Parallelizing Neural Networks with OpenMP and CUDA

A Comparison Of Implementations and Optimizations

Luis Guzman
University of Oregon
lguzmann@uoregon.edu

Steven Walton
University of Oregon
swalton2@cs.uoregon.edu

Abstract

With the major advances and gain in popularity of machine learning it has become increasingly important to optimize and parallelize these types of code. Much of the mathematics behind machine learning is based on tensor operations, specifically tensor multiplication. This creates a ripe environment for the study of parallelization techniques. Within this project we study the effects of OpenMP and CUDA acceleration within an Artificial Neural Network. We draw on design inspiration from popular machine learning frameworks such as PyTorch and TensorFlow to better understand how they work under the abstraction layers that they provide. Through our implementations we found an overall speed-up of $4.5x$ *through the use of OpenMP but were unable to find speedup due to CUDA implementation. Within this paper we discuss our successes and failures in the implementations of accelerating technologies.*

1. Introduction

In the past decade, Artificial Neural Networks (ANNs) have become increasingly popular, with advances in the area affecting almost every sector of computing. Deep Neural Networks (DNNs) – deeper, more complex models than their 'shallow' counterparts – have progressed from being academic curiosities to being the cornerstone of many modern technologies [9]. Given this newfound relevance, many researchers have devoted a lot of time optimizing these algorithms, constantly trying to improve them for different accelerator technologies and platforms. Even long before accelerators, many efforts in trying to parallelize neural networks were made [2]. While the new resurgence of machine learning (ML) partially came from the generalized availability of hardware accelerators like GPUs, there are still many innovations being made for CPUs. An example of this is Facebook's new Speech-to-Text (stt) system that runs on CPUs [3] and is aimed at home assistants that cannot rely on large and power-hungry accelerators. As ML algorithms are getting larger, there are also continued efforts to apply machine learning techniques to simpler hardware. Thus, for the foreseeable future, optimizing ML algorithms on various platforms will remain an active area of research.

In this section we will discuss the various algorithms used in machine learning. In Section 1.1 we will discuss Artificial Neural Networks and the underlying theory behind them. In Section 2 we will discuss how we implemented our network and our design choices. In Section 3 we will discuss the results and the optimizations we achieved from our parallelization efforts. Finally, in Section 4 we conclude the work and summarize our findings.

1.1. Artificial Neural Networks

This section presents the theory behind ANNs and the Gradient Descent (GD) algorithm that is commonly used to train them.

1.1.1 Perceptrons

The Perceptron, also referred to as "neuron", is the basic unit of an ANN. It is a mathematical function that was inspired by the biological neurons in the human brain [7]. Its mathematical formula is presented in Equation 1. It consists of a dot product between a feature vector x and a weights vector w. The resulting

dot product of these vectors is then fed into a function g known as *activation* [5] which yields the output $\hat{y}$ of the neuron.

$$\hat{y} = g\left(\sum_{i=0}^{n} w_i x_i\right) \tag{1}$$

Perceptrons can be used to solve regression (estimating real values) or classification problems (discrete number of classes). The feature vectors are obtained directly from the function being estimated or the dataset being classified. The weights vector can be initialized to a vector of zeros, however, it is common to initialize it using random numbers in the range $[0, 1)$ drawn from a uniform distribution. By providing random initializations networks typically converge faster and generalize better.

As for the activation function, Figure 1 presents a few examples of commonly used ones [5]. The bottom-right image in Figure 1 shows an example of a linear activation in the form of the Identity function. The other three examples represent non-linear activations: the Sigmoid (bottom-left) function yields an output between 0 and 1; the Hyperbolic Tangent function (up-left) has an output between -1 and 1; and the Rectified Linear Unit, or ReLU, (up-right) is a modified Identity that changes negative values to 0.

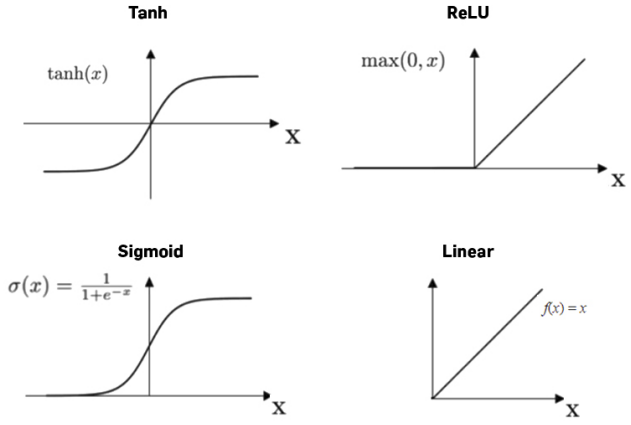

Figure 1. Examples of activation functions.

The activation function plays a crucial role because it actually determines the sort of problems that a perceptron is able to solve. If a linear (Identity, Step, Sign) activation function is used, a single perceptron is only able to solve "linearly separable" problems, that is, problems in which the classes can be successfully separated by a hyper-plane (Figure 2 left). On the other hand, by using a non-linear (Sigmoid, Tanh, ReLU) activation a more flexible decision boundary is created and, thus, more complex problems can be solved (Figure 2 right).

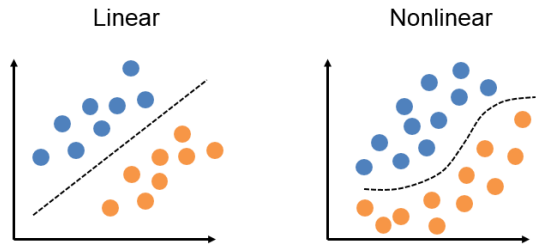

Figure 2. Linearly-separable vs non-linearly separable.

1.1.2 Training Perceptrons with Gradient Descent

It is safe to assume that the initial output of the perceptron $\hat{y}$, obtained with the initial state of the weights vector, will be different than the expected output y (label). So, in order to improve these predictions, the perceptron needs to be "trained". Training a perceptron basically means iteratively modifying the weights vector so that the outputs start to approximate the expected ones. This training process is illustrated in Figure 3: using the the expected output y and the obtained output $\hat{y}$ an error measure is computed and used to modify the weights. This process is repeated until the outputs fulfill a given convergence criteria or a maximum number of iterations is reached.

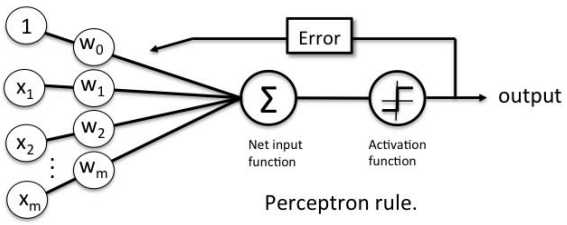

Figure 3. Perceptron training.

The Gradient Descent (GD) algorithm [8] is an optimization technique that is used to train perceptrons.

The main idea behind GD is to update the weights in the direction of the gradient of the function used to compute the error between the perceptron's outputs and the expected ones. This function is commonly referred to as *Loss* function [4]. The actual problem being solved determines the type of loss function that should be used. For example, in regression problems the Mean Squared Error (MSE) loss function is a typical choice. For binary classification problems, Binary Cross-Entropy loss or Hinge loss are good options. In multi-class classification problems, Cross-Entropy loss is the default alternative.

The general formula for the GD method is shown in Equation 2. First, the derivative of the chosen loss function L with respect to the weights vector w is obtained. Then it is multiplied by a scalar parameter γ known as *learning rate*. Finally, the weights are updated using this product.

$$w = w - \gamma \frac{d}{d_w} L \tag{2}$$

While the formula for GD is straightforward, obtaining the derivative of the loss function with respect to the weights is not always trivial as the chain rule of derivatives is needed. For example, Equation 3 shows the mathematical formula of the Binary Cross-Entropy loss function.

$$L(w) = y * \log(\hat{y}) + (1 - y) * \log(1 - \hat{y}) \tag{3}$$

If we assume a perceptron using the sigmoid activation function, the derivative of the loss is obtained in the following manner:

$$d = w^T x \tag{4}$$

$$\hat{y} = \sigma(d) = \frac{1}{1 + e^{-d}} \tag{5}$$

$$\frac{\partial L(w)}{\partial w} = \frac{\partial L(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial d} \cdot \frac{\partial d}{\partial w} \tag{6}$$

$$\frac{\partial L(w)}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right) = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \tag{7}$$

$$\frac{\partial \hat{y}}{\partial d} = \hat{y}(1 - \hat{y}) \tag{8}$$

$$\frac{\partial d}{\partial w} = x \tag{9}$$

$$\frac{\partial L(w)}{\partial w} = x^T(\hat{y} - y) \tag{10}$$

There are some variations to the GD algorithm, each with its own strengths and weaknesses. In Stochastic Gradient Descent (SGD) [8], the weights are updated for each training sample in the dataset, i.e. the output of one feature vector is obtained, the derivative of the loss function for that output is computed, and the weights are updated according to the GD formula. This version is, of course, suitable for online training or when the training dataset is so big that it cannot fit in memory all at once. However, the constant weight updates produce a the chaotic behaviour shown in Figure 4 as each training sample attempts to pull in its own direction.

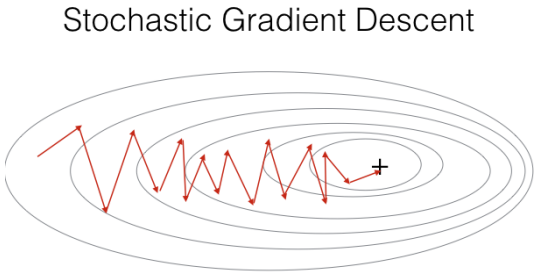

Figure 4. SGD convergence behaviour.

The (Mini) Batch Gradient Descent (BGD) [8] variation instead splits the training samples into small-sized batches and only updates the weights once per each batch. Thus, using BGD makes the algorithm behave more smoothly with less pronounced changes in its path towards convergence as it is shown in Figure 5.

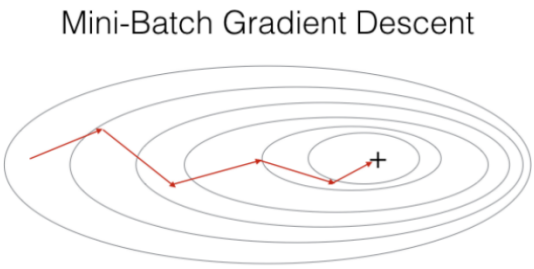

Figure 5. BGD convergence behaviour.

1.1.3 Multi-layer Perceptrons

There are, of course, limitations to what a single perceptron can accomplish no matter the activation function used. Figure 6 presents one of such cases in which a classification problem cannot be solved by a single perceptron. The dataset shown in the figure contains points in two dimensions labeled as two distinct classes identified by the colors blue and orange. As can be observed, the blue class is "contained" within the orange class and, as such, it is impossible for a single perceptron to define a decision boundary, even a non-linear one, that will correctly classify all of the points.

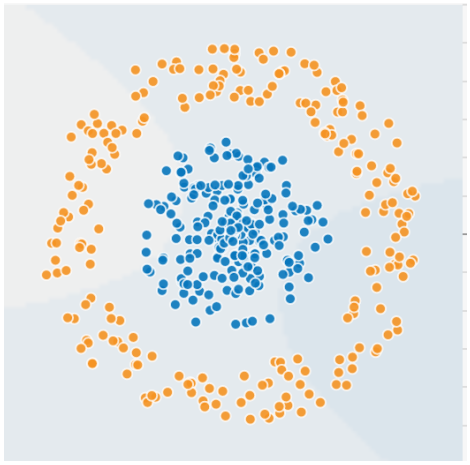

Figure 6. Class contained within the other.

Figure 7 presents the decision boundary created by a single perceptron using a sigmoid activation. In the figure, it can be observed that the resulting decision boundary correctly classifies most of the blue-class points. However, it incorrectly classifies the vast majority of the orange-class points.

In problems such as the this one using an ANN, also known as Multi-Layer Perceptrons (MLPs), be-

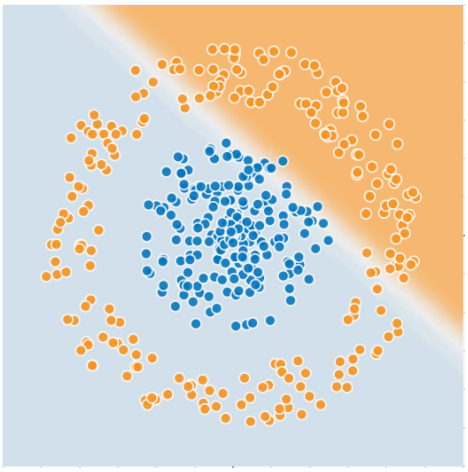

Figure 7. One perceptron solution.

comes necessary. A MLP consists in several perceptrons connected together to form a "network". In the literature, there exist multiple ways in which the perceptrons can be arranged to form different architectures [9]. However, in this work we focus on the architecture known as "Feed-Forward Neural Network" (FFNN) [10], shown in Figure 8. In FFNNs, all neurons in one layer are connected to all of the neurons in the following layer, hence the name Feed-Forward.

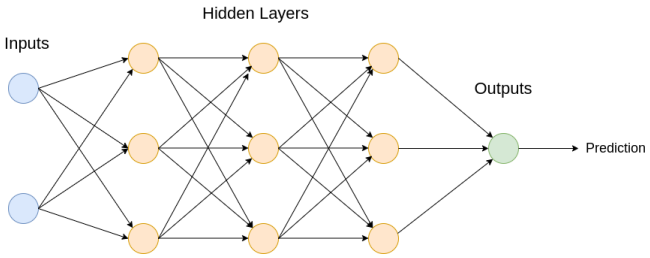

Figure 8. Structure of a FFNN.

In a FFNN, perceptrons are organized into 3 types of layers: input, hidden, and output layers. Input layers receive the feature vectors as inputs and connect to the first hidden layer. Hidden layers are intermediate layers that make up for the bulk of the network. Finally, the output layer is tasked with producing the result of the network.

Each layer in a FFNN can have a different number of neurons, and each neuron within a layer can have a different activation function. It is usual, however, for neurons in the same layer to have the same activation.

The number of neurons in the output layer and their activation types depend on the type of problem being

solved. For a regression problem, the network only has one output neuron, usually with an identity linear activation. In binary classification problems, one output neuron is sufficient. A sigmoid activation is also usually used in these cases, with labels 0 and 1 used to represent the two classes. For multi-class classification, the number of neurons in the output layer matches the number of classes being identified and a *Softmax* [1] (Equation 11) operation is applied to the results of the output layer neurons' to create a probability distribution vector. The class prediction is then obtained by obtaining the argmax of such probability vector.

$$Softmax(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_j e^{\hat{y}_j}} \tag{11}$$

Figure 9 shows the decision boundary created by a FFNN with 2 hidden layers of 3 neurons each trained on the dataset discussed previously.

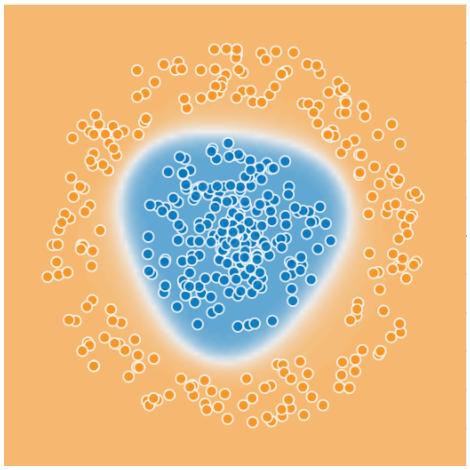

Figure 9. Solution with MLP.

To make predictions in a FFNN network, the outputs of each layer, also referred to as *activations*, are obtained sequentially. The activations of the input layer, obtained using the feature vectors, are in turn used as the inputs for the first hidden layer. This process known as the *Forward Pass* is repeated until the activation of the output layer is computed (Figure 10).

When using BGD, the forward pass through a FFNN becomes a series of matrix multiplications. The inputs become a matrix with dimensions [*Batch_Size* × *#_Features*] and at each layer, the activations are obtained by multiplying the input matrix by the weight

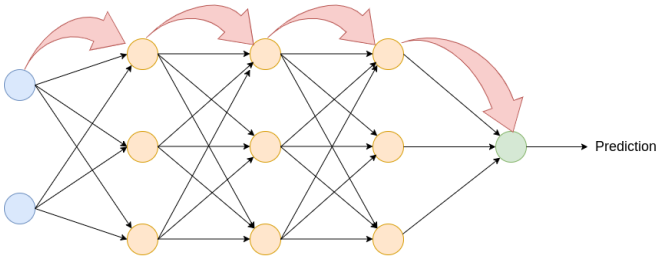

Figure 10. Forward pass.

matrix. Equations 12-15 show the forward pass for a network with 3 hidden layers and one output layer. Note: these equations purposely omit the neuron's activations as the objective is to highlight the matrix operations.

$$h_{1_{act}} = Inputs \cdot W_{h_1} \tag{12}$$
$$h_{2_{act}} = h_{1_{act}} \cdot W_{h_2} \tag{13}$$
$$h_{3_{act}} = h_{2_{act}} \cdot W_{h_3} \tag{14}$$
$$\hat{y} = h_{3_{act}} \cdot W_O \tag{15}$$

Updating the weights in a FFNN is more complex than with a single perceptron. The loss function merely determines the error at the output layer of the network. It is then necessary to propagate such error through every layer in the network all the way back to the input layer. To achieve this, the *Backpropagation* algorithm [6] is used (Figure 11).

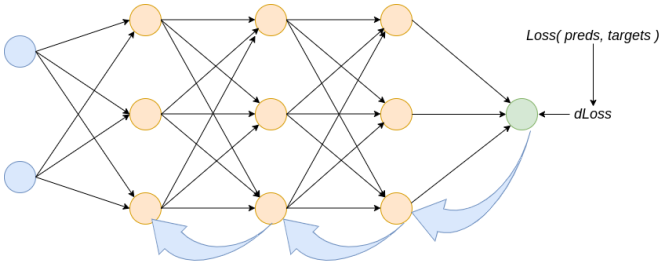

Figure 11. Backpropagation of the error.

Once the derivative of the loss function is obtained at the output layer, the error is propagated back to the preceding layer by multiplying it by the output layer weights. This same procedure is repeated until the error has reached the input layer (Equations 16-19). Note: as before, these equations omit the derivative of

the activation functions as the focus should be on the matrix operations.

$$O_{error} = dL_{oss} \tag{16}$$
$$h_{3_{error}} = O_{error} \cdot (W_O)^T \tag{17}$$
$$h_{2_{error}} = h_{3_{error}} \cdot (W_{h_3})^T \tag{18}$$
$$h_{1_{error}} = h_{2_{error}} \cdot (W_{h_2})^T \tag{19}$$

After the errors for all layers have been computed, the weights can be updated using the GD formula (Equations 20-23).

$$W_{h_1} = W_{h_1} - \gamma((I_{nputs})^T \cdot h_{1_{error}}) \tag{20}$$
$$W_{h_2} = W_{h_2} - \gamma((h_{1_{act}})^T \cdot h_{2_{error}}) \tag{21}$$
$$W_{h_3} = W_{h_3} - \gamma((h_{2_{act}})^T \cdot h_{3_{error}}) \tag{22}$$
$$W_O = W_O - \gamma((h_{3_{act}})^T \cdot O_{error}) \tag{23}$$

2. Methodology

For this project we set out to design a robust framework to build neural networks, drawing inspiration form industry and academic standards like PyTorch and TensorFlow. With this idea in place we set out to make a framework that allowed for 1) arbitrary network sizes 2) easy integration of different network features, such as different forms of gradient descent or regularizers, and 3) modularity that would allow for further expandability. We found these features to be important for several reasons, besides trying to mimic something that would be useful and not a one-trick pony. By having arbitrary network sizes this would allow us to test on a wide variety of datasets. Furthermore we had suspicions when starting the project that a network that could be trained well on simple datasets may also not be large enough to benefit much from parallelization. By allowing for arbitrary network sizes this would allow us to better test as the project progressed. Easy integration of new features and extendability this allowed us to quickly test more ideas and expand the project, as we did not know how much we would be able to implement. With this unknown factor and a short deadline being able to quickly add new features without major code rewrites is a valuable tool.

To create these features we broke out the network into multiple classes, again drawing on inspiration from PyTorch and HPC tools like VTK-m. We separated out all our math functions into one class, individual linear layers into another class, and finally the model into a super class. With this construction we could implement networks of arbitrary sizes, extend them, modify our algorithms, and so on with little conflict. In Figure 12 we show the code required to generate a simple model that will accurately classify the Moons Dataset. As can be seen, this allows for a very similar construction style to that of PyTorch, where one simply needs to define the Linear Layers with the input and output size and how they are connected. Similarly our Sequential class determines how to connect all the nodes together and creates all the connections that will be needed for the Forward and Backwards functions. In our example the construction under the "creating layers" section is similar to PyTorch's model constructor and our "Adding layers to the model" section is similar to PyTorch's "forward" function.

```
void defineModel(Sequential &model,
                 float learning_rate)
{
    //creating layers
    LinearLayer i_layer(2,3,
                    learning_rate,
                    model.getBatchSize());
    LinearLayer h1(3,3,
                    learning_rate,
                    model.getBatchSize());
    LinearLayer h2(3,3,
                    learning_rate,
                    model.getBatchSize());
    LinearLayer o_layer(3,1,
                    learning_rate,
                    model.getBatchSize());

    //Adding layers to the model
    model.add(i_layer);
    model.add(h1);
    model.add(h2);
    model.add(o_layer);
}
```

Figure 12. Example model construction

In a similar fashion we designed the rest of the net-

work. This was particularly helpful in that the math library was similarly fashioned. By the arbitrary input sizes we could simply focus on the operations that took the most time. Similarly, this made it simpler to optimize because we could directly find which mathematical operations were performing the most work and that it matched our intuition.

While this all worked as expected for our OpenMP implementation this methodology gave us problems when we started to build our CUDA implementation. With our inexperience in CUDA we had assumed that porting the majority of the network would be similar and we would only need to create new kernels for our math functions. By the time we had started to really understand CUDA we had a fully developed and working OpenMP implementation that was working well and showed significant performance increases through parallelization. We learned a valuable lesson about how good design choices for a CPU implementation did not necessarily equate to good design choices for GPU acceleration. Had we had more CUDA experience, or any, going into the project we would have likely made different design choices. Had we more time to spend on this project we would have spent more time learning about how to implement structures in CUDA and that would have helped integrate the designs.

2.1. Computing Resources

We performed our experiments on one system, UO's Alaska in the CDUX group. All OpenMP operations were performed on the head node, which has 2 Intel Xeon E5-2667v3's that have 8 physical cores and 16 threads, each, for a total of 32 threads. All CUDA operations were performed on compute node 2 which has a single Intel Xeon E5-1650 with 6 physical cores and 12 threads as well as an Nvidia GeForce RTX 2080Ti.

3. Results

In this section we investigate the how parallelisms affected our program. We break the report into three sections. In the first section (Sec 3.1) we discuss the optimization techniques we performed, in the second section (Sec 3.2) we discuss our OpenMP implementation, and finally in the third section (Sec 3.3) we discuss our CUDA implementation.

3.1. Serial Optimization

In common fashion, before we start implementing parallelization we first started optimizing the serial implementation of our code. We do this first because many optimization techniques are about memory access and reducing the number of times an address is accessed. This helps with both OpenMP and CUDA parallelization because in both cases we will still want to reduce our memory footprint. While CUDA optimization won't see as much benefits from this exercise, as CUDA optimization is more dependent upon multi-dimensional memory access, using blocks and grids, it will still be beneficial overall.

To do this first took our working code and profiled it. Because our code allows for arbitrary sized neural networks we increased the number of hidden layers and neurons to get better profiling, though this decreased our accuracy. Profiling the code this way we looked for areas where we could first pre-allocate memory, This had significant effects in the speedup. We then looked for loops where we could reduce redundant computation, for example calculating the vector size every loop iteration. An example of this shown in Figure 13. These types of optimizations significantly improved our results.

(a)

```
for(size_t i = 0; i < v.size(); ++i)
 for(size_t j = 0; j < v[0].size(); ++j)
  computation
```

(b)

```
size_t v_size = v.size();
size_t v0_size = v[0].size();
for(size_t i = 0; i < v_size; ++i)
 for(size_t j = 0; j < v0_size; ++j)
  computation
```

Figure 13. Reducing loop lookups. (a) unoptimized (b) optimized

Previous to these, and other, optimizations our functions were not near the worst performers according to *gprof*. After these we had our matrix multiply function, our transpose, and our vector addition near the top. All these optimizations ensured that there was proper pipelining and that we could fully exploit our parallelism. One final optimization that we did is opti-

mize the matrix multiplication to get the best indexing we could, over a naive version that was initially implemented. This provided the largest speedup.

3.2. OpenMP Implementation

After we have ensured that proper pipelining and serial optimization are implemented the next task is to parallelize. Because of the shared memory in OpenMP our optimizations in Section 3.1 we only have to access these memory points once and then we can share that memory across the threads. Because most of our computation was in the form of vector or matrix operations we can pipeline like above, share the memory, and parallelize the outer most loop so that we can exploit the faster stride on the inside of the loop, since those points are contiguous in memory and the outer loop isn't.

To parallelize our code we focused on our math library, which held all our mathematical functions and performs all our tensor operations. We investigated parallelizing every operation and found that most operations had little to no effect. A major hurdle we had was that we found that we needed hundreds of neurons and a fair amount of layers for parallelism to create several hidden layers. For the problems we are solving, such as the half moon, this is far too large to learn the dataset and will over fit. For example, to learn the half moon dataset we only needed 2 input neurons, a single hidden layer with 3 neurons, and an output layer. With this network, not even a "deep" neural network, we quickly converged on the answer, typically below 100 epochs. The parallelization helps significantly more when increasing the number of neurons as opposed to the depth of the network, or the number of hidden layers. But since our goal was to build a framework, similar to PyTorch, we just needed to test the parallelization rather than solve a specific problem. Thus we created experiments with 300 neurons in each hidden layer and 5 hidden layers. We found this to be a good balance between the program not taking too long to run and having enough work to see speedups from parallelism.

3.2.1 Ablation Study

To see the effect of parallelism we perform a small ablation study to see what was needed for speedup and what wasn't. We show the results of this in Figure 14. As we can see here, the matrix multiply parallelization had the largest impact on speedup. Without this the other cases actually increase as the number of CPUs increase. Note that we are parallelizing most functions and these do have some overhead. Interestingly we find that the run without either matrix multiply nor vector add implemented runs slightly faster than when just matrix multiply is removed. We believe that this is because of the lack of work. One other factor to note is that at 16 threads we note a slight increase, where previously we were strictly decreasing. This is because our machine has 16 physical cores 16 but they have 2 threads per core. Some work performs better on physical cores rather than hyper-threading, which is why it is turned off in many HPC applications, and we see that this holds true here. To aid in this analysis we draw a vertical gray line to bifurcate where hyper-threading begins. We note that our fastest run was at 16 threads, executing at 40.128s (4.47x speedup) and the 32 thread implementation ran at 46.072s (3.89x speedup). We also note that the fastest hyper-threading implementation happened at 28 threads, 42.938s (4.17x speedup). We believe that this is due to some peculiarity in how hyper-threading works. We notice that there was more uncertainty within this region. This gives evidence that for faster and more stable computation one should disable hyper-threading.

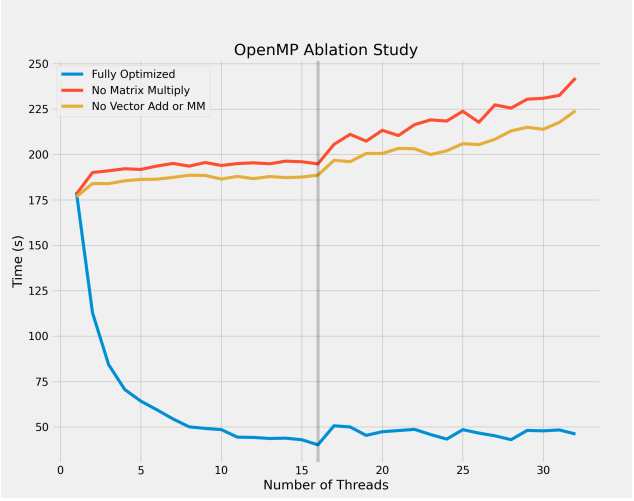

Figure 14. Comparison of how different functions affect parallelization results, using OpenMP

3.2.2 Matrix Multiplication

Additionally we look at our matrix multiplication vs using a cache optimized version. To create our cache optimized matrix we divide our i,j, and k indices into smaller blocks. This allows for the multiplication to be better localized in memory. We can see the results in Figure 15. Here we see that our cache efficient matrix multiply actually receives better performance increase under initial parallelization, achieving roughly a 14.5x speedup at 5 threads, while our naive implementation only achieves a 4.5x speedup at best. This suggests that the blocking version of matrix multiply could potentially be more efficient but that we do not have sufficient workloads to benefit from this type of parallelization. We would expect this version to be more efficient because it is more cache efficient. Unfortunately, in our workload we have matrices where the number of rows is far larger than the number of columns, often as small as 2, and that causes more cache misses than hits. Even worse, in our implementation many times we have vectors be multiplied and thus we do not gain any benefit at all. This type of multiplication has much larger overhead and we would not expect it to perform better than the naive version unless we were dealing with substantially larger matrices where we had issues fitting a row of data into memory. Interestingly in the cache efficient version we see that the computation diverges upon evoking hyper-threading as well as becomes less stable. We believe that hyper-threading suffers much harsher penalties to cache misses than physical cores do and that this causes the divergent behavior. We predict that if we were learning on larger datasets, such as ImageNet, we would see better performance increases from these type of optimizations.

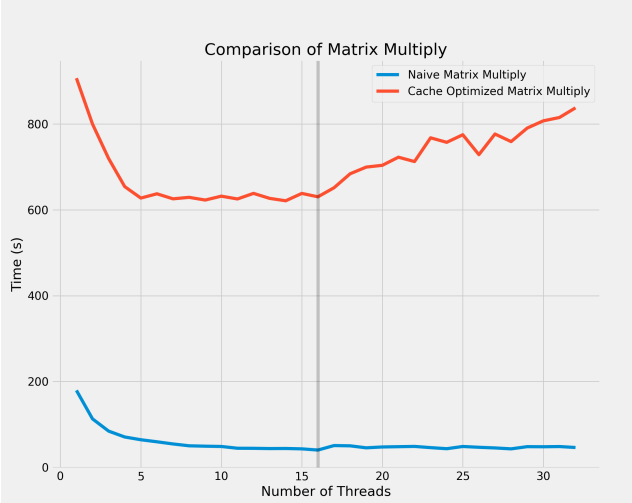

Figure 15. Comparison of different Matrix Multiply methods

This is with the fully optimized OpenMP version of the code.

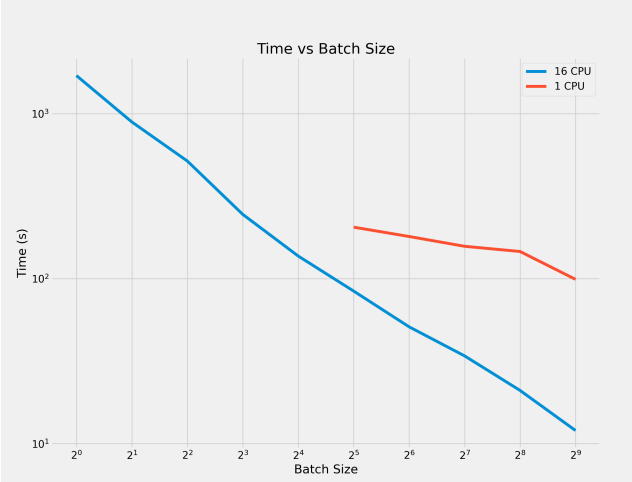

Figure 16. Time vs Batch Size using 2^n batch size on log-log scale

3.2.3 Batching

The largest improvement we made to our network was incorporating batching. Not only does batching allow for smother convergence but it also allows for more efficient parallelism. Batching has been shown to greatly speed up machine learning algorithms and we observe similar characteristics in our code. By implementing batching we observe a smooth decrease in time. Using power of 2 batch sizing we observe log scale decreasing in compute time, shown in Figure **??**. We can see that through batching we can get a 142x speedup.

One reason we added batching is because it also allows us to better exploit parallelism and increase our computation size. While this does not increase the problem size it does increase the amount of work done by each individual mathematical operation. This is why we see such a large time difference between a batch size of 1 vs a batch size of 512, which takes almost no time at all. We stop at a maximum of 512 because the moon dataset has 1000 data points and when using the typical 80% training size that's only two batches. We tested a 1024 batch size and the en-

tire computation took sub 1 second. This shows how proper algorithm design is needed for good parallelization. We also show a partial test when running 1 CPU and similarly looking at batching. We were not able to run a full suite because another student needed the machine for research work. Still, we can see in the 512 batch case that there is a difference of 8.25x by using 16 CPUs vs 1. This illustrates the power of batching and why it is a necessary algorithm for both speedup and parallelization.

3.3. CUDA

We were unfortunately unable to get the entire model into CUDA but we still wanted to provide some motivation as to how GPU acceleration would help with our problem. Knowing that PyTorch and TensorFlow benefit a lot from GPU acceleration we investigated how performing our matrix multiplication and transpose functions would help. For similar reasons as in the OpenMP example we didn't implement a block matrix setup in CUDA, and actually struggled with trying to implement this. The same is true for the efficient matrix transpose. Instead we just kept the naive algorithms. To make a fair comparison we timed our results in two different ways. The first we did was timing the matrix multiple with the communication time and we also timed just the kernel execution. We figured that testing the kernel function would give us a good indication of how fast a fully GPU implementation would run compared to the CPU version. Unfortunately we found that we were still slower in the GPU implementation. While our OpenMP solution ran in $1.213\mu s$ our CUDA kernel function ran in about $4.98\mu s$ and including communication time $121.734\mu s$.

Our working theory of why we are not seeing improvements with GPU acceleration is two fold. We believe that the largest contributor to our problem is actually the problem type. Because we are trying a simple classification we have matrices where $m >> n$. The optimized matrix multiplications do not perform well in these shaped matrices, performing best in square like matrices. The second issue is that the work itself isn't very large, only having 300 neurons per layer. We saw similar relationships with around 1000 neurons per layer, testing with a small number of epochs, but did not record the times because we could not do a full run in a reasonable amount of time. Linear layers of this size are also atypical in modern networks so we decided it was not a good avenue to pursue.

4. Conclusions

Through this work we were able to achieve a speedup of 4.5x over our serial implementation. Through this work we found many challenges within parallelizing seemingly simple mathematical functions. Learning about how certain algorithms perform better in different computational regimes as well as how the physical architecture affects different algorithms. We also learned a lot about how design patterns change as problems scale and the challenges in designing CUDA parallelization techniques and how writing good CUDA design patterns differs from that of conventional software engineering. With our analysis in our OpenMP study we saw how important proper algorithms are for exploiting parallelism. This is why we first focused on on our serial optimization before we moved to parallelism. Unfortunately we were not able to perform this same kind of parallelism in CUDA and due partially to our problem as well as our lack of experience in CUDA programming. Through this exercise we learned a lot about a few of the underlying challenges in accelerating Artificial Neural Networks and gained a greater appreciation for the work that goes into writing frameworks such as PyTorch and TensorFlow, and have a better appreciation for the utility that they provide to our community. If we were to do this kind of exercise again we would design things differently, keeping in mind our lessons learned from CUDA before beginning the designing process.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. 5

[2] Alexandra I. Cristea and Toshio Okamoto. A parallelization method for neural networks with weak connection design. In Constantine Polychronopoulos, Kazuki Joe, Keijiro Araki, and Makoto Amamiya, editors, *High Performance Computing*, pages 397–404, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. 1

[3] Quin He, Thilo Koehler, Antony D'Avirro, and Chetan Gupta. A highly efficient, real-time text-to-speech system deployed on cpus, May 2020. 1

[4] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *CoRR*, abs/1702.05659, 2017. 3

[5] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018. 2

[6] Raul Rojas. The backpropagation algorithm. in: Neural networks. pages 149–182, 1996. 5

[7] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. 1

[8] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. 2, 3

[9] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015. 1, 4

[10] Andreas Zell. *Simulation neuronaler Netze. [Simulation of Neural Networks.]*. Oldenbourg, 2000. 4