Project Neuronmancer:

GPU-Accelerated Backpropagation with C / CUDA

By Drew Hans

Table of Contents:

1. INTRODUCTION

Parallel programming with graphics processing units has interested me for a long time. I have always thought it was really cool how programmers were able to use specialized pieces of hardware to tackle huge problems. So naturally, when I entered CS491 I knew that I wanted to research parallel computing platforms and come up with some related project to work on for CS492. Since I started programming five years ago it has always been a dream of mine to be able to say that I once, in the past, built a program that utilized massive parallelism in some way. The key phrase in that last sentence is "in the past" because I knew that implementing such a program would be a giant pain. I was not wrong.

My senior seminar project is a C / CUDA program that I call Neuronmancer. This program is capable of generating feedforward artificial neural networks that learn to recognize handwritten digits by using the backpropagation learning algorithm. The name of my program comes from but shares no relation to, William Gibson's novel Neuromancer. I just thought it sounded cool.

2. BACKGROUND

A. Fundamentals of Machine Learning

Learning is a complicated process to describe but in a nutshell, it is turning experience into knowledge. Machine learning is a field of computer science that uses statistical techniques to give computer systems the ability to learn from data, without being explicitly programmed[1]. In general, machine learning is about creating a statistical model that can learn features of a dataset and then apply that feature knowledge to new data samples. We can usually separate machine

learning approaches into three major categories: supervised learning, unsupervised learning, and reinforcement learning.

In supervised learning, samples in the dataset come with one or more other attributes that we want our model to predict. Problems that fall into this category can be either classification problems, where data samples belong to one or more classes, or regression problems, where the desired output is a function of one or more continuous variables generated from the data sample. With unsupervised learning, samples in the dataset have no corresponding target values and our model attempts to learn patterns solely from the input samples. There are many types of unsupervised learning tasks but the most common is clustering, where the model automatically divides sample data into groups of similar items. Then there is reinforcement learning, where the model receives "rewards" and "punishments" for its actions. Reinforcement learning differs from supervised learning in that correct input/output pairs need not be presented, and sub-optimal actions need not be explicitly corrected.

B. Artificial Neural Networks

Artificial Neural Networks (ANNs) are a type of machine learning model vaguely inspired by the biological neural networks in our brains. Development of the first ANNs began in the 1940s. It was motivated by a desire to understand the human brain and to emulate some of its strengths. In 1943, McCulloch and Pitts set the foundation for neural network research when they proposed a simple mathematical model for describing neurons[2].

Figure 1. A mathematical model for a single neuron.

In this model, a neuron unit "fires" when a linear combination of its inputs exceeds some activation threshold. Mathematically, neuron unit j's output activation $a_j = f(\sum_{i=0}^{n} w_{ij} a_i)$, where a_i is the output activation of neuron unit i, w_{ij} is the weight on the link from neuron unit i to this neuron unit j, and f is the activation function of neuron unit j. Using this mathematical model, we can describe an ANN as a collection of connected artificial neurons.

There are two fundamentally distinct ways to connect neurons together to form a network: feedforward and feedback. Feedforward artificial neural networks (FF-ANNs) are organized such that neuron signals travel only one way, from input-layer to output-layer. Feedback artificial neural networks (FB-ANNs) are organized such that some neuron outputs feedback into their own inputs. Unlike FF-ANNs, which have no internal state other than their weights, FB-ANNs have an internal memory of previous input. Both ANN models have their strengths and weaknesses and which one you use will depend on the nature of the problem you are

tackling and the type of model you want to create. For the remainder of this section, I will focus on FF-ANNs since they are the focus of my research.

Figure 2. Visualization of a 3-layer feedforward artificial neural network.

FF-ANNs are usually arranged in layers, such that each neuron in a layer receives input only from the preceding layer. Organizing neurons in this way makes modeling simple. In an FF-ANN the first layer of neurons is called the input layer, the last layer of neurons is called the output layer, and all layers in between are called hidden layers. Layers can be fully-connected, where each neuron in one layer is connected to every neuron in the preceding layer, or partially-connected, where each neuron in one layer only connects to a few neurons in the preceding layer. The number of layers and the number of neurons per layer are important

decisions that need to be made when designing an FF-ANN. There is no "best answer", only general guidelines followed by researchers applying FF-ANNs to their problems.

Before an FF-ANN can be used to classify input, it has to be trained. Training an FF-ANN is basically a weight calibration process. In supervised learning, the network is exposed to a set of examples for which the desired outputs are known. Passing an example input into the network results in the activation of neurons in the output layer. A cost function compares the network's output activations with the expected activations and calculates an error value for each output layer neuron. These error values are then fed back into the network and used to modify the weights so that the next time it is given the example its output activations are closer to the expected activations. This process of error calculation and weight adjustment is called backpropagation and was first described in the paper Learning Representations by Back-propagating Errors[3] by Rumelhart, Hinton, and Williams in 1986. For supervised learning, backpropagation is the most commonly used training algorithm for feed-forward ANNs. The number of backpropagation iterations, so one forward pass of input and one backward pass of error signals for each example in our training set is called an epoch.

A detailed overview of biological neural networks and the theory behind ANNs can be found in Domen Verber's paper Implementation of Massive Artificial Neural Networks with CUDA[4]. Verber also provides an interesting discussion of the strengths and weaknesses of electrical hardware based versus software-based implementations. In Neural Networks and Deep Learning[5] Michael Nielsen introduces many of the fundamental mathematical concepts behind ANNs and deep learning by exploring the problem of teaching a computer to recognize handwritten digits. Nielsen shows how this problem is extremely difficult to solve using

conventional programming techniques yet can be "solved pretty well" using a simple ANN composed of a few dozen lines of code and no special libraries. This simple ANN is then improved through many iterations by gradually incorporating more of the core concepts of ANNs, including modern techniques for deep learning.

C. GPU-Accelerated Computing with CUDA

CPUs are sequential processing units designed to handle general computing workloads. CPUs are typically composed of a few cores and can handle only a few software threads at a time. In contrast, GPUs are massively parallel processing units adept at solving problems that fall into the single instruction multiple data (SIMD) model. The Compute Unified Device Architecture (CUDA) framework developed by Nvidia enables us to have a high-level view of the GPU as a massively multi-core computing platform with an interface close to the traditional multi-threaded programming style used with CPUs. The CUDA API allows the programmer to define kernel functions that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular functions.

Parallel computations are organized using the abstractions of threads, blocks, and grids. A thread is an execution of a kernel with a given index, a block is a group of threads, and a grid is a collection of blocks. Threads in a block can execute concurrently, serially, or in no particular order and can be told to stop at a certain point in the kernel until all other threads in its block reach the same point. Execution of a grid is handled by a single GPU chip. The GPU chip is organized as a collection of symmetric multiprocessors (SMs), with each multiprocessor responsible for handling one or more blocks in a grid. Each SM is further divided into a number

of CUDA Cores, with each Core handling one thread in a block at a time. Once a block is distributed to an SM the memory resources for the block are allocated and its threads are divided into groups of 32 called warps. Once a warp's memory resources have been allocated it is considered active. Schedulers inside the SM pick active warps and dispatch them to execution units inside the SM.

In combination with the hierarchy of processing units, CUDA-enabled GPUs also have different types of memory at each level (presented largest and slowest to smallest and fastest): global memory, constant/ texture cache, shared memory, and registers. Global memory can be accessed by any thread in any SM at any time and is the largest and slowest memory available. Each SM contains constant and texture cache read-only memory that can be accessed by threads across different blocks running on the same SM. Shared memory can be used by any thread in a block to read or write data. Registers are shared between cores in each SM and are the fastest memory available.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU, which is optimized for single-threaded performance, while the parallel part of the workload is made to run on the CUDA-enabled GPU. A strong argument for using GPU based backpropagation can be found in A Survey of GPU based Back Propagation Algorithm[6] by R.R. Chhajed, S.C. Dharmadhikari, and S.H. Chandak. The major purpose of this survey was to offer a comprehensive reference source for researchers analyzing CPU and GPU ANN implementations. The authors provide comparisons between the times taken to do the training and testing of backpropagation ANNs on different datasets using both the CPU and GPU. They found that GPU-enabled backpropagation yielded greater performance than CPU implementations for large

ANNs. Other optimizations for GPU ANN implementations are presented in Accelerating Forwarding Computation of Artificial Neural Network using CUDA[7] and in Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform[8].

3. IMPLEMENTATION

A. The C Programming Language

Originally I planned to use CUDA Python with the NUMBA Python compiler as described in the tutorial on Nvidia's website[9]. Python is my favorite programming language so I was excited to have the chance to use it for my senior seminar project. Unfortunately, setting up CUDA Python was more trouble than it was worth. The programming environment was very fragile and I had difficulty getting example programs to run.

After much thought, I decided to learn how to program in C since CUDA at its core is simply a minimal set of extensions to the C language and a runtime library[10]. Learning how to write C code required a weekend of research but learning how to think like a C programmer took me over four months. I bring this up because my biggest problem coding Neuronmancer was Java, even though I never wrote a single line of Java code for it. At Blackburn College, we learn how to program using Java. We think of problems in terms of Java and object-oriented programming. And Java is great - in an object-oriented world. But C is not Java. C has no objects. My biggest mistake was thinking about C code like Java without the objects. And I paid for it with time.

It was not until the mid-April that I realized my mistake. I started programming Neuronmancer in January and after two and a half months of programming, my code had become monolithic

and difficult to read. More importantly, model evaluation and training did not work. On April 5th I decided to rewrite Neuronmancer line-by-line and forty-five hours of programming later my project was cleaner and easier to read but training still did not work. After a few days of debugging, if you want to call it that, I realized I was getting nowhere and on April 12th, one week before our final senior seminar presentation, I realized my mistake and decided to rewrite my program again. This time I was going to do it the C way. Two days later I had Neuronmancer version 3.0 with sequential training code in C that worked for the host machine and parallel training code in C / CUDA that ran on the GPU device and sorta worked (the accuracy does go up after training on the GPU device but not as much as when we train on the host machine).

B. Ubuntu, Make, NVCC, & Other Tools

In addition to learning how to code in C, I got to use a bunch of different software tools during Neuronmancer's development.

When I first began, I used the Visual Studio 2015 Community Edition IDE for Windows 7 since it had native support for CUDA projects and could compile my project without me needing to know anything about NVCC - the NVIDIA CUDA Compiler. However, working in Visual Studio was very unintuitive so I talked Dr. Coogan into letting me setup Ubuntu 16.04 LTS on one of the R-Lab machines and continued my work there. One unforeseen benefit of switching to Ubuntu was the NVIDIA Nsight Eclipse Edition IDE that came bundled with the CUDA Toolkit version I installed. While not perfect by any means, coding inside Nsight Eclipse was much more enjoyable than using GNOME's gedit text editor.

Before I could get my C / CUDA program to run on Ubuntu I had to learn about GNU Make, Makefiles, and compiling programs with GCC, the GNU Compiler Collection. It took me a few days to wrap my head around but eventually, I got the gist of it. Once I knew how to compile C programs I started researching how to compile C / CUDA programs with NVCC. Initially, I used a Makefile I modified from one of the CUDA Toolkit sample projects to build Neuronmancer on Ubuntu but I got around to writing my own in mid-April. I have to say typing "make" to build my project and "make clean" to remove compiled files is much easier than typing out the full commands.

C. MNIST Database of Handwritten Digits

To benchmark the performance of models created by my program I needed a labeled dataset to learn from. Ultimately I decided to use the MNIST dataset of handwritten digits. MNIST's name comes from the fact that is is a modified subset of two datasets collected by the United States National Institute of Standards and Technology (NIST).

Figure 3. Samples from the MNIST dataset (colors are inverted for legibility).

The MNIST dataset comes in two parts. The first part contains sixty-thousand sample images intended to be used for training models and the second part contains ten-thousand sample images for testing models. Each MNIST sample is a grayscale image of a single digit 0 - 9 and is twenty-eight pixels by twenty-eight pixels in size. Since there are 784 pixels in every image, the

input-layer for any artificial neural networks created by Neuronmancer needs to have exactly 784 neurons in size. Similarly, given that there are only ten possible classifications for any given MNIST sample (digits 0 - 9) the output-layer should be exactly 10 neurons in size.

D. Project Design

Neuronmancer has gone through several design changes to get where it is today. In earlier versions, the user would enter this information through text-prompts given by the program as it ran but this method proved to be too time-consuming for the user and inflated the total lines of code making the program difficult to read. I found out the hard way that robust user interfaces are difficult to program using only C. In its current form, the user is given a message at the beginning of every run that they may define the parameters for creating artificial neural networks in the main.h source code file. This approach has several advantages, the biggest being that the size of the layers and number of neurons to allocate memory for on the host machine is known at compile time. The second biggest advantage is code readability. Since key parameters are defined as macros in the main header file, the number of parameters we must pass to each function is reduced (as the preprocessor replaces every macro variable with its defined value directly into the code).

One thing you will notice in Neuronmancer's source code is the lack of included libraries for fast matrix operations. Early on I considered using libraries for fast matrix operations such as GNU Scientific Library (GSL) and Basic Linear Algebra Subprograms (BLAS) for the host code and NVIDIA cuBLAS for the GPU device code. After much thought, I decided against using these libraries and instead chose to create my own functions using simple for-loops. I'm sure the

performance of my program suffered from this decision but I did not want to just "copy and paste" premade example matrix operations for feeding input forward and back-propagating errors into my program. I wanted to prove that I really understood what was going on inside the artificial neural network. It was important to me that I demonstrated mastery over these operations in my code.

The key to making Neuronmancer fast, at least on the CPU side, was the use of structs for organizing variables. In the first two versions of my program, I sometimes had to pass half a dozen variable pointers or more into functions. This may not sound like much of a bottleneck but it is. C is known as a "pass by value" programming language meaning that on every function call it has to copy over every single function argument while it puts a new frame on the stack. By using structs I only had to pass two or three struct pointers to functions rather than a dozen variables and this led to speed improvements all around.

One problem I kept running into on the GPU side was determining how many blocks and threads to use for different kernel launches. I tried a few different approaches. First, I tried setting the number of blocks equal to the number of thread-multiprocessors on the GPU device and the threads equal to some multiple of the warp size. This worked but wasted a lot of GPU resources with small kernel launches and was inadequate for large workloads. My second approach was to use a heuristics function to make a "best guess" and this ended up being the right way to go. I am sure that the heuristics I created could be improved but for my application, they worked fine.

When Neuronmancer is started, the neural network parameters will be displayed to the user along with a reminder that these parameters can be changed in main.h.

Figure 4. Visualization of artificial neural network models created by Neuronmancer.

After starting Neuronmancer a new feedforward artificial neural network will be created using the defined parameters in main.h. For working with the MNIST dataset the input-layer size parameter IL_SIZE should be exactly 784 neurons and the output-layer size parameter OL_SIZE should be exactly 10 neurons. The hidden-layer size parameter HL_SIZE and epoch parameter EPOCHS can be any size greater than 0. The learning rate parameter LEARNING_RATE can be any floating-point number. When initializing the network, the weights will be assigned random values in the range -1.0 to 1.0 inclusive. Originally the weights were initialized in the range 0.0 to 1.0 but this caused network predictions to skew towards only one class.

After the new neural network is initialized, the user is prompted with the following message:

What would you like to do?

Enter 0 to quit, 1 to train on host, 2 to train on GPU device, or 3 to evaluate:

~

If the user enters 0 then the program will end. No surprises there.

If the user enters 3 the program will perform an evaluation of the new network using the MNIST testing set. After the evaluation is complete the user will be shown the results in confusion matrix form. A confusion matrix, also known as an error matrix, is a table used to describe the performance of a classification model on a set of test data for which the true values are known[11]. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class (or vice versa).

Figure 5. Example of a confusion matrix generated after an evaluation.

If the user enters 1 the program will begin training the network on the host machine using the MNIST training set. Originally a model's neuron, weight, and bias values were kept in arrays and referenced by a single pointer but this ended up being more trouble than it was worth. Offsets had to be calculated and more arrays were needed to store those offset values. It was a mess, to be honest. I knew I needed a better way to organize the model's values so during my third rewrite of Neuronmancer I started using structs.

The structs defined in main.h are fundamental building blocks of models and you will need to have a good understanding of each one to fully grasp the function design in functions_core.c and functions_mnist.c. The structs MNIST_ImageFileHeader and MNIST_LabelFileHeader are used in functions_mnist.c to navigate through the ubyte files that contain the pixel data and label values of all MNIST samples. The structs InputLayer, HiddenLayer, OutputLayer, and ExpectedOutput are used to organize large arrays of data and are often used with pointers to simplify passing parameters to functions. The structs HLNeuron and OLNeuron are used to store important neuron values for the hidden-layer and output-layer respectively. Since the layer sizes are known at compile time, the structs are initialized without the use of malloc when the program starts. The weight array, bias value, and output values should be self-explanatory. The delta variable stores the calculated delta value of the neuron during the backpropagation step and the weightedSum variable stores the value of the combined input into the neuron before the activation function was applied. It might seem strange to store the weighted sum since we have the output variable but there is a good reason for it. Both the weighted sum and the delta value of a neuron are needed to update the weights and bias associated with it. By storing the weighted sum values we avoid having to waste time recalculating them later.

If the user enters 2 the program will begin training the network on the GPU device using the MNIST training set. To keep Neuronmancer's source code readable, I made sure to isolate all CUDA code in the functions_cuda.cu file. The cu extension is used to tell NVCC that the file contains CUDA functions, variables, and kernels. To avoid using linkers, which I have little experience with, the main file also has the cu extension so that functions_cuda.cu could simply be included in main's header file. The CUDA functions mirror the core functions described earlier with only a few minor deviations.

4. RESULTS

I tested several different hidden layer sizes and found that the host machine (CPU) was faster for all but the largest network. This indicated (1) the overhead associated with setting up the GPU device was greater than I had anticipated and (2) that my "optimal" block and thread size calculations were not as optimal as I had hoped.

seconds (lower is better)

Figure 6. Table of training times for epoch = 1

seconds (lower is better)

Figure 7. Table of training times for epoch =5

5. FUTURE WORK & CONCLUSION

I am planning on making my project open-source through GitHub with the hope that more experienced CUDA programmers will offer suggestions for improving training on the GPU device. Looking back on it, I'm proud of all I was able to accomplish in these past few months. It was frustrating at times but it was also a lot of fun. I learned a lot.

6. ACKNOWLEDGMENTS

Special thanks to all my friends who encouraged me when I felt like giving up.

Also, to my professors for giving me the freedom to explore a topic I found interesting.

7. REFERENCES

[1] Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. Retrieved from ieeexplore.ieee.org/document/5392560/

[2] McCulloch, W. S., Pitts, W., (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics. Retrieved from link.springer.com/article/10.1007/BF02478259

[3] Rumelhart, D. E., Hinton, G. E., Williams, R. J., (1988). Learning representations by back-propagating errors. Neurocomputing: Foundations of Research, Pages 696-699, Retrieved from dl.acm.org/citation.cfm?id=65669.104451

[4] Verber, D. (2012). Implementation of Massive Artificial Neural Networks with CUDA. Retrieved from www.intechopen.com/books/cutting-edge-research-in-new-technologies/implementation-of-massive-artificial-neural-networks-with-cuda

[5] Nielsen, M. (2015). Neural Networks and Deep Learning. Determination Press, Retrieved from www.neuralnetworksanddeeplearning.com

[6] Chhajed, R. R., Dharmadhikari, S. C., Chandak, S. H. (2017). A Survey of GPU based Back Propagation Algorithm. International Journal of Innovations & Advancement in Computer Science, Vol6(Issue10), Retrieved from www.academicscience.co.in/admin/resources/project/paper/f201710081507480904

[7] Park, J. H., Ro, W. W. (2016). Accelerating Forwarding Computation of Artificial Neural Network using CUDA. 2016 International Conference on Electronics, Information, and Communications, doi: 10.1109/ELINFOCOM.2016.7562974, Retrieved from ieeexplore.ieee.org/document/7562974

[8] Zhang, Y., Zhang, S. (2013). Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform. IEEE 25th International Conference on Tools with Artificial Intelligence, doi: 10.1109/ICTAI.2013.21, Retrieved from ieeexplore.ieee.org/document/6735232

[9] GPU Accelerated Computing with Python. Retrieved from developer.nvidia.com/how-to-cuda-python

[10] The CUDA C Programming Guide. Retrieved from docs.nvidia.com/cuda/cuda-c-programming-guide/

[11] Simple guide to confusion matrix terminology. Retrieved from www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

Appendix A

Pseudo-code for the Backpropagation Learning Algorithm

```
function backpropagationLearning(examples, network, epochs, α)

inputs: examples, a set of examples, each with input vector x and output vector y
        network, a multilayer network with L layers, neuron inputs in_i, neuron outputs a_i,
                 weights w_{i,j}, activation function g, and derivative of activation function g'
        epochs, the number of times all training vectors are used once to update the weights
        α, the learning rate

local variables: Δ(L), a vector of errors for layer L indexed by network node
                 error, a temporary floating-point variable that holds the cumulative error
```

```
  while count < epochs
      for each example (x, y) in examples do

          /* Load example's input vector x into input-layer */
          for each node i in the input layer 1 do
              a_i ← x_i

          /* Feed inputs forward to compute the outputs */
          for l = 2 to L do
              for each node j in layer l do
                  in_j ← ∑ w_{i,j}*a_i

                  a_j ← g(in_j)

          /* Calculate the output-layer's delta value */
          for each node j in the output layer L do
              error = e(y_j,a_j)  // e is a cost/loss/error function
              Δ(L)[j] ← g'(in_j) × error

          /* Propagate deltas backward */
          for l = L - 1 to 2 do
              for each node i in layer l do
                  Δ(l)[i] ← g'(in_i) * ∑ w_{i,j}*Δ(l+1)[j]


          /* Update every weight in network using calculated deltas */
          for l = 2 to L do
              for each weight w_{i,j} in l do
                  w_{i,j} ← w_{i,j} + α × a_i × Δ(l)[j]

      count ← count + 1
```