**Parallelization of Neural Networks**

**Team: Ajay Mandadi, Kartik Angadi, Sumedh Ravi**

**Github link:** [**https://github.gatech.edu/sravi71/ParallelNeuralNetwork**](https://github.gatech.edu/sravi71/ParallelNeuralNetwork)

**Background**

A neural network contains layers of nodes, that take input and in conjunction with information from other nodes, develop output without programmed rules. A **multi-layered perceptron** is one of the most common neural network models used in the field of deep learning. It is used for a variety of tasks, such as stock analysis, image identification, spam detection, and predicting house prices. The multi-layered perceptron can be divided into 3 main layers, the input layer, the hidden layer(s), and the output layer. The forward propagation algorithm is used to transmit data from the input layer to the output layer by calculating outputs for its output nodes, given the input. The backpropagation algorithm is used to optimize the performance of the model by sending the difference between the predicted outputs and the actual outputs, back through the network, and accordingly readjusting the weight matrices so that eventually, the error between the actual and expected output is minimized.

**Approach**

Making neural networks more computationally efficient will allow us train larger networks on bigger datasets in a shorter amount of time. Although, there are sequential dependencies between each layer in the network because the computation depends on the results of the previous layer, neurons within the same layer are independent from each other so their computations can be done in parallel. Our approach was to parallelize the training stages of a neural network ***to classify handwritten digits from the MNIST dataset***, with multi-threading using **OpenMP** and try to utilize multiprocessors on the GPU using **CUDA**.

We created classes to represent a neural network, a layer in a network and a node in a layer. (*Refer Perceptron.cpp Lines* 2*7-60*). We defined functions to read the image data which is stored as pixel values and the label data. We created classes to represent a neural network, a layer in a network and a node in a layer. We prepared the data in a format that is suitable to be fed into the model. (*Refer Perceptron.cpp Lines 228-318*).

We used a Stochastic Gradient Optimization Policy, which means that gradients are calculated after processing each image. Initially, we implemented a sequential neural network model in C++ which served as the baseline model to compare with. This implementation takes in an input image, runs the matrix computations, predicts the output, calculates the error and then backpropagates this error to update the weights. It then moves to the next training image, and so on. One run through all the training samples constitutes one epoch. Our training dataset consists of 60,000 such images and the test set has 10,000 images. We used a single hidden layer of size 100 and one output layer that outputs the probability of the image being each of the 10 digits. (*Refer Perceptron.cpp Lines 326-450*)

Next, we moved on to the OpenMP implementation, where we train the same model on different images in parallel on different threads. But this also means that the backpropagation of the errors from different images will result in the models on different threads having different weights. Hence, we synchronize these different weights by averaging out the corresponding weights from the models on the different threads. To reduce the time lost due to synchronization, we run a batch of images of size 100, sequentially on each thread and average the weights after a batch of images complete execution on each thread.

In the CUDA implementation, we parallelized the forward propagation step by running the computations of one node in the layer on a different thread. (*Refer cuda\_perceptron\_single\_thread.cu lines 142-201*). This allows us to compute the outputs of even larger layers in almost the same time as a single node of computations. However, we noticed that the performance was held back by the time taken to copy the inputs and weights to and from the GPU and this leaves us room for further optimizations in the future.

**Experimental results and analysis**

Chart, line chart

Description automatically generated

We can observe that the training duration in the OpenMP implementation is significantly faster than the sequential implementation. The training in the OpenMP version completes almost 16 times faster than a sequential model, when executed on 24 threads. However, the CUDA implementation of the Stochastic Gradient Descent training process took much longer than both the sequential and OpenMP implementations. We process each node in a layer of the model on a different thread and copy the results back to the device. One of the main reasons for this is the time taken to copy the weights and image input from the host memory to the device memory. To alleviate this problem, training on a GPU is usually carried out on a batch of inputs parallelly, and propagating the average of the gradients, resulting in a different type of optimization technique called Mini-batch Gradient Descent. Here each block in a GPU would be allocated a training sample to process. This would allow us to process a batch of images in almost the same time as a single image which would considerably reduce the model training time. However, we might have to train the model for a greater number of epochs.

Chart, line chart

Description automatically generated

In the OpenMP implementation, we can observe a slight drop-off in the accuracy metrics compared with the sequential and CUDA implementations, when trained for the same number of epochs. This is because we run a batch of images sequentially on different threads and average the weights at the end of the batch on all the threads. Hence, the same model does not process all the images, and by averaging the weights we tend to reach the optimum at a slower rate. In the sequential and CUDA implementation, a single model processes each of the 60,000 training images. Therefore, all the images are used to backpropagate error, calculate gradients, and update the weights of the model. However, the drop-off is not very alarming, and the results are comparable with the sequential implementation of the model. To obtain better results, we might have to consider using a different learning rate or training the model for a higher number of epochs.

**Future Scope**:

We were able to just parallelize the forward pass of the neural network on CUDA, however parallelizing all the matrix operations while optimizing the data transfer process between device and host would give us a better performance with respect to speed. Also processing the images in a batch would help us to fully utilize the compute power of the GPU.

**References**:

<https://github.com/BobbyAnguelov/NeuralNetwork/tree/master/Src/NeuralNetwork>

<https://github.com/mitesh1612/Parallel-Neural-Networks>

<https://medium.com/@udaybhaskarpaila/multilayered-neural-network-from-scratch-using-python-c0719a646855>