Implementing Neural Networks on GPU Hardware

Nate Gauntt
University of New Mexico
Dept. of Computer Science
Email : negaunt@cs.unm.edu

September 20, 2010

***Abstract* - Modern graphics processing units (GPUs) are cheap, ubiquitous, and increasing in performance at a rate two to three times that of traditional CPU growth. As GPUs evolve in function and become more programmable each generation, they are increasingly put to use in various high throughput parallel calculations, such as solving fluid flow equations or simulating plate tectonics. In this paper, we discuss the recent trend in implementing Neural Network architectures on the GPU, allowing in some cases real time input processing, and in other cases dramatic speedup of traditional algorithms.**

Keywords: Neural Networks, GPU, parallel processing, biological neuron, Cg, CUDA, BrookGPU

Introduction

In the early 1940's and throughout the 50's, partially as a result of American and British codebreaking activities, the theory of computation became an area of active research in the scientific community. A 1943 paper [McCullogh/Pitts 43] establishes a formal system that mathematically describes the behavior of biological neurons and networks of neurons. Using this system, they show that, with sufficient neurons and proper connections, any computable function can be represented by such a network of neurons. This work remains largely theoretical until serious work starts in the 60's attempting to model intelligence with computers. Early AI researchers in [Minsky/Papert 69] explain the serious mathematical limitations of a single artificial neuron, including it's inability to learn the trivial XOR boolean function. This symbolist critique cast doubt on the efficacy of using neural networks (NNs) for more than a decade, until an efficient method of error feedback learning called back-propagation was discovered [Davis 01].

Connectionist methodology flourished in the 80's and early 90's after this method was discovered, and modern graphics hardware provides a similar opportunity, allowing neural networks to be applied easily and cheaply to certain real-time problems and other throughput-sensitive domains. It is only within the last several years that GPUs could store 32-bit numerical results efficiently to textures (GeForce 8, 2006), could be programmed by high level languages such as Cg (2002), or have had the tool support to be used together in robust high-performance architectures like CUDA (2006) [4, 1, Halfhill 08]. Section 1 discusses economic advantages of GPU clusters, section 2 discusses GPU neural network implementations, and section 3 concludes. In a companion paper, 'Visualizing Neural Networks', we survey visualizing the often non-linear dynamics of various network architectures.

1 GPU Economic Advantages

Recently, much research has been devoted to middleware that coordinates the operation of many GPUs to perform large parallel calculations. This software has been used in industry to make high performance 'cluster of clusters', using commodity game consoles as cost-effective supercomputers. Little success was achieved by networks of the first-generation XBox [3], though the University of Illinois - Urbana Champaign developed a productive 64 node Playstation 2 cluster for $15,000. Using data from [Levine 09], this translates into $39/GFLOP using Playstation GPUs, as opposed to the estimated price of $312/GFLOP using Pentium4-3200 CPUs. Thus, in 2005, there was already an order of magnitude price difference per GFLOP in using GPUs vs CPUs for some cluster applications. Meuth summarizes the FLOPS per dollar ratio of modern CPUs vs various console generations of GPUs over time in [Meuth 07].

2 Neural Network Implementations

Neural Network implementations on the GPU have evolved with the changing state of the underlying hardware. One of the earlier implementations implements Kohonen feature mapping on an SGI workstation [Bohn 98]. Up to a 5 fold speed increase over CPU was possible using SGI's accelerated implementation of key OpenGL functions like glColorMatrix, glminmax, and various texture blending functions. Given the minor programmability of hardware at the time, Bohn relies heavily on such functions, tying the

implementation to vendor support of a specific API running on custom hardware. Modern OpenGL hardware largely sacrifices accelerated 2D functions for better 3D performance, thus justifying research into more general GPU techniques.

In the early 2000's, much effort was spent investigating increasingly programmable GPUs and their application to scientific computing [Larsen 01, Kruger 03]. Fast linear algebra for graphics on the GPU required fast matrix multiply implementations, leading to several neural network GPU implementations utilizing matrix multiply routines. One such implementation uses a blend of DirectX and hand-tuned GPU assembly code [Rolfes 04]. Unlike Bohn, Rolfes relies less on accelerated DirectX calls, relying instead on fast implementation of certain GPU instructions and hand-tuned assembly code. The implementation is still dependent on specific hardware, but at a lower level than the graphics API, thus arguably less likely to change as basic GPU instructions probably will always need to be accelerated. The main contribution of this implementation technique is to accelerate multi-layer perceptron (MLP) neural networks on commodity graphics hardware. For another matrix-based computation of MLP, see [Kyong-Su 04], used for real-time text detection in a video stream. Kyoung-Su achieves a 20-fold performance increase over an equivalent CPU implementation.

As the language of high level shader programming became standard and GPU compiler optimization improved, implementers began moving to higher level implementations of neural networks. Both self organizing map (SOM) and MLP neural networks were implemented in the shader language Cg in [Zhongwen 05]. Despite the high level language, some knowledge of GPU instructions is needed to construct efficient algorithms. For example, to find the minimum value (or best match) in the SOM, Zhongwen uses a recursive algorithm dividing the activation space into four quadrants at each step, until only scalar 2x2 matrices are left. This may seem puzzling without the knowledge that most GPUs have an efficient **min(v1,v2)** function that computes the minimum of two 4-vectors, which can be used to find the minimum values of two 2x2 matrices quickly. Zhongwen achieves a 2 to 4 fold performance advantage in SOM over the CPU, and 200 fold performance advantage in MLP over a CPU implementation. Such performance claims must be carefully evaluated; in this case no CPU code is given and it is not said whether the CPU implementation is optimized. That said, Zhongwen's MLP was able to compute an object tracking function in 46 ms, which they note is theoretically fast enough to be useful for real-time response. Zhongwen also notes several pitfalls of high level language implementations. In one case the same Cg code yielded correct results on an ATI card and subtly incorrect results on an NVidia card, and code changes were needed to work around a probable hardware defect. Also, the performance of GPU code is very sensitive to the number of rendering passes and amount of data transfer to the CPU, which they minimized where possible. Finally, they note that 32-bit floating point textures were critical to the accuracy of the MLP, a feature that is not fully supported on many graphics cards, and may also suffer from patent restrictions [5].

While Cg provides a language low level enough to be portable across graphics APIs (such as DirectX and OpenGL), but high enough to avoid hardware-dependent and cumbersome GPU assembly code, it is not the only language used for writing flexible shader programs. BrookGPU is a stream programming framework designed for use specifically on the GPU [2]. BrookGPU is higher level than Cg, abstracting the details of shader and vertex programs running on the GPU, instead focusing on input streams being operated on by special functions called kernels. The compiler enforces limitations inside kernel functions to ensure only efficient operations are executed on the GPU. The main benefit, as explained in [Davis 01], is cross-platform portability, as a single BrookGPU program can be compiled and run on both DirectX and OpenGL layers. In contrast, Cg only defines the low-level shader program, and the programmer must write 'glue' code with either DirectX or OpenGL function calls. Similarly, BrookGPU allows the programmer to focus on algorithm design, rather than on handling graphics API calls or deciphering shader language syntax. Davis was able to quickly implement Hopfield, MLP, and Fuzzy ART neural networks, and compare the BrookGPU to CPU implementations. For MLP and Hopfield networks, ATLAS BLAS optimized linear algebra suite was used in the CPU implementation, and complete code for both BrookGPU and CPU implementations is given. Davis was able to realize a modest 2 fold performance increase over CPU for MLP and Hopfield networks; however, the Fuzzy ART GPU implementation was dramatically slower than the CPU. The reason given for this is that BrookGPU's stream model doesn't support certain useful texture memory updates, and presumably other low level optimizations, that Cg allows. Similar to Zhongwen, Davis finds significant performance differences between NVidia and ATI performance for the same code, suggesting that high level implementations are quite sensitive to hardware differences, or perhaps differences in the two respective optimizing compilers for shader languages.

Spiking neural networks are often more interesting to neurobiologists than the MLP neural network, as they preserve more information about the dynamics of neurons and more closely resemble biological neurons. With additional fidelity comes cost; MLP neuron activations are computed each timestep, whereas spiking networks can take hundreds of timesteps to compute a single neuron activation [Bernhard 06]. To this end, several researchers have developed GPU algorithms for simulating these networks more efficiently. Bernhard uses integrate-and-fire spiking networks to solve an image segmentation problem in machine vision. Neurons are connected locally for excitation response to regions of similar color, and are connected globally for inhibition response regardless of color. Reading locally from a small number of excitatory neurons in parallel is efficient for

fast GPU texture memory, but reading all other neurons for global inhibition is not. Bernhard solves this by adding an additional pass where a texture pixel is colored only if its corresponding neuron has fired. Then, using an *occlusion query*, he counts the number of pixels written, and this computes the global inhibition for the neuron. Since GPUs accelerate occlusion queries, this results in a 5 to 7 times speed increase versus CPU timings. An unrelated benefit of simulating neural networks on the GPU is ease of implementation visualization. Since Bernhard simulates neurons with textures, it is straightforward to display these for debugging and publication purposes, as shown in many instances in [Bernhard 06].

Integrate-and-fire is a computationally simple model of spiking neurons where neurons are modeled as a charging capacitor that discharges when a threshold is reached [Abbott 99]. A more recent technique takes the canonical biochemical model presented in [Hodgkin/Huxley 51] and simplifies the partial differential equations to be more computationally tractable. The resulting Izhikevich model neurons are able to more accurately model the natural variation of different neuron types seen in in-vitro experiments and brain scans [Izhikevich 03].

Nageswaran et. al. model large numbers of interconnected Izhikevich neurons on a single GPU, on the order of 10^5 neurons with 10^7 synaptic connections [Nageswaran 09]. In order to do this, they use CUDA, similar to BrookGPU in that it abstracts the GPU as a multiprocessor, capable of running lightweight threads simultaneously and accessing shared memory via textures. Threads in CUDA can be scheduled in groups by the on-GPU hardware scheduler, such that if a thread in one group stalls reading GPU texture memory, another non-stalled group can run. Classical critiques of parallel distributed systems, such as synchronization costs, apply to CUDA as well. Despite this, Nageswaran was able to simulate large numbers of spiking neurons and synapses with a 25 fold speedup over a CPU implementation, with an average firing rate of 9Hz, within an order of magnitude of real-time brain wave frequencies. It is interesting to note that in both Bernhard's and Nageswaren's spiking network implementations, both the calculation and data structures for learning are present entirely on the GPU. This suggests having simple, more localized learning updates like weight to nearby firing frequency correlation (as opposed to complex global strategies like back-propagation) are more amenable to implementation on GPUs, which inherently must have simpler instruction and data paths than CPUs.

Fuzzy ART is a middle ground between the minimal analytic model of MLP networks and the biologically correct spiking networks, and its implementation on the GPU derives in part from various clustering algorithms on the GPU, such as SOM in [Zhongwen 05, Bohn 98]. Martinez-Zarzuela et. al. implement Fuzzy ART on the GPU in [Zarzuela 07]. Their technique uses two ART networks, one for training and one for testing, and exploits different kinds of parallelism for each task. The training ART parallelizes calculating the input activations and performs parallel sorting for the 'winning' category match. The parallel sort closely resembles the MLP reduction Zhongwen uses for SOM, and the updating of weights is accomplished by rendering into a sub-region of a texture via scissoring. Similarly, the testing ART parallelizes category calculation for many input vectors concurrently. For example, one calculates the match value of the first category to all n input vectors simultaneously, storing these values to n subregions of a texture. The same is done for the second category, stored into another texture. The max response between both textures is calculated and stored into the first texture, and this method is repeated for all remaining categories. Unfortunately, updating the weights during training tended to be the performance bottleneck, with the GPU training ART running 10 to 100 fold slower than a CPU implementation, whereas the GPU testing ART had no such bottleneck and ran 25 to 46 times faster than the CPU. The authors note that un-optimized memory transfer from CPU to GPU is a potential source of improvement for the testing ART.

There are several GPU implementations of clustering algorithms that are not neurally based, see [Hall 04, Harris 05]. A summary of timing results for GPU neural network implementations is shown in table 1.

Neural Arch.	Implementation	Speedup over CPU
SOM[1]	OpenGL, Cg	0.8-5.3, 2-4
MLP[2]	Cg, Cg, BrookGPU	20, 200[†], 2
Hopfield	BrookGPU	2
Fuzzy ART	BrookGPU, Cg	$2.2e^{-6}$[‡], 25-46[§]
IF[3] Spiking NN	Cg	5-10
IZ[4] Spiking NN	CUDA	25

Table 1: Timing results for neural network implementations

3 Conclusion

One key factor of designing GPU algorithms in general is choosing which level of the software stack to implement upon. Early implementations used graphics API calls directly, relying on implicit assumptions about which ones were accelerated by hardware. Later implementations rely on the somewhat stronger assumption that matrix multiply must always be fast on GPUs, but constrain an algorithm to be cast as matrix vector products. Still later implementations use GPU assembly code directly, freeing the problem from formulation as matrix multiplication and greatly expanding the number of accelerated operations available to the algorithm developer. Higher level shader language (Cg, GLSL) based implementations were developed to enable hardware portability, avoid assembly language, and push tricky parts of the optimization to GPU compiler writers. Even higher level platforms such as BrookGPU and CUDA were built on top of shader languages; the first puts limits on shader programs to ensure efficient compilation and avoid specific graphics APIs, and the second provides efficient on-GPU thread scheduling.

All of these levels have resulted in successful GPU neural network implementations, with different trade-offs associated with the different software layers. For example, BrookGPU allowed Davis to quickly create and test MLP, Hopfield, and

Fuzzy ART implementations, where most studies present a single optimized implementation. However, BrookGPU has issues optimizing memory transfer, resulting in a Fuzzy ART implementation that was dramatically slower than the optimized CPU one.

Many researchers state the necessity of minimizing or optimizing CPU to GPU data transfer, typically during the learning phase of the neural algorithm. Additionally, most implementers make clever use of newly added hardware functionality designed to enhance graphics performance. Bernhard uses occlusion queries, for example, in order to efficiently implement global inhibition. Being aware of a wide variety of GPU 'tricks', as well as having a solid understanding of GPU hardware architecture, is therefore key to creating state of the art neural simulators.

4 Future Work

As CPU and GPU designers attempt to push the performance curve by adding parallelism instead of clock speed, it becomes more important to develop robust, fault tolerant computational paradigms that can make use of this new hardware. Neural networks have properties amenable to filling this niche, as they are inherently fault tolerant and often easily parallelized. Much work remains, however, in creating efficient, portable implementations of neural architectures, especially for unsupervised learning algorithms like SOM and ART based networks. Even well-studied networks like the MLP are very performance sensitive to network geometry; more general codes are needed that would allow researchers to reap the benefits of GPU parallelism without having to rewrite complicated GPU code for networks with novel structure.

[1] Self-Organizing Map [2] Multi-Layer Perceptron [3] Integrate and fire neuron [4] Izhikevich neuron
† CPU implementation unavailable ‡ implementation lacks memory optimization § testing only, training is 14-87 times slower than CPU

References

[Abbott 99] Abbott, L "Lapicque's Introduction of the Integrate-and-Fire Model Neuron (1907)"
Elsevier Brain Research Bulletin vol. 50, No. 5/6, pp. 303-304, 1999

[Bernhard 06] Bernhard, F; Keriven, R "Spiking Neurons on GPUs"
International Conference on Computational Science, 2006

[Bohn 98] Bohn, C.A. "Kohonen Feature Mapping Through Graphics Hardware"
Proceedings of Third International Conference on Computational Intelligence and Neuroscience 1998

[Campbell 05] Campbell, A.; Berglund, E.; Streit, A.
"Graphics Hardware Implementation of the Parameter-less Self-organizing Map"
IDEAL pp.343-350 2005

[Davis 01] Davis, C "Graphics Processing Unit Computation of Neural Networks"
Master's Thesis, UNM Press 2001

[Halfhill 08] Halfhill, Tom "Parallel Processing with CUDA"
Microprocessor Report [Reprint] January 2008
www.nvidia.com/docs/IO/47906/220401_Reprint.pdf, accessed 9/10

[Hall 04] Hall, J; Hart, J "GPU Acceleration of Iterative Clustering"
Manuscript Accompanying Poster at GP2̂: The ACM Workshop on General Purpose Computing on Graphics Processors and *SIGGRAPH 2004 Poster*, 2004

[Harris 05] Harris, C; Haines, K "Iterative Solutions Using Programmable Graphics Processing Units"
The 14th IEEE Intl. Conf. on Fuzzy Systems (FUZZ) 2005 May 22-25 pp. 12-18, 2005

[Hodgkin/Huxley 51] Hodgkin, A. L.; Huxley, A. F.
"Measurement of Current-Voltage Relations in the Membrane of the Giant Axon of *Loligo*"
Journal of Physiology, London 116:424-448, 1951

[Izhikevich 03] Izhikevich, E "Simple Model of Spiking Neurons"
IEEE Trans. on Neural Networks vol. 14, no. 6, Nov. 2003

[Kruger 03] Kruger, J; Westermann, R "Linear Algebra Operators for GPU Implementation of Numerical Algorithms"
SIGGRAPH 2003 Conference Proceedings

[Kyong-Su 04] Kyoung-Su Oh; Keechul Jung "GPU implementation of Neural Networks"
Pattern Recognition Vol. 37, Issue 6, June 2004

[Larsen 01] Larsen, E.S.; McAllister, D "Fast Matrix Multiplies Using Graphics Hardware"
Super Computing 2001 Conference Denver, CO 2001

[Levine 09] Levine,B; Schroeder,J; et. al. "PlayStation and IBM Cell Architecture"
http://www.mcc.uiuc.edu/meetings/2005eab/presentations/PlaystationIBMCell.ppt, accessed 12/09

[Meuth 07] Meuth, Ryan; Wunsch, Donald "A Survey of Neural Computation on Graphics Processing Hardware"
IEEE 22nd International Symposium on Intelligent Control pp. 524-527, 2007

[McCullogh/Pitts 43] McCullogh,W; Pitts,W "A Logical Calculus of the Ideas Immanent in Nervous Activity"
Bulletin of Mathematical Biophysics 1943

[Minsky/Papert 69] Minsky, M; Papert, S "Perceptrons: an Introduction to Computational Geometry"
MIT Press, 2nd edition 1969

[Nageswaran 09] Nageswaren, J; Dutt, N; Krichmar, J
"Efficient Simulation of Large-Scale Spiking Neural Networks Using CUDA Graphics Processors"
accepted in *International Joint Conference on Neural Networks*, 2009

[Pietras 05] Pietras, K; Rudnicki, M "GPU-based Multi-Layer Perceptron as Efficient Method for Approximation Complex Light Models in Per-Vertex Lighting"
Studia Informatica Vol. 2(6) pp 53-63, 2005

[Rolfes 04] Rolfes, T "Artificial Neural Networks on Programmable Graphics Hardware"
Game Programming Gems 4 2004 pp. 373-378

[Rolfes 03] Rolfes, T "GPU Implementation of Checker Board Evaluator for GPG4"
http://www.circensis.com/gg4.html, zip archive, 2003, accessed 9/10

[Steinkraus 05] Steinkraus, D; Simard, P. Y.; Buck, I. "Using GPUs for Machine Learning Algorithms"
ICDAR '05 Proc. of 8th Intl. Conf. on Document Analysis and Recognition , Washington DC, USA pp. 1115-1119 2005

[Zhongwen 05] Zhongwen, Luo; Hongzhi, Liu; Xincai, Wu
"Artificial Neural Network Computation on Graphics Process Units"
IEEE Intl. Joint Conference on Neural Networks 2005 Vol. 1 pp. 622-626

[Zarzuela 07] Martinez-Zarzuela, M; Pernas, F "Fuzzy ART Neural Network Parallel Computing on the GPU"
IWANN 2007, LNCS 4507 pp. 463-470

[1] "Cg Homepage"
http://developer.nvidia.com/page/cg_main.html, accessed 12/09

[2] "BrookGPU", *Stanford University Graphics Lab*
http://graphics.stanford.edu/projects/brookgpu, accessed 12/09

[3] arch3angel@gmail.com "12 Node Xbox Linux Cluster"
http://www.xl-cluster.org/index.php, accessed 12/09

[4] "OpenGL NVidia depth_buffer_float Documentation"
http://www.opengl.org/registry/specs/NV/depth_buffer_float.txt, accessed 12/09

[5] "OpenGL ARB texture_float Documentation"
http://www.opengl.org/registry/specs/ARB/texture_float.txt, accessed 12/09