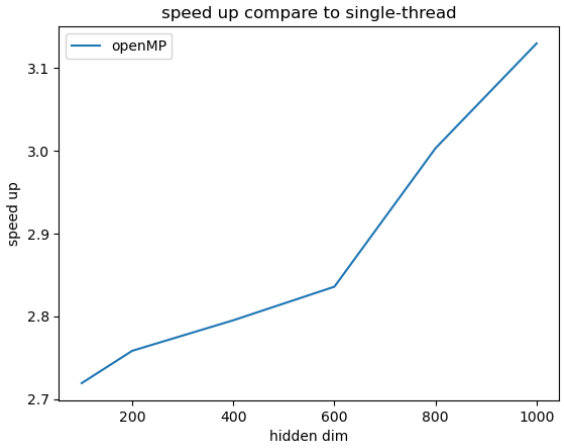

Fig. 3. Speedup of OpenMP (hidden dimension)

In the result of the experiment, we can see that with fixed batch size (1024), the degree of parallelism increases as the size of the hidden dimension increases. This result still show the importance of distribuing enough workload to each processor.

Therefore, in a system with multiple high performance processors, the workload is important when distributing the computation tasks, or else the speed up gained from using multiple processor could be slow down by the overhead of parallel programming.

4.2 Single-threaded, OpenMP, CUDA and Other Techniques - Batch Size

In this section, we will compare the time between single-threaded, OpenMP, CUDA, and CUDA with additional techniques. These techniques include memory padding, tiling, and tunning the size of the thread block.

4.2.1 Thread Block. In the CUDA programming language, a thread block is an abstrction that represents a group of threads that can be executed together. These threads can access a shared local memory and uses a barrier synchronization. In our implementation, for the matrix multiplication, we uses a 2D thread block for distributing the task, while for the other computation need, we uses 1D thread blocks. In our experiment, We tunned the shape of the 2D thread block from 4×4 to 32×32 and set the 1D thread block the same size of the 2D thread blocks.

In the result of the experiment, we can see that the performance is the best when the thread block is 8×8. We think that the reason behind this could be that this block size is close to the the size of a **warp**, which is the streaming execution unit in a NVIDIA GPU. While the other possibility for the performance decrease with larger block size and the bump in the figure is that the waste of computing resource. When we uses a large thread block, there could be more margin in the edge of the array, which would caused some threads have to wait for other threads to complete their job.

4.2.2 Padding. Padding is a common technique used to optimize memory usage and performance in parallel programming. It involves adding extra space between data elements in a data structure to avoid cache line conflicts, also known

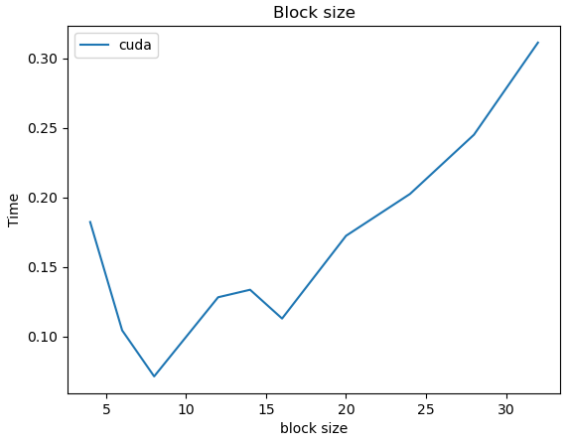

Fig. 4. Execution time of CUDA (block size)

as false sharing. False sharing occurs when multiple threads try to access different elements of a data structure that share the same cache line. This can cause the cache line to be repeatedly invalidated and fetched from main memory, leading to a performance penalty.

To avoid false sharing and reduce the time fetching data from memory, we add padding technique to our CUDA version. Compare with the CUDA(green line) and CUDA-padding(red line), CUDA-padding is faster than CUDA for batch size ≤ 512, and slightly slower than CUDA for batch size ≥ 1024. We thought that the curves are very close because the influence of padding is tiny.

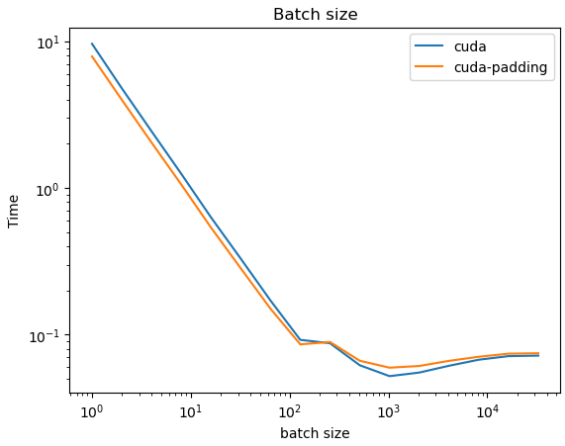

Fig. 5. Execution time of CUDA (batch size)

4.2.3 Tiling. Tiling is a common technique for GPU computing. This computing technique aim to leverage the fast local memory in the same thread block for fetching data that need multiple access to the global memory. Namely, this

technique copies the data from the global memory to the local memory so that the access of the data can be much faster. With such technique, we can speed up the matrix multiplication in our forward process.

From the result of this experiment, we can see that the acceleration with tiling actually save some time for out matrix multiplication.

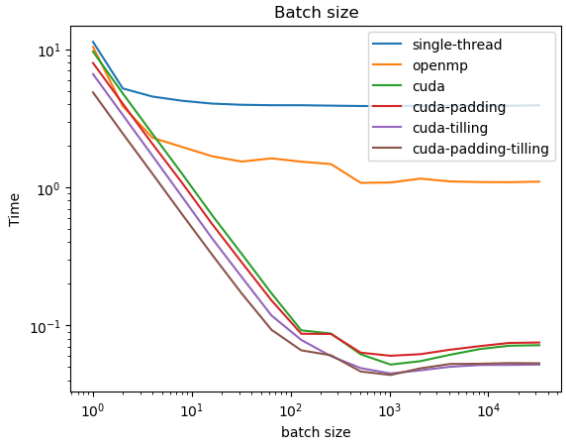

Fig. 6. Comparison of all approaches (batch size)

The single-thread and OpenMP curves are existing for time reference, in order to show how fast the CUDAs are.

4.2.4 Single Thread, OpenMP, CUDA and other techniques - Hidden Dimension. As the hidden dimension increases, the computation complexity should increase significantly, but it did not. The reason is that the increasing of hidden dimension also has contribution on the degree of parallelism, which reduce the time it need to do the large computation. As a result, the slope of our curves is not too large as we expected.

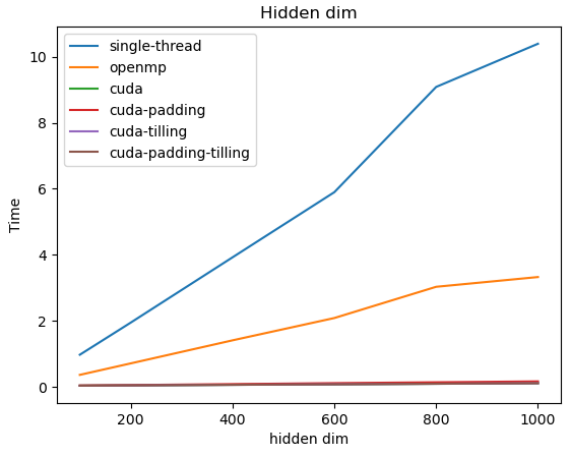

Fig. 7. Comparison of all approaches (hidden dimension)

4.2.5 OpenMP, CUDA with Other Techniques and LibTorch-GPU. In this section, we compare the best performing version of our implementation with torch library. The torch library have a great speed up comparing with our single-thread version. However, with our CUDA implementation with tiling, we have a better performance comparing with the torch library.

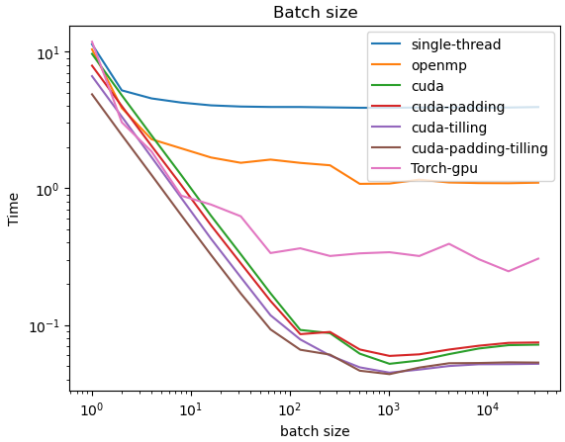

Fig. 8. Comparison of all approaches (comparison with LibTorch)

5 RELATED WORKS

We chose to compare our work with PyTorch[1], one of the most popular machine learning frameworks which is implemented using `cuDNN`[**?**], to see how our approach compares in terms of performance. We trained our model on the same dataset and hyperparameters as a PyTorch version. We run the comparison on the 8-core, 16GB RAM, RTX 3090 machine.

Our implementation was relatively fast when compared to the PyTorch version, but we attribute this to the overhead of the Python runtime. To eliminate this factor, we used libtorch, the C++ frontend of PyTorch, for the comparison.

Our implementation was slightly faster than the libtorch version, the difference in runtime between our implementation and the libtorch version was not as large as the difference between our implementation and the PyTorch version, which supports our assumption that the Python runtime contributes to the slower performance of PyTorch.

Overall, our implementation was competitive with PyTorch and libtorch in terms of performance, and the results of the comparison confirmed our assumption that the Python runtime contributes to the slower performance of the PyTorch version.

6 CONCLUSION

From the result of our experiments, we found several points when using CUDA with nerual network or other similar computation task, including:

- Distributing enough task for your processor to fully utilize the strength of your system.
- Tunning the thread block size of your GP-GPU or changing the way that you partition your workload may gain a fairly nice speed up.

- The tiling technique is useful when the computation task requires multiple access to the global memory.

Finally, we compare our implementation with the torch library and gain a decent speed up. This result shows that implementing your own version of the library and optimize the specific task you are dealing with may have the potential to beat the existing library built for solving more complicated task.

REFERENCES

[1] Francisco Massa Adam Lerer James Bradbury Gregory Chanan Trevor Killeen Zeming Lin Natalia Gimelshein Luca Antiga Alban Desmaison Andreas Köpf Edward Yang Zach DeVito Martin Raison Alykhan Tejani Sasank Chilamkurthy Benoit Steiner Lu Fang Junjie Bai Soumith Chintala Adam Paszke, Sam Gross. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. (Dec. 2019). https://doi.org/arXiv:1912.01703

[2] Geoffrey E. Hinton & Ronald J. Williams David E. Rumelhart. 1986. Learning representations by back-propagating errors. (Oct. 1986). https://doi.org/10.1038/323533a0

[3]]cuDNN NVIDIA. [n. d.]. *NVIDIA cuDNN - GPU accelerated deep learning.* https://developer.nvidia.com/cudnn