Graphics Processing Unit Computation of Neural Networks

by

Christopher Edward Davis

B.S., Computer Science, University of New Mexico, 2001

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

October, 2005

Dedication

To my parents, my wife and my friends for their never ending support.

Acknowledgments

It is not enough to help the feeble up, but to support him after.
William Shakespeare (1564 - 1616)

My career, ideas, and hopes have been shaped by so many people over the course of my life. It is impossible to list all these people here, but I can list the people who have helped inspire me to pursue this line of research, the people who have guided my direction, and the people who have kept me going when life looked bleak. Without these people the ideas contained in this thesis may have never seen the light of day. In particular, I would like to thank:

Professor Edward Angel, my advisor, for his support, advice, testing platform access, and perhaps most importantly patience which allowed me to explore a wide range of topics;

Amy Davis, my wife, for her unconditional support, understanding, guidance, companionship and excellent editing skills;

Professor Thomas Caudell, my on and off again employer and committee member, for opening my mind to a rich and worth-while field of study, providing guidance through the complexities of research, and providing me with enriching and challenging employment;

Professor George Luger, my friend and committee member, for broadening my understanding of the field of Artificial Intelligence, sharing entertaining stories, and providing helpful information on anything I ever inquired about;

Steve Smith, my current employer, for giving me enough intellectual rope to hang myself (or not), tolerating my sometimes whimsical seeming explorations into new or novel approaches to problems, and always having some obscure metaphorical reference or pun that makes me smile;

Takeshi Hakamata, for being an engaging consult on many of the topics and approaches presented in this thesis;

Steve Linger, my Los Alamos National Lab mentor, for helping guide me through the complexities of employment at LANL, and providing words of encouragement about my research and my scientific vigor;

Henry Davis, Irene Davis, Jennifer Foraker, my family, for always being there to support me financially and emotionally, shaping the person I am and the way I approach the world, and providing immeasurable support though my life;

Mark Fleharty, Bertram Goodman, Clint Sulis, my friends, for helping me experience adventures and fun, even in the most trying of times;

Los Alamos National Laboratory, for providing complementary work and employment during the last half of this research;

To all who have been influential figures in my life, provided spiritual or intellectual guidance, appealed to my senses or my desire for adventure, or have otherwise challenged me to think in new ways, I thank you.

1

2

3

[1]ATI, the ATI logo, and other ATI marks are used under license and are registered trademarks of ATI Technologies Inc. in the United States and other countries.

[2]NVIDIA, the NVIDIA Logo, Cg, and other NVIDIA Marks are registered trademarks or trademarks of NVIDIA Corporation in the United States and other countries.

[3]Microsoft, Windows, DirectX, Direct3D, and High Level Shader Language (HLSL) are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Graphics Processing Unit Computation of Neural Networks

by

Christopher Edward Davis

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico

Albuquerque, New Mexico

October, 2005

Graphics Processing Unit Computation of Neural Networks

by

Christopher Edward Davis

B.S., Computer Science, University of New Mexico, 2001

M.S., Computer Science, University of New Mexico, 2005

Abstract

This thesis outlines, discusses, and presents several artificial neural network architectures that are amenable to execution on a Graphical Processing Unit. Traits are identified that lead to good or poor performance of the artificial neural network simulation.

Contents

List of Figures **xiv**

List of Tables **xvi**

Glossary **xvii**

1 Introduction **1**

1.1 Overview 1

1.1.1 The Problem 2

2 Background **4**

2.0.2 History of the Artificial Neural Network 4

2.0.3 Anatomy of a Artificial Neuron 6

2.0.4 Similarity of Artificial and Biological Neurons 9

2.0.5 History of the GPU 11

2.0.6 What is a pipeline? 14

2.0.7 Vertex and Fragment processors 17
2.0.8 Moore Power: Programmable Graphics Hardware 19
2.0.9 So what? 21
2.1 Previous Work 22
2.1.1 General Purpose Graphical Processing Unit Computation . . . 22
2.1.2 How general purpose computation occurs on the GPU 23
2.1.3 Artificial Neural Network Graphical Processing Unit Simulation 24
2.1.4 These approaches are not enough 25
2.2 Approach 25
2.2.1 Virtual Machine 26
2.2.2 Assembly Language 28
2.2.3 ATLAS BLAS 29
2.2.4 BrookGPU 29
2.2.5 Stream Based Processing and Data Types in BrookGPU . . . 30
2.2.6 Algorithms Investigated 31
2.2.7 Hopfield Network 34
2.2.8 Adaptive Resonance Theory (ART) 36

3 Findings **38**

3.1 Results 38
3.1.1 Hardware Used 38

3.1.2 Software Used 39
3.1.3 Multi-Layer Perceptron Results 39
3.1.4 Little model Hopfield Network Results 40
3.1.5 Adaptive Resonance Theory Results 41
3.2 Discussion 43
3.2.1 Interpretations 43
3.2.2 What was right/wrong 44
3.2.3 Improvements 45
3.2.4 MLP Results Discussion 46
3.2.5 Hopfield Results Discussion 47
3.2.6 F-ART Results Discussion 47
3.2.7 What makes an ANN amenable to GPU execution? 48

4 Summary and Conclusion **50**

4.1 Summary 50
4.1.1 The Area 51
4.1.2 The Problem 51
4.1.3 Other's Solutions 51
4.1.4 This Solution 52
4.1.5 What was right/wrong? 52
4.1.6 The Study 52

4.2 Conclusion 53

5 Future Work **54**

Appendices **56**

A BrookGPU Multi-Layer Perceptron **57**

B ATLAS BLAS Multi-Layer Perceptron **64**

C BrookGPU Little model Hopfield Network **70**

D ATLAS BLAS Little model Hopfield Network **78**

E BrookGPU Fuzzy Adaptive Resonance Theory **83**

F CPU Fuzzy Adaptive Resonance Theory **92**

G Thomas Rolfes matrix product implementation **98**

References **102**

List of Figures

2.1 Biological Neuron 6

2.2 Artificial Neuron 7

2.3 Example of a artificial neural network 8

2.4 GeForceFX board 12

2.5 GeForceFX chip 13

2.6 ATI Radeon X850 board 14

2.7 ATI Radeon X850 chip 15

2.8 Diagram of the fixed pipeline architecture 16

2.9 Diagram of pipeline arithmetic 17

2.10 Diagram of the programmable pipeline architecture 18

2.11 Graph of GPU vs CPU performance increases 20

3.1 Comparison of two independent simulations of the MLP on the GPU 40

3.2 Comparison of MLP network simulated on GPU vs CPU using the number of multiplication and addition operations 41

3.3 Comparison of MLP network simulated on the ATI Radeon 800 GPU comparing OpenGL vs DirectX using the number of multiplication and addition operations 42

3.4 Comparison of Hopfield network simulated on GPU vs CPU 43

List of Tables

2.1 A small chart of some of the more commonly studied animals neuron counts 10

2.2 A performance chart of some of the more popular processors 19

3.1 A chart of Fuzzy ART network performance 42

Glossary

ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
API	Application Programming Interface
ART	Adaptive Resonance Theory
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
Cg ™	C for graphics, a shader language developed by NVIDIA Corporation
GPU	Graphical Processing Unit
HLSL ™	High Level Shading Language, a shader language developed by Microsoft Corporation
MLP	Multi-Layer Perceptron
$\vec{x}$	the vector $\vec{x}$ consisting of elements $(x_1, x_2, x_3, ..., x_N)$

$z_j = \sum_i^N W_{ji} \cdot x_i$ the matrix product of a two dimensional matrix W and a vector $\vec{x}$. W_{ji} measures the effect of component i of $\vec{x}$ on component j of $\vec{z}$

Chapter 1

Introduction

There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things.
Niccolo Machiavelli "The Prince" 1532

1.1 Overview

Artificial neural networks have held the attention of researchers for over 60 years because of their simplicity, robustness, and similarity to biological structures present in our own brains. Over the years, researchers have applied neural solutions to many fields; from medicine to weather forecasting, autonomous airplane control systems and simplistic robot controls.

Over the past 30 years, graphics cards have become useful and powerful processors found in most commercial off the shelf computers. These computers can range from

console game systems to a commodity desktop, and if applied correctly are currently capable of serious computation.

A GPU can be defined as a single chip processor used for processing two and three dimensional transform and lighting operations in the context of computer graphics. These processors are typically moderately parallel and contain several ALUs (Arithmetic Logic Units), several pixel pipelines (around 16), several vertex pipelines (around 6), and fast access to a relatively small amount of on card memory (32 to 512 MB). Modern 3-D graphics cards typically contain one or more of these GPUs.

In this thesis, simulation of artificial neural networks (ANN) on commodity graphics processor units (GPU) is explored, measured, and analyzed to provide some general observations about characteristics that make an ANN well or ill-suited to simulation on the GPU.

1.1.1 The Problem

Graphics processing units are being considered for many fields of computation due to their many attractive traits. One example field is ANNs. Artificial neural networks offer some attractive solutions to many real world problems. The field of artificial neural networks is an area of computation that has been by in large unexplored in comparison to other fields of GPU application. Because of these facts, the marriage of ANNs and the GPU would seem to be a worth-while area of research; to establish or invalidate particular architectures amenability to GPU simulation, to provide a fair and unbiased comparison of CPU and GPU execution of these architectures, and to establish some general traits that make ANNs more or less amenable to simulation on a GPU.

In this thesis, a brief history of GPUs and Artificial Neural Networks is presented.

Introductions to both GPUS and ANNs provide a basic overview of the information needed to understand the application of ANNs on GPUs. A survey of current GPU and CPU programming languages with ANN supporting features is presented and briefly discussed. Finally, a comparison between several ANNs implemented on both the CPU and the GPU are presented, allowing analysis and generalization about the applicability of ANNs on GPUs.

Chapter 2

Background

2.0.2 History of the Artificial Neural Network

In the early 1940s, researchers began investigating computational models that mimic the neural connections present in all animals. If creatures as simple as flat worms can perform complex tasks with seemingly simplistic and relatively small (in comparison to *Homo sapiens*) neuronal connections, why couldn't this design be mimicked *in silico*?

In 1943 Warren McCulloch and Walter Pitts published the seminal paper, "A Logical Calculus of Ideas Immanent in Nervous Activity"[13] in which they outline ten theorems and a propositional calculus explanation for the functionality of the biological neuron network. They also show that their neural network propositions are Turing complete, and as a result so are biological entities. They note from observation that neurons exhibit differential behavior when repeatedly fired, or when neighboring neurons are repeatedly fired[13, 28, 27]. This behavior would go without plausible explanation for another 6 years until Donald Hebb published his book entitled *The Organization of Behavior*[9]. In his book Hebb presented his theory

on learning involving the strengthening and weakening of neural weights in response to coordinated or uncoordinated firing (respectively)[9, 22]. These three researchers together established the neuron as the atomic unit of computation within the brain, and as a result solidified the use of artificial neurons as a valid approach.

In the mid 1950s the first simulations of artificial neural networks began to appear. Theoretical work still had a great lead on the actual implementation of networks. In 1969 Minsky and Papert published their book *Perceptrons*[15] which seemed to place significant limitations on the functionality of the perceptron. They even went as far in their condemnation as to say that there was no reason to believe that the limitations found in the single layer perceptron would be overcome by a multi-layer solution. This book dealt a great blow to the entire field of connectionist networks and is often cited as one of the instigators of the connectionist counterrevolution. In the 1980s the credit assignment problem was solved, and Minsky and Papert's prognostication about multi-layer perceptrons was proven false[8]. Multi-layer perceptrons could indeed solve more complex problems than were possible to solve with the single layer perceptron, and credit or blame could be appropriately spread through the network with some simple calculus.

In spite of the blow dealt by Minksy and Papert, theoretical work continued and resulted in several interesting models including self-organizing maps and adaptive resonance theory. In 1982 John Hopfield presented his ideas on using energy functions from physics to compute recurrent networks. Initially known as "auto-associative networks" they are now commonly known as "Hopfield Networks"[8, 24].

While this is only a very brief history of artificial neural networks, it should serve as a good synopsis of the high points in history that have motivated this research. Many important innovations have occurred between 1982 and the present, however few (if any) are of importance to the direction of this research.

2.0.3 Anatomy of a Artificial Neuron

The artificial neuron is based in large on the structure of a biological neuron (Figures 2.1 & 2.2). It consists of connections - analogous to the axon and dendrites, an activation potential - analogous to the soma, and a threshold function - analogous to the axon hillock. Like their biological counterparts the activation threshold is most commonly some non-linear function (tanh, logistic, atan), although it can be any function of choice.

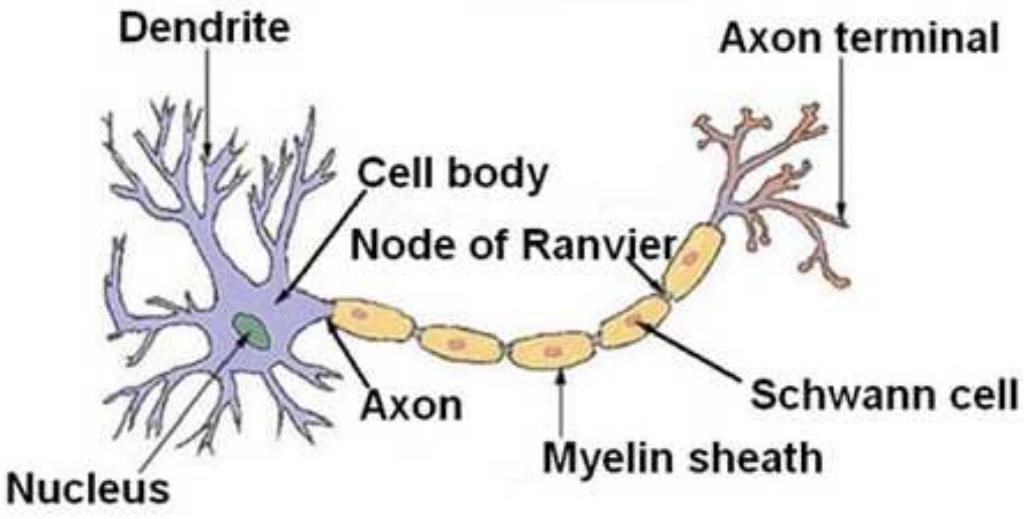

Figure 2.1: Biological neuron, image courtesy of U.S. National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program

Artificial neurons, like their biological counterparts, are most often composed into networks of neurons (Figure 2.3). Each neuron will have connection weights that indicate the propensity for that particular neuron to be excited or suppressed by the neurons that are sending signals to it. It is through the intelligent composition of these neuron networks that artificial neural networks are able to perform their tasks.

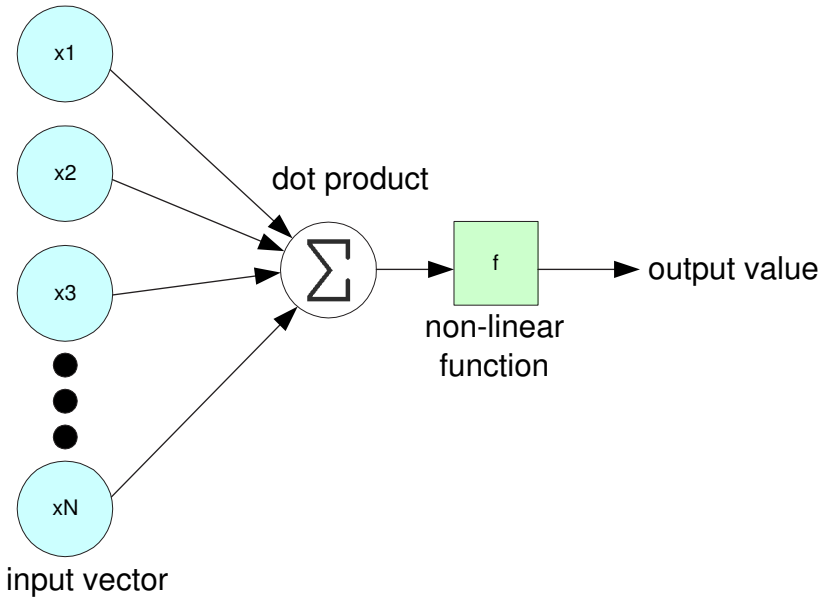

Figure 2.2: Artificial neuron

Some of the benefits of neural networks include: nonlinearity, input-output mapping, adaptivity, evidential response, contextual information, fault tolerance, uniformity of analysis and design, and the neurobiological analogy[8]. These benefits are discussed briefly below:

- The non-linear (or linear) nature allows the networks to process signals that are produced by some underlying non-linear process, such as a speech signal[8].
- The input-output mapping allows networks to make a literal mapping from given inputs to a desired output through either supervised or unsupervised learning. This mapping is obviously needed for classification problems, since to classify any input we must map the input to a given (or discovered) class[8].
- Adaptivity means that the network is able to adapt to the environment it exists within. This adaptivity is quite useful when the network is to be employed in

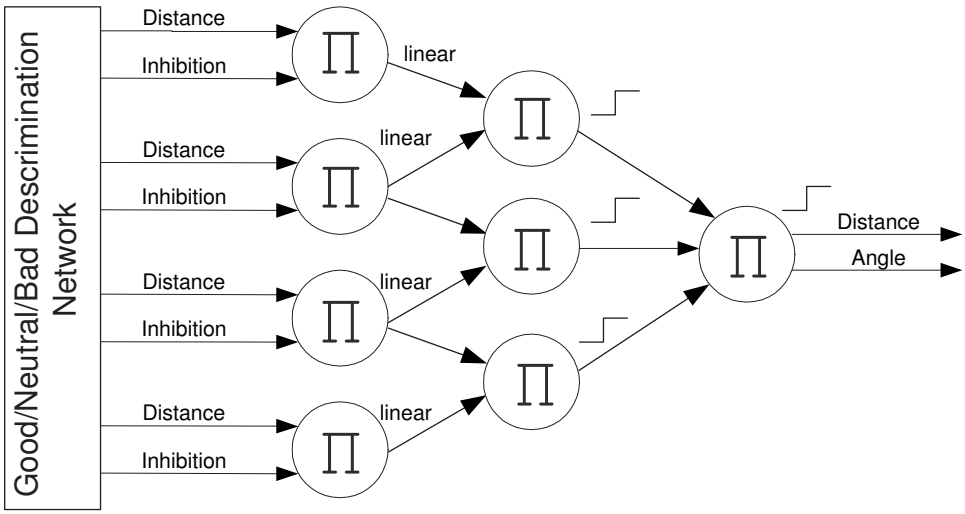

Figure 2.3: Example of a artificial neural network designed as part of a simulator robot control system. This network utilizes "Pi" (multiplication) neurons instead of the "Sigma" (summation) neurons used in this thesis research.

an environment in which all possible inputs can not be specified at the design stage[8].

- Evidential response allows the network to respond not only with a classification for a given input, but a confidence measure. This style of response can be useful for dealing with ambiguous inputs which might otherwise be misclassified, or result in more certainty being placed in an uncertain response[8].

- Contextual information refers to the network's natural encoding of information at a global scale. Because each neuron is potentially affected by the activity of every other neuron in the network, contextual information is in some sense inherently included[8].

- Because networks are composed of distributed neurons, each with its own piece of the body of knowledge, the performance can be called fault tolerant. If one

neuron or small group of neurons is for some reason disabled, the performance of the entire network will degrade gracefully[8].

- Uniformity of analysis and design refers to the neuron being the basic unit of computation present in most neural networks. This uniformity allows research and methods to be shared among various architectures, advancing larger portions of the field at once[8].

- The neurobiological analogy is convenient because it allows researchers in artificial and biological neural networks to look towards each other for explanation and inspiration[8].

2.0.4 Similarity of Artificial and Biological Neurons

The structural components of the biological neuron have simplistic counterparts in the artificial models, but the similarities would appear to end there. The human brain has approximately 2×10^{12} neurons, whereas the largest artificial networks are typically many orders of magnitude smaller. Biological networks engage in several modes of asynchronous parallel computation simultaneously through both axon firing locally and more global chemical reactions which alter the overall function of the brain.

While some researchers such as Rodney Brooks have offered up comparisons of the computational power of modern computers and their biological counterparts (Table 2.1), these claims are by and large misleading and probably mistaken. It has been argued that the brain has about 2×10^{12} neurons and the relaxation time of a neuron (time from firing to the ability to fire again) is about 10 milliseconds, the brain has a clock speed of approximately 100 Mhz. Using these figures results in a estimate of 2×10^{14} logical operations per second[25]. As a comparison, the Pentium4 running SSE2 instructions can churn out about 1.6×10^{10} operations per second. Even though

the Pentium4 performance is only slightly more than a factor of 1000 times slower, it would seem to be a great mistake to assume that we are even within a factor of 1000; not because human brains are some mythical supreme computational medium, but rather because the brain has had millions of years of evolutionary force behind its highly parallel and distributed architecture. The fact that we do not understand much of the brain's function at a low level makes these comparisons in computational power unfounded.

Neuron Count By Species	
Rotifers and Nematodes	less than 300 neurons
C. elegans	959 somatic neurons and 301 neurons
ANNs examined in this research	1024 to 6148 neurons
Drosophola	350,000 neurons
Small mammals	30 million neurons
Common Octopus	30 million neurons
Human	85 billion neurons
Whale and Elephant	over 200 billion neurons

Table 2.1: A small chart of some of the more commonly studied animals neuron counts

"Soma" is Greek for "Body". There are two main definitions for "soma" in the context of neuroscience. The first refers to the main body of the neuron which contains among other components the nucleus. The second definition refers to the function of the neuron, namely it refers to neurons that control the functions (mainly motor and sensory) of the body of the organism. In table 2.1 "soma" refers to this second definition, implying that the somatic neurons are mainly (but almost certainly not exclusively) involved in body control.

2.0.5 History of the GPU

The modern GPU is a descendant of the Geometry Engine, a very large scale integration (VLSI) based graphic card solution, and post-script raster processors in laser printers of the 1980s. By the early 1990s 2-D accelerated raster controllers had been successfully designed, marketed, and integrated into most people's graphics cards. These cards were capable of simplistic operations, constrained to 2-dimensional desktop type operations such as BitBlt (bit-blitting). High performance computing companies like Silicon Graphics (now SGI), HP, and Sun all had 3-D acceleration support for OpenGL by 1996; but these solutions were prohibitively expensive for the mass market. These cards generated a desire for accelerated 3D performance among the computer gaming community. In 1994 3dFX Interactive was incorporated and set out to design a graphics card that the masses could afford. In 1997, following a drastic drop in EDO RAM prices, 3dFX released their "Voodoo Graphics" chipset (later to be known as Voodoo 1). The Voodoo chipset was repackaged by industry leaders onto daughter cards that subsumed the traditional 2D graphics cards, performing acceleration only when running 3D applications. Due to the success of this chipset 3dFX released the Voodoo 2 a year later, to similar fanfare. Seeing the success of 3dFX, other industry leaders started formulating solutions to compete in this growing market. By late 2000 3dFX underwent one of the most renowned demises in the computer graphics industry, due to litigation delaying the release of lines, a shift from reselling chips to producing the full graphics card, and the release of the NVIDIA GeForce[23]. The spark of interest created by their early successes went on to fuel a very competitive market[23]. The new NVIDIA GeForce processor shifted much of the work from the CPU to the GPU and was the first GPU capable of useful and speedy GPU computation.

It is necessary to distinguish between the GPU (Figures 2.5 & 2.7) and the graphics card (Figures 2.4 & 2.6) that the GPU is integrated to. The GPU is simply

the processor utilized in many of the graphical operations performed by the computer, whereas the graphics card implements the full suite of graphical operations through the integration and utilization of many components including the GPU, memory, and often a suite of other small co-processors dedicated to video encoding and decoding, two dimensional operations, and other graphics related operations.

Figure 2.4: GeForceFX board (graphics card)

The early graphics architectures offered minimal computational ability. The processors were under-engineered in comparison to CPUs of the time, computations were limited to 8 bit precision, and more importantly the graphics pipeline was very rigid (Figure 2.8). The graphics pipeline will be discussed in more detail in section 2.0.6 but in brief in can be considered a streamlined series of processors capable of very efficiently performing specific special purpose operations on the vertex and pixel data, much like an assembly line in an auto manufacturing factory.

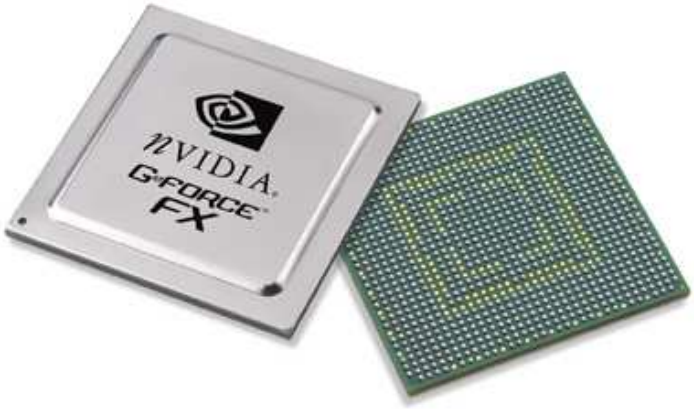

Figure 2.5: GeForceFX chip (GPU)

As the market has strengthened these issues have been addressed; GPU engineering has produced processors capable of well outperforming CPUs on linear algebra (Table 2.2) (in spite of Intel and AMD's attempts to integrate streaming instruction set math (see section 2.2.2 for further discussion)), current GPUs now support 16-bit floating point operations directly and 32 (NVIDIA) or 24 (ATI) bit floating point operations via additional programming, and NVIDIA, Microsoft, and eventually the architecture review board (ARB) (the industry organization responsible for controlling the OpenGL standard) have established standards that allow programmers much more access to the underlying hardware. As a result it allows much more flexibility in the graphics pipeline (Figure 2.10). It was this added functionality in the graphics pipeline first through the use of "extensions" then eventually through "shaders" coupled with increased performance and increased precision that gave GPUs their final push towards being a real option for computing.

Shaders are simple program kernels that are meant to allow the programmer to specify how the vertex and pixel handling are performed, but are flexible enough to be used in some general computation applications. Shaders can be written in high-

Figure 2.6: ATI Radeon X850 board (graphics card)

level programming languages and mapped onto underlying hardware and virtual machine by vendor specific compilers. The high-level language implementation and compilation approach allows developers to write code once, compile it to all future hardware revisions, and permits them to take advantage of new hardware innovations as they arrive without re-writing code.

2.0.6 What is a pipeline?

A pipeline (Figures 2.8 & 2.10) is an architecture that has risen out of the advances made in VLSI technology in order to facilitate higher throughput of data. Consider the operation $z = \alpha w + y$. Examining the work performed, there is 1 multiply performed and 1 addition. Now consider the execution of this formula x times on two different platforms, a serial processor and a pipelined processor. On the serial processor the multiplication is performed, then the sum. After this first result is returned the same execution occurs again; the multiplication first, then the sum. If

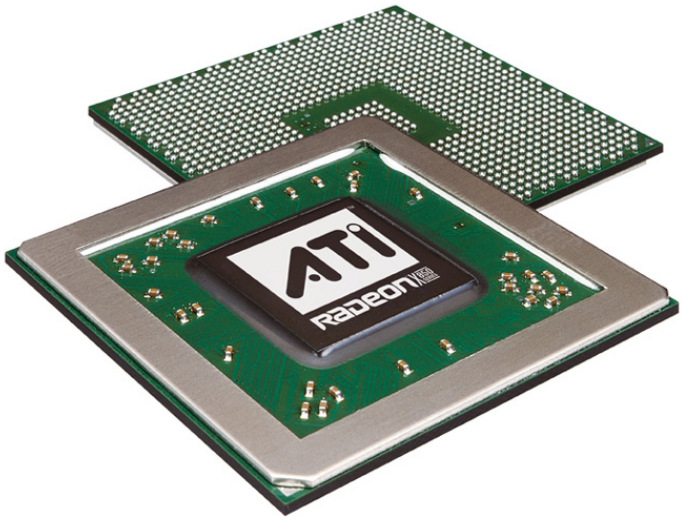

Figure 2.7: ATI Radeon X850 chip (GPU)

each operation takes 1 clock step, it is simple to see that the total time for execution of this formula is $2x$ clock steps. Now consider the pipelined processor. On the pipelined processor this formula can be broken into two steps that run independently; the multiply and the sum (see figure 2.9 for an example of this operation). This allows the pipelined processor to begin the multiplication of the next formula immediately after finishing the first multiplication and before the sum is completed. Now examine the result this has on the running time of the formula: the first result is returned after two time steps, but every successive time step has a result generated (total running time: $1x + 1$)! This is a great speedup as long as the operations performed are able to be decomposed into these independent computational steps.

To think of the pipelined architecture in a more abstract metaphor, think about an automotive assembly line. On the assembly line (invented by Henry Ford in 1914) a car moves from one station to the next, having a set of parts added or modified at each station. Each car has other cars in front of and behind it, so that there

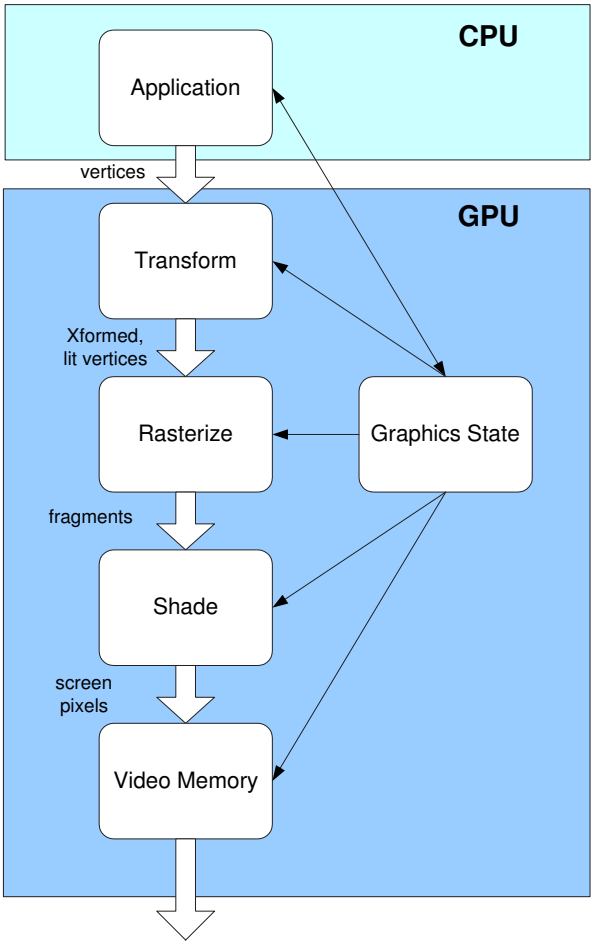

Figure 2.8: Diagram of the fixed pipeline architecture

is immediately another car for the employee at each station to work on as soon as they are done with the car currently at their station. In the same light, a pipelined processor (when correctly used) has data ready to be processed by each processing unit (like an employee at a station) as soon as it is done with its current operation.

If the pipeline architecture is also parallelized (as it is in GPUs), operations like $\overrightarrow{z} = \alpha\overrightarrow{w} + \overrightarrow{y}$ can be greatly sped up. For example consider a pipelined processor with 4 parallel pipelines. If the vector from the above formula is a vector with height

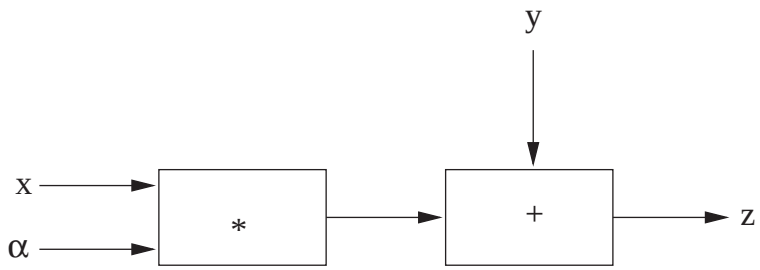

Figure 2.9: Diagram of pipeline arithmetic

4, this formula can now be run repeatedly in total running time $1x + 1$! Compare these results to a serial processor $(4 \times 2x)$ and the savings can be tremendous.

2.0.7 Vertex and Fragment processors

The two components of the graphics pipeline that are most heavily utilized during GPU computation are the vertex and fragment processors.

The vertex processor is the first transformation processor that a geometric primitive sent from the application encounters in the graphics processor. Basically the vertex processor takes in vertices from the host application then performs transform and lighting (T&L) operations on the vertex before passing it along to the rasterizer. At the moment very limited texture access operations are permitted at the vertex processing stage.

Once a vertex has been transformed (scaled, rotated, translated, converted into different coordinate systems, etc), the vertex is grouped into a primitive object and the primitive is rasterized. While rasterization is an important step, at the moment there is no programmable function in the raterizer and it is used very lightly in most GPU based applications.

Once the vertices have been rasterized into discrete geometric shapes in memory,

the fragment processor is invoked on the "fragments". A fragment can be thought of as a "potential pixel". The term potential is used because there is no guarantee that the pixel will actually finally be displayed. The fragment processor has much more flexibility in its operations than the vertex processor has. Particularly the fragment processor is the location where texture access, texture manipulation, and blending typically occurs.

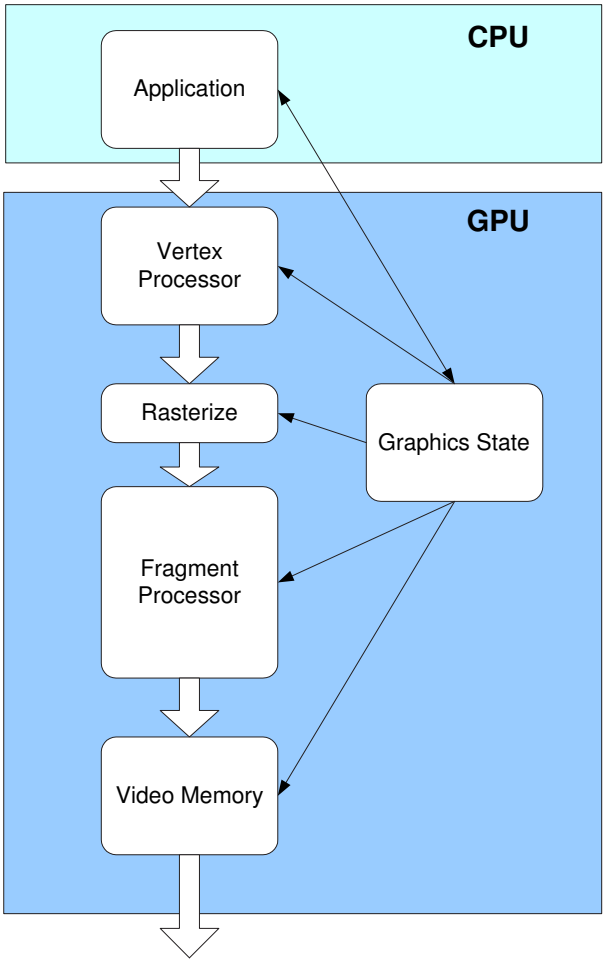

Figure 2.10: Diagram of the programmable pipeline architecture

Performance of popular processors

Processor	Peak Performance	Memory Bandwidth
3 GHz Intel Pentium 4 CPU	rated 12 GFlops	6 GB/sec
3 GHz Intel P4 CPU dual core	rated 24.6 GFlops	6 GB/sec
400 MHz NVidia 6800 GPU	measured 53.4 GFlops	36 GB/sec
430 MHz NVidia 7800 GPU	measured 165 GFlops	38.4 GB/sec
500 MHz ATI Radeon X800 GPU	measured 63.7 GFlops	32GB/sec

Table 2.2: A performance chart of some of the more popular processors

2.0.8 Moore Power: Programmable Graphics Hardware

A common usage of Moore's law states that transistor count (and hence computational power) will double every 18 months (Gordon E. Moore actually predicted a doubling every 2 years, but common use has now quoted his prediction as 18 months). Modern GPUs have been able to not only double in performance every 18 months, but have actually been increasing 5 fold every 18 months.

Pipelining allows the processor to handle data without extensive caching. CPUs access memory in a much less predictable pattern than the GPU, so they need caches. Because the GPU is able to reduce the number of on-chip caches, they are able to use that chip space to put in additional computational units or communication channels.

Not only is the processor pipelined and parallel, but the memory is also pipelined and parallel. This pipelining allows for very efficient access to large amounts of memory and helps to address the main issue affecting modern memory access, latency.

Within the pipeline processor there are two main classes of processor: vertex processors and fragment processors. Vertex processors are Multiple Instruction Multiple Data (MIMD) processors capable of performing all of the transformation operations required for geometry manipulation. At the moment the vertex processors are utilized very lightly for general purpose GPU programming. The fragment processors

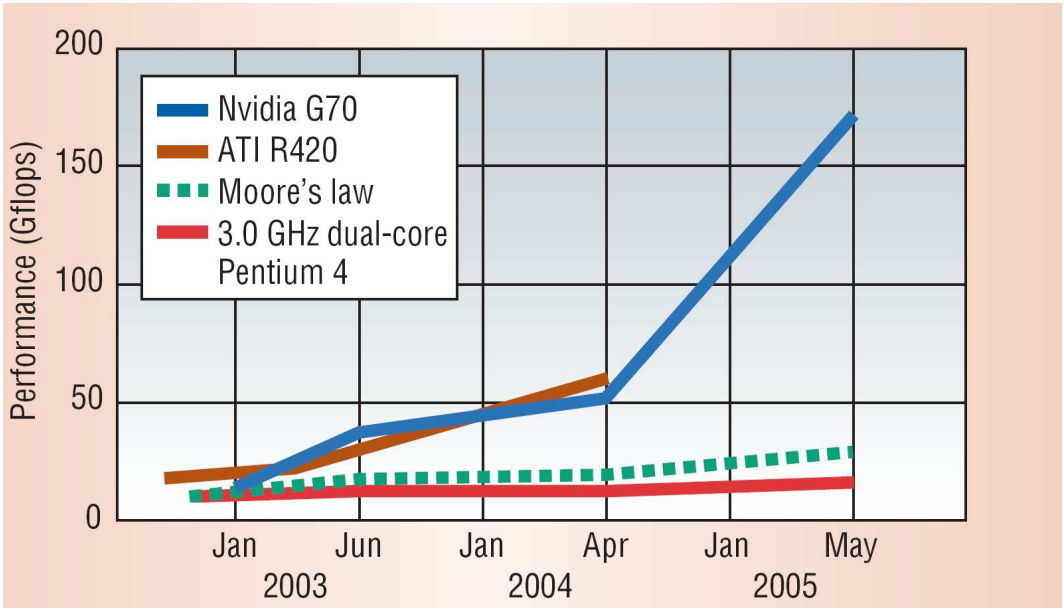

Figure 2.11: Graph of GPU vs CPU performance increases. This graph shows the maximum number of gigaflops attainable by two leading GPUs and the Pentium 4 compared to Moore's law. Courtesy of NVIDIA (see section 2.0.8 for further discussion)

are Single Instruction Multiple Data (SIMD) processors. The main computational task of the fragment processor is to assign color and other visual effects to each fragment or eventual pixel. By coupling the computational power of these two classes of processor with the large memory bandwidth substantial computational power is available.

It is the combination of these special purpose processors, pipeline architectures for processing and memory, and inherent parallelism that make these processors able to continually beat Moore's law and create an interesting niche for research.

2.0.9 So what?

You might be asking yourself about now, "So what?". Why should we care about GPUs coupled with Artificial Neural Networks? The answer has several parts to it.

1. The GPU is another processor in most systems today, and as such can be used as a co-processor to offload work from the CPU. The co-processor approach will allow you to better utilize the overall system as a whole rather than relying entirely on your CPU.

2. 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.

3. The GPU is composed of many parallel subcomponents which are easily replicated and integrated into the next generation of processor. As a result the generation to generation performance curve for the GPU is quite attractive compared to a CPU (Figure 2.11).

4. Artificial neural networks are capable of solving many interesting problems, but have been under utilized to date partially because of the computational cost.

5. Graphics programming languages are becoming more powerful and flexible. Programming environments like Brook are able to capitalize on the underlying "high level" shading languages like Cg, HLSL or GLSL to allow developers to remove themselves as far as possible from architecture specific assembly or other low-level constructs.

6. The GPU and CPU have fundamentally different approaches to architecture design and utilization. CPU development has been driven by the business and home computing industry, and is best suited to run diverse user applications. GPU development has been driven by games and graphics application, and is best suited to run specific graphical application operations. ANNs happen to have some operations and modes of execution that seem to share many of the operations and characteristics of graphics applications.

2.1 Previous Work

This section outlines the related previous research in this field. Although there has been little directly related research, the research presented below has laid much of the groundwork for this and future research into GPU ANN simulation.

2.1.1 General Purpose Graphical Processing Unit Computation

The research community has started several projects to utilize the GPU for general purpose computation. Individual research institutions and researchers have taken this initiative and been conducting interesting and useful investigation into the uses of the GPU. One of the most popular sites is GPGPU.org. This site is essentially a paper and application BLOG. Numerous people have published papers about potential applications of the GPU to real world computing problems, however to date there have only been two main papers (one has been reprinted with minor modifications) on ANN simulation on the GPU. GPGPU.org is host to the forums for BrookGPU, as well as several other community forums devoted to general computing on the GPU. It is through these forums and paper collections that ideas are shared

and extended, including the basis of this research.

Research topics that have been investigated for GPU implementation include: advanced rendering, global illumination, image-based modeling and rendering, audio and signal processing, computational geometry, GIS, surfaces and modeling, databases, sort and search, high-level languages, image and volume processing, computer vision, medicine and biology, scientific computing, data compression, data structures, dynamics simulation, numerical algorithms, and stream processing.

2.1.2 How general purpose computation occurs on the GPU

General purpose computation on the GPU necessitates using the built-in architecture API to accomplish operations. The graphics card (and more specifically the GPU) is designed to operate efficiently on graphics primitives such as polygons and textures. As such, computation occurs by mapping the problem at hand into the graphics primitives, performing graphics operations on the primitives, then re-mapping the resulting image into solution values. In most general purpose GPU programming at the moment computation starts with the generation of a quadrilateral by the vertex shader that fills the raster buffer (or other large buffer). The data for the problem is then loaded into the graphics card by encoding values from the user application into floats represented in one or more textures. The pixel (or fragment - depending on architecture) shader then does the computation using graphical convolution and blending operations. Finally, the resulting image is translated back into values in the user's application and results are obtained.

Computation is constrained by operations that are available on the GPU and by the size of the buffers available. For instance some GPUs implement operations not present on others, and some graphics cards (NVIDIA) are constrained to buffers of 16 million pixels while others (ATI) are constrained to buffers of 4 million pixels.

Care must be exercised when using the GPU to compute traditional CPU algorithms as the GPU has some functionality implemented differently than the CPU. For example, the GPU does not perform conditional branching efficiently (due to the pipelined architecture), and as such branching algorithms should be implemented traditionally through the CPU rather than the GPU. Other caveats include operations such as memory access. In most CPU based languages it makes no sense to attempt to access a fractional array value (for example $my\_array[1.5]$) however in graphics fractional memory access is a very standard way of implementing sampling or interpolating effects, and as such, a GPU program has no problem returning $my\_array[1.5]$ which may or may not be what was actually intended.

Most researchers perform their implementation of the computation in one of the approaches outlined in section 2.2.

2.1.3 Artificial Neural Network Graphical Processing Unit Simulation

Thomas Rolfes has published several versions of an article[16] on GPU simulation of the multi-layer perceptron (MLP). In these articles, he outlines an approach to the implementation of a MLP on the GPU through DirectX. In short, he uses matrix-matrix products to compute the activation levels for the neurons, then applies his non-linearity threshold function. Each layer in the network is a single matrix represented explicitly as a texture in DirectX, and inputs are aggregated into a matrix of compatible dimensionality. The matrix product is based on an approach presented in [14, 12, 20]. See Appendix G

Kyoung-Su Oh and Keechul Jung have also implemented a multi-layer perceptron for the GPU to perform text detection[10]. Again, they use a method for matrix products outlined in [14, 12, 20].

2.1.4 These approaches are not enough

While these papers are quite interesting, and both show promising results, both only examine one ANN architecture (the MLP) and by in large fail to identify the traits of the GPU that make certain ANN architectures well suited to GPU simulation. Perhaps more importantly, both papers constrain themselves to a batch processing mode of execution. While batch processing is a fine constraint in some applications and architectures, there are many classes of ANN architectures that can not be run in batch mode. A prime example is adaptive resonance theory (ART); because the templates may change between two consecutive input vector presentations, you can not present input in batch mode.

Lastly, while performance increases over the CPU are claimed, only Rolfes presents any source code, but even he lacks a published comparable CPU optimized version of the code. Correspondence with both sets of researchers has been attempted, but only Rolfes has responded. He has been kind enough to supply a copy of his revised work that was previously unavailable in the United States.

2.2 Approach

In light of the shortcomings of previous research in this field, the approach taken in this research is to explore multiple implementations of several ANN architectures, both on the GPU and on the CPU. To provide the most unbiased comparison between the relative performance of a GPU implementation and a CPU implementation the CPU code is implemented using ATLAS BLAS (see section 2.2.3) where applicable. While it is acknowledged that programming in a high level language does not provide the best performance possible, it is a convenient approach to exploring multiple implementations in a reasonable time period. Because of these constraints and ben-

efits, in this research both the GPU and CPU are programmed in the most efficient method available through C.

2.2.1 Virtual Machine

Hardware is often presented to the developer as a virtual machine at some level. In the following sections the most important virtual machine architectures used in this research are presented. The virtual machines are used by the developer through an API (application programming interface). The API can change as new techniques are developed or desired without requiring a change of the underlying hardware.

Cg

Cg is a shader language (API) developed by NVIDIA Corporation to allow developers to write code once and re-compile it for future architectures as they are released. Cg stands for "C for graphics", and was the first major shader language to be developed and released for real time graphics. It was inspired by the RenderMan shaders that were developed for non-interactive off-line rendering projects such as: Young Sherlock Holmes (1985), The Abyss (1989), Terminator II (1991), Jurassic Park (1993), Toy Story (1995), A Bugs Life (1998) (just to name a few of the highlights). The tremendous success of RenderMan shaders with artists, efficient implementation on hardware, and stunning visual result made this approach to graphics programming quite attractive. In short a shading language is typically used to determine the final surface properties of a scene or an object within a scene. These properties are manipulated by altering the properties of the vertices of the objects (location, color, etc), or the resulting pixels (color, alpha, location, etc).

DirectX & HLSL

DirectX™is a API developed by Microsoft® to facilitate a unified interface with the multimedia hardware present in modern PCs running Windows and gaming console systems. Of particular interest to this research is Direct3D™and HLSL™. Direct3D™is Microsoft's graphics portion of DirectX. It is one of the main competitors for OpenGL and provides many of the same style of 3D scene manipulation functionality. HLSL is Microsoft's version of Cg, a high level shading language for use with Direct3D. Both Direct3D and HLSL are ported to the PC and Xbox architectures, and thus present developers with a convenient and fairly standard virtual machine to write code for.

GLSL

GLSL, also sometimes called the ARB shading language, is the OpenGL shading language (API) that is currently being released. Over the past few years the architecture review board (ARB) has added various functions to the OpenGL extensions that approached a full-fledged shading language. In the last year the ARB finalized the standard for the new shading language extension, and released it. At the time of this research there were no reliable GLSL compilers, so it was not an option for research and experimentation. In spite of the lack of compiler, GLSL is worth noting here because a series of compilers and run-time environments have recently been released, it will be a viable virtual machine, and will have benefits and drawbacks when compared to the two currently available options (Cg and HLSL).

2.2.2 Assembly Language

While the previous sections presented several virtual machine views of the processors, it is also quite possible to view the hardware without any virtual machine abstraction. While assembly language implementation allows you direct, generally unhampered access to the hardware, it also ties you explicitly to a particular line of hardware. Early GPU programming was done exclusively in assembly language but proved to be an often arduous task for developers since every architecture needed its own assembly implementation of a shader for the shader to function optimally (or close to optimally). Assembly coding of the CPU and GPU are still popular routes to pursue when performance is of the upmost importance.

SSE and SSE2

For the CPU there are several ways to view its architecture and functionality. One of the more interesting views of the CPU is to view it as a streaming processor. SSE stands for ***S***treaming ***S***IMD ***E***xtensions and is Intel's solution to a streaming processor instruction set. SIMD stands for ***S***ingle ***I***nstruction, ***M***ultiple ***D***ata and is an approach to parallel processing that has become much more popular over the past 35 years. SIMD is a way of running small kernels of instructions over a set of data, with each data point receiving the same set of operations. In this method of processing, the pipelines of the modern processor are able to be better streamlined, the branches better predicted, and the overall performance improved. SSE and SSE2 are the successors to Intel's MMX (***M***ulti***m***edia E***x***tensions) and the main competitors to AMD's 3Dnow, and PowerPC's AltiVec instruction sets. These streaming instruction sets are mentioned here because the various high-performance linear algebra suites (like ATLAS BLAS) utilize these instructions heavily in order to achieve better performance. It is also worth noting that these are the direct analog to the

streaming kernels found in all shaders implemented on the GPU.

2.2.3 ATLAS BLAS

ATLAS stands for ***A***utomatically ***T***uned ***L***inear ***A***lgebra ***S***oftware and is a linear algebra package developed under the BSD license through Source-Forge (open source). As the name suggests it is a package that automatically tunes itself to the hardware that it is being run on. It attempts to perform tasks on the CPU in the most efficient method that it is aware of by testing parameters such as cache size, instruction sets supported, etc. One of the features that ATLAS provides is a BLAS implementation. BLAS stands for ***B***asic ***L***inear ***A***lgebra ***S***ubprograms. ATLAS BLAS provides a suite of functions callable from Fortran or C/C++ that allow for efficient execution of operations like vector-vector products, vector-matrix products, and matrix-matrix products. In this research the BLAS implementation utilizes the SSE2 instruction set (see section 2.2.2) in order to provide developers with an even more abstract view of the underlying hardware. Through this package complex operations are made available to developers with the simple call of a function.

2.2.4 BrookGPU

BrookGPU was developed at the Stanford University Graphics Lab as a GPU implementation of their streaming research language Brook. It is an extension to standard ANSI C and allows a simplified streaming interface to the programmable GPU. One of the main benefits to this approach to programming the GPU is that, unlike Cg, HLSL, GLSL, or assembly, this approach is highly portable, and far removed from the graphical constructs present in these languages. BrookGPU includes a compiler and runtime implementation, and utilizes the industry standard shader compilers to perform the low-level compilation to GPU assembly. Essentially BrookGPU maps

kernels and streams into one of the target shader languages (Cg or HLSL) for compilation. It is possible to test applications implemented in BrookGPU on Linux OpenGL, Windows OpenGL, and Windows DirectX because of the portability of BrookGPU code. This portability is quite convenient for an exploratory research project because it allows comparison of the various merits and shortcomings of each architecture and multiple graphic cards without code revision.

2.2.5 Stream Based Processing and Data Types in BrookGPU

Streams are a data type that lends itself to parallel processing. Streams are made up of elements, much like a standard array, however there are restrictions placed on how and when you can access the stream elements. Some of these restrictions include[5]:

> "indexing to gain access to stream elements is not allowed outside of kernels, no static initializers are allowed, streams must be local (stack) variables, streams can only be read and written inside kernel functions or through special operators that fill streams from regular pointers and vice versa."

In BrookGPU streams are converted into textures and loaded onto the graphics card's texture memory.

Streams are operated on by what are called "kernels". A kernel is a special function that is compiled into shaders and loaded onto the GPU. Each kernel invoked will operate on every stream element, but not necessarily in order. Kernels can not operate on global memory or static variables, but can be passed streams and scalars. Every kernel also has the restriction that it must have an output stream passed to it that matches the dimensions of an input stream.

Reduction kernels are a special type of kernel that can reduce the dimensionality of the data. In other words a reduction kernel does not need to meet the above restrictions of having a output stream that matches the dimensionality of an input stream, it can instead have an output stream that is smaller than the input streams. This kind of kernel is particularly useful for performing operations such as a summation of a vector into a scalar (used in operations like dot-products). Reduction kernels do have the restriction that they can only perform operations that are associative and commutative (like sum, min/max, OR, AND, XOR, etc).

A unique feature of stream based processing in BrookGPU is that stream shape determines the operation. Stream shape determining operation can be a bit difficult to understand at first (coming from a C/C++ background), but it allows for extremely general purpose kernels. A single kernel can operate on a multitude of different dimensional inputs, and depending on how the dimensions match-up to each other different operations may be performed. This generality allows programmers to operate on every element of a stream, or some sampled sub-set of the streams.

2.2.6 Algorithms Investigated

The algorithms investigated in this research are drawn from the popular ANN architectures. Through the implementation of three different algorithms a more complete characterization of the performance of the GPU as it pertains to ANN simulation can be developed.

Perceptron

The perceptron is one of the simplest neural models, and is typically the first model most people implement when studying neural architectures. As such, it would seem to be an obvious model to investigate for execution on the GPU.

The perceptron was originally developed by Rosenblatt in 1958[18]. Two years after its inception Rosenblatt published a proof of convergence[19], an important step towards proving the efficacy of the perceptron.

The perceptron is based on a series of dot products between an input vector and the connection weight vector of each neuron (Equation 2.1). The resultant scalar is then passed through a non-linear function (Equation 2.2) to obtain the actual value. The connection weight vectors can be supplied pre-set if you know how to solve the problem, or you can learn the weights through error correction methods[11, 8].

$$z = \sum_i^N w_i \cdot x_i$$
w_i is the neural weight for the i^{th} element of the input vector x (2.1)

$$y = \sigma(z)$$
y is the output value of the neuron and σ is the non-linear function (2.2)

The more interesting version of the perceptron is the multi-layer perceptron. In this model, perceptron neurons are connected in a layer to layer connection model. This means that the outputs of the first layer are fed as the input vector to the second layer, the outputs of the second layer to the inputs of the third and so on. The error correction method works slightly differently in this model than in the single layer perceptron. Because there are hidden nodes, you do not immediately know the weight correction that must occur for a given neuron. The credit assignment problem is easily solved by applying the corrections to each neuron based on a formula derived from the non-linear function used (Equations 2.3, 2.4, & 2.5).

$$\Delta w_{ki} = -c(d_i - y_i) * y_i(1 - y_i)x_k,$$
for output nodes (2.3)

$$\Delta w_{ki} = -c * y_i(1 - y_i) \sum_{i}^{N} -delta_j * w_{ij}, \tag{2.4}$$

for hidden nodes where j is the index of the nodes in the next layer

$$delta_j = (d_i - y_i) * (y_i(1 - y_i) \tag{2.5}$$

delta for Eq. 2.4

Because you must differentiate the non-linear function, it is important that the function be continuous - therefore a McCulloch-Pitts clamped neuron can not be used in this model. The multi-layer approach is more interesting not only because it is capable of solving more interesting problems, but because it is also more amenable to GPU execution.

The multi-layer perceptron is more amenable to GPU execution because the operations per byte of data transmitted to the graphics card increases. Because the bus to the graphics card is the slowest step in any computation, it is important that the operations per byte sent to the card or received from the card be as large as possible.

Because the GPU is better optimized for matrix-based math, another simplification and optimization can be made to improve the performance of the multi-layer perceptron. Instead of performing a series of dot products (one for each neuron) we can create a matrix of all the connection weights for a particular layer, and perform a single vector matrix multiply (Equation 2.6) followed by a non-linear activation function (Equation 2.7).

$$z_j = \sum_i^N W_{ji} \cdot x_i \tag{2.6}$$

$W_j i$ is the neural weight for the i^{th} element of the input vector x

$$y_j = \sigma(z_j) \tag{2.7}$$

y is the output value of the neuron and σ is the non-linear function

If the perceptron is run in batch mode, we can further optimize the execution by performing matrix-matrix products followed by a non-linear activation function[16].

The result from the first layer is then propagated to the next layer in the same fashion as the first. If a single input is being processed at a time, we perform another vector-matrix product, or in batch mode, another matrix-matrix product.

As with most neural models we can choose to turn learning on or off by deciding to update (Equations 2.3 & 2.4) or not update the connection weights.

2.2.7 Hopfield Network

The Hopfield network is based on a vector-matrix product with a non-linear function applied to the result. It is a single layer solution that is derived from energy minimization approaches in physics. It can be broken down into four basic stages: learning, initialization, iteration until convergence, and outputting.

In the learning stage, the fundamental vectors are presented to the system and the outer product rule (Hebb's postulate of learning) is used to compute the connection weight matrix. The fundamental vectors encode the data that you want the network to memorize and later recall in the form of ± 1. Once the weights have been computed they are kept fixed through the rest of the network's usage.

$$W_{ji} = \begin{cases} \frac{1}{N} \sum_{\mu=1}^{M} \xi_{\mu j} \cdot \xi_{\mu i} & \text{if } j \neq i; \\ 0 & \text{if } j = i. \end{cases} \tag{2.8}$$

W_{ji} is the weight matrix, N is the number of fundamental vectors,
ξ_μ is the fundamental vector

In the initialization stage, an unclassified vector is presented to the network and

the state vector is initialized to the value of the vector.

$$\begin{aligned} &x_j(0) = \ \xi_{i\text{probe}} \quad \text{for } j = 1, \ldots, N. \\ &x_j(0) \text{ is the input layer, } \xi_{i\text{probe}} \text{ is the probe vector} \end{aligned} \tag{2.9}$$

In the iteration until convergence step, the state vector is updated repeatedly asynchronously until it doesn't change from one step to the next. This iteration is done by performing a product between the state vector and the connection weight matrix, then applying a sigmoid function to the result.

$$\begin{aligned} &x_j(n+1) = f\left[\sum_{i=1}^{N} W_{ji} \cdot x_i(n)\right], \quad \text{for } j = 1, 2, \ldots, N. \\ &f \text{ is a non-linear function} \end{aligned} \tag{2.10}$$

Once the state vector has converged, the network enters the final stage: outputting. In this stage the state vector is simply returned as the result[11, 8].

A variation on the asynchronous update mentioned above uses a synchronous update and is known as a Little model. The main difference in performance is that while the Hopfield model will always converge to a stable state, the Little model will converge to a stable state or a limit cycle of length at most 2[8].

The Little model of the Hopfield architecture is amenable to GPU computation because it inherently uses the vector-matrix product. As in the multi-layer perceptron, it is also possible to amortize your work and perform matrix-matrix multiplication. However, in this model a matrix-matrix amortized approach seems to be less of an advantage because some vectors may converge well before other vectors do. Convergence and different points results in wasted cycles trying to converge already settled results.

2.2.8 Adaptive Resonance Theory (ART)

Adaptive resonance theory is based on the "winner takes all" model. In this architecture a series of vector templates is created, initially with value 1 or greater. A series of vector dot products is then performed to determine the template that has the closest value to the input vector. The closest template will have the largest dot product with the input vector. This template is then compared to see if it is "close enough" to the input vector via a user controlled variable called "vigilance". If the vector and template are indeed close enough, then the template is updated through a process called "erosion" in which a template monotonically decreases in zero or more dimensions. In Fuzzy ART the erosion is a fuzzy erosion (namely the min operation) whereas in ART1 a "logical and" is the erosion operator. If the vector and template are not close enough a new template is formed as a copy of the input[8, 17].

A brief overview of the Fuzzy ART algorithm follows:

$$\begin{aligned} &T_j(I) = \tfrac{|I \wedge w_j|}{\alpha + |w_j|} \\ &T_j \text{ is the choice function for input } I \text{ and category } j \end{aligned} \tag{2.11}$$

Where the fuzzy and operator $\wedge$ is defined by

$$(x \wedge y) \equiv min(x, y) \tag{2.12}$$

and the norm operator $|\cdot|$ is defined by

$$|x| \equiv \sum_{i=1}^{N} |x_i| \tag{2.13}$$

For a given input I the category choice is denoted as J where

$$T_J = max(T_j : j = 1...N) \tag{2.14}$$

Resonance occurs if the match is good enough

$$\begin{aligned} &\frac{|I \wedge w_j|}{|I|} \geq \rho \\ &\rho \text{ is the vigilance} \end{aligned} \tag{2.15}$$

Otherwise reset occurs and the value for T_J is set to -1 so that it will not be the max value ever again for this input.

Learning the weight vector occurs according to the following equation:

$$\begin{aligned} &w_J^{\text{new}} = \beta(I \wedge w_J^{\text{old}}) + (1 - \beta) w_J^{\text{old}} \\ &\beta \text{ is the learning speed} \end{aligned} \tag{2.16}$$

As in the previous two architectures, this architecture would seem to be amenable because it can be formulated as a vector-matrix dot product. This model can not be extended to the matrix-matrix multiplication form because the templates may change after each input.

Chapter 3

Findings

It is the tension between creativity and skepticism that has produced the stunning and unexpected findings of science.
Carl Sagan,

3.1 Results

The results will be presented on a architecture by architecture basis. Each set of results is the average time compiled over a number of runs for each datum. Specific dimensions, parameters, and iteration count will be given when appropriate.

3.1.1 Hardware Used

The hardware used in this research consisted of 3 graphics cards and three desktops. One desktop served as the CPU model and development platform. This desktop

has dual processor Xeon Pentium 4 2 GHz processors in it, 1 GB of RAM, and an NVIDIA 6800 AGP card. The other machines were utilized only for the GPU execution model. The graphic cards consisted of an NVIDIA 6800 Ultra AGP, and a ATI Radeon X800 PCI-X.

3.1.2 Software Used

The software used in this study consisted of three operating systems, one development and two testing. The development machine mentioned above was configured as a dual-boot Windows 2000™and Debian Linux "non-stable" machine running the 2.6.10 kernel. The testing machines consisted of a Windows XP Professional machine (the ATI x800 machine) running Cygwin, and a Linux machine (the NVIDIA 6800 Ultra) running "non-stable" and kernel 2.6.10. The Windows operating systems allowed for the testing of OpenGL and DirectX, while the Linux machine only utilized OpenGL.

3.1.3 Multi-Layer Perceptron Results

The multi-layer perceptron was the first architecture implemented in this research, and as such has had the most revisions made to it over time. Initially the model was implemented in vector-vector dot product form and it was approximately 20% slower than the same model implemented for the CPU using ATLAS BLAS . Following these results the perceptron was re-implemented in vector-matrix form to capitalize on the GPU's better performance in matrix-based math. Figures 3.1, 3.2, and 3.3 show the results from running a series of increasingly large multi-layer perceptrons. The algorithm as implemented involves three layers of size N, one output layer of size 4, and a input vector of size N. The ANN was run for 200 iterations per datum, where each iteration involves the loading of the weight vectors (included in

the running-time) and the presentation of 2000 unique input vectors to the ANN. The running time is then averaged across the 200 runs to produce a datum in the results. Averaging was necessary to help reduce the noise introduced by the system potentially being partially utilized by system level operations.

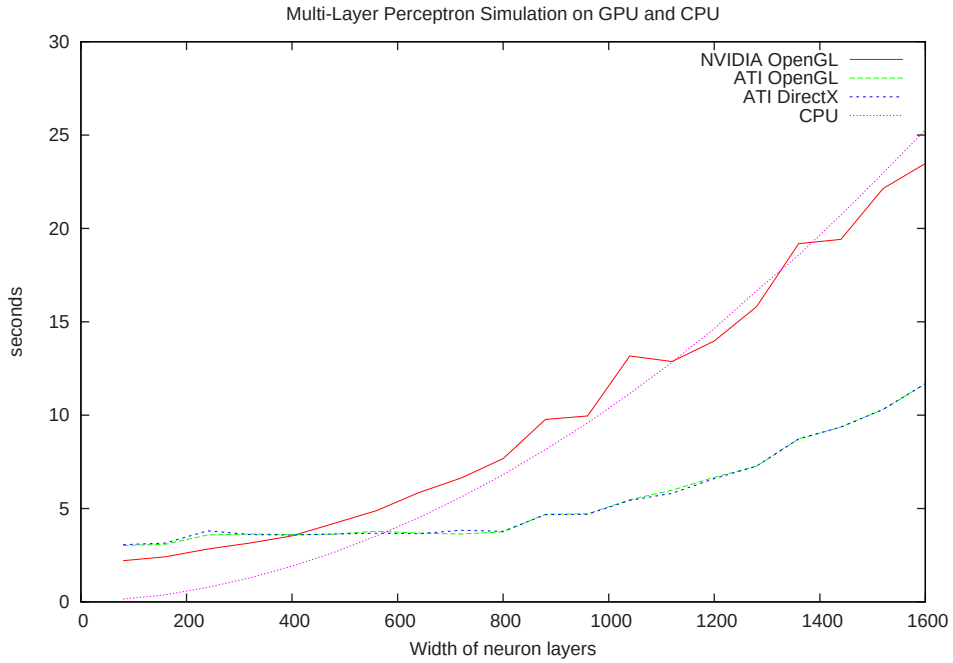

Figure 3.1: Comparison of two independent simulations of the MLP on the GPU. Notice that the "bumps" are present in all GPU simulation runs.

3.1.4 Little model Hopfield Network Results

The Little model of the Hopfield network was the second architecture implemented and measured. The initial implementation was made utilizing the vector-matrix product developed for the previous model. Because it is not simple to resize the memory matrix of a Hopfield network, the measurement made on this model instead shows the results of increasing the number of iterations performed in the "Iterate

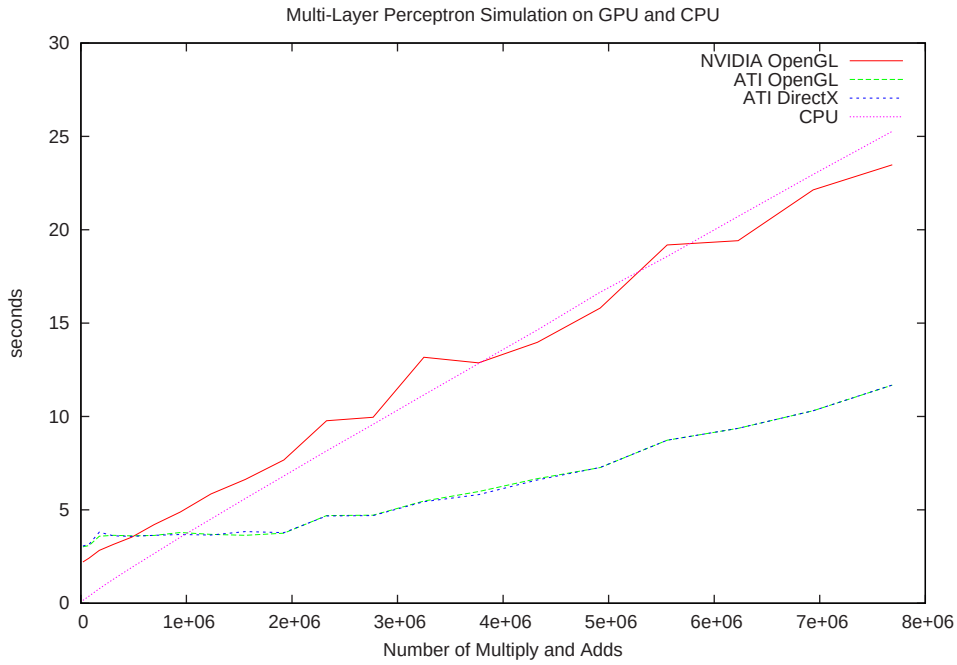

Figure 3.2: Comparison of MLP network simulated on GPU vs CPU using the number of multiplication and addition operations. As one might expect the performance scales linearly. (discussed in section 3.2.4)

until convergence" step (see section 2.2.7). The memory matrix represented a series of characters represented by a 32 by 32 pixel bit-map. The resulting memory matrix was 1024 by 1024 neurons (each character bitmap was turned into a single string of length 1024 for purposes of encoding and training). Figure 3.4 shows the results of increased iteration count run times.

3.1.5 Adaptive Resonance Theory Results

The final architecture implemented was F-ART. It was implemented using a fuzzy dot product kernel based on the vector-matrix product kernels used in the previous two implementations. Results of running the GPU version against the CPU version

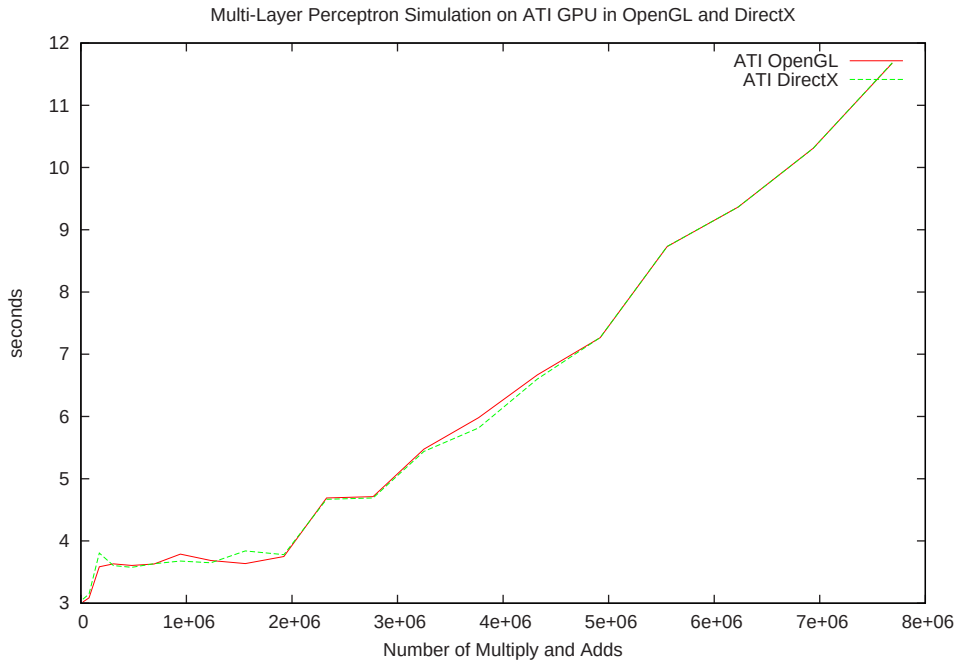

Figure 3.3: Comparison of MLP network simulated on the ATI Radeon 800 GPU comparing OpenGL vs DirectX using the number of multiplication and addition operations

showed that the GPU performance was very poor in comparison to the CPU (Table 3.1). In fact, the CPU was between 468024 and 679785 times faster than the GPU. No graph will be included in light of the poor performance, however the reasons the performance was so poor will be discussed in the next section. It should be pointed out that F-ART was the only on-line learning ANN implemented.

Fuzzy ART Performance	
ATI OpenGL	25.690157 seconds
ATI DirectX	25.831837 seconds
NVIDIA OpenGL	17.784929 seconds
CPU	0.000038 seconds

Table 3.1: A chart of Fuzzy ART network performance

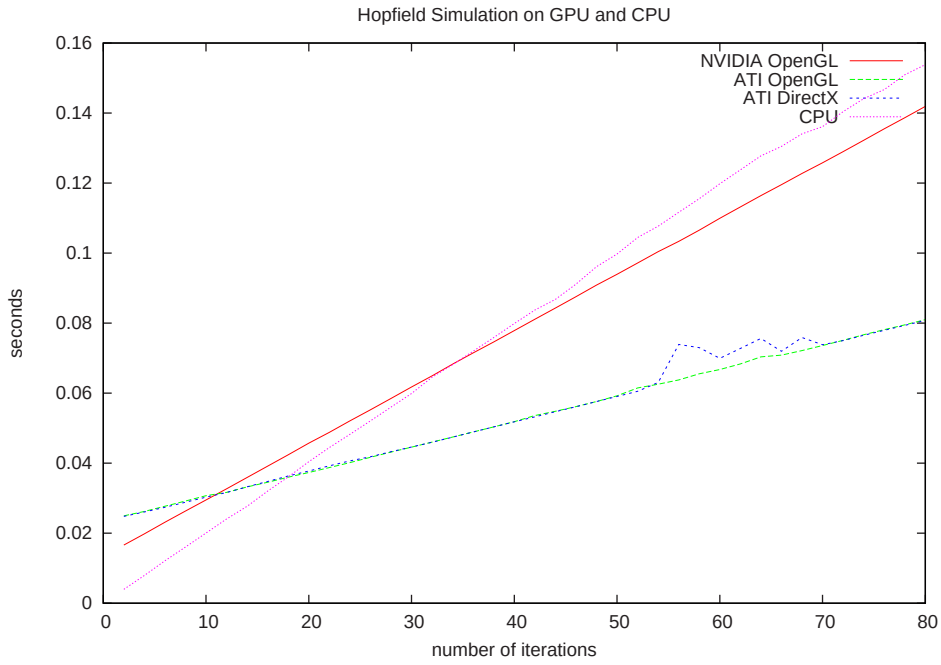

Figure 3.4: More evidence that the GPU overcomes the CPU's performance when enough operations per byte are performed

3.2 Discussion

What follows is a discussion of the results obtained from each architecture, discussion on the study, some potential interpretations of the results, what was done well in this research and what could have been done better or differently, and some connections back to the literature. This chapter is concluded by a discussion of what general characteristics make a ANN amenable or ill-suited to GPU execution.

3.2.1 Interpretations

The results presented above can generally be interpreted as the problem of overcoming the "operations per byte" threshold to make GPU simulation better performing

than the CPU counterpart. Researchers at Stanford refer to the threshold problem as the algorithmic intensity problem. Because the underlying hardware is constantly evolving, as well as the driver interfaces dramatically changing the interface to the hardware from revision to revision, it is extremely hard (and potentially misleading) to assign a specific operation per byte measure at which the GPU will surpass the CPU in performance.

One overarching high level interpretation that should be gathered from this research is that if one is interested in using the GPU as a serious computational platform, it is important to consider what kinds of operations you are performing, how many operations can be collapsed into larger operations, how many operations (roughly) per byte can be performed, how much traffic must occur on the system bus, and what your intended virtual machine representation is.

More detail into the specific traits that are desirable or undesirable in computation are described in section 3.2.7.

Comparing the ATI and the NVIDIA results from this experiment is of some interest. The large performance gap between the two cards is due to several factors. First, and probably most important, the I/O characteristics of the ATI card are far superior to that of the NVIDIA card due to the PCI-X bus. The PCI-X bus can transport 4.3 GB/sec of data in either direction while the AGP bus present on the NVIDIA card is limited to 2.1 GB/sec. Furthermore, the AGP bus has been measured to typically have good performance reading to the card, but poor performance reading back from the card to main memory.

3.2.2 What was right/wrong

Choosing BrookGPU as a language to develop these approaches in seems to have been a very good decision. Without BrookGPU the development of a particular

network architecture could have taken much longer, and poor approaches to the implementation of specific kernels would have resulted in much more lost research time. Debugging BrookGPU code also turned out to be much simpler than debugging assembly level code would have been.

Choosing BrookGPU as a language also limited the types of operations that could be performed. For example at the time of research there did not appear to be any way of performing a render to a sub section of a texture, and as such partial updates of networks is impossible. Recent driver updates by NVIDIA claim to support the FBO (Frame Buffer Object) which would allow for direct render to texture operations rather than using the P-Buffer, which may alleviate some of the issues in future research. Further discussion of this limitation can be found in the F-ART section below.

Starting at the smallest, most simplistic and direct implementations of the approaches to the linear algebra and the resulting networks and working up to more complex and involved models was a very good approach to have taken. This approach allows the performance at large to be characterized on the full range of operations one could desire to implement for most ANNs. This research examines much of the area not explored by the previous two sets of authors in the field[16, 10].

3.2.3 Improvements

One of the only serious potential improvements this research could benefit from would be a more in-depth and involved investigation of GLSL, Cg, HLSL, and assembly level implementation of the networks of interest. This improvement would allow for measurement and characterization of the ability of the GPU to simulate an actively learning ANN, since rendering to sub-texture regions could then be available.

One potential improvement to the MLP would be to implement matrix-matrix

products. However the matrix-matrix product approach to MLP has already been investigated by other researchers[16, 10].

3.2.4 MLP Results Discussion

The multi-layer perceptron was by far the most measured and improved upon model in this investigation. Although the MLP as implemented does show better performance on the GPU than on the CPU, it is not the largest performance gap.

Figure 3.1 shows the results of running increasingly large networks. The growth rate of the curve is expected because of the growing number of multiplications and additions. As mentioned above the output layer was fixed to 4 output neurons and the input and hidden layers were iteratively increased in size. The intersection points of the various curves show the points at which one processor overtakes another in performance. The ATI processor shows better performance than the CPU at just before layers of size 600 neurons, while the NVIDIA card shows better performance than the CPU just after 1100 neuron layers are run.

Figure 3.2 shows the more results of running increasingly large networks. In this graph the x axis is the number of multiplications and additions. As expected, this scale removes the majority of the curve from the results. It is interesting to note that the slope of the ATI results is much shallower than the slope for the NVIDIA results. As above the intersection points show when one processor overtakes another. In this view the ATI processor overtakes the CPU at approximately 1×10^6 multiply and add operations, while the NVIDIA card overtakes the CPU at approximately 4×10^6 multiply and add operations.

Figure 3.3 shows the same results as above, but only comparing the ATI results. This graph allows for more direct and unobstructed comparison of DirectX and OpenGL performance. It is interesting to note that there is very little difference

between the two curves. These results are particularly surprising because DirectX is supposed to have much more direct access to the hardware under Windows XP than OpenGL does. OpenGL is forced to go through Microsoft's HAL (Hardware Abstraction Layer) before reaching the hardware.

Of potential interest are the results shown in Figures 3.1 and 3.2. Initially it seemed as though the "bumps" in the curve were noise from system resources being consumed by other processes, but after several runs it must be concluded that these "bumps" are actually present in the performance. These perturbations are of some interest and are, as of yet, unexplained.

3.2.5 Hopfield Results Discussion

Figure 3.4 shows the results of running increasing numbers of iterations against the Little model Hopfield network. In this experiment the NVIDIA GPU performance exceeds the CPU performance at around 35 iterations which is well within normal operating conditions. The ATI performs even better; exceeding the CPU performance at around 19 iterations. These results make the Little model Hopfield network a very nice model for GPU simulation. The basic trait that enables this excellence in performance is the learn once, run many times aspect of the network. The Hopfield networks investigated involved only one matrix, and thus only one texture to operate on, while other architectures involved texture switching and re-texturing of the quad used to perform the actual low-level computation.

3.2.6 F-ART Results Discussion

Although F-ART possesses many traits that would seem to make it very GPU friendly, testing showed that within BrookGPU this was not the case. At the time

of implementation there was no apparent support for rendering to a sub-region of the texture in memory within BrookGPU, so to update the template matrix the entire matrix is repeatedly pushed across the bus. Repeated bus traffic and latency killed the performance, and made it impossible to ever do better than the CPU's performance.

3.2.7 What makes an ANN amenable to GPU execution?

About now hopefully you have an idea of what kinds of traits make a ANN ill or well suited to GPU simulation, but let's summarize some of the characteristics here:

- SIMD type operations – SIMD type operations are extremely well suited to programmable GPU execution. The basic form of the shader kernel and the hardware implementation are direct analogs of the SIMD architectures used on the CPU. The GPU has evolved to handle these kernel repeated convolutions in a parallel fashion to improve graphics performance and SIMD operations for computation can exploit this trait.

- vector-matrix or matrix-matrix style linear algebra – Linear algebra that utilizes a matrix (or a collection of vectors) scales nicely for the GPU. For vector-matrix operations as you increase the size of your matrix, you linearly increase the number of operations per value (float). If you are able to utilize matrix-matrix operations this scaling becomes N^2 where N is the size of your matrix. This scaling factor, or algorithmic intensity, is of upmost importance for the GPU. The design of the GPU is such that it is meant to be heavily utilized with a high operation per byte ratio (more on this below).

- minimal bus traffic – One of the major hurdles to GPU performance is the amount of data being sent to and from the card. The bus is by far the slowest

communication channel in the hardware of the graphics card, and as such should be avoided whenever possible.

- no learning (at least with Brook) – Although it is possible to implement learning for a ANN using some of the other languages, it is possible that learning could hurt the performance gap between the CPU and the GPU. Considering the previous item (minimal bus traffic), unless you can perform all the learning on the card (all the matrix or vector updates in texture memory) learning is something to be avoided.

- little or no branching – the branching performance of a GPU is poor in comparison to what is available on modern CPUs. The SIMD style of optimization is not well suited to conditional programming (if, while, for, switch, etc). The fact that the processors are named "pipelines" seems to hint at this, but tests confirm the suspicion.

- high operations to byte ratios – There are several factors that make high algorithmic intensity a desirable trait in GPU simulation. The bus issue (mentioned above) is one of them, however this is not the end of the story. Because the processors are aimed towards repetitive convolution kernels, or SIMD style operations, performing multiple operations on the same byte of data is highly advantageous and optimized. The increased instructions are not "free", but the resulting performance curve has a much shallower slope to it, and thus it scales very nicely.

Chapter 4

Summary and Conclusion

People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant.
Helen Keller (1880 - 1968)

There are many people who reach their conclusions about life like schoolboys: they cheat their master by copying the answer out of a book without having worked the sum out for themselves.
Søren Kierkegaard

4.1 Summary

In this section a brief summary of the thesis and research is reiterated.

4.1.1 The Area

Artificial neural networks are a useful approach to solving many difficult problems. ANNs are attractive because they posses many traits that allow them to make statements about neurobiology, statistics, classification, pattern recognitions, and many other interesting and useful fields.

Modern consumer graphics cards are rapidly becoming more powerful. The processors packaged on these cards are designed to dramatically improve graphics performance, but are applicable to many other useful types of computation as well.

4.1.2 The Problem

The problem as stated in this research is to implement artificial neural networks on the graphics card, characterize traits that help or hinder a ANN's performance on the GPU, and generally show that the simulation of ANNs on the GPU is a worthwhile endeavor, worthy of future study and experimentation.

4.1.3 Other's Solutions

Other researchers have implemented the multi-layer perceptron using techniques described in [14, 12, 20, 16, 10]. There have been many other researchers investigating other general purpose computing lines of research for the GPU (see http://www.gpgpu.org).

4.1.4 This Solution

This solution was implemented in BrookGPU, a stream programming C language extension developed at Stanford University. A series of architectures and revisions were developed to allow for the characterization of GPU performance as it relates to ANN simulation. Results were presented, discussed, and the efficacy and applicability of GPU simulation established.

4.1.5 What was right/wrong?

BrookGPU allowed several rapid prototypes of artificial neural networks to be implemented for both OpenGL and DirectX. This portability allowed measurements to be taken on both Windows and Linux, and comparisons made.

BrookGPU while a useful tool for this exploratory work, seems to have some shortcomings. The largest deficiency was the apparent lack of ability to update subsections of streams (in graphics parlance this is updating a sub-texture). Recent forum discussion points to the possibility of some solutions to this problem, however investigating this possibility is beyond the scope of the research in the time given.

BrookGPU and shader programming at large is still a very young field. As such many of the development tools are wanting, features are bug-filled, and performance is likely to change drastically over the course of a few months.

4.1.6 The Study

Generally speaking the study can be considered a success. Some architectures were shown to be well suited to GPU simulation using BrookGPU, and others were shown to be ill suited. Strong results do not always need to be positive to be good, and

this research points out some of the benefits and strengths, as well as shortcomings and weaknesses of GPU programming. This research shows some of the potential of GPU simulation of ANNs and leaves the field open for future research.

4.2 Conclusion

The field of artificial neural networks is an interesting approach to some difficult problems. It has held the attention of researchers for over 60 years now, and seems to be worthy of many more years of study. There is a multitude of reasons to consider using a neural based solution to real-world problems, and several of them have been outlined in this thesis. Coupled with commercial off-the-shelf graphics hardware found in many personal computers, significant improvements in simulation performance can be realized. Although more power is not a panacea, it is a good step towards allowing us to solve larger, harder, and ultimately more interesting problems and questions.

Chapter 5

Future Work

"This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."
Winston Churchill, Speech given at the Lord Mayor's Luncheon, Mansion House, London, November 10, 1942.

This research is worth much future investigation. Among the endeavors that are worthy of study are:

- assembly level implementations of ANNs to establish close to optimal performance measurements
- assembly level implementations of "high level primitives" to allow for more optimal implementations
- other high level language implementations to explore benefits and drawbacks of each language's capabilities

- other architectures beyond the popular consumer ATI and NVIDIA graphics cards (perhaps examine gaming console graphics cards)
- re-running the currently implemented tests with the new FBO options
- implement learning and determine efficacy of on-board learning
- applying this technology to interesting data sets
- building powerful hybrid CPU/GPU ANNs that capitalize on each component's strengths

Appendices

A BrookGPU Multi-Layer Perceptron **57**

B ATLAS BLAS Multi-Layer Perceptron **64**

C BrookGPU Little model Hopfield Network **70**

D ATLAS BLAS Little model Hopfield Network **78**

E BrookGPU Fuzzy Adaptive Resonance Theory **83**

F CPU Fuzzy Adaptive Resonance Theory **92**

G Thomas Rolfes matrix product implementation **98**

Appendix A

BrookGPU Multi-Layer Perceptron

```
// mlp.br
// Christopher Davis
// chris2d@cs.unm.edu
// Final BrookGPU implementation of a multi layer perceptron.
// This code utilizes vector matrix dot product in a parallel fashion
// to attempt to realize better performance.

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <lin_time.h>

reduce void rsum4(float4 x<>, reduce float4 result<>) {
  result += x;
}
```

```
kernel void forceGPUFlush_float4(float4 input<>, out float4 result<>) {
  result = input;
}

kernel void tanh4(float4 x<>, out float4 result<>) {
  result = tanh(x);
}

// A is (m/4)-by-n
// x is (n/4)-by-1
// result is (m/4)-by-(n/4)
kernel void denseMatVecKernel(iter float2 it1<>, iter float2 it2<>,
                              iter float2 it3<>, iter float2 it4<>,
                              float4 A[][], float4 x<>, out float4 result<>) {
  float4 data1 = A[it1.xy];
  float4 data2 = A[it2.xy];
  float4 data3 = A[it3.xy];
  float4 data4 = A[it4.xy];

  result  = x.x * data1;
  result += x.y * data2;
  result += x.z * data3;
  result += x.w * data4;

}

kernel void multKernel4(float4 x<>, float4 y<>, out float4 result<>) {
```

```
  result = x * y;
}

void do_MLP(int test, int num_iter, int length, int dim,
            float fWideDim, float fDim ){

  float *x, *y, *A, *B, *C, *D, *oArray, *timeArray;

  int size_of_comp;

  int i,j,k;

  float4 xStrm<1, dim>;
  float4 yStrm<dim, 1>;
  float4 AStrm<dim, length>;
  float4 BStrm<dim, length>;
  float4 CStrm<dim, length>;
  float4 DStrm<dim, 1>;
  float4 oStrm<1, 1>;

  float4 tmpStrm<dim, dim>;
  float4 resultStrm<dim, 1>;

  float4 flushStrm<1,1>;
  float4 flush[1];

  iter float2 it1<dim, dim> = iter( float2(0.0f, 0.0f),
                                    float2(fWideDim, fDim) );
```

```
  iter float2 it2<dim, dim> = iter( float2(1.0f, 0.0f),
                                    float2(fWideDim+1.0f, fDim));
  iter float2 it3<dim, dim> = iter( float2(2.0f, 0.0f),
                                    float2(fWideDim+2.0f, fDim));
  iter float2 it4<dim, dim> = iter( float2(3.0f, 0.0f),
                                    float2(fWideDim+3.0f, fDim));

  x = (float*)malloc(sizeof(float) * length);
  y = (float*)malloc(sizeof(float) * length);
  A = (float*)malloc(sizeof(float) * length * length);
  B = (float*)malloc(sizeof(float) * length * length);
  C = (float*)malloc(sizeof(float) * length * length);
  D = (float*)malloc(sizeof(float) * length);
  oArray = (float*)malloc(sizeof(float) * 4);
  timeArray = (float*)malloc(sizeof(float) * test);

  /* fill in matrix and x-vector values */
  srand(time(NULL));
  for (i=0;i<length;i++) {
    x[i] = (float)(rand()-(float)(RAND_MAX/2))/(float)RAND_MAX;
    D[i] = (float)(rand()-(float)(RAND_MAX/2))/(float)RAND_MAX;
    y[i] = 0.0f;
  }

  for (i=0;i<length;i++) {
    for (j=0;j<length;j++) {
      A[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                        (float)RAND_MAX;
```

```
100       B[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                            (float)RAND_MAX;
          C[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                            (float)RAND_MAX;
        }
      }


      InitTime();
      for (k = 0; k < test; k++){
110   UpdateTime();

      streamRead(AStrm, A);
      streamRead(BStrm, B);
      streamRead(CStrm, C);
      streamRead(DStrm, D);

      for (i=1;i<=num_iter;i++) {
        streamRead(xStrm, x);
        denseMatVecKernel(it1, it2, it3, it4, AStrm, xStrm, tmpStrm);
120     rsum4(tmpStrm, resultStrm);
        tanh4(resultStrm, yStrm);

        denseMatVecKernel(it1, it2, it3, it4, BStrm, yStrm, tmpStrm);
        rsum4(tmpStrm, resultStrm);
        tanh4(resultStrm, xStrm);

        denseMatVecKernel(it1, it2, it3, it4, CStrm, xStrm, tmpStrm);
```

```
    rsum4(tmpStrm, resultStrm);
    tanh4(resultStrm, yStrm);

    multKernel4(DStrm, yStrm, resultStrm);
    rsum4(resultStrm, oStrm);
    tanh4(oStrm, oStrm);
  }

  forceGPUFlush_float4(oStrm, flushStrm);
  streamWrite(flushStrm, flush);
  streamWrite(oStrm, oArray);

  UpdateTime();

  timeArray[k] = GetElapsedTime();

}
  size_of_comp = length*length + length*length + length*length + 4*length;
  for (k = 1; k < test; k++){
        timeArray[0] += timeArray[k];
  }
  printf(" %d %d %f \n", length, size_of_comp, (timeArray[0]/(float)test));

free(x);
free(y);
free(A);
free(B);
free(C);
```

```
    free(D);
    free(oArray);
    free(timeArray);

}


int main() {
  int num_iter = 2000;
  int test = 200;
  int base_dim = 20;

  int length, dim;
  float fWideDim, fDim;

  int i;

  for (i = 20; i < 21; i++){
        dim = base_dim * i;
        length = base_dim * i * 4;
        fWideDim = (float) length;
        fDim = (float) dim;
        do_MLP(test, num_iter, length, dim, fWideDim, fDim );
  }

  return 0;
}
```

Appendix B

ATLAS BLAS Multi-Layer Perceptron

```
// mlp_BLAS.c
// Christopher Davis
// chris2d@cs.unm.edu
// My final version of a multilayer perceptron written for ATLAS BLAS
// I use sgemv and sdot to achieve maximal CPU performance

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cblas.h>
#include "lin_time.h"
#include <time.h>
#include <assert.h>

void do_MLP(int test, int num_iter, int length){
```

```
int size_of_comp;
int sizeX = length;
int sizeY = length;
int dim = length/4;

float *x, *y, *A, *B, *C, *D, *x1, *x2, *x3, *x4;
float *d1, *d2, *d3, *d4, *timeArray;

float oArray[4] = {0.0f, 0.0f, 0.0f, 0.0f};

int i,j,k;

x1 = (float*)malloc(sizeof(float) * length/4);
x2 = (float*)malloc(sizeof(float) * length/4);
x3 = (float*)malloc(sizeof(float) * length/4);
x4 = (float*)malloc(sizeof(float) * length/4);
d1 = (float*)malloc(sizeof(float) * length/4);
d2 = (float*)malloc(sizeof(float) * length/4);
d3 = (float*)malloc(sizeof(float) * length/4);
d4 = (float*)malloc(sizeof(float) * length/4);
x = (float*)malloc(sizeof(float) * length);
y = (float*)malloc(sizeof(float) * length);
A = (float*)malloc(sizeof(float) * length * length);
B = (float*)malloc(sizeof(float) * length * length);
C = (float*)malloc(sizeof(float) * length * length);
D = (float*)malloc(sizeof(float) * length);
timeArray = (float*)malloc(sizeof(float) * test);
```

```
  /* fill in matrix and x-vector values */
  srand(time(NULL));
  for (i=0;i<length;i++) {
    x[i] = (float)(rand()-(float)(RAND_MAX/2))/(float)RAND_MAX;
    D[i] = (float)(rand()-(float)(RAND_MAX/2))/(float)RAND_MAX;
    y[i] = 0.0f;
  }

  for (i=0;i<length;i++) {
    for (j=0;j<length;j++) {
      A[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                        (float)RAND_MAX;
      B[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                        (float)RAND_MAX;
      C[ i*length+j ] = (float)(rand()-(float)(RAND_MAX/2))/
                        (float)RAND_MAX;
    }
  }

  InitTime();

  for (k = 0; k < test; k++){
  UpdateTime();
  for (i=1;i<=num_iter;i++) {

    cblas_sgemv(CblasRowMajor, CblasNoTrans, sizeX, sizeY,
                1, A, sizeX, x, 1, 0, y, 1);
    for (j=0; j < sizeX; j++){
```

```
        y[j] = tanh(y[j]);
    }

    cblas_sgemv(CblasRowMajor, CblasNoTrans, sizeX, sizeY,
                1, B, sizeX, y, 1, 0, x, 1);
    for (j=0; j < sizeX; j++){
        x[j] = tanh(x[j]);
    }

    cblas_sgemv(CblasRowMajor, CblasNoTrans, sizeX, sizeY,
                1, C, sizeX, x, 1, 0, y, 1);
    for (j=0; j < sizeX; j++){
        y[j] = tanh(y[j]);
    }

    for (j = 0; j < dim; j++){
        x1[j] = y[j];
        x2[j] = y[j+dim];
        x3[j] = y[j+dim];
        x4[j] = y[j+dim];
        d1[j] = D[j];
        d2[j] = D[j+dim];
        d3[j] = D[j+dim];
        d4[j] = D[j+dim];
    }

    oArray[0] = tanh(cblas_sdot(dim, x1, 1, d1, 1));
    oArray[1] = tanh(cblas_sdot(dim, x2, 1, d2, 1));
```

```
100         oArray[2] = tanh(cblas_sdot(dim, x3, 1, d3, 1));
            oArray[3] = tanh(cblas_sdot(dim, x4, 1, d4, 1));
        }

        UpdateTime();
        timeArray[k] = GetElapsedTime();
        }
        size_of_comp = length*length + length*length + length*length + 4*length;

        for (k = 1; k < test; k++){
110         timeArray[0] += timeArray[k];
        }
        printf(" %d %d %f \n", length, size_of_comp, (timeArray[0]/(float)test));

        free(x);
        free(y);
        free(A);
        free(B);
        free(C);
        free(D);
120     free(x1);
        free(x2);
        free(x3);
        free(x4);
        free(d1);
        free(d2);
        free(d3);
        free(d4);
```

```
  free(timeArray);

}

int main() {

  int base_length = 20;
  int test = 100;
  int num_iter = 2000;
  int length, i;

  for (i = 1; i < 21; i++){
        length = base_length * i * 4;
        do_MLP(test, num_iter, length );
  }

  return 0;
}
```

Appendix C

BrookGPU Little model Hopfield Network

```
// hopfield1.br
// Christopher Davis
// chris2d@cs.unm.edu
// Hopfield network
// This code utilizes vector matrix dot product in a parallel fashion
// to attempt to realize better performance.

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <lin_time.h>
#define EPS 1e-9f

void printdat(float *a){
        int i, j;
```

```
        for (i = 0; i < 32; i++){
            for (j = 0; j < 32; j++){
                if(a[i*32+j]>0) printf("1");
                else printf("0");
            }
            printf("\n");
        }
}

reduce void rsum4(float4 x<>, reduce float4 result<>) {
  result += x;
}

kernel void forceGPUFlush_float4(float4 input<>, out float4 result<>) {
  result = input;
}

kernel void tanh4(float4 x<>, out float4 result<>) {
  result = tanh(x);
}

kernel void minus4(float4 x<>, float4 y<>, out float4 result<>) {
  result = x - y;
}

kernel void plus4(float4 x<>, float4 y<>, out float4 result<>) {
  result = x + y;
}
```

```
// A is (m/4)-by-n
// x is (n/4)-by-1
// result is (m/4)-by-(n/4)
kernel void denseMatVecKernel(iter float2 it1<>, iter float2 it2<>,
                              iter float2 it3<>, iter float2 it4<>,
                              float4 A[][], float4 x<>, out float4 result<>) {


  float4 data1 = A[it1.xy];
  float4 data2 = A[it2.xy];
  float4 data3 = A[it3.xy];
  float4 data4 = A[it4.xy];

  result  = x.x * data1;
  result += x.y * data2;
  result += x.z * data3;
  result += x.w * data4;
}

kernel void multKernel4(float4 x<>, float4 y<>, out float4 result<>) {
  result = x * y;
}

kernel void equals4(float4 x<>, out float4 result<>){
        result = x;
}
```

```
kernel void ones4(out float4 result<>){
        result = 1.0;
}

int main() {

  int length = 1024;
  int dim = 256;
  int tests = 1000;
  int baseval = 1;
  float fDim = 256.0;
  float fWideDim = 1024.0;

  float *x, *y, *z, *A, *ones, *timeArray;
  float tempF = 0.0;

  int i,j,k,l,m;

  float4 xStrm<1, dim>;
  float4 x1Strm<1, dim>;
  float4 yStrm<dim, 1>;
  float4 y1Strm<dim, 1>;
  float4 AStrm<dim, length>;

  float4 tmpStrm<dim, dim>;

  float4 flushStrm<1,1>;
  float4 flush[1];
```

```
100
    iter float2 it1<dim, dim> = iter( float2(0.0f, 0.0f),
                                      float2(fWideDim, fDim) );
    iter float2 it2<dim, dim> = iter( float2(1.0f, 0.0f),
                                      float2(fWideDim+1.0f, fDim));
    iter float2 it3<dim, dim> = iter( float2(2.0f, 0.0f),
                                      float2(fWideDim+2.0f, fDim));
    iter float2 it4<dim, dim> = iter( float2(3.0f, 0.0f),
                                      float2(fWideDim+3.0f, fDim));

110 printf ("# Dimensions :%d by %d \n",length, length );

    ones = (float*)malloc(sizeof(float) * 2);
    x = (float*)malloc(sizeof(float) * length);
    y = (float*)malloc(sizeof(float) * length);
    z = (float*)malloc(sizeof(float) * length);
    A = (float*)malloc(sizeof(float) * length * length);
    timeArray = (float*)malloc(sizeof(float) * tests);

    /* fill in matrix and x-vector values */
120 srand(time(NULL));

    ones[0] = 1.0;
    ones[1] = -1.0;

    FILE *datafile;
    datafile = fopen("hopfield.dat", "r");
```

```
  if (!datafile)
  {
130       printf("Unable to open data file: %s \n", "hopfield.dat");
          exit(1);
  }

  for (i=0;i<length/4;i++) {
    for (j=0;j<length;j++) {
      k = fscanf(datafile, "%f", &tempF);
      A[ 4*(i*length + j) ] = tempF;
      k = fscanf(datafile, "%f", &tempF);
      A[ 4*(i*length + j)+1] = tempF;
140       k = fscanf(datafile, "%f", &tempF);
      A[ 4*(i*length + j)+2] = tempF;
      k = fscanf(datafile, "%f", &tempF);
      A[ 4*(i*length + j)+3] = tempF;
    }
  }

  InitTime();
  for (i = 0; i < 40; i++){

150 for (l=0;l<length;l++) {
    x[l] = ones[(rand()%2)];
    y[l] = ones[(rand()%2)];
  }
  for (m = 0; m < tests; m++){
  UpdateTime();
```

```
    streamRead(xStrm, x);
    streamRead(AStrm, A);

    forceGPUFlush_float4(xStrm, flushStrm);
    forceGPUFlush_float4(AStrm, flushStrm);
    streamWrite(flushStrm, flush);

    k = baseval + ( i);
    j = 0;
    while(k){

    denseMatVecKernel(it1, it2, it3, it4, AStrm, xStrm, tmpStrm);
    rsum4(tmpStrm, yStrm);
    //plus4(yStrm, y1Strm, yStrm);
    tanh4(yStrm, yStrm);

    denseMatVecKernel(it1, it2, it3, it4, AStrm, yStrm, tmpStrm);
    rsum4(tmpStrm, xStrm);
    //plus4(xStrm, x1Strm, xStrm);
    tanh4(xStrm, xStrm);

    j+=2;
    k--;
    }

    forceGPUFlush_float4(yStrm, flushStrm);
    streamWrite(flushStrm, flush);
```

```
  streamWrite(yStrm, y);

  UpdateTime();
  timeArray[m] = GetElapsedTime();
  }
  for (m = 1; m < tests; m++){
190         timeArray[0] += timeArray[m];
  }

  printf ("%d %f \n", j, (timeArray[0]/(float)tests));

  // printdat(y);
  }
  return 0;
}
```

Appendix D

ATLAS BLAS Little model Hopfield Network

```
// hop_cpu1.c
// Christopher Davis
// chris2d@cs.unm.edu
// A Hopfield network
// This code utilizes ATLAS BLAS vector matrix dot product
// to attempt to realize better performance.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cblas.h>
#include "lin_time.h"
#define MAX_DIM 2048
#define EPS     1e-9f
```

```
void printdat(float *a){
        int i, j;

        for (i = 0; i < 32; i++){
            for (j = 0; j < 32; j++){
                if(a[i*32+j]>0) printf("1");
                else printf("0");
            }
            printf("\n");
        }
}

int main() {

  int length = 1024;
  int baseval = 1;

  float *x, *y, *z, *A, *ones;
  float tempF = 0.0;

  int tests = 1000;

  float timeArray[tests];
  float oArray[length];

  int i,j,k,l,m;
```

```
printf ("# Dimensions :%d by %d \n",length, length );

ones = (float*)malloc(sizeof(float) * 2);
x = (float*)malloc(sizeof(float) * length);
y = (float*)malloc(sizeof(float) * length);
z = (float*)malloc(sizeof(float) * length);
A = (float*)malloc(sizeof(float) * length * length);

/* fill in matrix and x-vector values */
srand(time(NULL));

ones[0] = 1.0;
ones[1] = -1.0;

FILE *datafile;
datafile = fopen("hopfield.dat", "r");

if (!datafile)
{
      printf("Unable to open data file: %s \n", "hopfield.dat");
      exit(1);
}

for (i=0;i<length*length;i++) {
    k = fscanf(datafile, "%f", &tempF);
    A[i] = tempF;
}
```

```
InitTime();
for (l = 0; l < 40 ; l++){
for (m=0;m<length;m++) {
   x[m] = ones[(rand()%2)];
}
for (m = 0; m < tests; m++){
UpdateTime();

k = baseval + (l);
j = 0;
while(k){

cblas_sgemv(CblasRowMajor, CblasNoTrans, length, length,
            1, A, length, x, 1, 0, y, 1);
for (i=0; i < length; i++){
    y[i] = tanh(y[i]);
}

cblas_sgemv(CblasRowMajor, CblasNoTrans, length, length,
            1, A, length, y, 1, 0, x, 1);
for (i=0; i < length; i++){
    x[i] = tanh(x[i]);
}

j +=2;
k--;
}
```

```
100     UpdateTime();

        timeArray[m] = GetElapsedTime();
        }
        for (m = 1; m < tests; m++){
            timeArray[0] += timeArray[m];
        }
        printf ("%d %f \n", j, (timeArray[0]/(float)tests));

        // printdat(x);
110     }
        return 0;
    }
```

Appendix E

BrookGPU Fuzzy Adaptive Resonance Theory

```
// fart1.br
// Christopher Davis
// chris2d@cs.unm.edu
// Fuzzy ART
// This code utilizes vector matrix fuzzy dot product in a paralell fashion
// to attempt to realize better performance.

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <lin_time.h>
#define MAX_DIM 2048

static inline float mymin(float a, float b){
    float r;
```

```
    r = a;
    if (b < a) r = b;
    return r;
}

reduce void rsum4(float4 x<>, reduce float4 result<>) {
  result += x;
}

kernel void div4b(float4 x<>, float alpha, out float4 result<>){
        result = x / alpha;
}

kernel void div4a(float4 x<>, float4 y<>, float alpha, out float4 result<>){
        result = x / (y.x + y.y + y.z + y.w + alpha);
}

kernel void div4(float4 x<>, float4 y<>, out float4 result<>){
        result = x / y;
}

kernel void forceGPUFlush_float4(float4 input<>, out float4 result<>) {
  result = input;
}

// A is (m/4)-by-n
// x is (n/4)-by-1
// result is (m/4)-by-(n/4)
```

```
kernel void fdot(iter float2 it1<>, iter float2 it2<>,
                 iter float2 it3<>, iter float2 it4<>,
                 float4 A[][], float4 x<>, out float4 result<>) {

  float4 data1 = A[it1.xy];
  float4 data2 = A[it2.xy];
  float4 data3 = A[it3.xy];
  float4 data4 = A[it4.xy];

  // This is slower to do the float4 for some reason
  // result  = min(x, data1);
  // result += min(x, data2);
  // result += min(x, data3);
  // result += min(x, data4);

  // Do the faster min
  result.x  = abs(min(x.x, data1.x));
  result.x += abs(min(x.y, data2.x));
  result.x += abs(min(x.z, data3.x));
  result.x += abs(min(x.w, data4.x));
  result.y  = abs(min(x.x, data1.y));
  result.y += abs(min(x.y, data2.y));
  result.y += abs(min(x.z, data3.y));
  result.y += abs(min(x.w, data4.y));
  result.z  = abs(min(x.x, data1.z));
  result.z += abs(min(x.y, data2.z));
  result.z += abs(min(x.z, data3.z));
```

```
  result.z += abs(min(x.w, data4.z));
  result.w  = abs(min(x.x, data1.w));
  result.w += abs(min(x.y, data2.w));
  result.w += abs(min(x.z, data3.w));
  result.w += abs(min(x.w, data4.w));
}

int main() {

  const static int length = 1024;
  const static float fWideDim = 1024.0;
  const static int dim = 256;
  const static float fDim = 256.0;
  const static int num_iter = 1;
  const static int test = 1000;



  // choice parameter
  float alpha = 1.0f;
  // learning rate
  float beta = 1.0f;
  // vigilance
  float rho = 0.5f;
  float max = 0.0f;

  // the uncommited template value, larger than 1 leads to deeper
  // searches of previously coded categories
  float uncommited = 1.0f;
```

```
100
    float *x, *y, *z, *A, *tTime;
    float check_size = 0.0f;

    int i,j,testC,iterC,maxindex,reset_events,max_checks;

    float4 xStrm<1, dim>;
    float4 yStrm<dim, 1>;
    float4 zStrm<dim, 1>;
    float4 TSizeStrm<dim, 1>;
110 float4 AStrm<dim, length>;
    float4 tmpStrm<dim, dim>;
    float4 flushStrm<1,1>;
    float4 flush[1];

    iter float2 it1<dim, dim> = iter( float2(0.0f, 0.0f),
                                      float2(fWideDim, fDim) );
    iter float2 it2<dim, dim> = iter( float2(1.0f, 0.0f),
                                      float2(fWideDim+1.0f, fDim));
    iter float2 it3<dim, dim> = iter( float2(2.0f, 0.0f),
120                               float2(fWideDim+2.0f, fDim));
    iter float2 it4<dim, dim> = iter( float2(3.0f, 0.0f),
                                      float2(fWideDim+3.0f, fDim));

    x = (float*)malloc(sizeof(float) * length);
    y = (float*)malloc(sizeof(float) * length);
    z = (float*)malloc(sizeof(float) * length);
    tTime = (float*)malloc(sizeof(float) * test);
```

```
  A = (float*)malloc(sizeof(float) * length * length);

130 /* fill in matrix and x-vector values */
  srand(time(NULL));
  reset_events = 0;
  max_checks = 0;
  for (i=0;i<length;i++) {
    y[i] = 0.0f;
    z[i] = 0.0f;
  }

  for (i=0;i<length/4;i++) {
140   for (j=0;j<length;j++) {
      A[ 4*(i*length + j) ] = uncommited;
      A[ 4*(i*length + j)+1] = uncommited;
      A[ 4*(i*length + j)+2] = uncommited;
      A[ 4*(i*length + j)+3] = uncommited;
    }
  }

  InitTime();
  for (testC = 0; testC < test; testC++){
150 UpdateTime();

  for (i=0;i<length/2;i++) {
    x[i] = (float)rand()/(float)RAND_MAX;
    x[(length/2)+i] = 1.0f-x[i];
  }
```

```
    check_size = 0.0;
    for(i = 0; i < length; i++){
            check_size += x[i];
160 }

    for (iterC=1;iterC<=num_iter;iterC++) {
    streamRead(xStrm, x);
    streamRead(yStrm, y);
    streamRead(zStrm, z);
    streamRead(AStrm, A);

    forceGPUFlush_float4(xStrm, flushStrm);
    forceGPUFlush_float4(yStrm, flushStrm);
170 forceGPUFlush_float4(AStrm, flushStrm);
    streamWrite(flushStrm, flush);

    // Compute the category choice
    fdot(it1, it2, it3, it4, AStrm, xStrm, tmpStrm);
    rsum4(tmpStrm, yStrm);
    rsum4(AStrm, TSizeStrm);
    div4a(yStrm, TSizeStrm, alpha, zStrm);

    // Compute the resonance or reset
180 div4b(yStrm, check_size, yStrm);

    forceGPUFlush_float4(yStrm, flushStrm);
    forceGPUFlush_float4(zStrm, flushStrm);
```

```
streamWrite(flushStrm, flush);
streamWrite(yStrm, y);
streamWrite(zStrm, z);
// Don't need to read this out each time,
// just read it in each time since
// I do the update on the CPU side
// streamWrite(AStrm, A);
}

j = 1;
while(j){
max = 0.0f;
maxindex = 0;
for (i = 0; i < length; i++){
      if (max < z[i]){
            max = z[i];
            maxindex = i;
            max_checks++;
      }
}

if (y[maxindex]<rho){
      z[maxindex]= -1;
      reset_events++;
}
else {
      //we are close enough, let's learn
      for (i = 0; i < length/4; i++){
```

```
                    A[ 4*((maxindex/4)*length+i)+maxindex ] =
                        beta * (mymin(x[i],
                                A[ 4*((maxindex/4)*length+i)+maxindex ]))
                        + (1-beta)*A[ 4*((maxindex/4)*length+i)+maxindex ];
            }
            j = 0;
        }
    }
220

    UpdateTime();
    tTime[testC] = (float) GetElapsedTime();

    }
    for (i=1;i<test;i++) {
      tTime[0] += tTime[i];
    }
    printf("Total time = %f \n", tTime[0]);
230 return 0;
}
```

Appendix F

CPU Fuzzy Adaptive Resonance Theory

```
// fart_cpu.c
// Christopher Davis
// chris2d@cs.unm.edu
// This code implements a CPU version of Fuzzy ART.
// No ATLAS BLAS implementation of min existed so I wrote my own.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "lin_time.h"
#include <string.h>

static inline float min(float a, float b){
    float r;
```

```
    r = a;
    if (b < a) r = b;
    return r;
}

int main() {
    InitTime();
    int record_count,i,tempTime,j,k,l;
    float zTime = 0.0;
    float smaxRatio = 0.0;
    float dmaxRatio = 0.0;
    float sminRatio = 100000;
    float dminRatio = 100000;

    char source_file[60];
    char source_mfile[60];
    const static int length = 10;
    int templates = 1024;
    int templates_used = 0;
    int templates_ud = 0;
    int test = 1000;

    float *x, *y, *z, *w;
    float ** A;
    double alpha = 1.0;
    double beta = 1.0;
    double rho = 0.5;
    float SizeI = 0.0f;
```

```
    float uncommited = 1.0f;
    int notsat, maxindex = 0, reset_events = 0, max_checks = 0;
    float max = 0.0f;
    record_count = 0;

    A = (float **)malloc(templates * sizeof(float *));
    for (int i = 0; i < templates; i++)
        A[i] = (float *)malloc(length * sizeof(float));

    x = (float*)malloc(sizeof(float) * length);
    y = (float*)malloc(sizeof(float) * templates);
    z = (float*)malloc(sizeof(float) * templates);
    w = (float*)malloc(sizeof(float) * templates);


  printf ("# Dimensions :%d by %d \n",templates, length );

  /* fill in matrix and x-vector values */
  srand(time(NULL));
  printf("\n \n \n");

  for (i=0;i<templates;i++) {
    for (j=0;j<length;j++) {
      A[i][j] = uncommited;
    }
  }

  for(k = 0; k < test; k++){
```

```
UpdateTime();
record_count++;

for (i=0;i<length/2;i++) {
    x[i] = (float)rand()/(float)RAND_MAX;
  x[(length/2)+i] = 1.0f-x[i];
}

for (j=0;j<templates;j++) {
  y[j] = 0.0f;
  w[j] = 0.0f;
}

    for (j = 0; j < templates_used+2; j++){
        for (l = 0; l < length; l++){
            y[j] += fabs(min(x[l], A[j][l]));
            w[j] += fabs(A[j][l]);
        }

        z[j] = y[j] / (alpha + w[j]);
    }

    SizeI = 0.0f;
    for (j = 0; j < length; j++){
            SizeI += fabs(x[j]);
    }

   notsat = 1;
```

```
100     while(notsat){
        max = 0.0f;
        for (j = 0; j < templates_used+2; j++){
            if (max < z[j]){
                max = z[j];
                maxindex = j;
                max_checks++;
            }
        }

110     if (y[maxindex]/SizeI < rho){
            // Reset condition
            reset_events++;
            z[maxindex] = -1;
        }
        else {
            // close - let's learn
            templates_ud++;
            notsat = 0;

120         for (l = 0; l < length; l++){
                A[maxindex][l] = beta * (min(x[l], A[maxindex][l])
                                + (1 - beta) * A[maxindex][l]);
            }
        }
        if (templates_used < maxindex){
            templates_used = maxindex;
```

```
        if (templates_used == templates - 1){
            printf("\n \n Ran out of template space - exiting\n \n ");
            exit(1);
        }
      }
      }
    UpdateTime();
    zTime += GetElapsedTime();
    }

    printf("Processed %d records \n in %f seconds \n",
        record_count, zTime/test);
    return 0;
}
```

Appendix G

Thomas Rolfes matrix product implementation

The following code was presented in Mr. Rolfes' paper [16]. The pixel shader is presented first followed by the vertex shader. The entire source is available for download from www.circensis.com.

```
// Partial matrix product
// oC0 <- A1.x*B1 + A1.y*B2 + A1.z*B3 + A1.w*B4 + C1*scale + bias,
// with scale in c0.x and bias in c0.y

ps_2_0                  // shader version

dcl_2d s0               // texture stage A
dcl_2d s1               // texture stage B
dcl_2d s2               // texture stage C

dcl t0.xy               // texture coordinate A1
```

```
dcl t1.xy               // texture coordinate B1 row 1
dcl t2.xy               // texture coordinate B2 row 2
dcl t3.xy               // texture coordinate B3 row 3
dcl t4.xy               // texture coordinate B4 row 4
dcl t5.xy               // texture coordinate C1

texld r0, t5, s2        // C1
mad r0, r0, c0.x, c0.y  // C1*scale + bias ...
texld r1, t0, s0        // A1
texld r2, t1, s1        // B1
mad r0, r1.x, r2, r0    // ... +A1.x*B1 ...
texld r2, t2, s1        // B2
mad r0, r1.y, r2, r0    // ... +A1.y*B2 ...
texld r2, t3, s1        // B3
mad r0, r1.z, r2, r0    // ... +A1.z*B3 ...
texld r2, t4, s1        // B4
mad r0, r1.w, r2, r0    // ... +A1.w*B4.

// Add more blocks here and modify Shaders::Gemm() accordingly ...

// write result to output register
mov oC0, r0
```

The following code is the vertex shader component of the matrix product.

```
// Vertex shader for the matrix product A*B+C where:
// A is an m by k matrix,
```

```
// B is an k by n matrix,
// C is an m by n matrix.
// k/4 passes are needed in order to complete the matrix product.
// Each pass multiplies a column of 4 cells of A with 4 rows of B.
// The texture coordinates B1 ... B4 are 4 subsequent rows of B.
// Constant registers:
// c0: 2, -1, 0.5, 1
// c1: adjustments for out
// c2: adjustments for A
// c3: adjustments for B
// c4: adjustments for C
// c5: dx for B, dy for B, step*dy for A, step*dx for B

vs_1_1

dcl_position v0  // v0.z holds the vertex index

// adjust and transform vertex position
mov r0, c0
mov r1, c1
mad r2.xy, v0.xy, r1.zw, r1.xy  // adjust subpos within [0, 1]
mad r2.xy, r2.xy, r0.xx, r0.yy  // scale and translate to [-1, 1]
mov r2.zw, c0.zw  // z <- 0.5, w <- 1
mov oPos, r2

// adjust A texture coordinate
mov r0, c2
mad r2.xy, v0.xy, r0.zw, r0.xy // subpos
```

```
mad r2.x, c5.z, v0.z, r2.x
// plus index dependent shift in x-direction
mov oT0, r2

// adjust B texture coordinates
mov r0, c3  // scale and translation
mad r2.xy, v0.xy, r0.zw, r0.xy  // subpos
mad r2.y, c5.w, v0.z, r2.y
// plus index dependent shift in y-direction
mov oT1, r2.xy
add r2.y, r2.y, c5.y  // next row
mov oT2, r2.xy
add r2.y, r2.y, c5.y  // next row
mov oT3, r2.xy
add r2.y, r2.y, c5.y  // next row
mov oT4, r2.xy

// adjust C texture coordinate
mov r0, c4
mad oT5, v0.xy, r0.zw, r0.xy  // subpos
```

References

[1] E. Angel. *Interactive computer graphics: a top-down approach with OpenGL.* Addison Wesley Longman, Reading, Massachusetts, 2nd edition, 2000.

[2] Valentino Braitenberg. *Vehicles: Experiments on Synthetic Psychology.* The MIT Press, 1987.

[3] R. A. Brooks and A. M. Flynn. Fast, cheap and out of control: A robot invasion of the solar system. *Journal of the British Interplanetary Society*, pages 478–485, October 1989.

[4] R. A. Brooks and L.A. Stein. Building brains for bodies. *Autonomous Robots*, 1(1):7–25, 1994.

[5] I. Buck. Brook language. web, October 2004.

[6] J. Clark. The geometry engine: A vlsi geometry system for graphics. *Computer Graphics*, 16(3):127–133, July 1982.

[7] Daniel Drubach. *The Brain Explained.* Prentice-Hall Inc, 1 edition, 2000.

[8] Simon Haykin. *Neural Networks: a Comprehensive Foundation.* Prentice Hall, second edition edition, 1999.

[9] Donald O. Hebb. *The Organization of Behaviour.* John Wiley & Sons Inc, December 1949.

[10] K. Oh & K. Jung. Gpu implementation of neural networks. In *Pattern Recognition*, volume 37, page 1311 1314. Pergamon, January 2004.

[11] George F. Luger. *Artificial Intelligence Structures and Strategies for Complex Problem Solving.* Addison Wesley, Harlow, England, fifth edition, 2005.

[12] E. S. Larsen & D. McAllister. Fast matrix multiplies using graphics hardware. In *Super Computing 2001 Conference*. Available online at www-2.cs.cmu.edu/ alokl/15745/larsen_mcallister.pdf, Denver, CO, November 2001.

[13] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 1943.

[14] Á. Moravanszky. Dense matrix algebra on the gpu. In *ShaderX2*. Wordware Publishing, 2003.

[15] M. L. Minsky & S. A. Papert. *Perceptrons: an Introduction to Computational Geometry*. MIT Press, 2nd edition, June 1969.

[16] Thomas Rolfes. Neural networks on programmable graphics hardware. In *Game Programming Gems 4*. Charles River Media, 2004.

[17] G. Carpenter & S. Grossberg & D. Rosen. Fuzzy art: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, 4:pp. 759–771, 1991.

[18] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. In *Psychological Review*, volume 65, pages 386–408. 1958.

[19] F. Rosenblatt. On the convergence of reinforcement procedures in simple perceptrons. Technical Report VG-1196-G-4, Cornell Aeronautical Laboratory Report, Buffalo, NY, 1960.

[20] J. Krüger & R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH 2003 conference proceedings*. Available online at wwwcg.in.tum.de/Research/Publications/LinAlg, 2003.

[21] Wikipedia. Artificial neural network. http://en.wikipedia.org/wiki/Artificial_neural_network, 2005.

[22] Wikipedia. Donald olding hebb. http://en.wikipedia.org/wiki/Donald_Hebb, 2005.

[23] Wikipedia. Graphics processing unit. http://en.wikipedia.org/wiki/Graphics_processing_unit, 2005.

[24] Wikipedia. John hopfield. http://en.wikipedia.org/wiki/John_Hopfield, 2005.

[25] Wikipedia. Neural network. http://en.wikipedia.org/wiki/Neural_network, 2005.

[26] Wikipedia. Perceptron. http://en.wikipedia.org/wiki/Perceptron, 2005.

[27] Wikipedia. Walter pitts. http://en.wikipedia.org/wiki/Walter_Pitts, 2005.

[28] Wikipedia. Warren sturgis mcculloch. http://en.wikipedia.org/, 2005.