High Performance Computing

Project: Accelerating Neural Network Training

Members: Ariyan Chaudhary(i221035) & Ali Naveed (i220778)

Abstract

This report shows analysis of acceleration attempt at the Neural Network training program. Using Gprof Profiler, the program was profiled and *hot-spots* were analyzed. The acceleration targets three compute intensive functions: *forward()* and *backward()*. There were 4 different attempts made at accelerating the same algorithm. Starting with a naive implementation that achieved merely ~0.5 second speed-up per epoch on both functions, the final version (v4) effectively parallelized the computations to almost negligible compute time, leaving only the overhead of data communications as time consuming. Additionally, two separate versions were created for exploring further modes of parallelism. It was shown in v3.1 instead of processing one image per iteration, using a batch of images provided more room for acceleration. The v5 used high-level directives instead of explicitly defining the device kernels and parallelizing strategy.

Introduction

The original program is a typical Neural Network used in deep-learning implemented purely in C language. It uses MNIST dataset for training and testing separately. Sixty thousand labeled images are used for training the neural network, and ten thousand separate images are used to test the accuracy of the trained network. Setting the *srand* seed to be 777, the accuracy of the program reaches up to ~96% and takes around ~23 seconds to finish three epochs.

This project focuses on accelerating two main algorithms implemented in the program, namely the forward propagation and backward propagation. Using Gprof profiler, it was revealed that both of these functions are the main compute intensive components of the program. It was observed that their total compute percentage reaches almost to 99%. By offloading these two functions to the CUDA device, the program could effectively be parallelized and used in practical applications where increasing number of epochs and dataset exponentially raises the computations.

Application Profile and Analysis

Profiling Methodology:

Tool: Gprof profiler used to analyze the CPU implementation.

Findings: *The results were averaged over multiple iterations.*

- forward: **~72%** of total runtime.
- backward: **~27%** of total runtime.

Combined: *~99%* of total runtime.

The remaining time is negligible and distributed among I/0 functions like loadMNISTImages that reads the dataset and the parent train function which loops over the images to call these forward and backward functions.

Estimated Speedup Limit (Amdahl's Law):

The target functions (forward and backward) add up to approximately 99% of the total compute time. As such, **P** would be *0.99.*

Applying Amdahl's law: $\frac{1}{(1-P)+P/S}$

To find the speedup limit, putting S = *inf*. As such: $\frac{1}{(1-0.99)+\frac{0.99}{inf}}=\frac{1}{(0.01)+0}=100$

Following these calculations, the theoretical *speedup limit* was calculated to be around ***100x.***

However, that is almost impossible to achieve as no part of the program could be sped up to infinity. The communication latency between CPU and GPU is also a big consideration. It was observed that the target functions are called inside a loop that iterates over each image in the dataset *(60k)*, as such the kernel call overhead was bound to limit the acceleration.

Implementations

1. Naive Approach (V2):

This version was implemented to only compare against nuanced optimizations integrated in further versions.

Data management: Transferring the images and labels onto device was done at the start of the *train()* function. Sending each image separately in the loop would only pile up overhead latency.

It must be noted that the network weights and biases were transferred separately as contiguous memory.

Additionally, two arrays, '*output'* and '*hidden'* were created directly on the device side. After backward function finishes, cudaDeviceSynchronize is called and the '*output*' is copied to host for calculations.

Parallelizing Strategy: The forward function was divided into two kernel calls: *compute_hidden* and *compute_output*. The backward function, on the other hand, was divided into six kernels.

compute_hidden kernel also computes the *relu* function as it has the same loop with same access pattern.

compute_output kernel allows only the first thread to calculate the softmax functionality inside itself. Creating a separate kernel for it added more overhead.

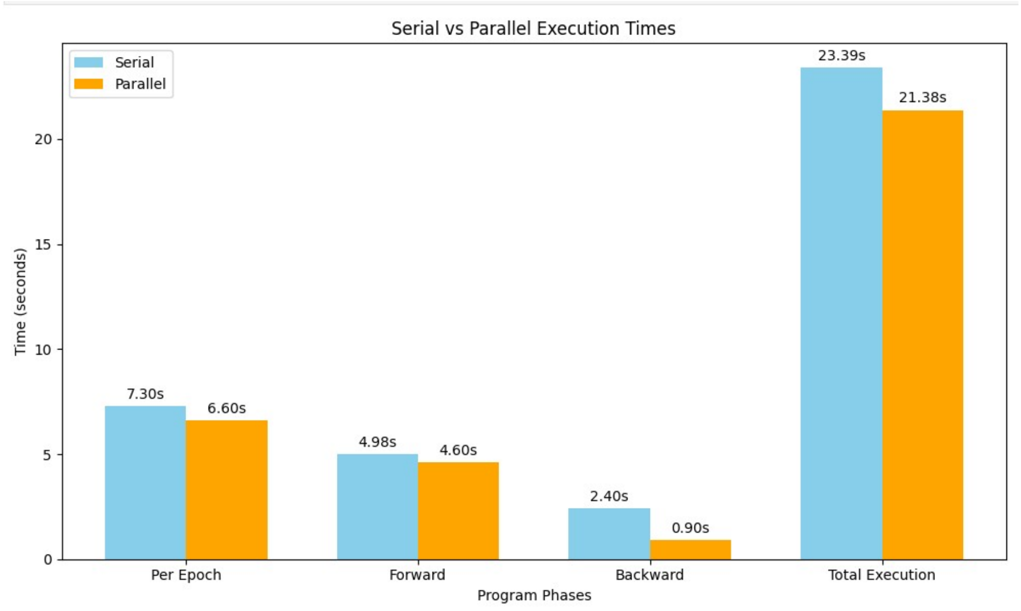
Same with the six kernels inside the backward function, each main loop was created into a kernel.

Results: After this implementation, even as a naive implementation, the program was accelerated by a slight margin. Meanwhile, the accuracy of the neural network remained the same.

	V1 (in seconds)	V2 (in seconds)	Speed up
Per Epoch	~7.3	~6.6	1.1x

Forward	~4.98	~4.6	1.08x
Backward	~2.4	~0.9	2.6x
Total Execution	~23.39	~21.38	~1.1x

It can be visually seen below:



As can be seen from the graph, there was slight increase in all components. However, the effort required to parallelize this program and the outcome achieved does not make it seem worth it. Further optimizations would fix that.

2. Optimized Approach (V3):

This version uses advanced optimization techniques to further accelerate the program.

Data management: Similar to the naive approach, the images and labels were transferred to the device, the weights and biases were sent there as contiguous memory.

Optimizations:

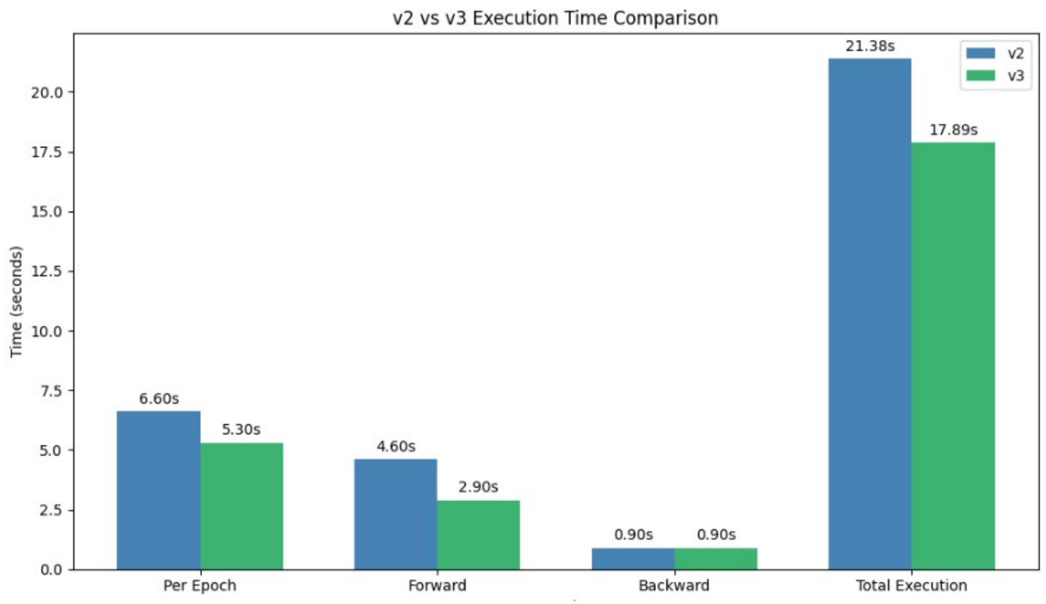
- Shared memories were used in bother kernels of forward(compute_hidden, compute_output)
- Occupancy was calculated for different launch configurations and used the most optimal ones.
- Cuda streams were used to asynchronously copy the output array back from the backward kernel while letting the other three do their computations.
- Memory coalescing was also performed, although it did drop the accuracy by ~4 percent.

Results:

	V2 (in seconds)	V3 (in seconds)	Speed up
Per Epoch	~6.6	~5.3	1.24x

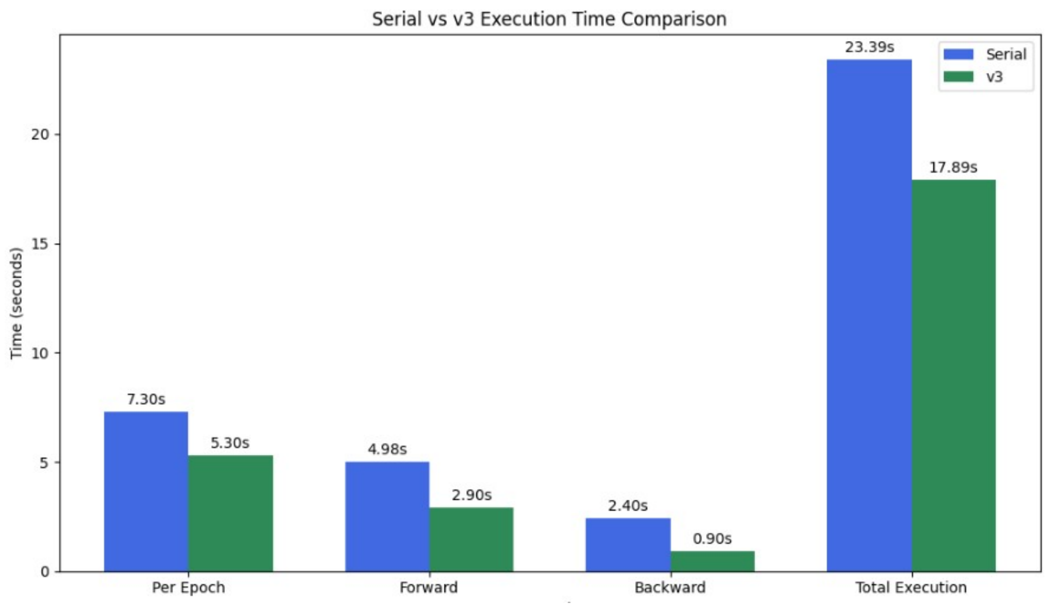
Forward	~4.6	~2.9	1.58x
Backward	~0.9	~0.9	1x
Total Execution	~21.38	~17.89	~1.2x

It can be visually seen below:



As can be seen, compared to the naive implementation, there is further speed-up achieved.

If compared to the serial version, the speed-up appears more pronounced.



Still, It should be noted that despite optimizations, the acceleration remains minimal.

Further analysis showed that with 60k images, and a total of 8 kernels, there was bound to be a limit to how much speed-up could be achieved as total kernel calls reaches up to 480k!

Considering these limitations, the efforts were focused on parallelizing the very loop(iterating over each image) that calls the forward and backward functions.

3. Further Optimizations (v 3.1):

It remains a separate version because the parallelizing strategy steps outside the domain of HPC and focuses on the algorithm used in the Neural Network training called Stochastic Gradient Descent (SGD). This algorithm used in our program goes over each image separately to update the weights and biases.

As it turns out, this strategy provides poor parallelization room. If the loop over images is divided such that a batch of images are processed at the same time, most of the calculations remains same, but the soul of the algorithm changes to that of the Mini Batch Gradient Descent.

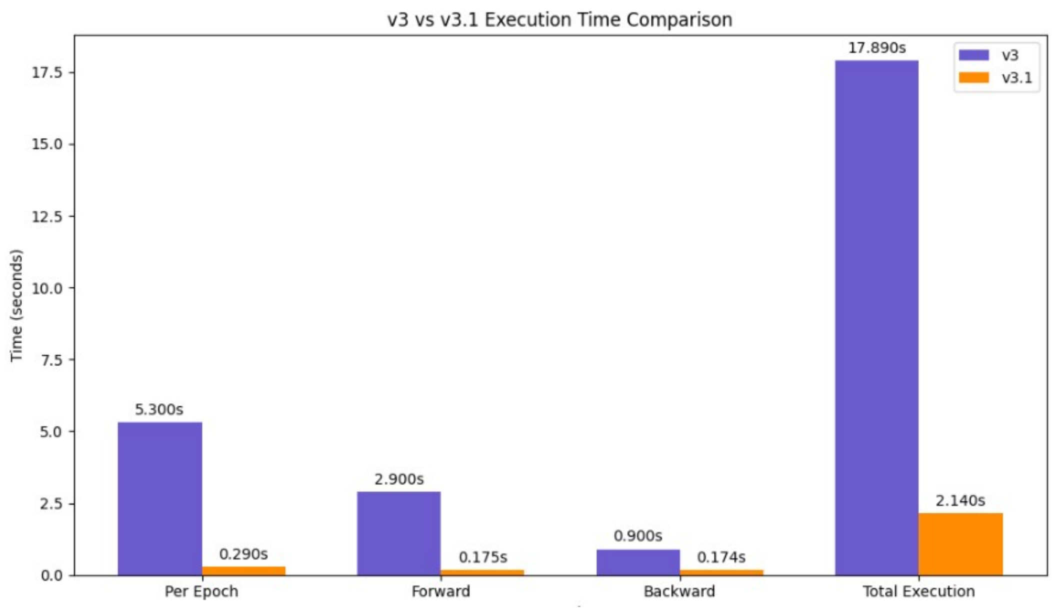
Optimization: In this version, the program uses a batch of images to update the weights instead of single image each time.

This, in effect, not only reduces the kernel calls by a margin of BATCH_SIZE, but also utilizes better resources from the device.

However, as the algorithm itself has changed, there are also variations in the accuracy. That isn’t a problem in this program since adding a few more epochs has a manageable trade-off.

Results: *5 epochs were used instead of 3 to maintain accuracy.*

	V3 (in seconds)	V3.1 (in seconds)	Speed up
Per Epoch	~5.3	~0.29	18.27x
Forward	~2.9	~0.175	16.5x
Backward	~0.9	~0.174	5.17x
Total Execution	~17.89	~2.14	~8.36x



Speed up from Serial: *23.39/2.14 → **~10.9x***

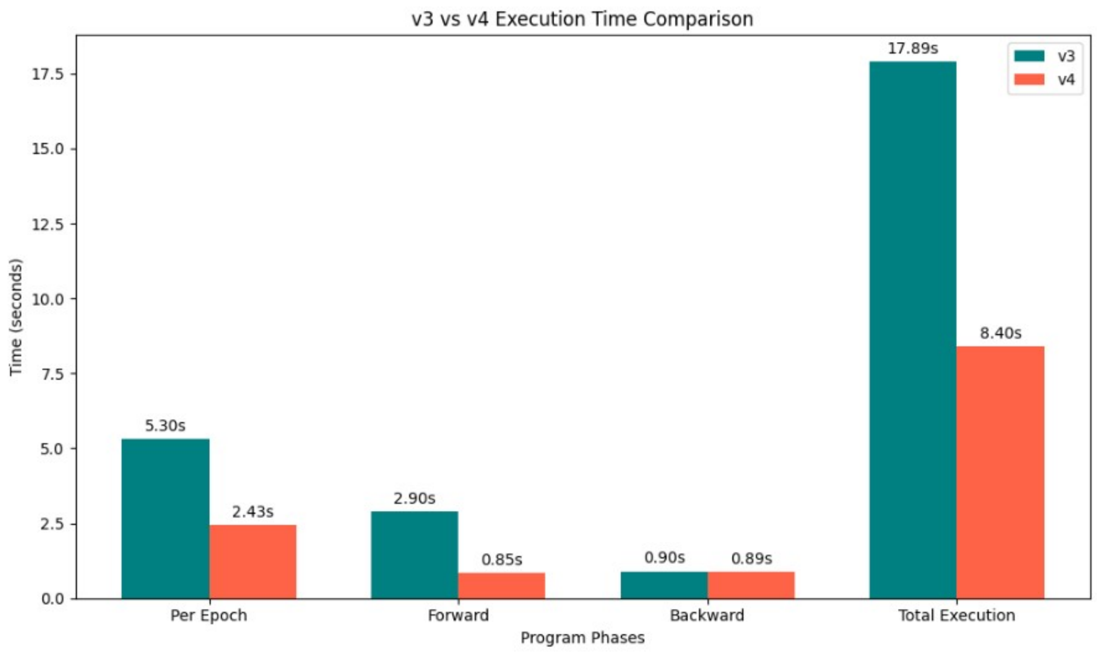
4. Further Optimizations (v 4):

In this version, the program effectively utilizes the tensor cores using cublas functions. It uses the *cublas_v2.h* library for using the functions. This tensor core implementation is applied on the original algorithm SGD.

Optimization: The matrix multiplications from both forward and backward functions were identified and using cublasSgemm wrapped in an error checking function definition, they were offloaded to the tensor cores.

Results:

	V3 (in seconds)	V4 (in seconds)	Speed up
Per Epoch	~5.3	~2.43	2.2x
Forward	~2.9	~0.85	3.6x
Backward	~0.9	~0.89	1.01x
Total Execution	~17.89	~8.4	2.12x



Speed-up from the serial version: *23.39/8.4 → **2.78x***

5. OpenACC implementation (v 5):

This version builds upon the serial version and uses the pragma directives over loops of the forward and backward functions to parallelize them.

Optimization: Using copyin pragma directions, the dataset images, labels, and network weights and biases are offloaded to the device at first.

Following that, the compute-intensive loops inside both forward and backward functions are wrapped inside “acc parallel loop” pragma directive which automatically offloads these functions to the device.

Results:

	V1 (in seconds)	V5 (in seconds)	Speed Up
Per Epoch	~7.3	~13.3	0.54x
Forward	~4.98	~4.63	1.08x
Backward	~2.4	~8.01	0.3
Total Execution	~23.39	~40.8	~0.57x

As can be seen, we could not achieve a speed up using the pragma directives provided by OpenACC.
There was some speed-up in case of the forward function, but the backward function slowed down the over all program.

Conclusion: The naive implementation served as the basis over which further optimizations were performed. However, even the optimizations such as shared memory and memory coalescing did not provide significant improvements. Calling the kernels hundred of thousands of times put a bottleneck at the acceleration which could not be overcome without changing the algorithm itself. Using mini batch gradient descent to process multiple images at the same time allowed effective parallelization. In the fourth version, using the tensor cores for matrix multiplications also showed promising results. Finally, the fifth version that used pragma directives provides great abstraction, but limited to using the same algorithm, we could not achieve a speed-up. Instead, our program slowed down. It was concluded that the best implementation was still the v3.1 which processed multiple images at the same time as it provided *~10.9x* speed up!