ABSTRACT

OPTIMIZING ARTIFICIAL NEURAL NETWORKS ON A GPU PLATFORM

by

Brian Schmidt

Chair: Roy Villafane

ABSTRACT OF GRADUATE STUDENT RESEARCH

Thesis

Andrews University

College of Technology

Title: OPTIMIZING ARTIFICIAL NEURAL NETWORKS ON A GPU PLATFORM

Name of researcher: Brian Schmidt

Name and degree of faculty chair: Roy Villafane, Ph.D.

Date completed: September 2011

Problem

The size and speed required by modern applications of neural networks is growing continuously. In the last decade, it has become possible to run general-purpose algorithms in GPUs, and much research has been done on running artificial neural networks on this new platform, making them faster and more efficient.

The purpose of this thesis is to design an optimization to an Artificial Neural Network algorithm running on a GPU. The optimization involves using a representation of the data specifically designed for sparse data. Its performance was measured on different topologies, to determine in what situations the optimization is effective.

Method

The research was pursued by creating versions of the algorithm in question that functioned with and without the optimization, and gathering performance data while also varying several parameters of the algorithm, such as: number of neurons, number of layers, and level of connectivity. At the end, performance data were gathered and compared.

Results

The test showed that the optimization performed only as expected in very specific topologies and situations. In all tests, the optimized version of the algorithm performed the feed-forward operation in less time than the un-optimized version of the algorithm. In some tests, the optimized version of the algorithm was able to store the weight matrices of the test artificial neural networks in less space than the un-optimized version of the algorithm.

Conclusions

The optimized GPU algorithm saved execution time and memory space when compared with un-optimized versions of the same algorithm when the proportion of connected neurons between layers was below 30%, and when there were many small layers connected together. It would be useful to apply the optimization discussed in this thesis in situations like these.

Andrews University

College of Technology

OPTIMIZING ARTIFICIAL NEURAL NETWORKS ON A GPU PLATFORM

A Thesis

Presented in Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Brian Schmidt

2011

OPTIMIZING ARTIFICIAL NEURAL NETWORKS

ON A GPU PLATFORM

A thesis
presented in partial fulfillment
of the requirements for the degree
Master of Science

by

Brian Schmidt

APPROVAL BY THE COMMITTEE:

_____
Roy Villafane, Ph.D., Chair

_____
Stephen Thorman, Ph.D.

_____ _____
William Wolfer, M.S. Date approved

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ABBREVIATIONS

3D	3 Dimensional
ALU	Arithmetic and Logic Unit
ANN	Artificial Neural Network
ART	Adaptive Resonance Theory
Cg	C for graphics
COO	Coordinate format
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate (memory)
DIA	Diagonal format
ELL	ELLPACK format
FANN	Fast Artificial Neural Network Library
FICNN	Fully Interlayer Connected Neural Networks.
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphical Processing Unit
HYB	Hybrid Format
MIMD	Multiple Instruction Multiple Data
MLP	Multi-Layer Perceptron
OpenCL	Open Computation Language

OpenGL	Open Graphics Library
PCNN	Partially Connected Neural Network
PKT	Packet Format
SIMD	Single Instruction Multiple Data
SLP	Single Layer Perceptron
SOM	Self-Organizing Map

ACKNOWLEDGMENTS

I would like to thank my parents for supporting me through this long process. I would also like to thank my brother Roger for reading multiple drafts of this thesis.

CHAPTER 1

INTRODUCTION

Neural Networks Introduction

The Connectionism Paradigm

The connectionism paradigm is used to solve problems in several fields. It uses emergent qualities in networks of simple processing units to model behavior and find working solutions. Neural networks are an application of this field of study.

Connectionist models are made up of many simple processing units that follow a common set of rules and the connections between them. One of the strengths of the connectionist paradigm is its ability to learn. Learning is accomplished by modifying the connections between units.

Connectionist models are based on emergence. Emergence is the way complex systems and patterns arise out of many simple interactions ("Emergence," n.d.). According to Rumelhart and McClelland (1986), connectionist systems have the following parts:

1. A set of processing units.
2. An activation state for each unit.
3. An output function for each unit.
4. A pattern of connectivity among units, represented by a matrix of real numbers indicating connection strength.
5. A propagation rule spreading the activations via the connections.
6. An activation rule for combining inputs to a unit to determine its new activation.
7. A learning rule for modifying connections based on experience.
8. An environment which provides the system with experience (p. 46).

Neural networks are considered connectionist systems because they implement each of these features. Other examples of connectionist systems are: self-organizing maps and adaptive resonance theory.

Biological Neural Networks and the Brain

The brain is the most complex object known to science, and its most basic processing unit is the neuron. A neuron is an electrically excitable cell that processes and transmits information by electrical and chemical signaling. The brains of all animals are made up of networks of interconnected neurons that communicate to process information ("Brain," n.d.). As an example, the human brain has 10^{11} neurons, arranged in a huge network, and works as a parallel system (Kriesel, n.d.). By definition, the brain is a connectionist system with amazing emergent qualities.

The processing capabilities of the human brain far outperform even the most powerful computers in certain tasks, but are themselves outperformed by computers in other tasks. The function of any brain is to control the actions of an animal; to do this it must extract information from the senses and control the movements of the body. The brain is an information processing organ (Kriesel, n.d.).

So, why is the brain of interest to computer scientists? Kriesel (n.d.) has a very good argument for studying the information processing capabilities of the brain. The hundred-step rule describes the basic difference between the biological processing that is done by neurons and the digital processing that is done by computers. As a rule, a simple modern computer can be performing only one operation at a certain moment in time, because of its sequential organization. However, a brain can be performing many operations in a certain moment in time, because of its parallel organization. According to

Kriesel (n.d.): “Experiments showed that a human can recognize the picture of a familiar object or person in 0.1 seconds, which corresponds to a neuron switching time of 10^{-3} seconds, or 100 discrete time steps of parallel processing” (p. 5). Furthermore, a modern digital computer, while able to accomplish many more operations in the same 10^{3} seconds, is not able to recognize a human face with the same efficiency of 100 time-steps. The parallel nature of the brain is its most important quality when it comes to its information processing capabilities. As a side note, its biological nature also allows the brain to recover from injuries and rearrange itself to the changing requirements of the environment (Kriesel, n.d.).

A neuron is made up of the cell body, dendrites which receive impulses from other neurons, and an axon, which sends information to other neurons. Neurons can have more than one dendrite, but they never have more than one axon. Electrical potential is maintained across the cell membrane by pumping neurons through. This is the basis of information processing and communication. When the electrical potential changes by a large enough amount, a signal is sent along the axon to the other neurons connected to it. The ability to learn is implemented through a mechanism that makes neurons that cause each other to fire, to grow stronger connections (Kriesel, n.d.). See Figure 1.

Early Work

The use of biological neurons as models for information processing has a history going back to the early 1940s. The first work was done by W.S. McCulloch and W. Pitts in 1943, who worked on models of neurological networks; they were able to show how simple networks were able to calculate many functions. In 1949, Hebb created

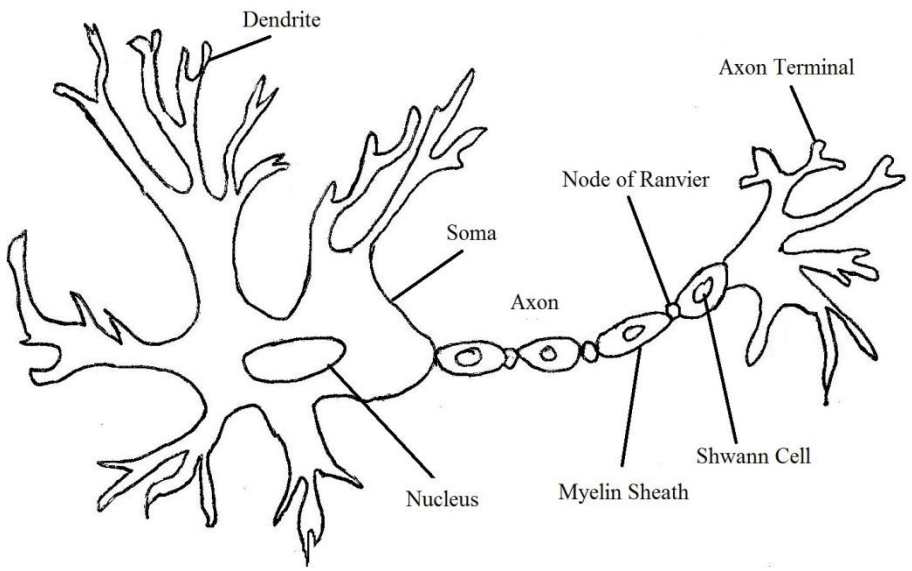

Figure 1. Structure of a neuron.

a theory on the learning ability of neural networks; he postulated what is now known as the Hebbian Rule. In 1969, Minsky and Papert published a complete mathematical description of perceptrons. In 1974, Werbos published the first description of the back propagation algorithm, the main algorithm used for learning in neural networks (Werbos, 1974; Davis, 2001).

After a slow period of several decades, neural network research increased again in the late 1970s. Stephen Grossberg and Gail Carpenter began research that led to models of adaptive resonance that would lead to the creation of adaptive resonance theory. In 1982, Teuvo Kohonen researched self-organizing maps (SOMs) and published his results. At the same time, John Hopfield created a new type of neural network that now bears his name: the Hopfield network (Davis, 2001; Kriesel, n.d.).

Mathematical Basis

A neuron in a biological network follows rules that are encoded in its DNA, for processing information. A neuron in an artificial neural network follows mathematical rules to do the same thing. Just like a biological neuron, an artificial neuron accepts information from other neurons, processes it, and provides information to other neurons. Also, just like its biological counterparts, a biological neuron can learn by modifying its connections to its input neurons, but instead of a chemical process, it changes parameters and performs computations to do so.

The input to a neuron can most readily be described as a vector, whose value is determined by the outputs of the neurons that are connected to it, or an outside input to the network. Kriesel (n.d.) calls this input:

$$\overline{x}$$

However, biological neurons are able to modify their connections; this is achieved mathematically by associating a value called a weight with each input to a neuron. The weights can be stored in a corresponding vector $\overline{w}$. To apply the associated weight to the value of an input, the input vector $\overline{x}$ and the weight $\overline{w}$ are multiplied together:

$$\overline{wx}$$

In order to reduce the information, the values of the weighted inputs are summed together, and the result is now called a weighted sum, represented as such:

$$\Sigma(\overline{wx})$$

Lastly, to provide a bounded output to other neurons, the weighted sum is passed through an activation function called Ω:

$$\Omega(\Sigma(\overline{wx}))$$

which produces the scalar output of the neuron, here called Y.

$$Y = \Omega(\Sigma(\overline{wx}))$$

The activation function is usually one of three functions: $1/(1+e^{-x})$, $e^{x\wedge 2}$, or tanh(x) (Moore, 2009). I chose the first function for this thesis's tests, because of its simplicity.

This is a simple definition of the mathematical workings of a single neuron; however, one neuron working alone cannot accomplish very much, so now I am going to mathematically describe a network of neurons. As written in Kriesel (n.d.), the definition of a neural network is:

> A neural network is a sorted triple (N, V, w) with two sets N, V and a function w, whereas N is the set of neurons and V is a sorted set {(i, j) | i, j Є N} whose elements are called connections between neuron i and neuron j. The function w: V -> R defines the weights, whereas w((i, j)), is the weight of the connection between neuron i and neuron j. Depending on the point of view, the function is either undefined or 0 for connections that do not exist in the network. (p. 36)

Most importantly for the subject of this thesis, the weights, inputs, and outputs of a neural network are most efficiently stored as matrices and vectors. Furthermore, the most computationally intense part of executing a neural network is calculating the weighted sum, which is most easily described as a linear algebra problem.

The Basic Perceptron

The most basic type of neural network is the perceptron, and it is widely implemented. While there is not one standard to define what is meant with the term perceptron, most of the time it denotes a feed-forward network with shortcut connections (Kriesel, n.d.).

A perceptron uses two types of neurons, input neurons and information processing neurons. These are organized into layers within which neurons are not connected to each

other. To receive information into the network, an input neuron (also called an identity neuron) is used. An input neuron has only one input with a fixed weight, and it serves as a placeholder for the network input. An information processing neuron works as described in the previous section, and is often fully connected to the previous layer. A more concise definition of a perceptron is: a feed-forward neural network in which the first layer is composed of input neurons, and there is one or more layers of input processing neurons. Finally, one layer is fully connected to the previous one (Kriesel, n.d.). Perceptrons can contain shortcuts, which are connections that skip a layer, but these are kept from the definition and will not be dealt with in this thesis.

The most basic case of a perceptron is the single-layer perceptron (SLP), which contains one input layer and one output layer, and only one set of changeable weights. However, perceptrons can have more than one layer, and are then called multi-layer perceptrons (MLP). See Figures 2 and 3.

SLPs are not as powerful as MLPs, and are limited in the input that they can recognize. SLPs can only recognize output that can be linearly separable. To explain, let us use the AND and the XOR functions. An MLP can recognize both the AND and the XOR functions, but an SLP can recognize only the AND function, because the output of the XOR function is not linearly separable, meaning it cannot be separated into two different areas by a line. See Figures 4 and 5.

As can be seen in the figures, the input/output map for the AND function can be separated into two areas, but the input map for the XOR function cannot, and therefore requires an MLP to be processed. Furthermore, an MLP is a universal function approximator, which is proven by the Theorem of Cybenko (Kriesel, n.d.). It is possible

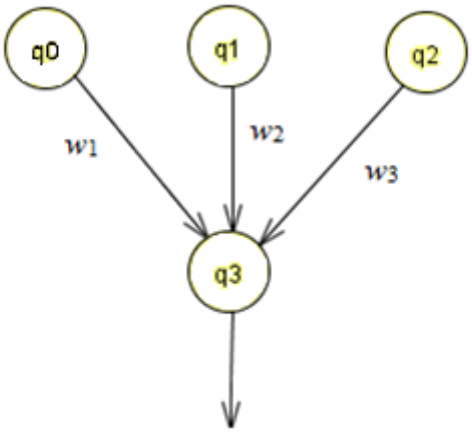

Figure 2. Single-layer perceptron.

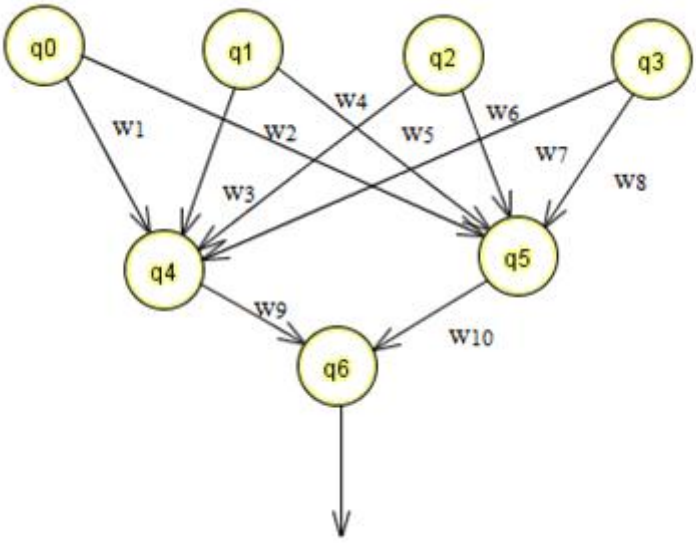

Figure 3. Multi-layer perceptron.

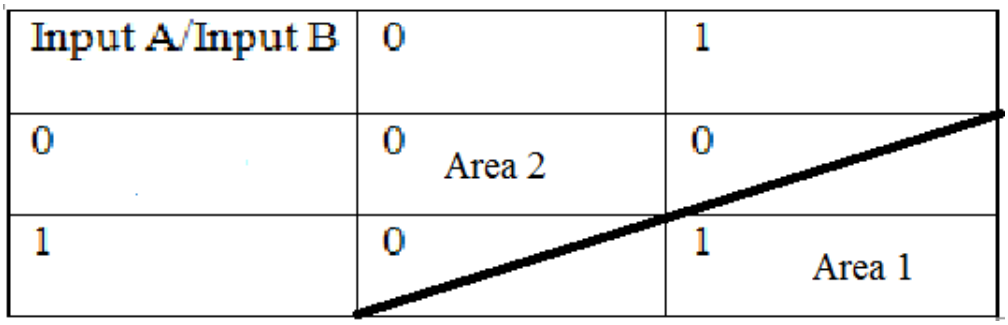

Figure 4. Input/output map for the AND function.

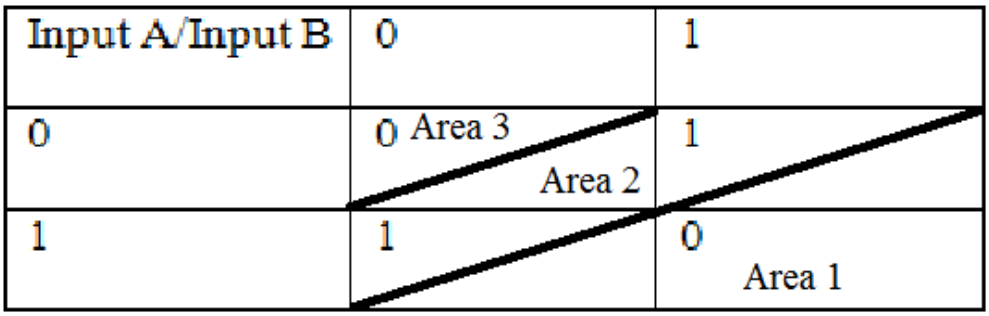

Figure 5. Input/output map for the XOR function.

for an MLP with one hidden layer to approximate a function with a finite number of points of discontinuity, with the appropriate number of neurons (Kriesel, n.d.). There is no advantage for a perceptron to have more than three layers, since any function can be approximated by a three-layer MLP with the appropriate number of neurons, and an MLP with more than three layers does not have more power.

Artificial neural networks have a capacity for modeling functions. The capacity of an ANN is determined by the number of neurons and the number of connections between them. A very complex function requires more neurons to be learned correctly.

The three-layer MLP is very commonly used. Therefore the three layers have standard names. The first layer is called the *input* layer, the second is the *hidden* layer,

and the last is the *output* layer. For the purpose of this thesis, there are then only two changeable set of weights, between the input and hidden layers, and the hidden and output layers. The weights connecting the input layer to the actual input are not trained, since they are used to accept input into the network.

Different Topologies

The perceptron is the simplest type of neural network, because neurons are grouped into layers, layers are connected in a simple fashion, and information flows in a single direction through the network. The perceptron model is a very constrained model, and more interesting opportunities become available once these constraints are lifted. Some of the different topologies that neural networks can take will be described.

A perceptron is a feed-forward network, since connections are allowed to neurons that are in the following layers. Shortcut connections are connections that skip one or more layers. Networks with a shortcut connection are considered to be MLPs.

Recurrent networks are networks in which neurons are able to influence themselves, by either having a direct connection to their own input, or by having a cycle in the graph describing their connections. Lateral recurrence occurs when neurons from the same layer are allowed to connect to each other. Laterally recurrent networks are not considered to be MLPs. Recurrent networks are more powerful than MLPs because they are able to influence themselves by using the results of a calculation in later calculations. Recurrent networks can either converge or not converge. If a network converges, then it returns a given output for the same input at any time. If it doesn’t converge, then it can

return different outputs for a given input. Some examples of recurrent networks are Jordan networks and Elman networks.

Completely connected networks are networks in which all neurons can be connected to all other neurons. Therefore it is impossible to group neurons into layers. An example of completely connected, recurrent ANNs is the Hopfield network, named after John Hopfield, who developed them. Hopfield networks are inspired by physics models of the magnetic energy interactions of gases, which cause every gas particle to arrange itself with every other gas particle in the lowest energy state. In a Hopfield network, every particle influences every other particle in the same way, and the network can be thought of as "a cloud of neurons" (Kriesel, n.d.). Hopfield networks are used to find the local minima of a defined function (Hopfield, 1982).

Another type of ANN is Self-Organizing Maps. SOMs work within two spaces: the N-dimensional input space and the G-dimensional space on which the neurons are organized. The G space is usually one-, two-, or three-dimensional, because these are easy to visualize (Kriesel, n.d.). The output of the network is the state of the network itself after a number of iterations. SOMs are able to perform unsupervised learning, and they are used to map a high dimensional input into a low-dimensional map (Kriesel, n.d.).

Adaptive Resonance Theory (ART) is an extension of the basic ANN template to assimilate biological features of a natural brain. An ART network has two layers, the input layer and the recognition layer. The input layer is completely linked with the recognition layer, and the recognition layer is linked to the input layer. Because of this layout, during the operation of the network both layers affect each other, and this leads to resonance. The network functions by comparing the information coming in through the

input layer with the information stored in the recognition layer. Through this process it is able to accomplish unsupervised learning and pattern recognition (Kriesel, n.d.).

A bias neuron is sometimes used to modify the behavior of a network, while not changing its topology. A bias neuron is a neuron that always has an output of 1. It is used to simplify the training and representation of a network.

Introduction to Back Propagation and Learning

There are two ways to accomplish learning: supervised learning, in which an outside agent provides the knowledge needed to learn; and unsupervised learning, in which the network is able to teach itself, if it is given enough data from the domain to be learned.

A neural network learns by modifying the weights of the connections between its weights. To accomplish this, the back propagation algorithms is most often used, so called because it propagates the error of the output of the network backwards through the layers to minimize the error. When using back propagation to train a neural network, two things are needed: a set of network inputs, paired with a set of desired network outputs. Back propagation is therefore a supervised learning procedure. From these two inputs, the algorithm calculates the total error of the current configuration of the network and then the weights can be changed to minimize that error. Back propagation requires the activation function used by the neurons to be differentiable (Kriesel, n.d.).

The back propagation algorithm is composed of two steps: feed-forward and weight update. The feed-forward portion is the normal operation of the network, and it requires the training input. The weight update requires the desired output and the actual output of the network; from this the error can be calculated.

To define the average error for the whole network mathematically: the set of training example pairs is called P(p, t), where p is the correct output, and the associated input is t, $\bar{x}$ is the input vector, and $\bar{y}$ is the output vector of the neural network. The error function is defined as:

$$ERR = \frac{1}{2}(\Sigma((t_p - y_p)^2))$$

The error function is defined as the mean squared sum of the difference between the actual and desired outputs (Kriesel, n.d.). The goal of learning is to minimize this error.

Another type of error is calculated for each individual neuron; it is called the error gradient, and it is needed to calculate how to modify the neurons' weights. The error gradient is calculated using the derivative of the activation function, hence the necessity for a differentiable activation function, like this:

$$ErrorGradient_j = (IdealOutput_j - Output_j) * F'(Output_j).$$

As an example, the sigmoid activation function will be used:

$$F(x) = 1/(1+e^{-x})$$

$$F'(x) = F(x)*(1 - F(x))$$

Then the error gradient function becomes:

$$ErrorGradient_j = (IdealOutput_j - ActualOutput_j) * ActualOutput_j * (1-ActualOutput_j)$$

Where j is a neuron in the network.

The gradient of each neuron depends on the previous layer's error, so the calculation happens backwards, from the last layer to the first layer (so it is called back propagation). The error of a neuron in a hidden layer is calculated like this:

$$HiddenError_i = \sum(OutputError_j * w_{ij}) * F'(HiddenOutput_i)$$

Where j is a neuron in the network, and i is another neuron, and w_{ij}, is the weight for that input.

The error is the summation of the output errors of the next layer and the weights of the connections to those neurons, multiplied by the value of the derivative of the activation function at the hidden neurons output.

To calculate the amount of change needed for each individual weight, we need this formula:

$$\Delta w_{ij} = Output_i * Error_j$$

where $output_i$ is the neuron's output value and $error_j$ is the error of the output neuron for that connection. Therefore, the new value for the weight will be:

$$w_{ij} = \Delta w_{ij} + w_{ij}$$

Back propagation can be implemented in two ways: as either online learning or batch learning. During online learning, the weights are updated after every feed-forward pass. During batch learning, the changes to the weights are saved and the weights are updated after a determined number of forward passes (Angelou, 2010).

Applications of Neural Networks

Artificial neural networks are classified as statistical or data mining algorithms, and are useful in many tasks. *Wikipedia* lists some general categories of problems:

1. Function approximation, or regression analysis, including time series prediction, fitness approximation and modeling.
2. Classification, including pattern and sequence recognition, novelty detection and sequential decision making.
3. Data processing, including filtering, clustering, blind source separation and compression.
4. Robotics, including directing manipulators, Computer numerical control.

("Artificial Neural Networks," n.d.)

Neural networks have been used in the following applications, and many more: radar systems (Lee, Choi, & Kim, 2003), speech recognition (Veselý, Burget, & Grézl, 2010), handwritten text recognition (Ciresan, Meier, Gambardella, & Schmidhuber, 2010), medical diagnosis(Sidiropoulos, Cavouras, Pagonis, Dimitropoulos, & Stonham, 2009), image recognition(Scherer, Schulz, & Behnke, 2010), and anomaly detection in computer networks (Bastke, Deml, & Schmidt, 2009).

Partially Connected Neural Networks

A partially connected neural network (PCNN) is defined as a network that contains only a subset of the entire set of possible connections for a particular neural network model. They have fewer connections between layers than fully interlayer connected neural networks (FICNN) (Elizondo & Fiesler, 1997). Having fewer connections in a neural network allows for less network complexity, less storage requirements, less processing time, and less training and recall time (Elizondo & Fiesler, 1997). PCNNs have been used to improve the performance of neural networks used in image processing, speech recognition, and linguistics. Elizondo and Fiesler's (1997) study contains a good overview of PCNNs.

GPU Programming Introduction

Stream Processing

GPU programming is an application of the concept of stream processing. Stream processing is a computer programming paradigm ("Stream Processing," n.d.). An application that uses stream processing is able to more easily and efficiently use hardware resources. The stream processing paradigm is made up of a set of data and the operations

that are to be applied to it, and is usually executed in MIMD architecture. The data used by the program are separated into chunks that can be processed in parallel, and the code that acts on these data is able to be executed transparently on many processors at once.

Stream processing works well with applications that have three characteristics:

1. Compute Intensity, or, a high ratio between the number of arithmetic operations per memory access (Fatahalian, Sugerman, & Hanrahan, 2004)

2. Data Parallelism, which exists when an operation within the program does not rely on previous results to be executed and therefore allows greater flexibility in processing ("Stream Processing," n.d.)

3. Data Locality, which refers to a uniform way of accessing data within the program. This allows for more efficient caching ("Stream Processing," n.d.).

Stream processing does not excel in general-purpose computations. However, if an algorithm can be designed to cluster its data into a set of elements, called a stream, and define its operations to be done on the stream into segments of code called kernels, then it can use specialized processors called stream processors in its execution, sometimes achieving significant speed-ups (Che et al., 2008).

Stream processors contain specialized hardware that can execute stream processing programs, for example: the Cell processor from IBM, the Imagine and Merrimac projects from Stanford University, and most importantly for this thesis: GPUs.

There are many programming languages and programming language extensions that specialize in stream processing, for example: CUDA from NVidia, Brook from Stanford, and OpenCL by the Khronos group, being some of the more well-known alternatives.

General Description

A GPU is a highly specialized processor, designed for the work of rendering graphics. Because the high compute intensity and the parallel nature of graphics calculations, graphics cards started including support for these operations by adding GPUs in the mid-1990s. HP, Sun Microsystems, and SGI were the first to include 3D acceleration support in their workstations in 1996; however, they were not available to normal users. The company 3dFX introduced the first true commercial GPU for the general public in 1997, with the release of their Voodoo Graphics chipset. This was later followed by the NVidia GeForce, and many others (Davis, 2001). These days, every gaming system of sufficient complexity contains a 3D accelerator, which has caused the per-unit cost of a GPU to drop because of the economies of scale (Boggan & Pressel, 2007).

The first GPUs were not programmable and were not useful for general purpose computations. However, in 1999, NVidia introduced the first programmable operations on a GPU. In 2002, ATI introduced floating point calculations on their Radeon 9700 model (Owens et al., 2007). Using these hardware capabilities and native graphics interfaces, users started to use GPU hardware for general purpose calculations. GPUs were not easily programmable, however, until the introduction of the CUDA framework by NVidia in 2006, which was the first framework for doing GPGPU calculations.

Hardware Model

Modern GPUs are implemented as circuits that can reside in the same die as the host processor, board, with shared or separate memory. GPUs and graphics cards are often connected to the host CPU using PCI Express, AGP, or PCI interfaces.

The graphics pipeline is built from vertex processors, rasterizers, fragment processors, and a frame buffer. The two components of the graphics pipeline that are most heavily utilized during GPU computation are the vertex and fragment processors (Davis, 2001). At the beginning of the pipeline, the GPU accepts vertex data from the application running on the CPU. These data go into the vertex processor, which does transform and lighting operations. The modified vertices are grouped into graphics primitives and are then sent to the rasterizer where they become a stream of image fragments for each pixel covered by the primitive. The rasterizer is not programmable and therefore is not useful for GPGPU. The fragments then enter the fragment processor. The fragment processor performs tests to determine if the fragment should be shown in the final image. If a fragment passes the tests, then it is written to the frame buffer for display (Boggan & Pressel, 2007; Davis, 2001). See Figure 6. The vertex and fragment processors are most heavily used during GPU computation (Davis, 2001). However, modern GPU architectures have now combined the fragment and vertex processor into one type of processor called a Streaming Multiprocessor. The architecture of GPUs is SIMD, and they fall into the category of vector processors.

As an example, the NVIDIA Fermi architecture will be used, which is designed to adapt the GPU graphics capabilities to be used for scientific calculations. The GPU chip contains the L2 cache, the processing cores, the memory interface, the global scheduler, and the host interface. The first Fermi GPU contained 16 SMs with 32 cores each, for a total of 512 processing cores. The memory interface is 384 bits wide, and supports up to 6 gigabytes. The host interface is PCI Express. Each SM contains a common instruction cache, schedulers, dispatch units, a register file, shared load/store units, shared special

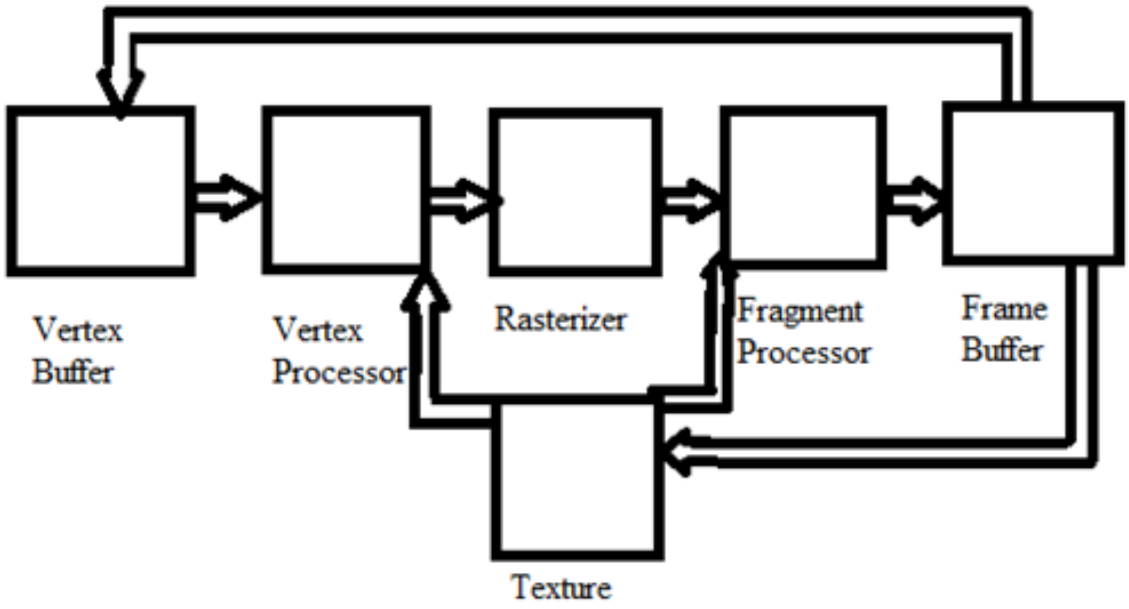

Figure 6. The graphics pipeline. From "A Survey of General-Purpose Computation on Graphics Hardware," by J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, & T. Purcell, 2007, *Eurographics 2005, State of the Art Reports*, 2005, pp. 21-51. Copyright 2005 by the Eurographics Association. Reproduced with permission.

function units, an L1 cache, and the interconnect network. The L1 cache allows for faster communications between threads working in the same SM. The special function units calculate functions like sine, cosine, and square root. Since each SM has two thread schedulers and dispatch units, each SM is able to execute two different threads on its cores concurrently. The Fermi architecture supports double precision floating point operations.

Each processor core contains one operand collector, a floating point ALU unit, and an integer ALU. The processor cores are extremely simple, when compared to normal CPU cores. Since cores within an SM operate in lockstep fashion, the two schedulers can be shared between all of the cores. The load/store units are also shared

between all of the cores. Finally, the special function units are shared between the cores. All of these architectural features allow the GPU to dedicate many more transistors to its ALUs, giving it the arithmetic throughput it is known for (NVidia Corporation, 2008). See Figure 7.

Software Model

As mentioned before, a GPU follows the stream processing model. Therefore, it works by invoking one instance of a kernel on a single stream element. The operations that are possible for kernels to do on a GPU fall under several categories: map, reduce, stream filtering, scatter, gather, sort, and search.

The map operation applies a function to every input element, producing a modified output of the same size. The reduce operation takes an input stream and outputs a smaller output stream; it can be used to compute the sum or the maximum of a stream. Read and write operations on a GPU are called scatter and gather operations when memory is accessed indirectly. A gather operation is easily done, but a scatter operation is not implemented and requires extra programming steps. The stream filtering operation removes elements from the stream based on given criteria. The sort operation transforms an unordered set of elements to an ordered set of elements. The search operation is used to find a certain element within a stream, and possibly the nearest neighbors to that element (Owens et al., 2007).

Because of the limitations of the memory model of GPUs, the data structures most often used are two-dimensional arrays, also called textures. With these, it is possible to represent dense or sparse multidimensional arrays that are static or dynamic. Also, it is possible to store adaptive structures such as: quadtrees, octrees, kNN-grids, and k-d trees

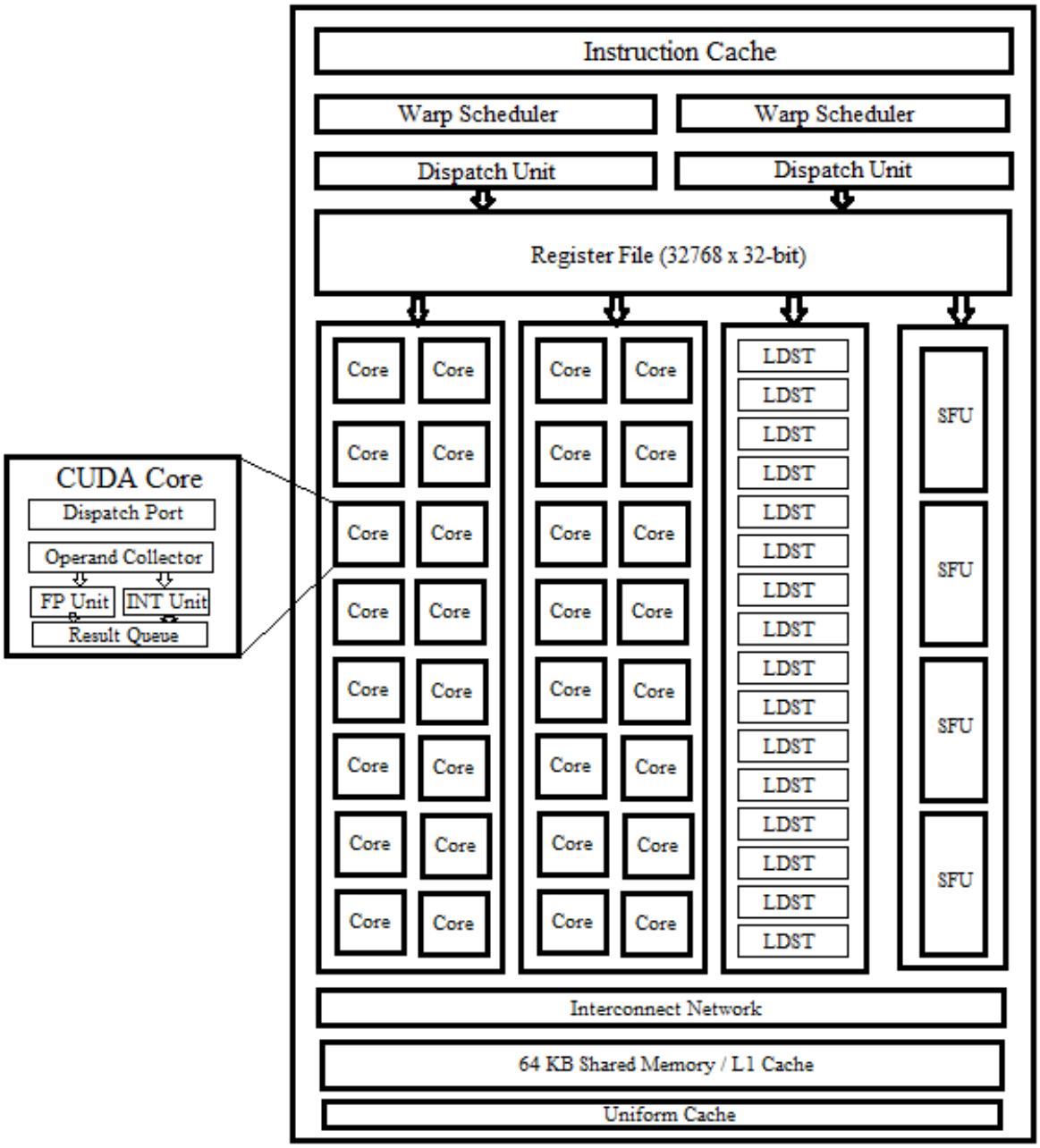

Figure 7. Fermi streaming multiprocessor. From "NVIDIA Fermi Compute Architecture Whitepaper," 2009, retrieved from NVidia website: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Copyright 2009 by NVIDIA Corporation. Reproduced with permission.

(Owens et al., 2007). All of these data structures require programming effort to implement, and are abstractions.

The programs written for this thesis will use the CUDA framework. The CUDA framework uses threads. A CUDA thread is organized by the compiler or the programmer into thread blocks and grids of thread blocks, where each thread belongs to a block, and each block belongs to a grid. Each thread is an instantiation of a kernel, with its own thread id, program counter, registers, private memory, inputs, and outputs. Each thread block contains shared memory for communication between threads. Also, grids can share results in global memory after thread synchronization. See Figure 8.

Strengths and Weaknesses

Computer graphics chips are today's most powerful computational hardware for their cost. Because of the economy of scale of the video game industry and its strong demand for 3D computer graphics, GPU cards have become relatively cheap (Boggan & Pressel, 2007). In 2005, the ATI X800 XT GPU was available for $447, and it could outperform a comparable CPU four times over (Owens et al., 2007). As of 2011, GPUs are used to power the most powerful computer on earth ("Tianhe-1," n.d.).

However, they have several weaknesses. Because of their lack of independent schedulers for each core, modern GPUs cannot perform conditional branching efficiently. If a conditional branch on a core diverges during execution from the rest of the cores in the SM, then the performance of the SM will degrade (Davis, 2001). This limits the types of algorithms that can achieve good performance on a GPU.

Because of its data access and caching architecture, GPUs require coalesced memory accesses to achieve the maximum throughput available. In practice this means

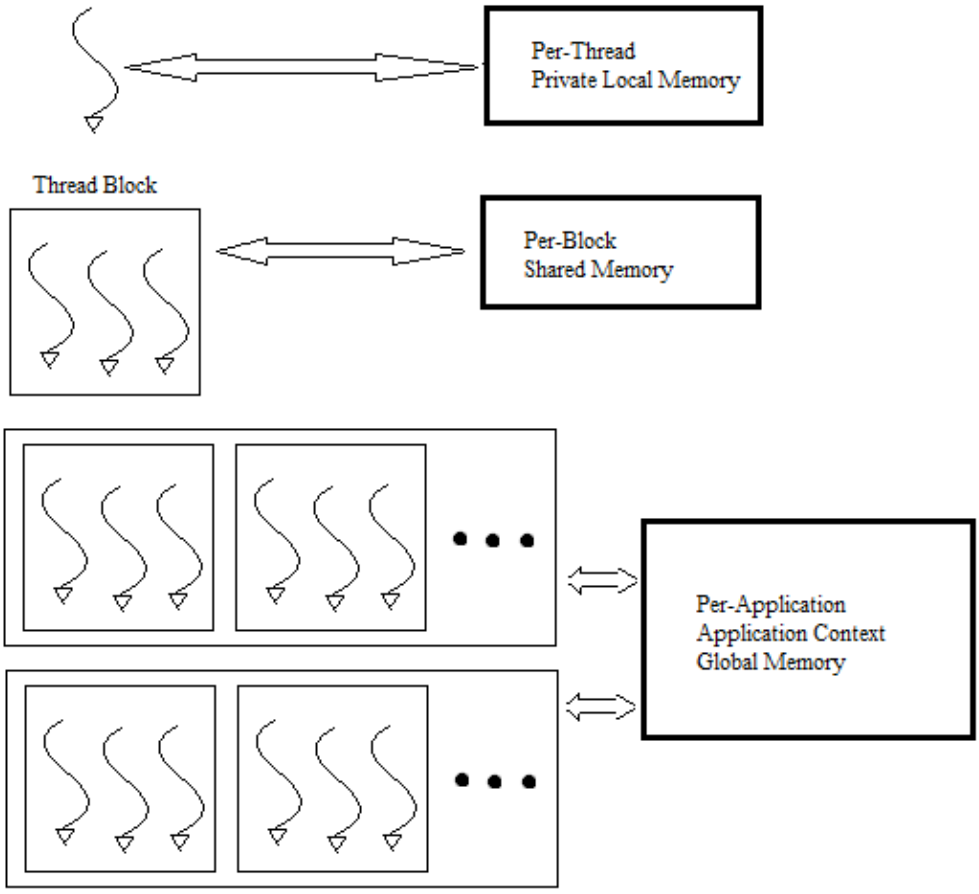

Figure 8. CUDA thread and memory hierarchy. From "NVIDIA Fermi Compute Architecture Whitepaper," 2009, Retrieved from NVidia website: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Copyright 2009 by NVIDIA Corporation. Reproduced with permission.

designing kernel programs so that each thread created from them will access adjacent memory locations which can be easily joined into one memory operation (Bell & Garland, 2008). This requirement limits the number of algorithms that can be ported to the platform.

GPUs are by design co-processors, and require a communication channel with the host processor. Therefore, time is required to transfer results back to main memory.

However, this limitation has been largely lifted by the introduction of the PCI-Express Bus (Boggan & Pressel, 2007).

Programming Recommendations

The following recommendations have been gathered from previous research. First, keep memory transfers between the CPU and GPU to a minimum. Second, keep the number of kernels used in a program to a minimum. And third, since modern GPU programming tools do Just-In-Time compilation for kernel programs, we should perform these only once in every program.

Hardware Used Throughout This Thesis

To test the claims of this thesis a workstation was used. The CPU is an AMD 6 core Phenom X6 running at 2.80 GHz and is a 64-bit CPU; it has 16 GB of memory. The same workstation is connected to an NVIDIA GEFORCE GTX 560 Ti graphics card, with a GF100 Fermi core running at 900 MHz. It has 1 GB of memory running at 1026 MHz; the memory interface is 256 bits wide and the memory is DDR5. The maximum theoretical bandwidth is 128.27 GB/second. The compute capability of the card is 2.1.

Suitability for Neural Networks and Reasoning

Because of the nature of neural networks, they are very well suited for execution on GPU. As previously mentioned, an algorithm must fulfill three requirements to be amenable to execution on a stream processor. Neural networks have high compute intensity per memory access. Neural networks have a high level of data parallelism since a neuron can execute separately from all others if its input data are available. And lastly,

neural networks access data in regular patterns, which helps in caching. Davis (2001) explains it like this:

> The GPU is very good at linear algebra operations because of the pipelined and parallel architecture. Most artificial neural networks rely heavily on linear algebra to implement their neurons, connection weights, activation functions, etc.; and as a result artificial neural networks are generally more amenable to execution on the GPU than general purpose computations. (p. 21)

Because of these reasons, the purpose of this thesis is combining the areas of neural networks and GPU programming, and finding ways to make the combination better.

Review of Previous Work

A simple feed-forward perceptron has been implemented for a GPU many times, and several papers have been published about it. Prabhu (2007) published a paper that included a description of back propagation learning on the GPU. He found that the CPU version performed better when the network was smaller than 100-200 neurons, because the GPU requires time to launch kernel programs and perform memory transfers. For larger networks, however, the GPU gained a marked advantage. Luo, Liu, and Wu (2005) also describe another implementation of an MLP on a GPU, resulting in a speedup of around 200 times. Lahabar, Agrawal, and Narayanan (2008) showed an implementation of an MLP with back propagation. Training of the network was 90-110 times faster than a MATLAB implementation and 120-140 times faster than a FANN implementation. The network ran 40 times to 50 times faster on the GPU than in MATLAB, and 55 times to 70 times faster than a FANN implementation. Both Moore (2009) and Hovorka (2010) used a GPU to run recurrent NNs.

Bohn (1998) coded an implementation of a Kohonen map on a GPU. Because of

the lack of programming tools for GPUs at the time, the program was written with OpenGL function calls. The SOM was able to achieve a speedup of 500% on large dimensional inputs, when compared to a CPU implementation. Luo et al. (2005) describe another implementation of a SOM, with similar results as Bohn's (1998); however, they use newer hardware and are able to remove some of the restrictions that Bohn had. Raina, Madhavan, and Ng (2009) coded an implementation of a Deep-Belief Network and sparse coding on a GPU, used to perform unsupervised learning on large data sets.

Neural networks running on GPUs have been used in several applications. For example, Bastke et al. (2009) used probabilistic neural networks running on GPUs to detect network intrusions. Scherer et al. (2010) used convoluted neural networks running on a GPU to speed up image recognition 8.6 times over a CPU version. In a similar paper, Uetz and Behnke (2009) used a GPU to train convolutional neural networks to recognize handwritten characters, achieving a speed-up of 80 times during normal operation and a maximum speed-up of 110 times for the back propagation training. Ciresan et al. (2010) used GPUs to speed up the training of deep NNs, used for handwritten letter recognition.

Veselý et al. (2010) used a GPU to train neural networks for speech recognition, on the real-world phoneme-state classification task, showing a nearly 10 times speed-up when using a CUDA version, as compared to a single-thread version. Sidiropoulos et al. (2009) used GPU to speed up the training of a probabilistic neural network in the diagnosis of mammograms, achieving significant speedups with identical classification results.

Bernhard and Keriven (2006) use GPUs in the similar field of spiking neuron

simulation, which seeks to simulate the way that biological neurons operate, achieving a speed-up of 5 times to 20 times. Bhuiyan, Pallipuram, and Smith (2010) achieved a speed-up of 9.5 times for a network of 5.8 million spiking neurons. Nageswaran, Dutt, Krichmar, Nicolau, and Veidenbaum (2009) achieved a speed-up of 24 times faster over a CPU version for the simulation of 100K neurons with 50 million synaptic connections, firing at an average rate of 7Hz. Yudanov, Shaaban, Melton, and Reznik (2010) implemented a spiking neuron simulation on the GPU with real-time performance, achieving a speed-up of 8 times to 9 times compared to a CPU simulation.

The need for linear mathematics operations in scientific computing and the necessity for high performance computing in scientific computing has generated a lot of research in the area of matrix and vector operations on GPUs. There have been many papers written about performing linear algebra operations on a GPU, and a few papers specifically on efficient sparse matrix multiplication on a graphics card. In an early paper, Larsen and McAllister (2001) implemented matrix multiplication on a GPU and concluded that the algorithm was limited by the low precision hardware available at the time; his GPU implementation was not faster than a comparable CPU version. Hall, Carr and Hart (2003) did similar research, but took into account the GPU's memory layout and connection to the CPU, and found that the execution times where about the same as a CPU implementation, the reason being the constraints of GPU programming as well as the low-bandwidth connection of the GPU to the CPU. Fatahalian et al. (2004) analyzed the performance of matrix multiplication operations and the previous work, and confirmed that GPUs were not able to outperform an optimized CPU implementation, despite the fact that matrix multiplication is a natural fit for stream processors. They

found that the GPU algorithm was able to use only between 17% and 19% of the arithmetic unit's time, despite the bandwidth usage being near the maximum capacity of the card, while CPU versions had nearly 65% arithmetic efficiency. They concluded that the algorithm used was near-optimal for the current hardware, and without significant redesign to the architecture of GPU, the performance would not get any better. With the development of GPU architectures, linear algebra performance improved considerably. Volkov and Demmel (2008) did a comparison of the performance of dense linear algebra algorithms on several modern GPU platforms, as well as an optimized CPU version. They found that performance scaled linearly according to the number of SPs on the GPU, and although the CPUs were able to achieve greater efficiency, they were still more than two times slower than most GPU implementations. With the improvement of hardware, linear algebra problems can now be solved efficiently on GPUs (Moravánszky, 2003).

With the improvement of linear algebra calculations on the GPU, it can be seen that NNs will correspondingly perform better on a GPU than on a CPU. However, the subject of this thesis is sparsely connected neural networks, and therefore what is needed is to find out about the performance of sparse linear algebra operations on the GPU. Bell and Garland (2008) have written a detailed analysis of sparse linear algebra on GPU hardware, including memory-saving storage methods and bandwidth efficient kernels. Their study also includes a performance analysis, comparing GPU performance to CPU performance. The authors find that the double precision sparse matrix vector multiplication performance of their algorithm is generally two and a half times that of a Cell BE with 8 SPs and more than 10 times greater than that of a quad-core Intel Clovertown system (Bell & Garland, 2008).

In-depth Description of the Problem and Motivation

As mentioned, neural network computations reduce easily to linear algebra operations. However, not all neural networks have the same structure and, therefore, it might be possible to tailor the linear algebra operations used to execute the neural network so that performance can be increased.

The data representing the weights of a neural network are most easily stored in a matrix format, where the row number indicates the neuron where the connection begins, and the column number indicates the neuron where the connection ends. In this representation, a missing connection between neurons would be represented by a zero. By this description, an SLP would require just one weight matrix, and an MLP would require two or more weight matrices for storage. Furthermore, if two layers in the network are partially connected, then the weight matrix between would also be sparsely populated.

Sparse matrix computation is a special field of linear algebra that deals with the representation and computation of matrices that contain many elements that are set to zero ("Sparse Matrix," n.d.). While dense operations are regular and are often limited by floating point throughput, sparse operations are much less regular and are often limited by bandwidth (Bell & Garland, 2008). Because of these findings, it might be useful to try a specialized algorithm to gain the benefits of less memory usage and faster computation.

By applying an optimization from the field of sparse linear algebra computations on GPU processors, a feed-forward perceptron that is partially connected will gain two things: less space used because of the compressed structure of the data structures used, and faster performance, because of the parallel structure of GPUs and the fewer calculations required by the optimized algorithm.

CHAPTER 2

METHODS USED

Explanation of the Proposed Method

There are many ways to compress sparse matrices, based on their features. Sparse matrices can be classified into two general categories: structured and unstructured matrices. Structured matrices have an easily distinguishable pattern in their non-zero entries which can be used to compress their information. Unstructured matrices don't have a pattern in their non-zero entries and require a more general compression scheme.

Bell and Garland (2008) provide a very good description of many formats especially designed for sparse matrix storage. In this thesis, however, it is not necessary to cover formats specifically designed for quick insertion and deletion of elements. Some formats are: the diagonal format (DIA), which is specialized for matrices containing their non-zero entries along the center diagonal, the ELLPACK format (ELL), which is specialized for matrices in which every row has close to the same number of zeroes, and the coordinate format (COO), which is a general storage format, since it can be applied to a matrix of any structure, consisting of a tuple for every no-zero entry containing the elements: row, column, data. The Compressed Sparse Row format is an extension of the COO format; it sorts the data for faster access and it adds a compression scheme for oft-repeated row numbers. The hybrid format (HYB) combines the best features of the COO

and ELL formats. The packet format (PKT) is used for diagonal matrices and vector architectures.

The proposed solution is composed of a new data structure and a new algorithm. The data structure will be described first. Neural networks don't have a pattern in the non-zero entries of their weight matrices, since this is very unlikely to happen in real-world learning; therefore the matrices containing the weights are unstructured.

The ELL format is very effective when the matrix has close to the same number of non-zero entries in every row. For a sparse matrix that is to be stored, a matrix is allocated with the same number of rows as the original matrix, and the maximum number of non-zero entries in any of its rows as the number of columns. Then the non-zero entries are packed into the smaller matrix, and then padded with zero values. The row number of a non-zero element in the matrix is stored in another matrix. As an example of a matrix:

A = [1 7 0 0]
[0 2 8 0]
[5 0 3 9]
[0 6 0 4]

Becomes:

Data = [1 7 *]
[2 8 *]
[5 3 9]
[6 4 *]

Indices= [0 1 *]
[1 2 *]
[0 2 3]
[1 3 *]

The coordinate format (COO) is a simple data structure which holds a three-element tuple for every non-zero element in the matrix. Each tuple contains the row index, the column index, and the data. The COO format is easy to use, and its storage requirements rise linearly with the number of non-zero elements in the matrix. As an example:

A = [1 7 0 0]
[0 2 8 0]
[5 0 3 9]
[0 6 0 4]

Becomes:

Row = [0 0 1 1 2 2 2 3 3]
Column = [0 1 1 2 0 2 3 1 3]
Data = [1 7 2 8 5 3 9 6 4]

By combining both formats one is able to get the benefits of both, while minimizing the drawbacks. The formats are combined by finding the typical number of non-zero entries per row in the sparse matrix, and then allocating an ELL matrix to store these entries, then using the COO format to store the remaining entries (Bell & Garland, 2008).

Similar to the data structure, the algorithm used for this optimization is a combination of the COO and ELL algorithm. In the ELL algorithm, one CUDA thread is assigned to each row of the matrix; the additions are performed in a for-loop and are summed together within the kernel and stored in the result vector. The matrices are linearized into vectors to enable the GPU to easily coalesce memory accesses. The COO algorithm uses a segmented reduction to allow it to assign one thread block to work on multiple rows of the matrix, with a final reduction across blocks to sum the results together. The COO algorithm is usually slower than the ELL algorithm; however, by combining both data structures and corresponding algorithms, the HYB algorithm outperforms all others on a GPU. Assuming that most non-zeros belong to the ELL portion, performance of the HYB sparse matrix vector multiplication operation will most resemble that of the ELL kernel (Bell & Garland, 2008).

According to the hardware model, in order for an algorithm to fully use the

resources of the GPU, it needs to coalesce memory accesses and keep threads executing in the same block from diverging in execution. According to Bell and Garland (2008), the algorithm and data structure with the highest FLOP performance for single-precision unstructured matrices is the hybrid format, therefore this format was chosen for this thesis's tests. This format also has fairly good data compression.

Analytical Evaluation of the Proposed Method

Space Complexity

The memory usage of each weight matrix in the un-optimized algorithms is described by this formula:

$$\text{memory used}_{\text{layer}} = m^2$$

where m is the number of surons per layer. The total memory used for the neural network is:

$$\text{memory used}_{\text{total}} = \sum_{k=0}^{n} \text{memory used}_{k}$$

where n is the total number of layers in the neural network, and memory used_{k} is the amount of memory used in that particular weight matrix.

The data structure design for each un-optimized algorithm is slightly different, which will cause the programs to give slightly different total memory usages, when they should be the same. The GPU un-optimized version uses "flattened" matrices to more easily transfer its data, therefore it uses fewer pointers, which means it will use a different amount of memory than an equivalent CPU un-optimized neural network.

As mentioned, the compression scheme that was chosen for the tests is the HYB format. This format uses a combination of the ELL format and the COO format. The

memory usage of the optimized algorithm is described by the formula:

$\text{memory used}_{ell} = 2(i*j)$

where i is the number of columns in the ELL portion of the data structure, and j is the number of rows in the ELL portion of the array.

$\text{memory used}_{coo} = 3k$

where k is the number of elements in the COO portion of the matrix. To combine the two formulas:

$\text{memory used}_{layer} = \text{memory used}_{ell} + \text{memory used}_{coo}$

The heuristic used to divide a sparse matrix is described like this: "Our implementation computes a histogram of the row sizes and determines the largest number K such that using K columns per row in the ELL portion of the HYB matrix meets a certain objective measure" (Bell & Garland, 2009, p. 5). The objective measure is determined from statistics gathered beforehand. To calculate the amount of memory used by a neural network using this compression scheme, this formula is used:

$\text{memory used}_{total} = \sum_{k=0}^{n} \text{memory used}_{k}$

where n is the total number of layers in the neural network, and memory used_{k} is the amount of memory used in that particular weight matrix in the network.

The best compression ratio is achieved when the matrix has the same number of non-zero elements in every row, allowing the program to use the allocated ELL space fully, not needing to use the less-efficient COO format. The worst compression ratio occurs when all of the non-zero elements are grouped in one row of the matrix. This forces almost all of the elements to be stored in the COO portion of the data structure. The compression ratio is hard to determine beforehand without knowing details about the

structure of the sparse matrix to be compressed.

Time Complexity

I will now analyze the time complexity of every algorithm individually. The non-optimized CPU algorithm has a time complexity of:

$jm^2 + jn$

where j is the number of layers, m is the number of rows and columns (since all of the matrices in this thesis are square), and n is the computational cost of the transfer function (a fixed cost).

The non-optimized GPU algorithm has a time complexity of:

$$j\left(\frac{m^2+n}{k}\right)$$

where j is the number of layers, m is the number of rows and columns, n is the cost of the transfer function, and k is the thread parallelization factor. The thread parallelization factor is introduced to describe the hardware configuration of a GPU, where each row in a matrix vector product is calculated by a single thread running in parallel with many others. The factor k is affected by many runtime variables, such as the GPU model used, the presence of other programs in the system using the GPU, etc. In this formula, j also includes the kernel launch time, which is an overhead incurred by the hardware used.

As previously mentioned, the algorithm used for the HYB data structure is a combination of the ELL and COO kernels. Bell and Garland (2008) state that the performance of the HYB kernel is most influenced by the ELL portion of the combined kernel, since the ELL portion of the data structure usually holds more information. The best-case scenario for the algorithm would occur when the non-zero values are distributed

evenly across all rows, allowing each thread to process the same amount of elements, producing fully-coalesced memory accesses and non-divergent execution. The worst-case scenario would occur when all of the non-zero values are located on one row of the matrix, meaning that one thread would be required to handle all of the elements in the matrix.

To solve for the optimized version of the algorithm, a special variable needs to be introduced: n_z which stands for the proportion of elements in a certain matrix that are not zero.

$$j(\frac{n_z * m^2 + n}{k})$$

where j is the number of layers, m is the number of rows and columns, n is the cost of the transfer function, k is the thread parallelization factor, and n_z is the proportion of non-zero elements in the matrix to the total number of possible elements in the matrix. This factor can take any real value in the range: [0, 1].

The worst case for all algorithms can be described by the big O notation as:

$$O(x^2)$$

where x is the number of neurons per layer.

The hardware that I am dealing with in this thesis contains many cores, and must be dealt with specifically. Amdahl's law tells us what speed-up we can expect when using a single processor. Hill (2008) did further research that updated Amdahl's law to reflect the rise of multicore processors like GPUs. Figure 9 shows the predicted speed-up of a program running on multiple cores. Factors include: software fraction that is parallelizable (f), total chip resources in BCEs (n), and the BCE resources (r) devoted to increase the performance of each core. The x axis is the number of cores, and the y axis is

the predicted speed-up. The formula used to generate this graph came directly from Hill (2008, p. 2):

$$\text{Speedup}\ (\text{f}, \text{n}, \text{r}) = \frac{1}{(\frac{1-\text{f}}{\text{perf(r)}})+(\frac{\text{f}*\text{r}}{\text{perf(r)}*\text{n}})}$$

where perf(r) is equal to $\sqrt{r}$. For more details, see Hill (2008).

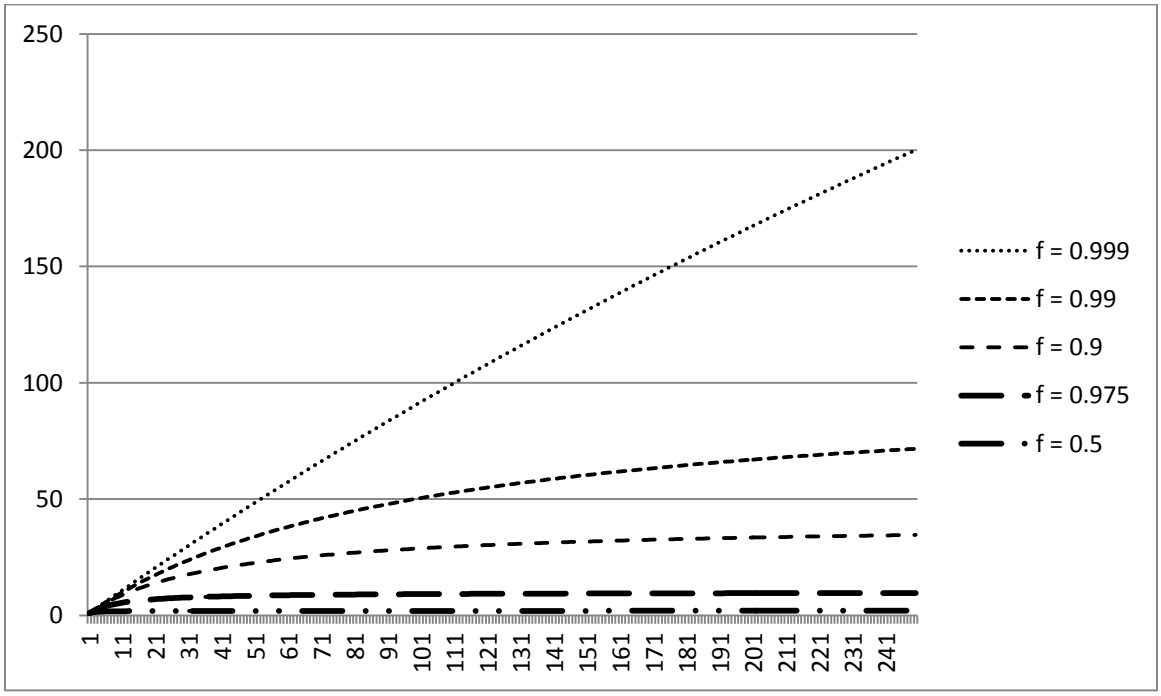

Figure 9. Speed-up of symmetric multicore chips.

Description of the Experiments

For this thesis, three versions of the feed-forward part of the ANN algorithm were tested. Specifically: the ANN feed-forward algorithm for the CPU without any optimization and the ANN feed-forward algorithm for the GPU with and without the optimization. In the GPU programs, the memory transfer time was not included in the execution time, but the kernel launch time was included.

I created three programs to perform the tests; the programs received parameters from a script, saving their results to a file. The first program uses only the CPU, and it is used as a baseline. I modified the code from Chhabra (2006) for my own purposes. The second program is an un-optimized, naïve implementation of the algorithm on the GPU using CUDA; I coded this program myself based on the work of Steinkraus, Buck, and Simard (2005), Scherer et al. (2010), and Chellapilla, Puri, and Simard (2006). The third program uses the CUSP library, which is a library specialized for performing sparse linear algebra on the CUDA framework, based on the research from Bell and Garland (2008). The programs run on a UNIX system and use a system service to measure elapsed time, accurate down to the nanosecond. Each program performs each test a certain number of times, and the results are then averaged and saved. This is done so that a single discrepancy in the results does not affect the final result. For every test, a neural network is generated according to the parameters passed to the program and then filled in with the correct number of connections; the connections are placed randomly between the layers in the ANN.

Two separate sets of tests were run: one set to measure the time complexity of the algorithm, and one set to measure the space complexity of the algorithms. While holding all other variables at the same level, these variables were changed: the number of connections between layers, the number of neurons in a layer, and the number of layers in the tested ANN. This makes for a set of 18 tests that were used to measure the performance of this optimization. Since the variables are the same for sets of three tests, one from each algorithm coded, the results are directly comparable; therefore, six separate charts have been generated to compare the results from the three algorithms. The

default values for each test were: 50 neurons per layer, 5 layers, and fully connected layers.

To simplify testing, all layers in an ANN had the same number of neurons, meaning that all of the weight matrices were the same size. The neural networks used in this thesis were specifically designed to show the performance of certain parts of each algorithm, and not to test every possible topology.

Analysis of the Results

In Figure 10, the bound variable is the number of neurons per layer, which ranges from 1 neuron to 100 neurons, giving 100 data points. The unbound variable is execution time. Other variables are: number of layers, which is equal to five in this test, and connection proportion, which is 100% in this test. The results for the CPU un-optimized and GPU un-optimized programs agree with both results from previous research. As shown in previous research it is more efficient to execute smaller neural networks in the CPU. The optimized GPU algorithm performed the feed-forward pass in less time than the other algorithms in every portion of this test.

In Figure 11, the bound variable is the number of neurons per layer, which ranges from 1 neuron to 100 neurons, giving 100 data points. The unbound variable is memory space. Other variables are: number of layers, which is equal to five in this test, and connection proportion, which is 100% in this test. The un-optimized CPU version and the un-optimized GPU version of the algorithm used about the same amount of memory, with the un-optimized GPU version using much more memory.

In Figure 12, the bound variable is the number of layers, which ranges from 1 neuron to 100 layers, giving 100 data points. The unbound variable is execution time.

Other variables are: number of neurons per layer, which is equal to 50 in this test, and connection proportion, which is 100% in this test. The results for the CPU un-optimized and GPU un-optimized programs are as expected from previous research, both of them increasing linearly with the number of layers. The optimized GPU program performed the feed-forward pass faster in every portion of this test.

In Figure 13, the bound variable is the number of layers, which ranges from 1 neuron to 100 layers, giving 100 data points. The unbound variable is memory space. Other variables are: number of neurons per layer, which is equal to 50 in this test, and connection proportion, which is 100% in this test. The un-optimized CPU and un-optimized GPU version of the algorithm used almost the same amount of memory. The optimized GPU version of the algorithm used significantly less memory than the others; however, the trend that it follows would cause it to use more memory that its counterparts if the test continued.

In Figure 14, the bound variable is the connection proportion, which ranges from 1% to 100%, giving 100 data points. The unbound variable is execution time. Other variables are: number of neurons per layer, which is equal to 50 in this test, and the number of layers, which is set to 5. The optimized GPU algorithm performed significantly better than the un-optimized CPU and un-optimized GPU versions. In all three tests that measured execution time, the optimized version of the algorithm performed the feed-forward pass faster than the un-optimized algorithms.

In Figure 15, the bound variable is the connection proportion, which ranges from 1% to 100%, giving 100 data points. The unbound variable is memory space. Other variables are: number of neurons per layer, which is equal to 50 in this test, and the

number of layers which is set to 5. The un-optimized CPU and un-optimized GPU versions of the algorithm used a constant amount of memory, which means that no matter how many connections exist in the layers of the networks, the same amount of memory will be used. The optimized GPU algorithm used less memory than the others, when the connection proportion was less than about 30%.

From the data produced by the programs, it is evident that the predictions of this thesis have proven to be true. This thesis claimed that any feed-forward perceptron that is partially connected would be able to benefit from less space used in memory and less processing time used in its feed-forward pass. As predicted, the partially connected networks used less memory than the fully connected networks. However, every network that executed with the optimized GPU algorithm was able to execute faster than the non-optimized CPU and GPU versions of the algorithm, regardless of the level of connectivity of the network. The results of three tests are graphed in Figures 10-15. In all tests, time was measured in nanoseconds, and space was measured in bytes.

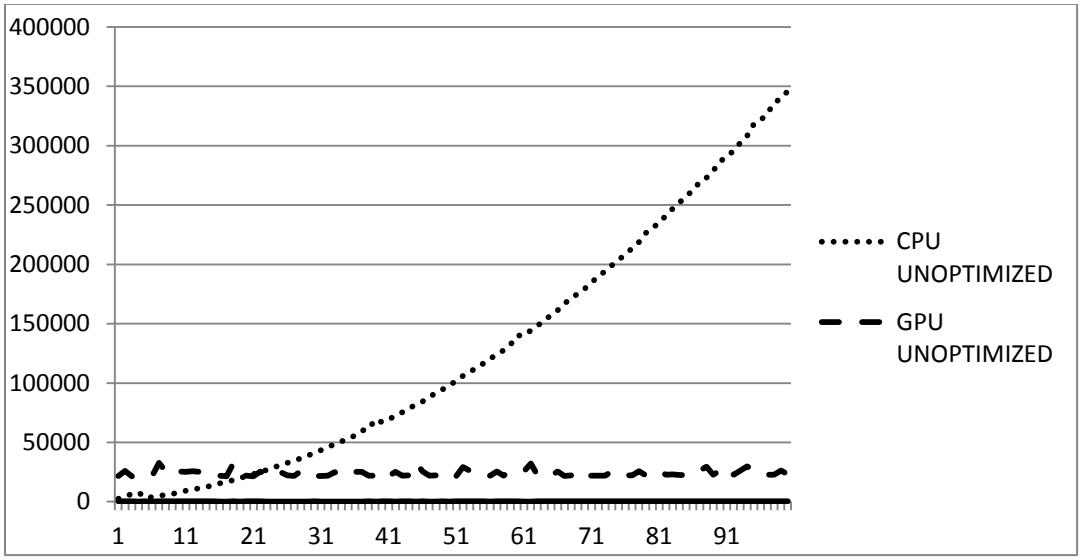

Figure 10. Test results, x: number of neurons per layer, y: execution time.

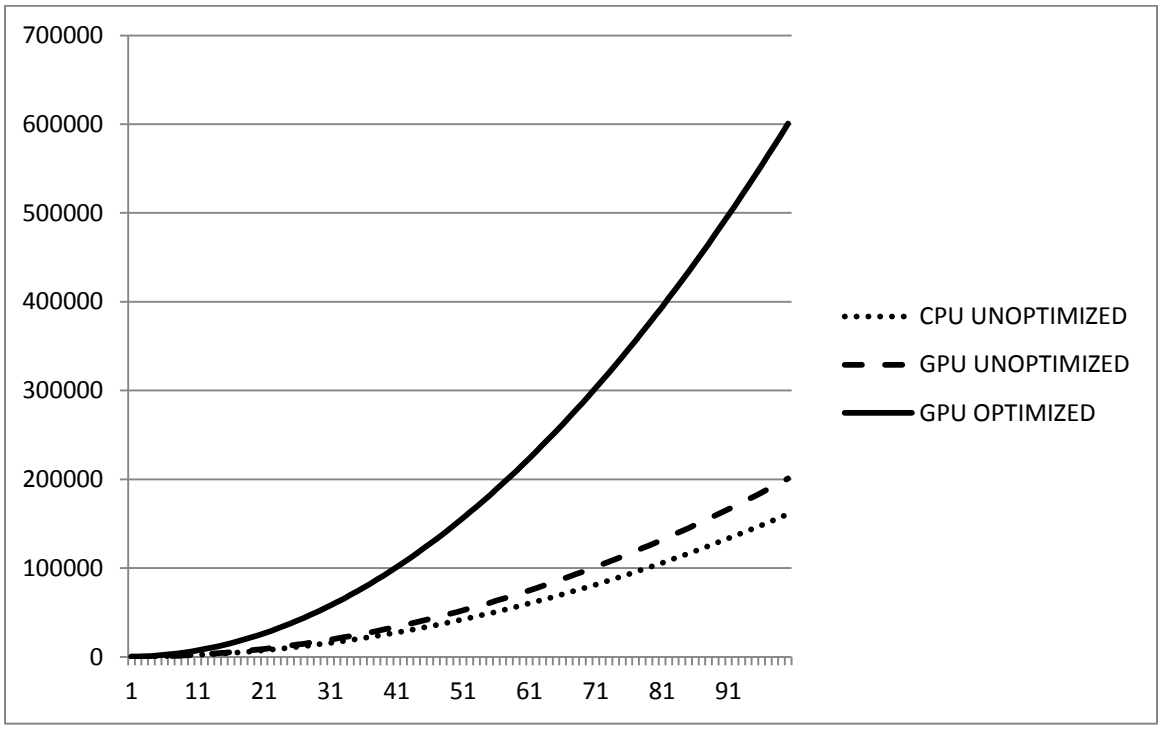

Figure 11. Test results, x: number of neurons per layer, y: memory used.

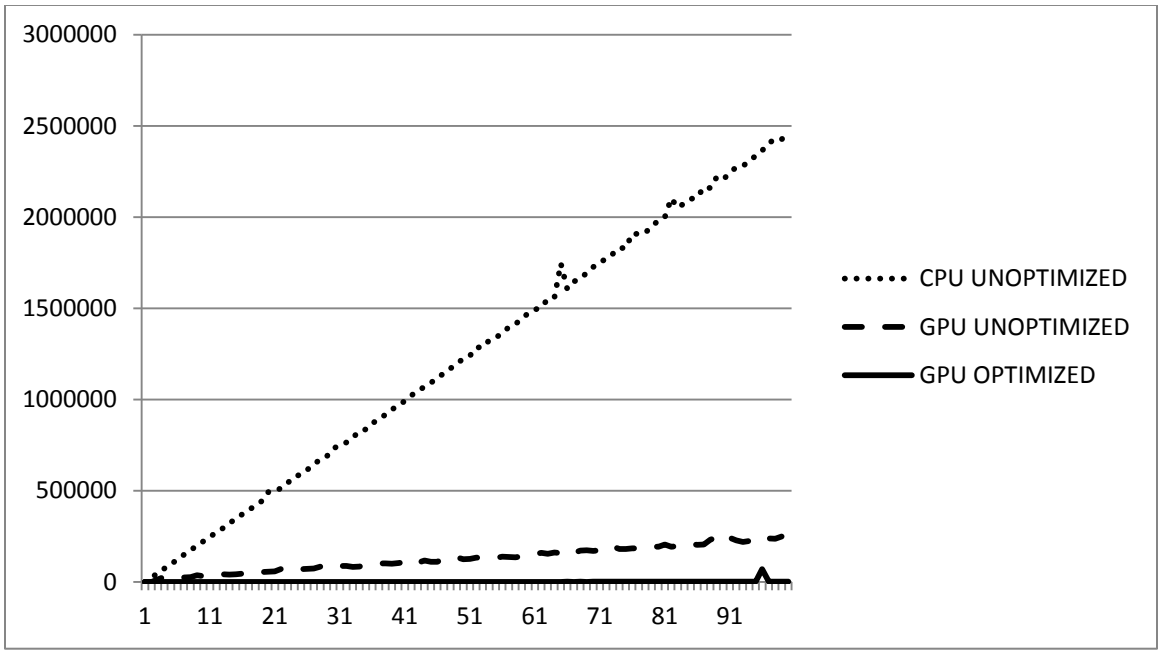

Figure 12. Test results, x: number of layers, y: execution time.

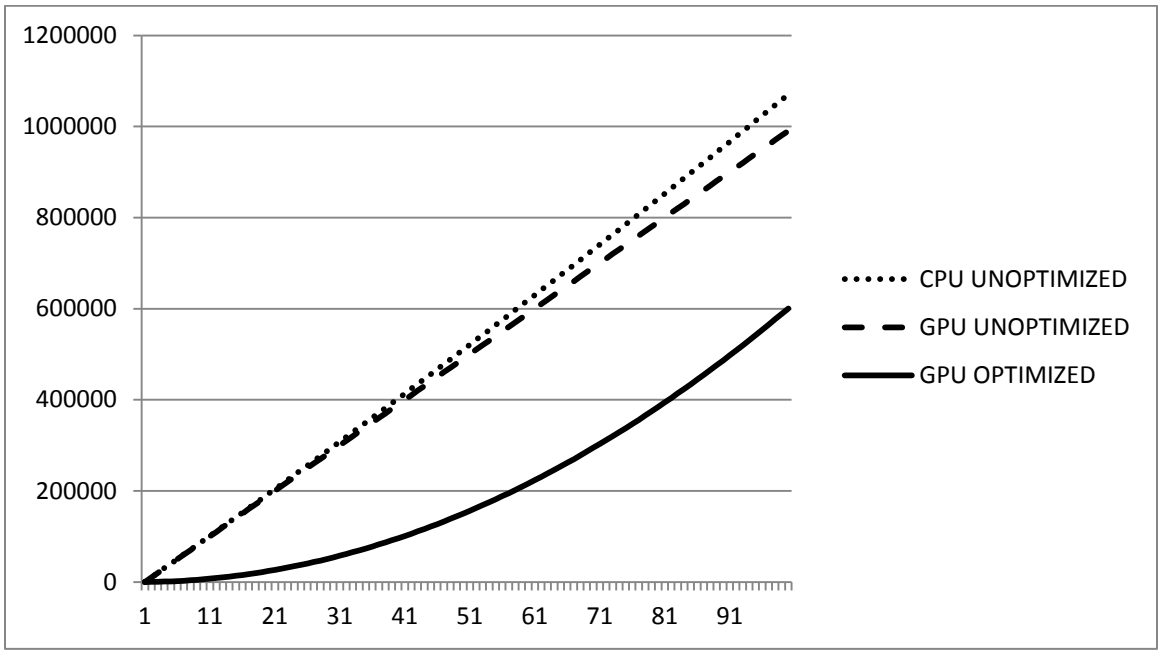

Figure 13. Test results, x: number of layers, y: memory used.

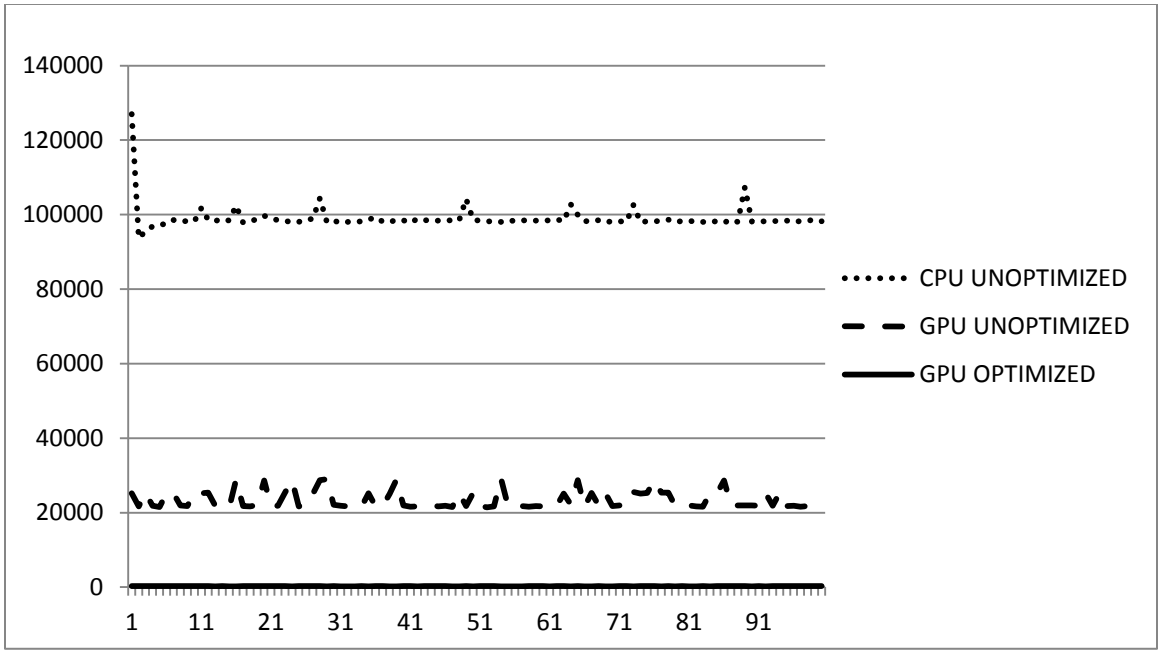

Figure 14. Test results, x: number of connections per layer, y: execution time.

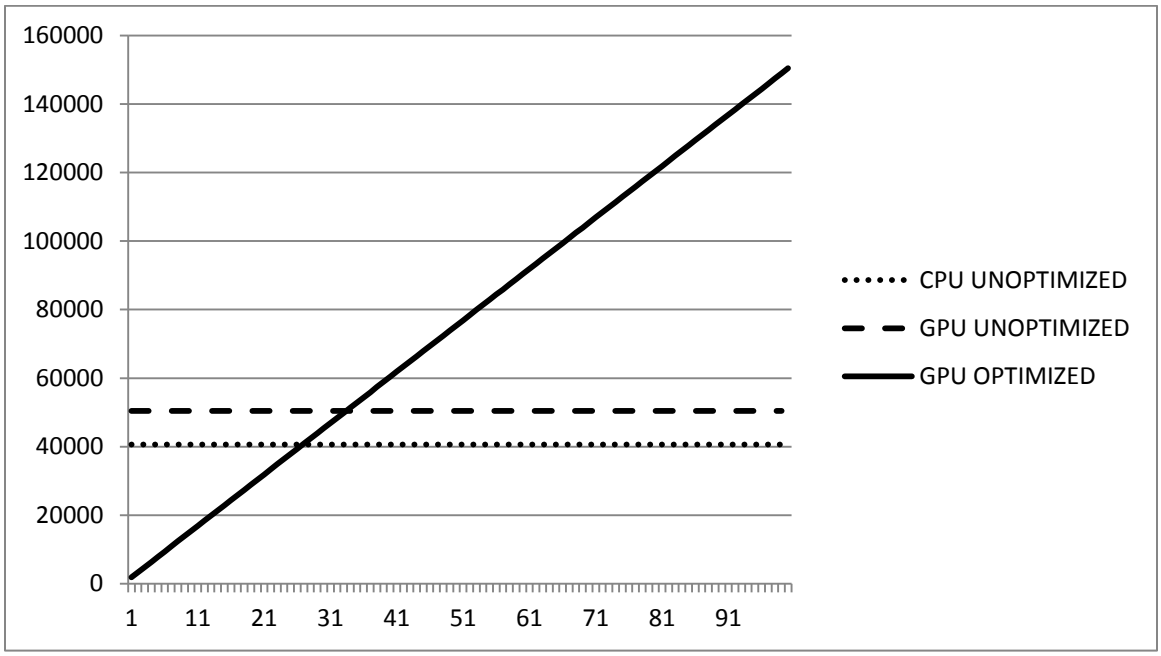

Figure 15. Test results, x: number of connections per layer, y: memory used.

CHAPTER 3

CONCLUSION

Statement of the Results

The data prove that it is feasible to use this optimization to perform the feed-forward pass on a perceptron faster when running in a GPU, in the right situation. Also, in situations where the inter-layer connectivity of the NN is less than 30%, it takes less space to use the optimized GPU algorithm.

Conclusion

Previous research dealing with neural networks running in GPUs has proven that it is a very good platform for the problem. This thesis has sought to extend the research by trying to apply an optimization from a related field.

In terms of execution time, the optimized algorithm proved to be faster than the un-optimized algorithms in all three tests. In terms of storage space used, the optimized algorithm used more space than the un-optimized algorithms in two of the three tests performed, and only used less memory than the un-optimized algorithms when the level of connectivity of the neural network tested was less than around 30%.

Future Work

There are several ways in which this research can be continued. There are many

different storage formats for sparse matrices, and this thesis used only the most promising one; other storage formats could be tested and their performance measured.

The number of tests that were run was limited by the time available to implement the programs and run the tests. To keep the results simple and easy to understand, this thesis deals only with a small part of the possible topologies of a neural network. Further research could be done on non-square matrices and their effect on the performance of the algorithm.

The neural networks programmed on the GPU required that a kernel be launched for every layer in the network. This is a shortcoming in the design of the GPU programs that were used. However, it might be possible to code the algorithm in one kernel and therefore make the execution time more efficient.

REFERENCE LIST

Angelov, R. (2010, April 3). *Basic neural network tutorial–theory*. Retrieved February 13, 2011, from Taking Initiative: Boby Angelov's Blog: http://takinginitiative.net/2008/04/03/basic-neural-network-tutorial-theory/

Artificial neural networks. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved March 22, 2011from http://en.wikipedia.org/wiki/Artificial_Neural_Networks

Bastke, S., Deml, M., & Schmidt, S. (2009). *Combining statistical network data, probabilistic neural networks and the computational power of GPUs for anomaly detection in computer networks.* In *Workshop on Intelligent Security*. Thessaloniki, Greece. Retrieved from http://www.tzi.de/~edelkamp/secart2/papers/Bastke.pdf

Bell, N., & Garland, M. (2008). *Efficient sparse matrix-vector multiplication on CUDA.* Santa Clara, CA: Nvidia Corporation.

Bell, N., & Garland, M. (2009). *Implementing sparse matrix-vector multiplication on throughput-oriented processors*. Retrieved from http://www.nvidia.com/docs/IO/77944/sc09-spmv-throughput.pdf

Bernhard, F., & Keriven, R. (2006). Spiking neurons on GPUs. *International Conference on Computational Science* (pp. 236-243). Reading, UK: Springer.

Bhuiyan, M., Pallipuram, V., & Smith, M. (2010, April 19). *Acceleration of spiking neural networks in emerging multi-core and GPU architectures*. Paper presented at the Nitnth IEEE International Workshop on High Performance Computational Biology, Atlanta, GA. Retrieved from http://www.hicomb.org/HiCOMB2010/papers/HICOMB2010-02.pdf

Boggan, S., & Pressel, D. (2007). *GPUs: An emerging platform for general purpose computation.* Department of the Army, Army Research Laboratory, Aberdeen Proving Ground, MD. Retrieved from http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA471188

Bohn, C. (1998, October). *Kohonen feature mapping through graphics hardware. Joint Conference on Information Sciences*. Paper presented at the Third International Conference on Computational Intelligence and Neurosciences, Research Triangle Park, NC. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=A001B748749C4CA7B1EC97839D91CC0F?doi=10.1.1.54.7109&rep=rep1&type=pdf

Brain. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved February 23, 2011 from http://en.wikipedia.org/wiki/Brain

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., & Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel Distributed Computing*, 68, 1370-1380. doi: 10.1016/j.jpdc.2008.05.014

Chellapilla, K., Puri, S., & Simard, P. (2006, October). *High performance convolutional neural networks for document processing.* Paper presented at the Tenth International Workshop on Frontiers in Handwriting Recognition, La Baule, France. Retrieved from http://research.microsoft.com/~kumarc/pubs/chellapilla_iwfhr2006_ConvNNGPU.pdf

Chhabra, T. (2006, March 28). *Back-propagation Neural Net*. Retrieved March 12, 2011, from Code Project: http://www.codeproject.com/KB/recipes/BP.aspx

Ciresan, D. C., Meier, U., Gambardella, L. M., & Schmidhuber, J. (2010, November). *Deep big simple neural nets excel on handwritten digit recognition.* Paper presented at the Tenth ACM/IEEE Supercomputing Conference, New Orleans, LA. Retrieved from http://arxiv.org/PS_cache/arxiv/pdf/1003/1003.0358v1.pdf

Davis, C. E. (2001). *Graphics processing unit computation of neural networks* (master's thesis). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.6961&rep=rep1&type=pdf

Elizondo, D., & Fiesler, E. (1997). A survey of partially connected neural networks. *International Journal of Neural Systems*, 8, 535-558. doi: 10.1142/S0129065797000513

Emergence. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved February 23, 2011from http://en.wikipedia.org/wiki/Emergence

Fatahalian, K., Sugerman, J., & Hanrahan, P. (2004, August 30). *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication.* Paper presented at the Eight ACM SIGGRAPH Conference on Graphics Hardware, Grenoble, France: Retrieved from http://graphics.stanford.edu/papers/gpumatrixmult/gpumatrixmult.pdf

Hall, J., Carr, N., & Hart, J. (2003). *Cache and bandwidth aware matrix multiplication on the GPU.* Retrieved from http://www.cis.upenn.edu/~suvenkat/700/lectures/7/UIUCDCS-R-2003-2328-1.pdf: UIUC Technical Report UIUCDCS-R-2003-2328

Hebb, D. (1949). *The organization of behavior: a neuropsychological theory.* New York: Wiley.

Hill, M. D. (2008, July). Amdahl's law in the multicore era. *IEEE Computer,* 41(7), 33-38.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, USA 79,* 2554-2558. doi:10.1073/pnas.79.8.2554

Hovorka, M. (2010). *Training of neural networks using CUDA technology* (bachelor's thesis). Retrieved from https://dip.felk.cvut.cz/browse/pdfcache/hovorma1_2010bach.pdf

Kriesel, D. (n.d.). *A Brief Introduction to Neural Networks.* Retrieved from: http://www.dkriesel.com

Krüger, J., & Westermann, R. (2005). Linear algebra operators for GPU implementation of numerical algorithms. *ACM SIGGRAPH,* 22, 908-916. doi: 10.1145/1201775.882363

Lahabar, S., Agrawal, P., & Narayanan, P. (2008, December). *High performance pattern recognition on GPU.* Paper presented at the Sixth National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics, Bhubaneswar, India. Retrieved from http://cvit.iiit.ac.in/papers/Sheetal08High.pdf

Larsen, E. S., & McAllister, D. (2001, November). *Fast matrix multiplies using graphics hardware*. Paper presented at the First Super Computing Conference, Denver, CO. Retrieved from http://www.sc2001.org/papers/pap.pap313.pdf

Lee, J., Choi, I., & Kim, H. (2003). Natural frequency-based neural network approach to radar target recognition. *IEEE Transactions on Signal Processing*, 51, 3191 – 3197. doi: 10.1109/TSP.2003.818908

Luo, Z., Liu, H., & Wu, X. (2005, August). *Artificial neural network computation on graphic process unit*. Paper presented at the 16th IEEE International Joint Conference on Neural Networks, Montréal, Canada. Retrieved from http://xgxy.cug.edu.cn/xgxynewweb/jsfc/lzw/ANN_GPU.pdf

McCulloch, W., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology,* 5(4), 115–133.

Minsky, M., & Papert, S. (1969). *Perceptrons.* Cambridge, Massachusetts: MIT Press.

Moore, C. (2009). *A recurrent neural network implementation using the graphics processing unit* (master's thesis). Retrieved from http://ir.library.oregonstate.edu/xmlui/bitstream/handle/1957/13658/MooreChristopherE2009.pdf?sequence=6

Moravánszky, Á. (2003). Dense matrix algebra on the GPU. In W. F. Engel (Ed.), *Direct3D ShaderX2* (pp. 1-22). Zurick, Switzerland: Wordware Publishing.

Nageswaran, J., Dutt, N., Krichmar, J., Nicolau, A., & Veidenbaum, A. (2009, August). Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. Paper presented at the 20th International Joint Conference on Neural Networks, Atlanta, Georgia. Retrieved from http://www.ics.uci.edu/~jmoorkan/pub/gpusnn-ijcnn.pdf

NVidia Corporation. (2009). *NVIDIA's Next Generation CUDA compute architecture: Fermi.* Retrieved April 22, 2011, from nvidia.com: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Owens, J., Luebke, D., N., G., Harris, M., Krüger, J., Lefohn, A., & Purcell, T. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 80-113. doi: 10.1111/j.1467-8659.2007.01012.x

Prabhu, R. (2007, November). *GNeuron: parallel neural networks with GPU*. Paper presented at the 19th International Conference on High Performance Computing, Reno, NV. Retrieved from http://www.hipc.org/hipc2007/posters/GNeuron.pdf

Raina, R., Madhavan, A., & Ng, A. (2009, June). *Large-scale deep unsupervised learning using graphics processors*. Paper presented at the 26th International Conference on Machine Learning, Montreal, Canada. Retrieved from http://www.machinelearning.org/archive/icml2009/papers/218.pdf

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing: explorations in the microstructure of cognition.* Cambridge, MA: MIT Press.

Scherer, D., Schulz, H., & Behnke, S. (2010, September). *Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors*. Paper presented at the 20th International Conference on Artificial Neural Networks, Thessaloniki, Greece. Retrieved from www.ais.uni-bonn.de/papers/nips09ws_scherer_behnke.pdf

Sidiropoulos, K., Cavouras, D., Pagonis, N., Dimitropoulos, N., & Stonham, J. (2009, July). *From the accelerating the design of probabilistic neural networks for computer aided diagnosis in mammography, employing graphics processing units*. Paper presented at the Third International Conference on Experiments/Process/System Modeling/Simulation/Optimization, Athens, Greece. Retrieved from http://e-jst.teiath.gr/issue_15/Sidiropoulos_15.pdf

Sparse Matrix. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved February 23, 2011from http://en.wikipedia.org/wiki/Sparse_matrix

Steinkraus, D., Buck, I., & Simard, P. (2005). Using GPUs for machine learning algorithms. *Proceedings of the Eight International Conference on Document Analysis and Recognition*, 2, 1115-1120. doi: 10.1109/ICDAR.2005.251

Stream Processing. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved February 22, 2011from http://en.wikipedia.org/wiki/Stream_processing

Tianhe-1. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved February 22, 2011from http://en.wikipedia.org/wiki/Tianhe-I

Uetz, R., & Behnke, S. (2009, December). *Locally-connected hierarchical neural networks for GPU-accelerated object recognition.* Paper presented at the 18th Workshop on Large-Scale Machine Learning: Parallelism and Massive Datasets, Whistler, BC. Retrieved from http://www.is.unibonn.de/papers/nips09ws_uetz_behnke.pdf

Veselý, K., Burget, L., & Grézl, F. (2010, September). *Parallel training of neural networks for speech recognition.* Paper presented at the 13th International Conference on Text, Speech and Dialogue, Brno, Czech Republic. Retrieved from http://www.fit.vutbr.cz/research/groups/speech/publi/2010/vesely_EEITC_2010_39.pdf

Volkov, V., & Demmel, J. (2008, November). *Benchmarking GPUs to tune dense linear algebra*. Paper presented at the Eight ACM/IEEE Conference on Supercomputing, Austin, TX. Retrieved from http://mc.stanford.edu/cgibin/images/6/65/SC08_Volkov_GPU.pdf

Werbos, P. (1974). *Beyond regression: new tools for prediction and analysis in the behavioral sciences* (Doctoral dissertation, Harvard University). Retrieved from http://www.google.com/url?sa=t&source=web&cd=1&ved=0CCQQFjAA&url=http%3A%2F%2Fciteseer.ist.psu.edu%2Fviewdoc%2Fdownload%3Bjsessionid%3D1E2022D8BF09C5B494B1CBDE0F9EE26D%3Fdoi%3D10.1.1.41.8085%26rep%3Drep1%26type%3Dpdf&ei=UhaNTo3EAu3LsQLd_IiWAQ&usg=AFQjCNGN_tmIgk2kX4ed_OGy4g7O1RevQg&sig2=9Q7FgyDXcXaymAfpWzdvjg

Yudanov, D., Shaaban, M., Melton, R., & Reznik, L. (2010, July). *GPU-based simulation of spiking neural networks with real-time performance & high accuracy*. Paper presented at the 21st International Joint Conference on Neural Networks, Barcelona, Spain. Retrieved from http://www.cs.ucl.ac.uk/staff/W.Langdon/cigpu2010/papers/n-0573.pdf