Examining Suitability of Multicore Processor Architectures for Solving Realistic Computationally Intensive Problems by Simulating Synaptic Behavior in Large Neural Networks

A Thesis
in STS 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment
Of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Steven Cook

April 27, 2007

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for papers in Science, Technology, and Society courses.

TABLE OF CONTENTS

Abstract	**iv**
Chapter 1: Diminishing Returns in Processor Improvements	**1**
Multicore Background Overview	3
Looking Forward	5
Chapter 2: Social Impacts of Advancing Multicore Processor Technology	**6**
Chapter 3: Project Background in Computer Architecture and Related Fields	**10**
Chapter 4: Processor Comparison Methodology	**17**
Code Analysis	17
Prototyping	19
Re-writing	20
Comparisons	21
Chapter 5: Comparison of GPU and CPU Calculation Speeds	**23**
GPU vs CPU: Conclusions	25
Lessons Learned for Cuda Programming	26
The Future of GPGPU's	29
Bibliography	**35**
Works Cited	**30**
Appendix A: Test Machine Specs	**42**

Appendix B: Selected NeuroJet Function Code **43**

Appendix C: Prototype Source Code **44**

Appendix D: Final Implementation Source Code **46**

Appendix E: Table of Results **50**

ABSTRACT

As the ability to increase commodity microprocessor performance via increased clock frequency becomes severely limited by physical constraints, new techniques become necessary to continue to improve performance. One such technique is to place more than one execution core on a processor chip, thus creating multicore processors. Making efficient use of multicore hardware is a difficult task, as not all computer algorithms can effectively be split up to take advantage of the number of available processing cores. Several designs have been released in recent years, but it remains to be proven that these multicore processors can handle a wide range of computationally-intensive real-world applications.

This paper describes the undertaking of a project to test one such processor, the Nvidia GeForce 8800 GTX, by measuring how quickly and efficiently it can run a selected portion of a program to model synaptic filtering in large neural networks. The impacts of this technology on society, as well as any ethical issues that may arise as a result, are also examined. After writing a portion of the neural simulation to run on the graphics card, its speed was compared to the speed of performing the same task on the CPU for different input array sizes. While the CPU finished in shorter time for every input array size, the GPU was in fact far faster at doing the calculations but was hampered by a severe penalty for moving the input arrays to and from the GPU card. Recommendations for future research were made based on this result, main among which was to explore ways to improve the latency and transfer speed of the GPU card.

CHAPTER 1: DIMINISHING RETURNS IN PROCESSOR IMPROVEMENTS

The field of microprocessor design is approaching a branching point: the traditional techniques for increasing the performance of microprocessors are beginning to reach severe physical limitations. For example, further increasing processor frequency now generates so much heat that it is quickly becoming an unviable option for continuing the current rate of microprocessor performance increases. If we are to continue to enjoy the benefits of ever more powerful processors, including increasingly detailed simulations to increase scientific and medical understanding, then new approaches to processor design will be needed. One promising technique is to increase the number of cores, or execution entities, on a microprocessor. This project aims to steer such multicore processors towards a more generalized form to be better suited for executing computationally intensive real-world programs. To test the effectiveness of a processor developed using this design technique, an attempt was made to speed-up a complex synaptic-modeling simulation by offloading a computationally-intensive section of the simulation code to an Nvidia™ general-purpose graphics processing unit (GPGPU).

The underlying concept to this “multiple cores” approach is simply that if a processor cannot be made to perform more operations per second, then more than one of them can be run in parallel to increase overall data throughput. This method allows multiple tasks, or “threads”, to be executed simultaneously. However, making effective use of multiple cores is a complex task because programs have to be structured so as to effectively split, or “parallelize”, the program into multiple threads that can run concurrently. Different algorithms parallelize with differing levels of effectiveness, and

there is a lack of information as to what a useful, general-purpose multicore processor architecture would look like.

The goal, then, is to try to develop this "generalized" multicore processor that can efficiently support the widest range of applications. The first step towards such a processor is to test how well different computationally-intensive real-world applications can be made to perform on current multicore processor offerings. The next step is to analyze performance results to see what, if any, architectural changes need to be made. This information can then be used to design increasingly useful multicore processors. This project will perform such an analysis by means of a program called "NeuroJet" that models synaptic filtering behavior in large neural networks. The analysis will be performed by re-writing some of the most computationally-intensive portions of NeuroJet to take advantage of the parallel-processing capabilities of an Nvidia multicore processor (in this case, the g80 GPGPU), running the newly re-written code, and identifying any performance bottlenecks so as to allow for necessary improvements to be made on future multicore designs. Results show how much of a performance increase was gained, as well as what changes to the GPGPU hardware or software implementations would allow even greater performance increases in the future.

The Nvidia g80 GPGPU was chosen as the target processor due primarily to the relative ease of programming for it. The recently-released programming interface developed for it, called the Compute Unified Driver Architecture (CUDA), adds several GPGPU-specific extensions to the C programming language. This allows the GPGPU to be programmed using language very close to C, which saves the programmer from many difficult details of GPGPU interaction. Given the already great complexity of the synaptic

simulation program, it was deemed prudent to use the multicore device that had the simplest interface available.

Before going further into the details of the project and its methodology, it would be useful to give a review of related advances to elucidate the current state of multicore processor development and how this project will relate and build on past work.

MULTICORE BACKGROUND OVERVIEW

The rapidly evolving field of multicore processor design contains many areas of research that have until very recently remained virtually uncharted. This project aims to steer future multicore hardware processor development towards a more generalized form to be more suitable for executing computationally intensive real-world programs. Aside from the hardware aspects, the techniques for writing software to most efficiently utilize the power of multicore processors are still under development.

It is an interesting time for the type of multicore processor being used in this project. graphics processing units, or GPU's, contain multiple processing elements that operate simultaneously on streams of data, traditionally graphics-related computation for entertainment and 3D visualization. Graphics processing units have recently become an attractive choice of processor for performing many kinds of heavy-workload computations, not just those related to 3D imaging. This is due to two main reasons. The first reason is that they offer theoretical peak performance that can already surpass that of CPU's, and they are improving at a rate exceeding that of CPU's as well. The second reason is that their programming interfaces are becoming more and more generalized,

which frees developers from having to structure their programs as graphics computations. One such interface is the "CUDA" application programming interface (API), developed by Nvidia corporation and being utilized in this project. CUDA stands for Compute Unified Device Architecture, and has been released by Nvidia for use with the 8-series GeForce (G8X) family of GPU's being also used in this project. Since CUDA allows GPGPU-controlling code to be written by the developer in a modified version of C, existing applications can be ported over to the GPU and new ones can be written by programmers with knowledge of C. This new type of card, which can essentially be programmed in a mainstream language, is referred to as a general-purpose GPU (GPGPU).

As a side benefit of this increase in accessibility, researchers with limited knowledge of the details of GPU hardware are becoming able to turn this processing power towards furthering scientific endeavors. To find an example of where such a crossover might occur one needs look no further than the synaptic-modeling research program this project uses as a case study.

The research effort deals with the hippocampus, a part of the brain located inside the temporal lobe. Inside lie many millions of neurons and even more junctions, or "synapses," that act as interconnects between those neurons. As the hippocampus is involved in illnesses including epileptic seizures and Alzheimer's, neuroscientists hope that an increased understanding of the functioning of these neural networks will lead to cures for these diseases. To that end, work has been done on simulating networks of neurons and their synaptic junctions. This early work attempted to form computer models that could accurately predict many facets of synaptic behavior and resulting neural

excitation, an endeavor to be furthered as a result of this research. The importance of hippocampal research increased as a theory was proposed that attempted to accurately define the role of the hippocampus in the association and disassociation of memories.

The equations currently available to govern synaptic behavior are difficult to work with since they contain large numbers of variables. This made it necessary to study ways to create models which could be more easily simulated yet retain their accuracy. The software model being used as a baseline for this project is the product of one such attempt at creating more simulation-friendly models of synaptic behavior in neural networks.

NeuroJet currently takes 64 minutes to run a sample script containing 8000 neurons. Accelerating the simulation on the GPGPU would permit the number of neurons, duration of simulation, and other variables to be increased and thus allow new types of experiments to be performed. This additional research motivation provides an example of the kind of impact that GPU development can have.

LOOKING FORWARD

This research project is but one example of computing technology progressing to meet the social demand for ever more powerful processing capability. The next section, Chapter 2, will be devoted to a more in-depth analysis of the social and ethical ramifications of contributing to the processing race. Chapter 3 will review the literature in related fields, Chapter 4 will discuss the project methodology, and Chapter 5 includes results and conclusions. The thesis will end with recommendations for furthering this research.

CHAPTER 2: SOCIAL IMPACTS OF ADVANCING MULTICORE PROCESSOR TECHNOLOGY

This project will expand the knowledge base regarding the viability of one type of multicore processor, the General Purpose Graphics Programming Unit (GPGPU). As a result of this line of research, GPGPU's will likely become more usable and widely adopted by organizations doing computationally-intensive research, a change already taking place. The most direct and early consequences of this shift in the computational paradigm appear near unanimously beneficial to both the environment and research as a whole. Other issues arise when the GPGPU developments are put in the broader context of society's continual push towards ever greater and faster information processing, and for completeness some issues corresponding to this generic digital advance will also be examined. Lastly, since this project draws on neuroscience modeling as a potential application on which to employ GPGPU technology, it will also be necessary to discuss the implications of furthering this research into the workings of the brain.

As general-purpose Graphics Processing Units have increased in both processing power and ease of programmability, companies and research organizations are able to replace entire supercomputers with single GPGPU's capable of performing the same workload for a fraction of the cost. In 2003, $400 could by a graphics card with processing power comparable to that of an image generator costing hundreds of thousands of dollars in 1999 (Macedonia, 2003, p. 1). With innovations being subsidized by the multi-billion dollar gaming industry, prices are kept relatively low and progress is constant (Macedonia, 2003, p. 1). One company, Acceleware, has made a business out of

selling machines that use Nvidia brand GPU hardware to accelerate electromagnetic simulations and seismic data processing for customers (Acceleware, 2007) at a cost far lower than purchasing a supercomputer. This could mean some loss of business for the supercomputer divisions of companies such as IBM, with an increase in demand for GPU's benefiting the likes of Nvidia and AMD/ATI. In addition, power consumption would likely drop as supercomputer clusters are replaced with single machines running GPU hardware, benefiting the environment.

Looking at the context in which GPGPU advances have been made raises deeper social issues. It is one thing to know that advancing graphics processing unit technology now benefits research as well as entertainment. But the driving force is that of a perceived need for ever faster and more powerful computers. This perceived need for greater convenience and more immersive entertainment is accelerating the development of ever more powerful computers whose repercussions cannot be fully known until after the hardware is available. Just as GPGPU's were the result of applying entertainment-oriented technology to research, it is difficult to foretell what other fields this massive information processing capability will find its use in.

In a famous *Wired* magazine article, Bill Joy describes how the advanced computers and robotics of the future will concentrate power in the hands of the few (Joy, 2000). As Joy foresaw, the social asymmetry in control over information technology already prevalent today will be exacerbated by an increased reliance on computers in the future. It is possible, Joy predicts, that the majority of humans will be effectively removed from the decision making process either by a ruling elite of humans or something more than human: Artificial Intelligence.

As noted by Fogel & Fogel (1995), the ultimate goal of artificial intelligence is to find “a theory of intelligence that accounts for the behavior of naturally occurring intelligent entities and can be used to guide the creation of artificial entities that are capable of intelligent behavior” (p. 3). Research is already underway to “use genetic techniques to grow silicon brains trillions of times as complex as human ones” (Port,1995, abstract). Creating more and more powerful computers may facilitate the eventual overlapping of neuroscience, A.I. theory, and information processing necessary to give rise to thinking machines that will dramatically alter the fabric of society.

While these issues affect society in general, certain aspects of advancing neuroscience specifically affect parents. As noted by Bruer (1998),

> a flood of policy reports, conference proceedings, and professional and popular articles have proclaimed that "new" discoveries in brain science will revolutionize how we think about children, parenting, and early education. We have at our disposal, enthusiasts claim, a neuroscientific basis for an action and policy agenda on behalf of young children (p. 1).

This demonstrates a trend towards citing neuroscience as a basis for child care techniques. Should improvements in GPU technology allow more advanced neuroscience modeling in the future, they could possibly contribute to this trend for better or worse. There could be other unforeseen consequences as simulations are applied to other social sciences.

By contributing to the development of increasingly powerful computer processors as well as indirectly furthering the study of neural networks and neuroscience in general, this project may decrease the various entry costs to performing intense computations,

provide parents and educators with more informed child development knowledge and techniques, and help lead to cures for neurological damage. The broader push towards ever-greater computational ability that this research is but a part of has deeper potential long-term consequences, such as the theoretical risk that advancing information processing and related fields will concentrate power in the hands of a ruling technocratic elite - or maybe not even in people's hands at all. More directly, it is impossible to prove at this early stage that GPGPU technology will not find a use in some malicious field at some point in the future. In the final analysis issues like these are far-fetched social concerns far beyond the scope of this paper, and the proven, concrete benefits of GPGPU development should not be foregone on their behalf.

CHAPTER 3: PROJECT BACKGROUND IN COMPUTER ARCHITECTURE AND RELATED FIELDS

The rise of multicore processor architectures has been remarkably similar to the process of evolution. Multicore processors are the like the newest species trying to thrive (gain market share) by overcoming new survival threats to older processor designs (for instance, massive heat buildup). Like with mammals and dinosaurs, changes in the environment provided the opportunity for these less-common but well-adapted types of processors to gain dominance while other designs went extinct. The first step in understanding this evolutionary development will be a more detailed look at the technical problems in the microprocessor industry that multicore architectures were so well-suited to solve. A brief chronology of multicore development will then provide insight into the different evolutionary paths these processors are taking.

To understand the importance behind the branch that this project is testing, the general-purpose graphics processing unit (GPGPU), a general background on them will also be supplied. While this approach will introduce the important aspects of the problem in a logical order, it does so without answering the all-important question of how these newer, better processors will benefit society. To this end, background on the neuroscience aspects of the simulation software being used will also be given. Speeding up the simulation code is an example of a technical problem this research is trying to address, as neuroscience is an example of a field that stands to benefit greatly from successfully advancing multicore processors.

Something needed to be done to address the physical limitations impeding the drive towards faster single-core processors, a struggle that an Intel white paper summed up well: “The difficulty is that researchers now are coming up against the physical limits of atomic structure for scaling transistors while still managing power and thermals” (Ramanathan, 2006, p. 9). Managing heat generation and power consumption while still increasing the amount of work done in a given time period is the main motivation behind the ongoing mainstream push towards multicore processors. The following example was given in a speech by Intel CTO Justin Rattner explaining how adding more processor cores can reach this goal. For an example processor, a 20% increase in frequency will correspond to a 73% increase in power consumption for a 13% increase in total performance. Turning the frequency down by 20% yields a performance hit of 13% for a power consumption drop of near 50%. With the power saved, another processor could be added. The end result is that the power consumption is the same as in the beginning, but the two processors working simultaneously could theoretically provide a 73% increase in total performance (Rattner, 2006, p. 11). While these numbers are merely examples, they illustrate how the various parameters scale and how powerful the multicore approach is. The industry-wide consensus on the strength of this underlying theory gave birth to many different types of multicore implementations in recent years.

Like the history of evolution, the differentiation between processor “species” is not entirely clear, and classification is difficult. Tracking the chronology of multicore processor development is made difficult by the disagreement over what qualifies as a true “multicore” processor. For example, the Pentium D contains two Pentium 4 chips left connected on one end, with all of the connections between them made externally

(Stokes, 2006). Since the connections between the two Pentium 4 cores are not technically on the same piece of silicon, it is arguable as to whether or not it should be called a multicore processor in the strictest sense (Stokes, 2006). One proposed definition is that the processor cores must all be general-purpose, and by this definition the first multicore processor was the IBM Power4 dual-core CPU in the year 2000 (Gardner, 2006). Changing the definition to include any types of cores connected on-chip, and the likely candidate becomes the 1995 TMS320C80, a 4-core video processor CPU from Texas Instruments (Gardner, 2006). To call a GPU a multicore processor, as is done in this paper, is stretching the definition of "core" to include the individual data pipelines that compose the GPU.

An early work in the field of multicore processor compiler theory stated that "the future direction of parallel computing is not clearly defined, in part because of our lack of understanding of what constitutes effective machine organization and good programming methodology" (Banerjee, Eigenmann, Nicolau, & Padua, 1993, p. 1). Fourteen years later these problems are still not fully understood: in a February 2007 interview, Intel executive Pat Gelsinger acknowledged in an interview that the debate between heterogeneous vs. homogenous multicore processor architectures within his own company would likely not be resolved before 2020 (Goodwins, 2007). Today the industry abounds with competing multicore processor approaches, such as Nvidia's general-purpose graphical programming units and Sony's Cell processor. Each approach, however, currently has weaknesses. Examples are that the GPGPU experiences slowdown with Discrete Event Simulation (Perumalla, 2006), and the Cell processor's double-precision arithmetic throughput could stand to be further improved by

architectural modifications (Williams et al., 2006). There is, then, still much room for testing and improvement. Aside from the hardware aspects, the techniques for writing software to most efficiently utilize multicore processors are still under development, though several new approaches have recently been proposed (Eichenberger et al., 2005).

As relates specifically to GPGPU's, they are becoming an attractive option for handling heavy computation loads due to two main reasons. First, they offer performance that can already surpass that of CPU's, and they are improving at a rate exceeding that of CPU's as well (2x / year for GPU's compared to 1.5x / year for CPU's) (Luebke, Harris, Kruger et Al., 2004). The following figure demonstrates the growth of the computational ability of Nvidia GPU's versus CPUs from 2003 onward. In early 2003, GPU potential throughput exceeded that of high-end CPUs (Green, Simon and Mark Harris, 2006).

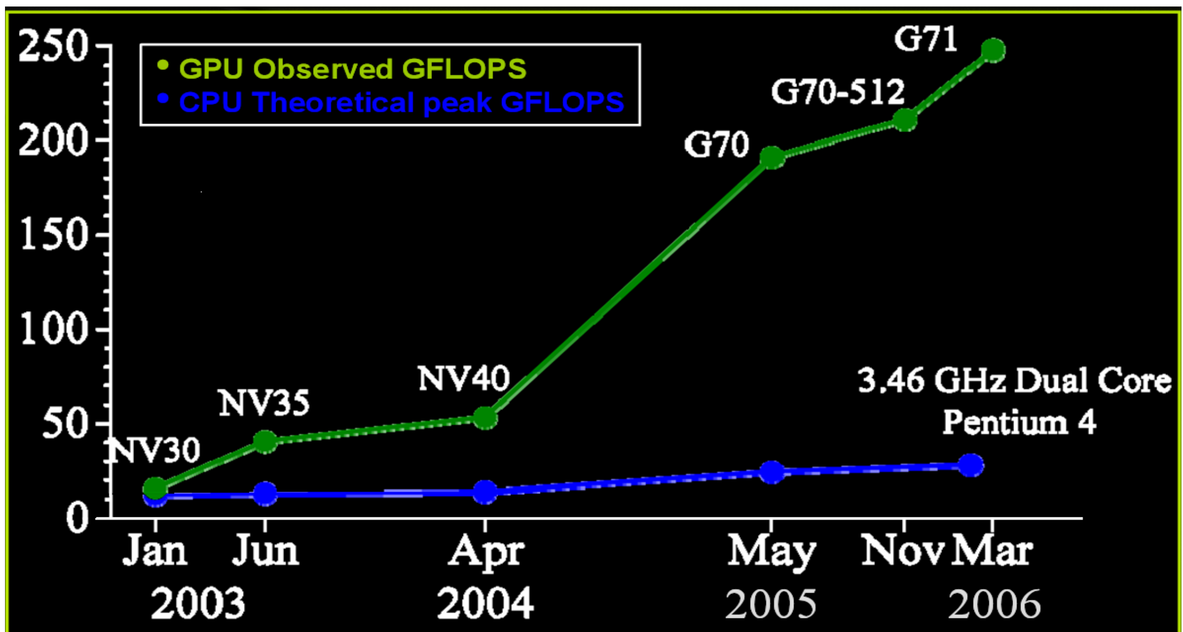

Figure 1: Comparison of GPU to CPU GFLOPS (giga floating point operations per second) from Jan 2003 to Mar 2006. Modified by Steven Cook from (Green, Simon and Mark Harris, 2006, slide 6).

Their power has grown so much that they are now used for applications besides graphics even within computer games, such as motion planning, AI, and physics (Macedonia, 2003, p. 2).

The second reason that GPGPU's are becoming more attractive is that their programming interfaces are becoming more and more general, permitting a broader range of applications to be employed (Luebke, Harris, Kruger et Al., 2004). A prime example of this is the "CUDA" application programming interface being utilized in this project. CUDA stands for Compute Unified Device Architecture, and has been released by Nvidia for use with the 8-series GeForce (G8X) GPU family being used in this project. Since CUDA allows GPGPU-controlling code to be written by the developer in a modified version of C, existing applications can be ported over to the GPGPU and new ones can be written by programmers with knowledge of C. Since CUDA was only released publicly on February 15, 2007, this thesis will be able to benefit the current literature by providing a real-world experience with CUDA development.

As a side benefit of this increase in accessibility, researchers with limited knowledge of the details of GPU hardware are able to turn this processing power towards furthering scientific endeavors. To find an example of where such a crossover might occur one need look no further than the synaptic-modeling research program this project uses as a case study. The research effort deals with the hippocampus, a part of the brain located inside the temporal lobe. Inside lie many millions of neurons and even more junctions, or "synapses," that act as interconnects between those neurons. As the hippocampus is involved in illnesses including epileptic seizures and Alzheimer's, neuroscientists hope that an increased understanding of the functioning of these neural

networks will lead to cures for these diseases. To that end, work has been done on simulating networks of neurons and their synaptic junctions as far back as 1988 (Traub, Miles, & Wong, 1988). This early work attempted to form computer models that could accurately predict many facets of synaptic behavior and resulting neural excitation, an endeavor to be furthered as a result of this research. The importance of hippocampal research increased as a theory was proposed that attempted to accurately define the role of the hippocampus in the association and disassociation of memories (Tsukada, 1993). The equations currently available to govern synaptic behavior are difficult to work with because they contain large numbers of variables, and as a result it became necessary to study ways to create models which could be more easily simulated yet retain their accuracy (Izhikevich, 1999). These equations are the subject of frequent re-workings; only two years ago a model of hippocampal function was proposed that included a special role for randomization (Levy, Hocking, & Wu, 2005). The software model being used as a baseline for this project is the product of one such attempt at creating more simulation-friendly models of synaptic behavior in neural networks.

The hippocampus (seen at right) also plays a vital role in spatial learning, or the mental mapping of an organism's environment (Image: Myers 2000). One recent model suggests that in rats, certain behaviors are triggered by recognition of landmarks, regardless

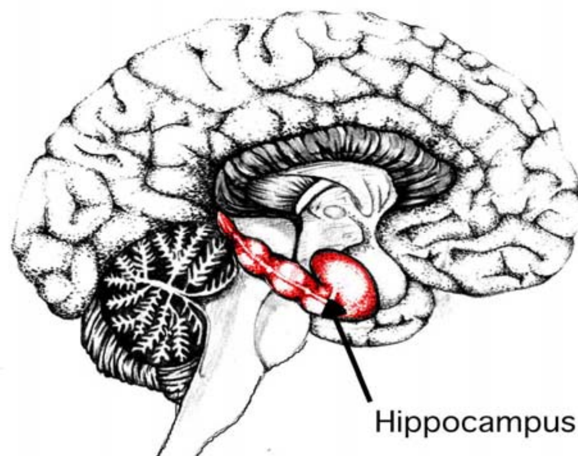

Figure 2: The location of the Hippocampus in the human brain. (Source: Myers, 2000).

of whether these landmarks have been moved (Denham & McCabe, 1998). Application of this model of learning to a mobile robot placed in an artificial environment has resulted in a machine able to successfully navigate back to a specified location (Burgess, Donnet, & O'Keefe, 1998). These two papers are examples of how research into the human brain has resulted in progress in artificial intelligence, an example of a potentially important link for research into advancing simulations.

The rapidly evolving field of multicore processor design contains many new developments that have yet to be fully explored and tested. Of the many different branches, the GPGPU family of multicore processors has just started to come into its own as a mainstream tool capable of handling massive calculations through use of parallel data pipelines. If it can be shown that computationally-intensive real-world simulations like the one used in this project for modeling synaptic filtering can in fact be sped up by using a GPGPU like the Nvidia G8X series, then GPGPU's could gain in prominence in neuroscience and other research fields. By contributing to the evaluation of GPGPU technology, this project shall help determine the future direction of GPGPU and, by extension, multicore processor adoption and development.

CHAPTER 4: PROCESSOR COMPARISON METHODOLOGY

The project can be thought of as composing three phases: code analysis, prototyping, re-writing, and comparing the CPU and GPU implementations. In the first phase, the NeuroJet program was studied to determine which functions took the most execution time – these would be the ideal candidates to speed up on a GPGPU. In the second phase, a prototype program was written that would allow me to gain experience with the CUDA programming interface as well as provide insight into how to parallelize loops onto the GPGPU. In the third phase, a replacement for the identified section of code was written to port the identified function's work over to the GPGPU to reduce the total run-time.

CODE ANALYSIS

To determine which functions in the large NeuroJet program would be ideal re-write candidates, it was necessary to know what portion of the total running time they each took. The size and complexity of NeuroJet made it difficult to analyze by hand; therefore, the decision was made to use a profiling tool instead. The GNU Profiler, or gprof, was used to gather statistics during program run-time that include the percentage of total run-time a function's execution occupies. Analyzing the output identified the most time-intensive functions, as shown by the following section of the gprof output:

CalcDendriticToSomaInput(xInput const&, bool) : 38.07% : 1468.88 secs : 297500 runs

NeuronType::forceExt() : 16.96% : 654.61 secs : 2380000000 runs

CalcSynapticActivation() : 13.43% : 518.30 secs : 297500 runs

CalcDendriticExcitation() : 11.97% : 462.01 secs : 297500 runs

This output lists the function name, followed by its total run time (percentage of total and in seconds), and finally how many times it was run. The code of these functions was then analyzed to determine how viable it would be to re-write them to be run on the graphics card under CUDA.

Each of these functions was then evaluated for their independence of other functions. This was deemed desirable since once the function was running on the GPU, it wouldn't be able to easily interact with the rest of the program. Due to the time cost of reading to and writing from the GPU, it was deemed unwise to offload any function that took very little time per run but ran billions of times, ruling out NeuronType::forceExt() as a viable option. Of these four functions, CalcDendriticExcitation() was chosen due to the fact that it had a computationally-intensive inner loop which did not interact with other functions. Its original code can be found in Appendix B, page 43. This inner loop was the portion re-written to run on the GeForce 8800 under CUDA.

The loop's task is to check all the elements in one float array, called sumwz[], to see if their absolute value is greater than a cutoff value. If an element is above the cutoff value, its value is written into the corresponding index in another array, dendExc[]. Loading this function onto the GeForce 8800 would test for an important bottleneck in the form of the delay in moving the two float arrays over to the graphics card, then

reading them back. Even if the GeForce GPU proved to be faster at completing the loop, it could still be slower overall if the transfer latency was too high.

PROTOTYPING

When the project was begun, Nvidia's CUDA API was a new development that was still under a Non-Disclosure Agreement. Neither I nor my thesis advisor had any previous experience with it, and despite its touted ease of use a steep learning curve was anticipated. A common programming practice in situations which require learning a new interface or model is to write a throw-away prototype program that addresses as many risk areas as possible in a simple, straightforward manner. I decided to write one such prototype program to help gain familiarity with the CUDA model.

Since the main feature of the portion of NeuroJet I was re-writing was a loop, I decided that it would be best to make my prototype also implement a loop. The example I settled on was a simple series summation, and successful implementation of this would allow me to gain an idea of how to make a programming loop work on the GPGPU. The example I settled on was a series summation, where all the values from 1 to a user-supplied number under 512 would be added together. The CUDA launch configuration was set up to launch a single block of 512 threads, and each thread would calculate based on its thread ID number. These values were summed into a "result" integer loaded to the card, and then read back to get the final answer. A code listing can be found in Appendix C, page 44.

RE-WRITING

With the prototype functioning successfully and basic concepts learned, it was time to expand the prototype into an implementation of the inner loop of the CalcDendriticExcitation() function. This required loading the two previously mentioned float arrays to the card, and reading one back when the card had finished calculating. The most significant difficulty of this step, and also the most important, was to modify the function run on the GPGPU so that it could support an arbitrary array size.

The GeForce 8800 is programmed under CUDA by writing a single function that every thread executes. Different threads are assigned to different data regions by means of their thread ID and their block ID. While there can be only up to 512 threads in a single block, many thousands of blocks created to allow for an extremely large number of threads to be dispatched. The task, then, was to launch a total number of threads as close as possible to the number of elements to be analyzed in the float array sumwz[]. I accomplished this by dividing the array size by 512, rounding up, and launching that many blocks with 512 threads each.

While this launch configuration always ensured that there were enough threads, it almost always results in having “leftover” threads. For instance, if there are 1000 elements in the array, 2 blocks of 512 threads each would be launched for a total of 1024 threads. If the last 24 threads naively tried to access their corresponding location in the array, this would cause an index-out-of-bounds error. To correct for this problem, a series of case statements was added to the code that would run on the GPGPU, and a “cutoff” thread value was passed so that the threads over the remainder would not perform any harmful action. The following pseudo code describes the checking algorithm used:

```
If(not last block)
    Execute code normally
Else ( in last block)
    If (thread ID < cutoff)
        Execute code normally
    Else ( thread ID >= cutoff)
        Do nothing
```

This algorithm was successful in computing all necessary values without exceeding the bounds of the index. After sample values were passed, the new code was found to be operating correctly in CPU emulation mode. Windows was the only environment available with a working GeForce 8800 card, so the code had to be ported over to a Microsoft Visual Studio 2005 project. The code was run on Windows under the Cygwin Linux emulation tool and found to execute properly on the actual GeForce 8800 card. This implementation was then deemed successful. A final code listing can be found in Appendix D, page 46.

COMPARISONS

Since Windows was the only environment available with a working GeForce 8800 card, and NeuroJet was written to be compiled under Linux, it was impossible to compare the program's runtime with and without using the GPU without making extensive changes to the NeuroJet simulation program. To provide a clear comparison, a simple CPU implementation of the inner loop of CalcDendriticExcitation() was written to

compare to the GPU implementation. After some false starts when using the Cygwin "time" function to measure execution time, CUDA's own timing functionality was decided upon. Timers would be created before the actual calculations were performed on the CPU and GPU, and stopped immediately upon completion. For details on timing implementation, refer to the final code listing in Appendix E, page 50. To measure for bottlenecks, the total GPU time was measured along with the setup time to load and read the float arrays from memory. The test was run for different sumwz[] loop sizes, with the loop always containing an alternating pattern of values above and below the cutoff threshold to ensure that both branches got equal coverage. Since it was mentioned that the array would always be at least 10,000 elements long, and longer in the future, the comparison was run over different array sizes. The sizes chosen were 10,000, 100,000, 1,000,000, 10,000,000, and 25,000,000. The results of the experiment are listed in the following section.

CHAPTER 5: COMPARISON OF GPU AND CPU CALCULATION SPEEDS

The first question to be answered in the comparison between CPU and GPU implementations of part of the NeuroJet synaptic simulation was simply which one could do the work in the shortest amount of time, for each array size. The following graph compares the total time of the GPU versus the CPU for the different array sizes tested, with lower times being preferable:

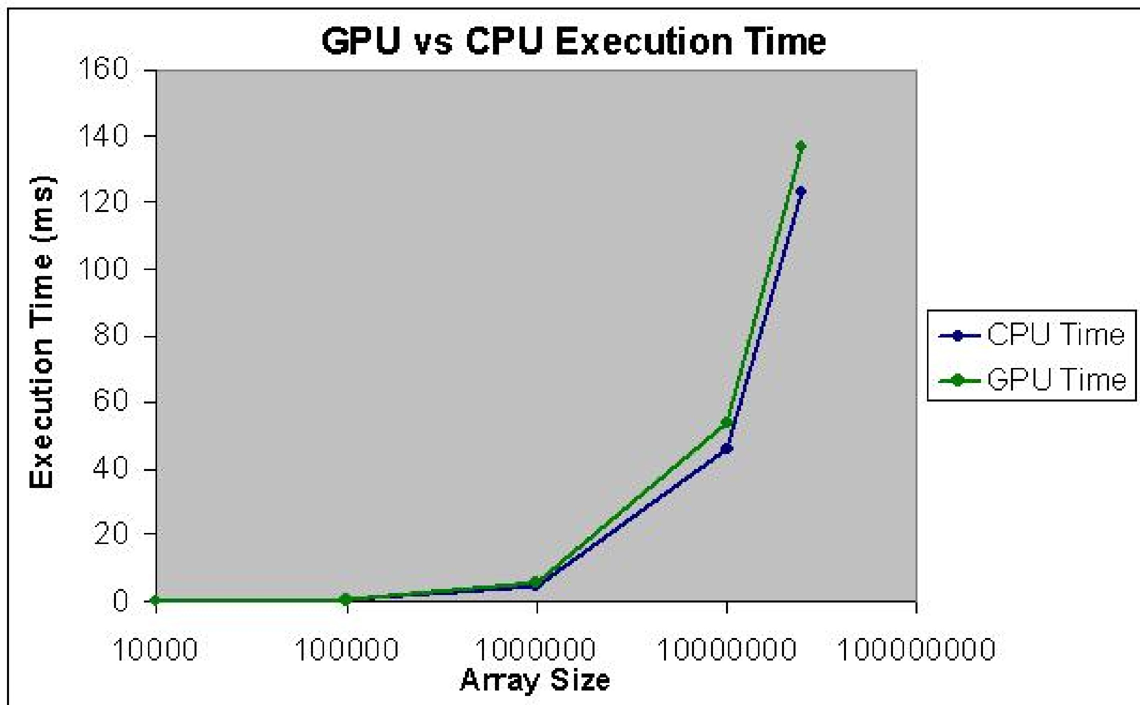

Figure 3: The GPU is never able to beat the CPU time. (Source: Steven Cook)

For all array sizes tested, the CPU finishes before the GPU does. The percentage difference starts out great at a size 10,000 array with the GPU being roughly 200% slower, but narrows over time until there is roughly a 10% speed difference at array size 25 million. Attempts to go far beyond 25 million resulted in the GPU crashing, presumably due to the array size exceeding the available memory.

These results tell that for this specific application, there is no compelling reason to employ the GeForce 8800 GPU as it will only slow the simulation down with the current on-card implementation I wrote. This is not entirely unexpected given the volume of data that had to be stored to and loaded from the GPU. The next step was to attempt to determine exactly how much loss was incurred as a result of moving the data.

Comparing the time to load and calculate on the GPU to the time to simply load the GPU was astonishing. The difference between the two values, the actual calculation time, was miniscule. A calculation that took an Intel Core 2 Duo processor 46 milliseconds could be completed on the GeForce 8800 GPU in only 2.3 milliseconds. Here is the same graph from above, with the GPU calculation-only time added:

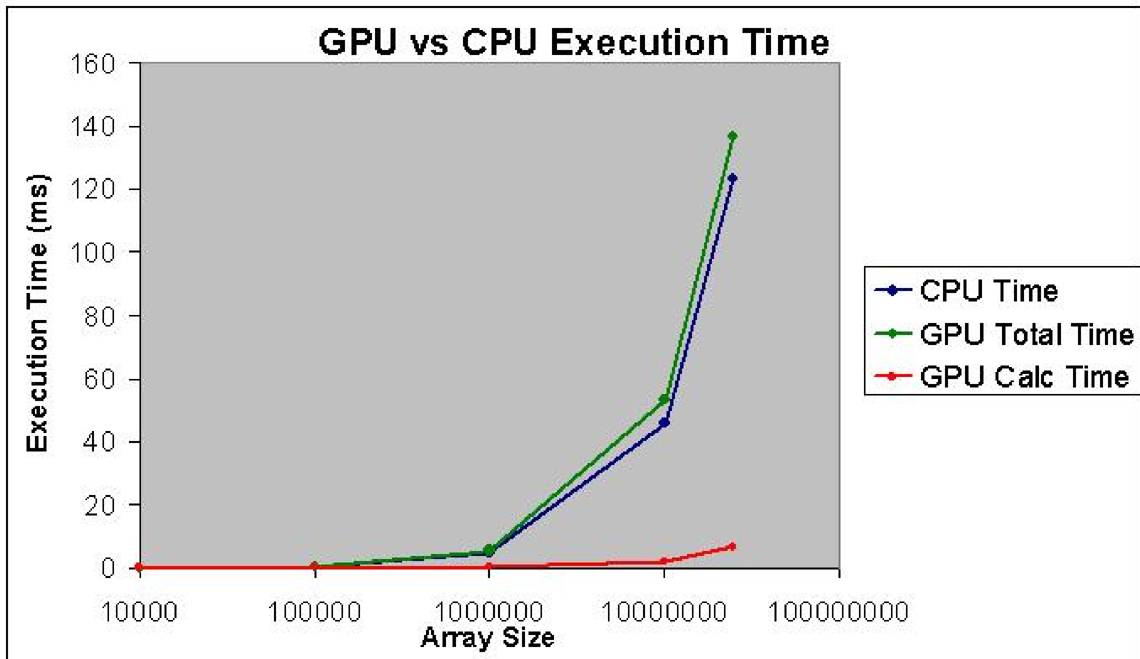

Figure 4: GPU computation-only time far faster than that of CPU (Source: Steven Cook)

Comparing the calculation-only GPU time to the CPU time shows that while they start nearly even, with the CPU slightly faster, the difference grows dramatically as array

size increases. Clearly, if more work was needed to be done on the data in the loop, the GPU would likely prove superior. As a simple test of this, I modified the program to run the calculation 100 times instead of once for an array size of 1,000,000. The CPU took 458.5 milliseconds, while the GPU's total time, including loading and reading, was 22.5 milliseconds. This extremely telling result indicates the potential of the GPGPU as well as opens up many further fields of study. Result data is listed in Appendix E, page 50.

GPU VS CPU: CONCLUSIONS

The project's goal of furthering multicore development by identifying bottlenecks was a success. For applications light in calculation but heavy in data transfer, such as this one, the GPU is badly limited by the memory transfer rate and should thus be avoided. Improving the latency and throughput of this connection in future designs will make the GPU more viable for these types of simulations as a whole, and NeuroJet in particular.

The results obtained are considered accurate due to the simplicity and accuracy of the timing methodology employed. Nonetheless, they only represent one program on one computer for a potentially nonoptimal GPU-side implementation. Varying these parameters over a very large number of repetitions would greatly strengthen and hone results concerning GPU viability for this particular program.

Many interesting areas of further research are suggested by the results obtained. First, the implementation I used to calculate the loop on the GPU might be able to be improved. It contained a number of conditional branches, which could prove expensive under further analysis. Additionally, the launch configuration might also be reconfigured

for greater speed, perhaps by setting 256 threads in a block instead of 512 and increasing the number of blocks. Studying how to achieve efficiency in loop computations under CUDA would benefit this and other applications.

Having only tested one function to see if an improvement might be brought about, the GPGPU could also prove useful in accelerating the other functions identified when analyzing NeuroJet. Passing the code I wrote to NeuroJet's author will give him an example to work from should he want to try to optimize NeuroJet or its successors for the GPGPU. The CUDA code I wrote has yet to be integrated into the actual NeuroJet program, and when resources allow, this would permit further studies done with actual program data instead of the simulated data used here.

LESSONS LEARNED FOR CUDA PROGRAMMING

For the GPGPU to be useful to a program such as NeuroJet, it will be necessary to make changes to the programming model. As testing has indicated, the GPU can only outperform the CPU when the calculation load is heavy enough – more precisely, when there is enough of a workload for the GPU's calculation speed advantage to eclipse the GPU's data transfer costs. Therefore, if NeuroJet could be restructured to decrease the size and frequency of data transfers with the GPU, it could be possible to match or even beat the CPU calculation times. One possible approach to minimize the data transfer delays would be to do more of the work on the data sets transferred to the card. This project only moved one loop over to the GPU; moving a greater portion of the calculations, including parent loops, would both increase the amount of data to be fed to

the GPU and decrease the total number of data transfers that needed to occur. These same lessons can be applied to other attempts to write (or re-write) programs for CUDA.

The first step for a CUDA programmer is to answer the question of whether the workload required is of a sufficient size and parallelizable structure so as to warrant the use of the GPU. Answering this question requires knowledge of both the volume of data employed in the calculation and the specific algorithm(s) to be used. The goal of the CUDA programmer here is to devise a program structure such that the number and size of data transfers to and from the GPU is kept to a minimum, which can be accomplished through doing as much of the calculation work as possible on the GPU while keeping the data resident on the GPU for as long as possible. Certain types of applications may not be suited to running in parallel, and may thus be unsuitable for CUDA implementation.

Once the most CUDA-friendly program structure possible has been defined, it needs to be seen whether it will run faster (or if necessary, significantly faster) on the GPU. To help determine if the GPU time will be less than the CPU time, the following formula must be true:

# of transfers * time cost per transfer + volume of computation / GPU speed
< volume of computation / CPU speed

Here the GPU time is the left half of the inequality, and the CPU time is on the right. An example from this project would be a test run on an array size of 1 million floats, looping 100 times. There is only one complete transfer to-and-from the card, with a time cost of 5.17ms. The time to do one computation was .47ms, so to do the computation 100 times would likely be no more than 100*.47ms, or 47ms. Adding these

together gives a predicted time of 52.17ms for the GPU. On the CPU side of the inequality, it took 4.6ms to do one calculation. Therefore, we would expect 100 calculations to take roughly 100*4.6ms = 460 ms. This is ten times as long as the predicted GPU time. If a CUDA programmer knew this information, he would be able to see that if a calculation needed to be done 100 times it would be far more efficient on the GPU, and the programmer could expect a break-even point between CPU and GPU around 10 calculations. As a final note, when this 100-loop experiment was actually run, the GPU time was 27.6ms while the CPU time was 458.5ms. The CPU time scaled very linearly, while the GPU time was only a fraction of what was predicted. This further illustrates the need for GPU testing, as its performance does not necessarily scale as predictably as the CPU's.

Depending on the application, calculating this formula may be straightforward or highly complex. Nevertheless, the programmer should keep the basic idea of this formula in mind if for no other reason than to be able to tell if modifications to the program will dramatically affect GPU performance.

A final word of advice to CUDA or any GPGPU programmers is to experiment on the card early and often. This will help tremendously in both debugging and gaining an idea of the timings involved. Early measurements of transfer times can help spot intractable speed issues and save time, and calculation time measurements can show what the GPGPU is best at and by how much. From my experience on this project I recommend that GPGPU programming be done in iterative cycles, with new code being tested out on the GPGPU on a regular basis.

THE FUTURE OF GPGPU'S

Looking to the future, there are signs to be watched for to ensure that this technology is moving in a socially beneficial direction. As GPGPU's like the Nvidia GeForce 8800 make their way into the mainstream, it will be possible for end users to turn their computational power towards their own devices. I am concerned that this may give hackers an extremely powerful tool with which they could use to brute-force passwords and other security protocols. It would be worthwhile for experiments to be done to determine just how much trouble and end user could cause with a few of these cards, and if necessary whether encryption routines need to be strengthened as a result.

At the same time, I feel that these GPGPU's represent a potentially historic milestone. As long as they cost in the hundreds of dollars and the API, CUDA, is publicly available, it will be possible for anyone to purchase one of these cards and effectively have a supercomputer at their disposal. This may one day be looked upon as a development similar to the printing press and personal computer in the way they enable people to process information in new and innovative ways. To fully reap the potential benefits of having millions of users doing research on their own, it is imperative that GPGPU's both remain affordable and retain publicly available programming tools.

In conclusion, I deem this project a success due to the extremely interesting GPU behaviors that were uncovered. While it was not possible to accelerate the neural simulation using the graphics card at this time, a path has been laid out for future research to allow this and other similar simulations to benefit from GPGPU development in the future.

WORKS CITED

Acceleware Corporation. Financial Update: Acceleware Announces $2,925,000 Investment by NVIDIA. January 22, 2007. Retrieved March 8, 2007 from <http://www.hpcwire.com/hpc/1214993.html>.

Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D. (1993). Automatic Program Parallelization. *Proceedings of the IEEE*, 81, 211 - 243.

Bruer, J.T. (1998). The brain and child development: Time for some critical thinking. Public Health Rep., 113, 388-397.

Burgess, N., Donnet, J., & O'Keefe, J. (1998). The Hippocampus and Navigation in Rats, Robots, and Humans. *Self-Learning Robots II: Bio-robotics (Digest No. 1998/248)*. London : IEE.

Day, M., Zhongfeng, W., Jun, D., et al. (Feb 2006). Selective elimination of glutamatergic synapses on striatopallidal neurons in Parkinson disease models. *Nature Neuroscience*, 9(2), 251-259.

Denham, M.J., & McCabe, S.L. (1998). A model of predictive learning in the rat hippocampal principal cells during spatial activity. *The 1998 IEEE International Joint Conference on Neural Networks Proceedings*, 2, 1547 - 1552.

Eichenberger, A.E., O'Brien, Kathryn, O'Brien, Kevin, et. Al. (2005). Optimizing Compiler for a CELL Processor. *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, New York: IEEE.

Fogel, D.B., Fogel, L,J. (Nov/Dec 1995). Evolution and Computational Intelligence. IEEE International Conference on Neural Networks, 4, 1938-1941.

Gardner, J Scott. Multicore Everywhere: The x86 and Beyond. ExtremeTech. January 9, 2006. Retrieved March 8, 2007 from <http://www.extremetech.com/article2/0,1697,1909147,00.asp>.

Goodwins, Rupert. *The Big Interview: Pat Gelsinger*. February 26 2007. Retrieved March 8, 2007 from <http://news.zdnet.co.uk/emergingtech/0,1000000183,39286049-1,00.htm?r=118>.

Green, Simon and Mark Harris. *Havok FX Physics on Nvidia GPU's*. March 31, 2006. Retrieved March 8, 2007 from <http://developer.nvidia.com/object/havok-fx-gdc-2006.html>.

Izhikevich, E.M. (1999). Class 1 Neural Excitability, Conventional Synapses, Weakly Connected Networks, and Mathematical Foundations of Pulse-Coupled Models. *IEEE Transactions on Neural Networks*, 10(3), 499 – 507.

Joy, B. (April 2000). Why the Future Doesn't Need Us [Electronic Version]. *Wired* Magazine, 8.

Levy, W., Hocking A.B., & Wu, X. (2005). Interpreting Hippocampal Function as Recoding and Forecasting. *Neural Networks 18*, 1242 - 1264.

Macedonia, Michael. The GPU Enters Computing's Mainstream. (2003). *Computer*, vol. 36, no. 10. 106-108.

Myers, Ann L. (2000). Memory Loss and the Brain (image). Retrieved March 26, 2007 from http://www.memorylossonline.com/glossary/hippocampus.html>.

Nussel, P., & Truett, R. (September 2006). Go inside the factory of the future; In coming decades plants will have more computers and fewer (but smarter) people. *Automotive News Europe*, Technology section, 14.

Perumalla, K.S. (2006). Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). 20th Workshop on Principles of Advanced and Distributed Simulation, 74- 81.

Port, O. (1995). Computers that think are almost here: The ultimate goal of artificial intelligence, human-like reasoning, is within reach. *Business Week*, 68(4) , 68-70.

Ramanathan, R M. *Intel Multi-Core processors Leading the next Digital Revolution*

(white paper). Retrieved 12, Mar 2007 from <ftp://download.intel.com/technology/computing/multi-core/multi-core-revolution.pdf>.

Rattner, Justin. *Intel IDF – Justin Rattner Keynote Presentation*. March 7, 2006. Retrieved March 8, 2007 from <http://www.intel.com/pressroom/kits/events/idfspr_2006/20060307_RattnerTranscript.pdf>.

Stokes, Jon. *What Counts as "multicore?"* Ars Technica. January 9, 2006. Retrieved March 8, 2007 from <http://arstechnica.com/news.ars/post/20060109-5937.html>.

Traub, R.D., Miles, R., Wong, R.K.S. (1988). Large scale simulations of the hippocampus. *Engineering in Medicine and Biology Magazine, IEEE* , 7, 31-38.

Tsukada, M. (1993). A theoretical model of the hippocampal-cortical memory system motivated by physiological functions in the hippocampus. *Proceedings of 1993 International Joint Conference on Neural Networks, IJCNN '93-Nagoya,* 2, 1120-1123.

Vogt, A., Lauer, L., & Offenhausser, A. (2002). Controlled outgrowth and synapse

formation of rat brain neurons by microcontact printing. *Proceedings of the IEEE-EMBS Special Topic Conference on Molecular, Cellular and Tissue Engineering,* 106 - 107.

Williams, S., Shalf, J., Oliker, L., et al. (2006). *The Potential of the Cell Processor for Scientific Computing*. California: Computational Research Division, Lawrence Berkeley National Lab. Retrieved September 12, 2006 from Berkeley Computer Science Department <http://www.cs.berkeley.edu/~samw/projects/cell/CF06.pdf>.

BIBLIOGRAPHY

Acceleware Corporation. Financial Update: Acceleware Announces $2,925,000 Investment by NVIDIA. January 22, 2007. Retrieved March 8, 2007 from <http://www.hpcwire.com/hpc/1214993.html>.

Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D. (1993). Automatic Program Parallelization. *Proceedings of the IEEE*, 81, 211 - 243.

Bronzino, J.D. (1990). *Medical Technology and Society: An Interdisciplinary Perspective*. Boston : MIT Press.

Bruer, J.T. (1998). The brain and child development: Time for some critical thinking. *Public Health Rep.*, 113, 388-397.

Burgess, N., Donnet, J., & O'Keefe, J. (1998). The Hippocampus and Navigation in Rats, Robots, and Humans. *Self-Learning Robots II: Bio-robotics (Digest No. 1998/248)*. London : IEE.

Day, M., Zhongfeng, W., Jun, D., et al. (Feb 2006). Selective elimination of glutamatergic synapses on striatopallidal neurons in Parkinson disease models. *Nature Neuroscience*, 9(2), 251-259.

Denham, M.J., & McCabe, S.L. (1998). A model of predictive learning in the rat hippocampal principal cells during spatial activity. *The 1998 IEEE International Joint Conference on Neural Networks Proceedings*, 2, 1547 - 1552.

De Waal, F.B.M. (1999). The end of nature versus nurture. *Scientific American*, 281, 94-99.

Eichenberger, A.E., O'Brien, Kathryn, O'Brien, Kevin, et. Al. (2005). Optimizing Compiler for a CELL Processor. *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, New York: IEEE.

Fogel, D.B., Fogel, L,J. (Nov/Dec 1995). Evolution and Computational Intelligence. *IEEE International Conference on Neural Networks*, 4, 1938-1941.

Gardner, J Scott. Multicore Everywhere: The x86 and Beyond. ExtremeTech. January 9, 2006. Retrieved March 8, 2007 from <http://www.extremetech.com/article2/0,1697,1909147,00.asp>.

Goodwins, Rupert. *The Big Interview: Pat Gelsinger*. February 26 2007. Retrieved March 8, 2007 from <http://news.zdnet.co.uk/emergingtech/0,1000000183,39286049-1,00.htm?r=118>.

Green, Simon and Mark Harris. *Havok FX Physics on Nvidia GPU's*. March 31, 2006. Retrieved March 8, 2007 from <http://developer.nvidia.com/object/havok-fx-gdc-2006.html>.

Gschwind, M.., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., & Yamazaki, T. (2006). Synergistic Processing in Cell's Multicore Architecture. *IEEE MICRO*, 26, 10 - 24.

Herzfeld, N. (June 2002). Creating in Our Own Image: Artificial Intelligence and the Image of God. Zygon Journal of Religion & Science, 37(2), 303 - 303.

Izhikevich, E.M. (1999). Class 1 Neural Excitability, Conventional Synapses, Weakly Connected Networks, and Mathematical Foundations of Pulse-Coupled Models. *IEEE Transactions on Neural Networks*, 10(3), 499 – 507.

Joy, B. (April 2000). Why the Future Doesn't Need Us [Electronic Version]. *Wired Magazine*, 8.

Kim, K., & Nicolau, A. (1991). Parallelizing tightly nested loops. *Parallel Processing Symposium, 1991. Proceedings., Fifth International*, 630-633.

Koch, Christoph (1999). *The Biophysics of Computation*. New York: Oxford University Press.

Kumar, R., Zyuban, V., & Tullsen, D.M. (2005). Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. *Proceedings, 32nd International Symposium on Computer Architecture,* 408 - 419.

Levy, W., Hocking A.B., & Wu, X. (2005). Interpreting Hippocampal Function as Recoding and Forecasting. *Neural Networks 18*, 1242 - 1264.

Macedonia, Michael. The GPU Enters Computing's Mainstream. (2003). *Computer*, vol. 36, no. 10. 106-108.

McKinney, J. (1992). *Microwave Harassment and Mind-Control Experimentation.* Silver Spring, MD : Association of National Security Alumni, Electronic Surveillance Project.

Montrym, J., & Moreton, H. (2005). The GeForce 6800. *IEEE Micro*, 25, 41 - 51.

Myers, Ann L. (2000). Memory Loss and the Brain (image). Retrieved March 26, 2007 from http://www.memorylossonline.com/glossary/hippocampus.html>.

Nishiyama, T., Ikeda, M., Iwata,N., et al. (2005). Haplotype association between $GABA_A$ receptor sγ2 subunit gene (GABRG2) and methamphetamine use disorder. *Pharmacogenomics Journal*, 5 , 89 - 95.

Nussel, P., & Truett, R. (September 2006). Go inside the factory of the future; In coming decades plants will have more computers and fewer (but smarter) people. *Automotive News Europe*, Technology section, 14.

Owens, J.D., Luebke, D., Govindaraju, N., et. al. (2005). A Survey of General-Purpose Computation on Graphics Hardware. *Eurographics 2005, State of the Art Reports*, 21-51.

Perumalla, K.S. (2006). Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). *20th Workshop on Principles of Advanced and Distributed Simulation,* 74- 81.

Port, O. (1995). Computers that think are almost here: The ultimate goal of artificial intelligence, human-like reasoning, is within reach. *Business Week*, 68(4) , 68-70.

Ramanathan, R M. *Intel Multi-Core processors Leading the next Digital Revolution* (white paper). Retrieved 12, Mar 2007 from <ftp://download.intel.com/technology/computing/multi-core/multi-core-revolution.pdf>.

Rattner, Justin. *Intel IDF – Justin Rattner Keynote Presentation*. March 7, 2006. Retrieved March 8, 2007 from <http://www.intel.com/pressroom/kits/events/idfspr_2006/20060307_RattnerTranscript.pdf>.

Schmid, G. (2001). Report on the Existence of a Global System for the Interception of Private and Commercial Communications (ECHELON Interception System) Pt. 1, Motion for a Resolution , Explanatory Statement. *Temporary Committee on the ECHELON Interception System*. Retrieved from <http://www.waronterrorismwatch.ca/Echelon_Interception_System.pdf>.

Schwartz, H. (1969). The Legitimation of Electronic Eavesdropping: The Politics of "Law and Order." *Mich Law Rev*, 67, 455-510.

Stokes, Jon. *What Counts as "multicore?"* Ars Technica. January 9, 2006. Retrieved March 8, 2007 from <http://arstechnica.com/news.ars/post/20060109-5937.html>.

Strom, David. *5 Disruptive Technologies to Watch in 2007*. January 1, 2007. InformationWeek. Retrieved March 8, 2007 from <http://www.informationweek.com/internet/showArticle.jhtml?articleID=196800208&pgno=3&queryText=>.

Traub, R.D., Miles, R., Wong, R.K.S. (1988). Large scale simulations of the hippocampus. *Engineering in Medicine and Biology Magazine, IEEE* , 7, 31-38.

Tsukada, M. (1993). A theoretical model of the hippocampal-cortical memory system motivated by physiological functions in the hippocampus. *Proceedings of 1993 International Joint Conference on Neural Networks, IJCNN '93-Nagoya,* 2, 1120-1123.

United States. Senate (1977). *Project MKULTRA, the CIA's Program of Research in Behavioral Modification: Joint Hearing, August 3, 1977, before the Select Committee on Intelligence and the Subcommittee on Health and Scientific Research of the Committee on Human Resources.* Testimony of Senator Inouye.

Vogt, A., Lauer, L., & Offenhausser, A. (2002). Controlled outgrowth and synapse formation of rat brain neurons by microcontact printing. *Proceedings of the IEEE-EMBS Special Topic Conference on Molecular, Cellular and Tissue Engineering,* 106 - 107.

Williams, S., Shalf, J., Oliker, L., et al. (2006). *The Potential of the Cell Processor for Scientific Computing*. California: Computational Research Division, Lawrence Berkeley National Lab. Retrieved September 12, 2006 from Berkeley Computer Science Department <http://www.cs.berkeley.edu/~samw/projects/cell/CF06.pdf>.

APPENDIX A: TEST MACHINE SPECS

Generated by running "dxdiag" in Windows, relevant output displayed:

```
------------------
System Information
------------------
Time of this report: 3/24/2007, 17:23:40
       Machine name: SKADRONDELL4
   Operating System: Windows XP Professional (5.1, Build 2600) Service Pack 2
(2600.xpsp.061012-0254)
           Language: English (Regional Setting: English)
System Manufacturer: Dell Inc.
       System Model: Dell DXG061
               BIOS: Phoenix ROM BIOS PLUS Version 1.10 1.1.3
          Processor: Intel(R) Core(TM)2 CPU          6300  @ 1.86GHz (2 CPUs)
             Memory: 1022MB RAM
          Page File: 440MB used, 2019MB available
        Windows Dir: C:\WINDOWS
    DirectX Version: DirectX 9.0c (4.09.0000.0904)
DX Setup Parameters: Not found
     DxDiag Version: 5.03.2600.2180 32bit Unicode


---------------
Display Devices
---------------
        Card name: NVIDIA GeForce 8800 GTX
     Manufacturer: NVIDIA
        Chip type: GeForce 8800 GTX
         DAC type: Integrated RAMDAC
       Device Key: Enum\PCI\VEN_10DE&DEV_0191&SUBSYS_039C10DE&REV_A2
   Display Memory: 768.0 MB
     Current Mode: 1280 x 1024 (32 bit) (75Hz)
          Monitor: Plug and Play Monitor
  Monitor Max Res: 1600,1200
      Driver Name: nv4_disp.dll
   Driver Version: 6.14.0010.9773 (English)
      DDI Version: 9 (or higher)
Driver Attributes: Final Retail
```

APPENDIX B: SELECTED NEUROJET FUNCTION CODE

“Inner loop” portion to be re-written is in green surrounded by /* and */. The section

```
if (useDendInh)
               dendExc[i] /= (dMult * numerator + BaseInhib);
```

was ommitted from the re-write as its useDendInh was not turned on in the NeuroJet version being used.

```
void CalcDendriticExcitation() {
   const float dMult = SystemVar::GetFloatVar("DenomMult");
   dendExc = DataList(ni, 0.0L);
   for (PopulationCIt PCIt = Population::Member.begin();
        PCIt != Population::Member.end(); ++PCIt) {
      const double FeedBackExcToInternrn = PCIt-
>getFeedbackInhibition();
      const double FeedFwdExcToInternrn = PCIt-
>getFeedforwardInhibition();
      const float BaseInhib = (KFBDend * FeedBackExcToInternrn) +
                              (KFFDend * FeedFwdExcToInternrn) +
K0Dend;
   /*   for (unsigned int i = PCIt->getFirstNeuron(); i <= PCIt-
>getLastNeuron(); ++i) {
         const float numerator = sumwz[i];
         if (abs(numerator) > verySmallFloat) {
            dendExc[i] = numerator;
            if (useDendInh)
               dendExc[i] /= (dMult * numerator + BaseInhib);
         }
      }*/

   }
   enqueueDendriticResponse(dendExc, sumwz_inhdiv, sumwz_inhsub);
}
```

APPENDIX C: PROTOTYPE SOURCE CODE

```
/*******************************************************
main.cpp: goal of this program is to demonstrate ability
to offload a simple series sum calculation to GPU and perform
calculation there

Steve Cook
*******************************************************/

#include <stdio.h>
#include <iostream.h>

// includes, project
#include <cutil.h>

// includes, kernels
//#include "main_kernel.cu"
using namespace std;
extern "C" void factorial_add_local(int j, int *result);
extern "C" int factorial_add_CUDA(int j, int* result);
extern int numArray[1000];

int main()
{
      int x = 0;
      int CPU_res = 0;
      int CUDA_res = 0;
      int *CPU_result_ptr = &CPU_res;
      int *CUDA_result_ptr = &CUDA_res;


      std::cout << "Program Initialized.\n";
      std::cout << "enter a number X less than 1000 ";
      std::cin >> x;
      // check bounds
      if((x >= 1000) || (x < 0)){
            std::cout << "x out of range, program exiting.";
            return 1;
      }

      std::cout << "calculating sum from 1 to " << x << ": ";

      // initialize numArray
      for(int k = 0; k < 1000; k++){
            numArray[k] = k;
      }
      // calculate result on CPU
      factorial_add_local(x, CPU_result_ptr);
      // calculate result on GPU
      factorial_add_CUDA(x, CUDA_result_ptr);

      std::cout << "\nResult from CPU: " << CPU_res;
      std::cout << "\nResult from GPU: " << CUDA_res;
      return 0;
}
```

```
#ifndef _MAIN_KERNEL_H_
#define _MAIN_KERNEL_H_
#include <cutil.h>
#include <stdio.h>

int numArray[1000];

__global__ void CUDA_func(int* arrayOnDevice, int* answerOnDevice){

       // want to get the value in the array at position x. Is this how
we do that?

       *answerOnDevice += arrayOnDevice[threadIdx.x+1];
//       *answerOnDevice = 1000;
             __syncthreads();
}
// indexing method?
extern "C" int factorial_add_CUDA(int j, int *result){
      int BSIZE = j;

      // Load numArray onto CUDA
      int arraySize = 1000 * sizeof(int);
      int* arrayOnDevice;
      cudaMalloc((void**)&arrayOnDevice, arraySize);
      cudaMemcpy(arrayOnDevice, numArray, arraySize,
cudaMemcpyHostToDevice);

      // allocate space for result

      int* answerOnDevice;
      int answerSize = sizeof(int);
      cudaMalloc((void**)&answerOnDevice,answerSize);

      dim3 dimBlock(BSIZE, 1,1);

      // QUESTION
      // do we want to do something about j divided by 768?
      dim3 dimGrid(1,1);      // temporary fix only!

      CUDA_func<<<dimGrid, dimBlock>>>(arrayOnDevice, answerOnDevice);

      // copy computed answer back to host
      cudaMemcpy(result, answerOnDevice, answerSize,
cudaMemcpyDeviceToHost);
       printf("%d\n",*result);
      return 0;
}


extern "C" void factorial_add_local(int j, int* result){
      for(int x = 0; x <= j; x++){
            *result += numArray[x];
      }
//    return result;
}

#endif // #ifndef _TEMPLATE_KERNEL_H_
```

APPENDIX D: FINAL IMPLEMENTATION SOURCE CODE

```
/********************************************************
thesis.cu

Steve Cook
*******************************************************/

#include <stdio.h>
#include <cutil.h>
//#include <stdlib.h>
#include <math.h>

#include <thesis_kernel.cu> //remove for linux?

extern "C" float CalcDendritic_CUDA(int firstNeuron, int lastNeuron,
int sumwzSize, float* sumwzPtr, float verySmallFloat, int dendExcSize,
float *dendExcPtr, int overhead_mode);
//extern "C" void CalcDendritic_CPU(int firstNeuron, int lastNeuron,
float* sumwzPtr, float verySmallFloat, float dendExcPtr);

const int loopSize = 1000000; // change this to vary num of parameters
used
float sumwzArray[loopSize];
float dendExcArray[loopSize];

int main(/*int argc, char *argv[]*/)
{
    float verySmallFloat = .00000001f;
      int firstNeuron = 0;
      int lastNeuron  = loopSize;
      //float[] sumwzArray;
      float *sumwzPtr  = sumwzArray;   // do I want the & there?
      float *dendExcPtr = dendExcArray;

      // initialize numArray
      for(int k = 0; k < loopSize; k++){
            sumwzArray[k] = 1 * (k % 2); // alternate 0/1
            dendExcArray[k] = 0;
      }

      // work section
      // calculate result on CPU
      unsigned int timer = 0;
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));

      for(int x = 0; x < 100; x++){
        for(int i = firstNeuron; i <= lastNeuron; i++){
                  const float numerator = sumwzArray[i];
                  if(fabs(numerator) > verySmallFloat) {
                        dendExcArray[i] = numerator;
                  }
            }
      }
```

```
            CUT_SAFE_CALL(cutStopTimer(timer));
            float CPU_time = cutGetTimerValue(timer);
            printf("on CPU: Processing time: %f (ms)\n",
            cutGetTimerValue(timer));
            CUT_SAFE_CALL(cutDeleteTimer(timer));

       // calculate result on GPU
            float fullTime = CalcDendritic_CUDA(firstNeuron,
lastNeuron, loopSize, sumwzPtr, verySmallFloat,loopSize , dendExcPtr,
0);
            float overheadTime = CalcDendritic_CUDA(firstNeuron,
lastNeuron, loopSize, sumwzPtr, verySmallFloat,loopSize , dendExcPtr,
1);

            printf("GPU card time(diff): %f (ms)\n", fullTime-
overheadTime);
            printf("GPU-CPU card time(diff): %f (ms)\n", fullTime-
overheadTime-CPU_time);
      //debugging output
/*    for(int k = loopSize - 1000; k < loopSize; k++){
          printf("%f ", dendExcArray[k]);
      }*/

      return 0;
}
```

```
#ifndef _thesis_kernel_H_
#define _thesis_kernel_H_
#include <cutil.h>
#include <stdio.h>

// extern const int loopSize;
extern float sumwzArray[];
extern float dendExcArray[];


__global__ void CUDA_func(float* sumwzOnDevice, float* dendExcOnDevice,
float verySmallFloat, int lastBlock, int lastThread){

	if(blockIdx.x < lastBlock - 1){
		__device__ __local__ float LocalVar =
sumwzOnDevice[(blockIdx.x * 512) + threadIdx.x];
		if(abs(LocalVar) > verySmallFloat){
			dendExcOnDevice[(blockIdx.x * 512) + threadIdx.x] =
LocalVar;
		}
	}
	else {	// in last block
		if(threadIdx.x < lastThread){
			__device__ __local__ float LocalVar =
sumwzOnDevice[(blockIdx.x * 512) + threadIdx.x];
			if(abs(LocalVar) > verySmallFloat){
				dendExcOnDevice[(blockIdx.x * 512) +
threadIdx.x] = LocalVar;
			}
		}
	}

}

extern "C" float CalcDendritic_CUDA(int firstNeuron, int lastNeuron,
int sumwzSize, float* sumwzPtr, float verySmallFloat, int dendExcSize,
float *dendExcPtr, int overhead_mode){

	// find launch configuration
	int numRuns = lastNeuron - firstNeuron;
	int BSIZE = 512; // most efficient block size?
	double b = 512;
	double n = lastNeuron - firstNeuron;
	double gs2 = ceil(n / b);
	int test = (int)gs2;

	int gridSize = /*ceil(double(numRuns / BSIZE))*/ test;
	int final_thread_num = numRuns % BSIZE;

	if(final_thread_num == 0) final_thread_num = 512; // do all
threads in end block if true
	dim3 dimGrid(gridSize,1);

	// Load array onto CUDA

	int sumwzMemSize = sumwzSize * sizeof(float);
	float* sumwzOnDevice;
```

```
	cudaMalloc((void**)&sumwzOnDevice, sumwzMemSize);
	cudaMemcpy(sumwzOnDevice, sumwzPtr, sumwzMemSize,
cudaMemcpyHostToDevice);


	// allocate space for result

	int dendExcMemSize = dendExcSize * sizeof(float);
	float* dendExcOnDevice;


	cudaMalloc((void**)&dendExcOnDevice,dendExcMemSize);

    unsigned int timer = 0;
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));

	//for(int k=0;k<100;k++)
	cudaMemcpy(dendExcOnDevice, dendExcPtr, dendExcMemSize,
cudaMemcpyHostToDevice);

	dim3 dimBlock(BSIZE, 1,1); // need math to calc right block size

	if(!overhead_mode){
		for (int x = 0; x < 100; x++)
	CUDA_func<<<dimGrid, dimBlock>>>(sumwzOnDevice, dendExcOnDevice,
verySmallFloat, gridSize, final_thread_num);

	}

	// copy computed answer back to host
	cudaMemcpy(dendExcPtr, dendExcOnDevice, dendExcMemSize,
cudaMemcpyDeviceToHost);
	CUT_SAFE_CALL(cutStopTimer(timer));
	float time_elapsed = cutGetTimerValue(timer);
	if(overhead_mode){
		printf("Processing time on GPU (overhead only): %f (ms)\n",
cutGetTimerValue(timer));
	}
	else{
		printf("Procesing time on GPU: %f (ms)\n",
cutGetTimerValue(timer));
	}
    CUT_SAFE_CALL(cutDeleteTimer(timer));
	return time_elapsed; // what goes here?
}

#endif // #ifndef _thesis_kernel_H_
```

APPENDIX E: TABLE OF RESULTS

- All times are in milliseconds (ms)

Array Size	CPU Total Time	GPU Total Time	GPU Calculation-only Time	GPU Time Percent Overhead
10,000	0.044871	0.134632	0.049875	62.95
100,000	0.444841	0.627043	0.062051	90.10
1,000,000	4.623071	5.645345	0.468405	91.70
10,000,000	45.928864	53.579575	2.316719	95.68
25,000,000	123.382126	136.620880	6.907059	94.95