Cortical Architectures on a GPGPU

Andrew Nere and Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin – Madison
nere@wisc.edu, mikko@engr.wisc.edu

ABSTRACT

As the number of devices available per chip continues to increase, the computational potential of future computer architectures grows likewise. While this is a clear benefit for future computing devices, future chips will also likely suffer from more faulty devices and increased power consumption. It is also likely that these chips will be difficult to program if the current trend of adding more parallel cores continues to follow in the future. However, recent advances in neuroscientific understanding make parallel computing devices modeled after the human neocortex a plausible, attractive, fault-tolerant, and energy-efficient possibility.

In this paper we describe a GPGPU extension to an intelligent model based on the mammalian neocortex. The GPGPU is a readily-available architecture that fits well with the parallel cortical architecture inspired by the basic building blocks of the human brain. Using NVIDIA's CUDA framework, we have achieved up to 273x speedup over our unoptimized C++ serial implementation. We also consider two inefficiencies inherent to our initial design: multiple kernel-launch overhead and poor utilization of GPGPU resources. We propose using a software work-queue structure to solve the former, and pipelining the cortical architecture during training phase for the latter. Additionally, from our success in extending our model to the GPU, we speculate the necessary hardware requirements for simulating the computational abilities of mammalian brains.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming-*Parallel programming.*

General Terms

Algorithms, Performance, Design.

Keywords

CUDA, GPGPU, cortical architecture, hypercolumn, minicolumn.

1. INTRODUCTION

Contemporary computer architectures are extraordinarily good at performing a variety of complex mathematical and scientific workloads with performance reaching over a petaflop in today's supercomputers [1]. As future technologies continue to scale down to ultra-small CMOS transistors, nano-tubes, or even individual molecules, we realize the number of devices available for future architectures will grow and the performance potential for these computing devices will increase as well. Current trends allocate these extra devices as multiple processors, and chip-multiprocessors are now commonly available in desktop and laptop computers. While the number of resources and available cores will continue to increase on future chips, programming these highly parallel devices is not getting easier. More devices also means that the energy consumed by these future chips will increase. Furthermore, as devices shrink and process variation grows, faults will become a more common problem.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GPGPU '10, March 14, 2010 Pittsburgh, PA, USA
Copyright (c) 2010 ACM 978-1-60558-935-0/10/03 $10.00

While chip-multiprocessors have only recently become commonplace, we consider that nature has provided a highly parallel processor that is easy to program (or *train*), energy efficient, and fault tolerant. The mammalian neocortex exhibits all of these qualities that are important to future computer designs, and recent work in the fields of neurobiology and neuroscience has provided the necessary insights into how the brain is able to harness such virtues [9]. Therefore, the neocortex should be considered as a promising candidate model for future computing devices.

The work in [5, 6] proposes an intelligent system design inspired by the mammalian neocortex. In this paper, we propose extending this model to the GPGPU. We describe how this neocortex-inspired architecture fits well to NVIDIA's CUDA framework and show some preliminary results indicating that, on the GPU, the model can be extended to interesting problems such as image recognition tasks and game-playing in real time with impressive performance results. We also examine some of the difficulties encountered with expanding this intelligent system model to the GPGPU and propose several solutions that we have implemented to overcome them.

The rest of this paper is organized as follows: Section 2 gives a high level description of the cortical architecture described in [5, 6]. Section 3 describes the methods used to extend these cortical architectures to the GPGPU using NVIDIA's CUDA framework as well as presents some performance results. Section 4 examines two of the major issues encountered with expanding this cortical architecture to the GPGPU: synchronization for data dependencies and unbalanced resource utilization. Section 5 describes a work-queue implementation used to reduce the overhead associated with such data dependencies and presents some performance results of this solution. Section 6 describes a method to pipeline the training stages of the cortical architecture to improve resource utilization on the GPGPU and presents some performance results. Section 7 presents our conclusion and proposes our future work.

2. CORTICAL ARCHITECTURE DESIGN

The neocortex is a component of the brain that is unique to mammals. This part of the brain is responsible for skills such as mathematics, music, language, and perception. The neocortex comprises around 77% of the entire human brain. For a typical adult, it is estimated the neocortex has around 11.5 billion neurons and 360 trillion synapses, or connections between neurons [14]. Mountcastle [11] observed that the neocortex is structurally very uniform, composed of millions of nearly identical functional units. He identified these functional units as cortical columns and later referred to them as *hypercolumns*. A hypercolumn is composed of smaller structures he coined *minicolumns* which in turn are collections of 80-100 neurons. The minicolumns within a hypercolumn share the same receptive field, meaning the same set of input synapses, and are tightly bound together via short-range inhibitory connections [12]. Using these connections, a minicolumn is able to alter the synaptic weights of its neighbors to identify unique features from the inputs observed in the receptive field.

The cortical architecture model detailed in [5, 6] and expanded in this paper is highly motivated by the properties of the neocortex and the structure of the hypercolumn. Historically, different levels of abstraction have been used in pursuit of intelligent systems. These have ranged from high level behavioral models all the way down to the cellular level designs such as artificial neural networks. Using hypercolumns as the level of abstraction, this model can still be rooted in biological plausibility yet avoid the computational complexity of a neuron-level model. Previous research has lead to designing many successful artificial neural networks capable of tasks such as handwriting recognition or playing checkers. In fact, some of these traditional neural networks have even been ported to the GPU [2, 7]. However, traditionally artificial neural networks are often trained via back-propagation for classification or recognition tasks: that is, the correct classification is known and adjusts the weights in each layer from the top down to reduce the classification error. This form of learning is known as supervised learning. In biology, it is much more likely that learning is accomplished via unsupervised learning, where labels are not provided, but classification is achieved through similarity of features, or semi-supervised learning, where a few labels are provided and classification is based on similarity to the labeled data. Our cortical architecture is able to learn features from its dataset in an entirely unsupervised fashion. However, in the future this model will likely use a semi-supervised learning rule that will make learning faster, more robust, and maintain biological plausibility.

Figure 1 shows the basic functional unit of our cortical architecture model, the hypercolumn. Each hypercolumn consists of multiple minicolumns that share a receptive field. These minicolumns are also strongly connected to neighboring minicolumns via inhibitory horizontal pathways shown in green. Each of the minicolumns contains a set of weights W initialized to random values. During operation, each of the minicolumns evaluate the dot-product between its inputs *X* and weights *W*.

$$DP = \sum_{i=1}^{N} X_i . W_i \quad \text{(Equation 1)}$$

The result of this dot-product is used as the input to a thresholded activation function. When this output exceeds the threshold, the minicolumn fires and feeds forward its output to another hypercolumn's receptive field.

Another unique feature of the neocortex is its ability to accomplish complex tasks using parallel hierarchical processing. A primary example is the visual cortex. At the lowest level, unique edges are recognized by different minicolumns, while upper levels of the hierarchy recognize unique shapes, objects, and ultimately full visual scenes [4].

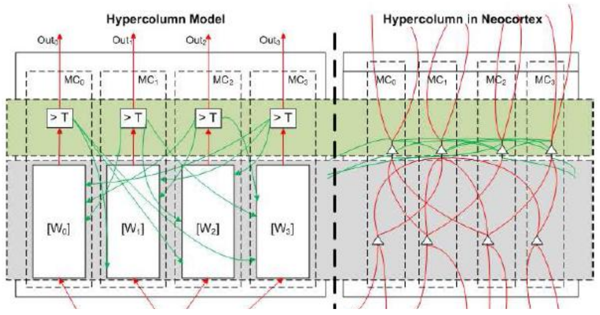

Figure 1. A hypercolumn is made up of minicolumns.

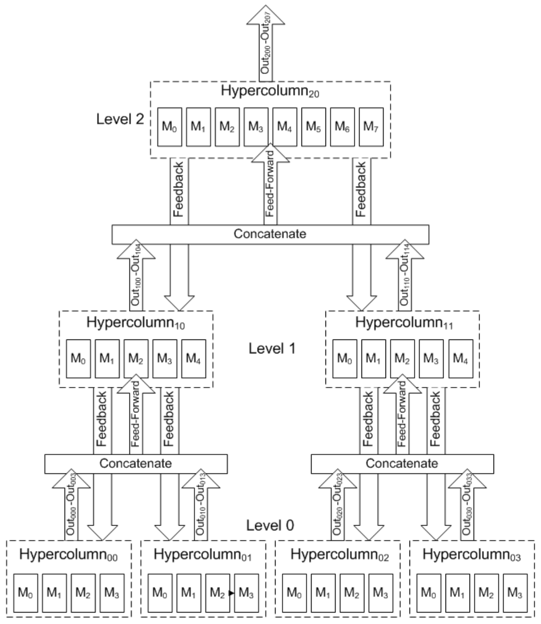

Figure 2. A hierarchy of hypercolumns.

In the same manner, the cortical architecture model also uses a hierarchical design to accomplish complex tasks. Figure 2 shows an example of a three level hierarchical cortical architecture. In the bottom level, each of the hypercolumns has a distinct receptive field to be shared by each of its internal minicolumns. The output of this hypercolumn feeds forward its input to the next level of the hierarchy, which in turn is structured similarly. Within the hierarchy, each of the higher level hypercolumns receives its inputs from the activations of the lower hypercolumns. The minicolumns in the top level hypercolumn train themselves to identify the entire complex unique features of the input. For this paper, we consider visual recognition tasks as the inputs to the cortical architecture. The scale and configuration of the hierarchy depend on the resolution and number of unique inputs. Figure 3 shows some of the inputs that various cortical architectures were trained on, ranging from small 8x8 pixel images to 28x28 pixel handwritten characters from the MNIST database (http://yann.lecun.com/exdb/mnist).

Figure 2 also shows feedback paths from higher levels of the cortical architecture to lower ones. These feedback paths play an important role in the recognition of noisy and distorted data by propagating "big picture" contextual information from the upper levels of a hierarchy to the lower levels. These feedback paths are known to exist in biological neural networks; we are currently working to extend our model to incorporate their functionality.

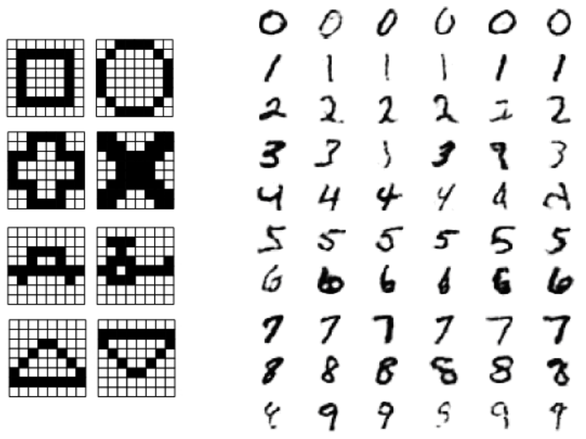
Figure 3. Recognition tasks range in scale from toy problems (left) to MNIST handwritten data (right).

3. CORTICAL ARCHITECTURE ON CUDA

While it may be possible to eventually create hardware designs inspired by the mammalian neocortex, we have spent considerable time investigating currently available hardware architectures that would be a good match for our existing software model. A major goal of the work described in the previous section is to design intelligent systems that are good at performing tasks such as playing a board game, speech to text translation, or recognizing handwritten characters. However, we would also like these tasks to be performed in real time. A major feature of the model is that, like the brain, a large amount of parallelism is inherent to the design of the structure. This large amount of parallelism has made supercomputers such as the Cray XD1 an excellent platform for cognitive models [13]. However, with a limited budged the GPGPU is an attractive hardware architecture that is still able to take advantage of parallel processing algorithms. Particularly, NVIDIA's CUDA framework was a viable option that allows programmers to take advantage of massive amounts of parallel processing units on a commercially available GPU. Using a modest set of extensions to the C-programming language, programmers can easily port their serial programs to parallel programs without any graphics knowledge.

CUDA-enabled GPUs contain a number of *Streaming-Multiprocessors* (SMs). The CUDA programming model is built around several layers of components which the programmer can explicitly configure. The CUDA-thread is the basic unit of execution, and these threads are organized into thread-blocks, or *Cooperative Thread-Arrays (CTAs).* Current generation CUDA enabled devices are capable of executing between 1 and 8 CTAs concurrently on each SM. Within a CTA, threads can communicate and share local data via a fast-access shared memory space. In current generation GPU hardware, there is 16KB of shared memory per SM. The number of CTAs that can concurrently execute on the GPU depends on a number of factors, including the number of threads scheduled per CTA, the number of registers used per thread, and the amount of shared-memory used during program execution [3]. CUDA applications also can be optimized by loading variables into the shared memory and optimizing global memory accesses with memory coalescing [15].

3.1 Implementing Hierarchy on CUDA

Like the cortical architecture described in this paper, the CUDA framework also has different levels of components. The cortical architecture has minicolumns, hypercolumns, and hierarchies, whereas CUDA has threads, CTAs, and groups of CTAs known as grids or kernels. Fitting the cortical architecture to the CUDA software model was achieved by mapping the different levels of components between the two. In our implementation, each minicolumn is mapped to a CUDA-thread and each hypercolumn to a CTA. This is a good fit because in CUDA, the basic building block for a unit of work is the CTA, and in the cortical architecture the basic building block is the hypercolumn. Using the local shared memory space, we are able to model the fast short-range lateral connections that connect the minicolumns within a hypercolumn. For a hypercolumn to learn more distinct features from a set of inputs, the number of minicolumns can be increased to recognize new features. This simply means adding more threads per CTA in the CUDA cortical architecture model.

Considering the hierarchical nature of the cortical architecture, we realize the inputs of the upper levels depend on the outputs from the lower levels. If we consider executing a cortical architecture such as Figure 2 on a GPGPU, we have no way of guaranteeing that a CTA executing a hypercolumn in the lowest level will finish before a CTA executing the hypercolumn at the highest level. For data dependencies such as these, the typical solution is to execute the structure as separate CUDA-kernels; that is, simply execute one level of the hierarchy on the GPU, return control to the CPU, and launch the next level of the hierarchy. Section 4 will detail some of the inefficiencies we discovered using this solution on this hierarchical data structure, and Sections 5 and 6 detail some of the solutions we have explored.

3.2 Kernel Configuration

A key design feature of the cortical architecture is that it is easily scalable to any size needed by the particular task of interest. Figure 3 shows two different scale recognition tasks that our cortical architecture has been applied to: simple 64-pixel images and 784-pixel handwritten digits. We make note that as the resolution of the inputs increases, the number of hypercolumns and minicolumns in the network increases likewise. In the simplest type of configuration, hypercolumns are arranged in a binary tree fashion like Figure 2. Table 1 details the input size, total number of hypercolumns (or CTAs) for each configuration, and the number of levels in the different cortical architectures. For example, 256-pixel input, the hierarchy has 31 hypercolumns across five levels, and for a 4096-pixel input, the hierarchy has 511 hypercolumns across nine levels using a binary tree configuration. As mentioned above, if we want to increase the number of distinct features learned by each hypercolumn, we configure the cortical architecture to encompass more minicolumns per hypercolumn. This in turn increases the number of threads per CTA and effectively reduces the number of CTAs that can be scheduled per SM. In the rest of the paper we examine both the upper and lower bounds of the cortical architecture configurations by testing the current range of schedulable CTAs per SM (1 to 8 in current hardware). For the lower bound (1 CTA), each hypercolumn consists of 256 minicolumn-threads, while the upper bound (8 CTAs) has 32 minicolumn-threads per hypercolumn.

Table 1. Kernel Configurations of Experiments

Input Size (Pixels)	# of Hypercolumns (CTAs)	# Levels in Hierarchy (Kernel Launches)
256	31	5
512	63	6
784	97	6
1024	127	7
2048	255	8
4096	511	9

3.3 Results of CUDA vs. C++ Model

Figure 4 shows the performance speedups of a highly-optimized CUDA model vs. our original unoptimized serial C++ implementation for a range of different scale inputs. The C++ implementation was run on an Intel Core2 Quad @ 2.4 GHz, and the CUDA version was executed on a GeForce GTX 280 with 30 SMs (total 240 cores) @ 1.46 GHz. To measure performance, we examined the execution time for 15000 image training iterations of different scale inputs ranging from 256 to 4096 pixels. For reference, the MNIST handwritten database characters are 28x28 (for a total 784) pixels and have served as the baseline input dataset for training our model while the other scale inputs have typically been image recognition training sets we have created ourselves.

In Figure 4, we see that the CUDA version of the cortical architecture renders some significant speedups. For the scales of inputs tested, the achievable speedups range from 9x–77x for cortical architectures scaled to run 8 CTAs/SM and 9x – 45x when scaled to 1 CTA/SM. Even more promising is the fact that the speedups continue to improve as the cortical architecture and input size increases.

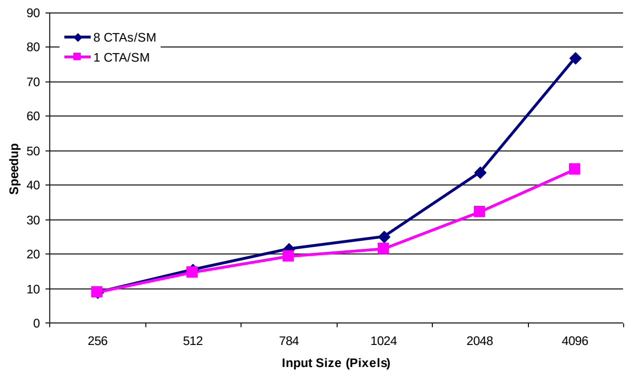

Figure 4. Speedups of CUDA vs. C++ model.

4. DIFFICULTIES WITH DATAFLOW/HIERARCHY ON CUDA

While it is clear from the speedups obtained in the previous section that our neocortex-inspired model ports well to the CUDA framework, we also make some observations on a couple inefficiencies in our implementation. When applications which contain data dependencies, the typical solution is to synchronize these dependencies with multiple CUDA-kernel launches. This lock-step method, similar in nature to Bulk Synchronous Processing [16], uses the end of one CUDA-kernel and the beginning of the next as a type of implicit global barrier. However, this solution for structures like the cortical architecture hierarchy results two problems: significant overhead and poor GPU resource utilization.

The first inefficiency is that using multiple kernel launches incurs the overhead of transferring control from the GPU to the CPU multiple times for a single hierarchy. Depending on the application, these kernel launch overheads can become a significant portion of the total execution time. Figure 5 shows the percent of execution time spent for kernel launch overhead for the different scale inputs and networks we simulated. We can see that 6.5-14.1% of the total execution time for a hierarchy is spent on kernel launch overhead.

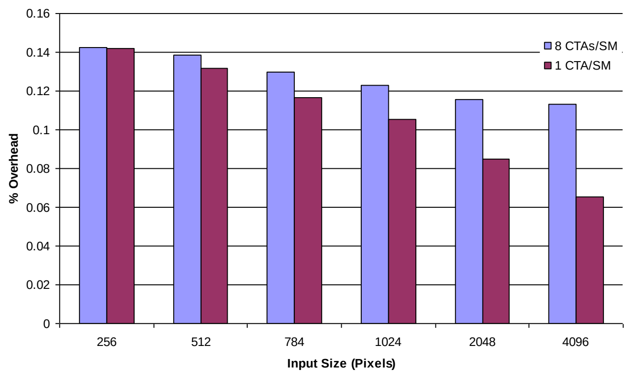

Figure 5. Overhead of using multiple kernel launches for different scale hierarchies.

The second inefficiency we observe is poor resource utilization on the GPGPU. As described in Section 2, the outputs from the lower levels of the hierarchy become the inputs to the upper levels. Therefore, the hypercolumn count decreases with each higher level of the cortical architecture. For the configurations considered for this paper, each layer reduces the number of hypercolumns by a factor of 2 (for example, see Figure 2). Using a single kernel launch per level means that the upper layers of the network, with very few CTAs, will use very little resources of the GPGPU. For example, consider a 7-level hierarchy capable of image recognition for an input size of 1024 pixels. At the lowest level, 64 CTAs can be executed in parallel. However, at the top level of the hierarchy, only a single CTA is executed. Figure 6 shows the level-by-level breakdown of speedups for a network scaled for 1024 pixel image recognition.

5. KERNEL FUSION USING QUEUE

From Figure 5 we understand the amount of overhead incurred by multiple kernel launches. Ideally we would like to be able to execute the entire cortical architecture on the GPU concurrently, reducing the overhead to a single kernel launch. Scheduling each

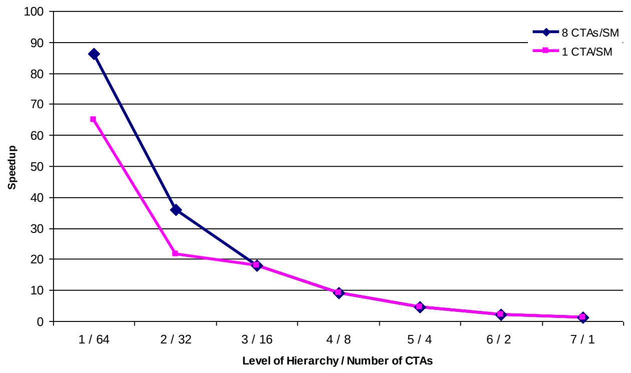

Figure 6. Speedup of each level in a 7-level hierarchy.

hypercolumn as a CTA, we could leave it up to the CUDA architecture to schedule each of the hypercolumns. However, a limitation of the CUDA architecture is that there is no guarantee as to the order in which CTAs are scheduled on the SMs [3]. Furthermore, once a CTA is scheduled on an SM, it will run until completion. This means there are no sleep-wait primitives to allow a hypercolumn to wait until its inputs from a lower level hypercolumn are ready. For a hierarchical data structure like the cortical architecture, this means there is no easy way to guarantee that upper level hypercolumns will be evaluated after the lower level hypercolumns have correctly provided the inputs to the upper levels.

Since we cannot control how CUDA schedules CTAs, we instead create a software work-queue to explicitly orchestrate the order in which hypercolumns are executed. The work-queue is managed directly in the GPU's global memory space. This work-queue method operates as follows: a single CUDA-kernel is launched to occupy only as many CTAs as can concurrently fit across all of the SMs in the GPGPU. This means on our current hardware, depending on the configuration of the cortical architecture, 30-240 CTAs are scheduled on a GPGPU. Each of the CTAs loop until all the hypercolumns in a hierarchy have executed. Each CTA first uses an atomic primitive to access into the work-queue. The work-queue returns the specific hypercolumn-ID that the CTA will execute next. Once the CTA finishes the execution of the hypercolumn, it in turn sets a flag to indicate to its parent hypercolumn that it has finished execution and schedules the parent hypercolumn on the work-queue. Finally, the CTA indexes again into the work-queue to get the next scheduled hypercolumn-ID. If a hypercolumn in the upper levels of the hierarchy get scheduled, they will wait to execute until each of its children hypercolumns have finished execution and indicated so via the set flags. Once the queue is empty, the kernel returns control to the CPU.

We extend the work-queue method to reduce the total number of global reads and writes performed during the cortical architecture's execution. Specifically, we made the observation that we could reduce the number of global memory reads and writes if children and parent hypercolumns were scheduled immediately after each other on the same SM. For example, we consider Hypercolumn$_{00}$, Hypercolumn$_{01}$, and Hypercolumn$_{10}$ in Figure 2. Using the work-queue method, Hypercolumn$_{00}$ and Hypercolumn$_{10}$ would be scheduled first on the queue, and after completing execution, would write their outputs to the global memory. Later, when Hypercolumn$_{10}$ is scheduled, it would read its input data from global memory. However, if Hypercolumn$_{00}$ and Hypercolumn$_{10}$ were scheduled back to back, the data transfer between the two hypercolumns could exist at the fast shared memory level as opposed to a write and read at the global memory level. In this optimization, which we term opt-queue, once a CTA has finished executing the workload Hypercolumn$_{00}$, it checks to see if its peer-hypercolums (Hypercolumn$_{01}$ in this case) have completed as well via a single global-memory flag. If the peer hypercolumns have completed and indicated via the flag, the CTA can simply begin execution of Hypercolumn$_{10}$ rather than fetching a hypercolumn off of the work-queue. The major advantage here is that only the data provided by Hypercolumn$_{01}$ must be read from the global memory since the data provided by Hypercolumn$_{00}$ exists in the shared memory. The major advantage here is that the overall bandwidth consumption is reduced by limiting the number of transfers to the global memory.

Figures 7 and 8 show the relative execution time of both the opt-queue and the standard work-queue versus the baseline (multiple kernel call) CUDA implementation. We also compare against this same CUDA baseline minus the kernel launch overhead which we observed in Figure 5. For cortical architectures configured to occupy one CTA per SM (Figure 7), we see that the work-queue and opt-queue implementations beat the baseline configuration for inputs of 512 pixels and larger. They also outperform the baseline minus the kernel-launch overhead for the 4096 pixel input. We also note that opt-queue matches or beats the performance of the work-queue implementation for all sizes of inputs tested.

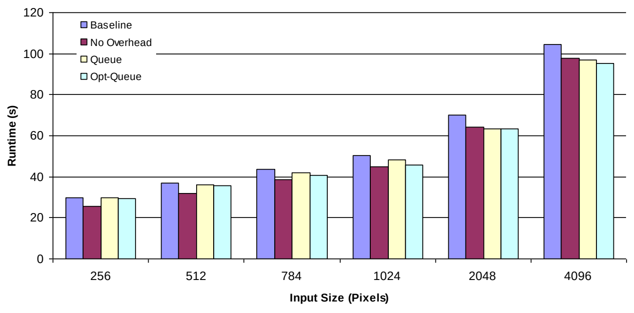

Figure 7. Execution time for 1 CTA/SM

Figure 8 shows the performance comparison for a cortical architecture configured to occupy 8 CTAs per SM. Here, we notice that the work-queue and opt-queue implementations actually match the performance or perform worse than the baseline. We note that once again, opt-queue's performance matches or beats that of the work-queue implementation. We hypothesize that these trends are likely due to an increased amount of concurrent atomic operations to the global memory space. The atomic primitives used for the queues are actually quite slow, typically on the range of 400-600 cycles [3]. Not only are these memory accesses slow, but we also encounter an effect known as memory-camping. Memory-camping refers to global memory bank access patters where there is more than one outstanding request at a time to a single bank, and the requests are queued up at the global memory level. Since we have a single global work-queue structure, we have many global memory accesses to the same memory bank by multiple SMs. We believe the major bottleneck here is that, when scheduling 8 CTAs per SM, we are performing 8-times as many concurrent atomic read-modify-write operations to the work-queue. The cumulative latencies of the global memory accesses in turn result in hindering performance to the point where the memory accesses accumulate to more than the original kernel-launch overhead.

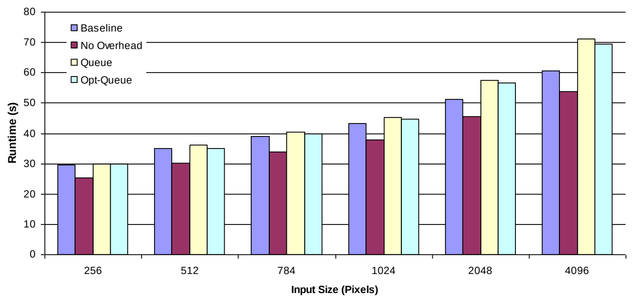

Figure 8. Execution time for 8 CTAs/SM.

6. PIPELINING TO INCREASE RESOURCE UTILIZATION

From Figure 6, we are able to see how the hierarchical design of our cortical architecture results in poor utilization of the GPGPU's resources. We see that for the lower levels of the hierarchy we are able to extract a large amount of parallelism, 66x and 87x speedups for the two configurations of cortical architectures. However, since upper levels of the hierarchy have fewer hypercolumns to evaluate, it is often the case we have less work than actual resources. We see the benefits of using the GPGPU quickly taper off. Once again, the optimal solution to maximize hardware utilization would be to have all hypercolumns across all levels of the cortical architecture execute concurrently. However, we are unable to do so due to aforementioned data dependencies between adjacent levels.

Our solution is to simply pipeline the training phase of the cortical architecture. In this method, a single kernel-launch executes all hypercolumns in the hierarchy, and we use multiple buffers between each hierarchy level to guarantee that outputs from one level do not corrupt the inputs to the next level during global memory accesses. For example, if we consider a cortical architecture from Table 1 that evaluates a 1024 pixel input, seven levels of the hierarchy execute on seven separate kernel launches. By pipelining, all seven layers are executed in a single kernel launch. Between any particular level of the cortical hierarchy, the inputs are read from buffer-0 and outputs of the level are written to buffer-1. Upon the next kernel launch, the inputs are read from buffer-1 and the outputs are written to buffer-0. While this method better utilizes the GPU resources and also improves training throughput, it still takes multiple kernel launches for any particular input to propagate the entire way through the hierarchy (seven kernel launches in the case of our example).

Figure 9 shows the speedups of the pipelined versus the baseline multiple kernel-launch implementation. For the cortical architectures scaled for 8 CTAs/SM, the achievable speedup is 3.5x – 4.5x over the baseline, resulting in up to a 273x speedup. When scaled to 1 CTA/SM, the achievable speedups were 1.55x – 3.1x, resulting in up to a 70x speedup over the C++ implementation. We note the dip seen between 2048 and 4096 pixel inputs is an indicator that we are asymptotically approaching a performance ceiling of this application on the GPU, with 202x to 273x speedups respectively.

The disadvantage of this implementation is that the amount of global memory dedicated to input/output data doubles for every buffer added between hierarchy levels. Furthermore, using this pipelined implementation is feasible only in the feedforward training-phase for the cortical architecture. During the testing phase of the cortical architecture, we may use feedback connections from the upper levels of the hierarchy back down to the lower levels, as described in Section 2. Using a pipelined implementation would become increasingly complex as every connection would need to be buffered at each level, and evaluation of feedback would induce pipeline bubbles. For these reasons, pipelining is really only a feasible method to speed up the training phase of a cortical architecture for a particular task.

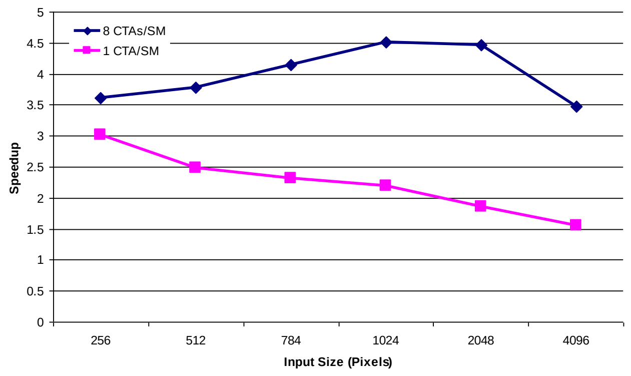

Figure 9. Speedup of pipelined training.

7. FUTURE WORK AND CONCLUSION

In this paper we have described a GPGPU parallelization extension to an intelligent system based on the mammalian neocortex. Using CUDA, we achieved 9x-77x speedups across different scale cortical architectures with a baseline design. We also examined a couple of inefficiencies observed on a data dependent hierarchy like our design. Our work-queue solution removes the overhead of multiple kernel-launches for some configurations of our cortical architecture, while our pipelining solution maximizes resource utilization and has up to a 273x speedup over the unoptimized C++ implementation. We also believe that these same solutions could be applied to other applications that have similar data dependencies and design structure. We currently are investigating the effectiveness of using a work-queue and pipelining for applications such as multilayer feedforward neural networks, array reductions, and parallel scans and joins.

Table 2. Estimated GPUs to model mammalian cortexes

	Number of Neurons in Neocortex (millions)	**Number of Tesla GPUs Required**
Rat	15	4
Cat	300	64
Chimp	6200	1325
Human	11500	2458

$$\frac{\text{Total \# of Neurons} \times \text{\# of Mult/Add Insts. per Minicolumn} \times \text{Biological Neuron Firing Rate}}{\text{240 Cores per GPU} \times \text{100 Neurons per Minicolumn} \times \text{Frequency of Core}} \quad \textbf{(Equation 2)}$$

With our initial success with the cortical architecture on visual recognition tasks and the promising speedups achieved by using NVIDIA's CUDA framework, we also consider the scale of GPGPUs needed to model different biological neocortex counterparts. Table 2 shows our estimates for the number of GPUs needed to fully realize the real-time compute capability of different scales of mammalian brains [14]. We consider the current top-of-the-line GPU available, the Tesla C1060, with 240 cores @ 1.3 GHz. The estimated number of GPUs for real-time performance is given by Equation 2. We assume that biological neurons have a maximum firing rate around 10-ms [10]. We also consider that each of our model's minicolumns must perform a multiply-and-add operation per unique synaptic input, estimating 10,000 synapses per minicolumn [8].

As future computing systems promise more devices and more parallelism, we consider alternatives to traditional computer architectures. Recent advances in neuroscience give inspiration towards future architectures based on the human brain with hopes of addressing issues like fault tolerance, energy efficiency, and programmability. While a hardware device based on the neocortex may be years down the road, we believe that the GPGPU paired with our cortical architecture is a formidable step towards realistic intelligent computing machines.

8. ACKNOWLEDGMENTS

We wish to thank Atif Hashmi for his assistance in porting the cortical model.

9. REFERENCES

[1] K. J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, J.C. Sancho, Entering the Petaflop Era: The Architecture and Performance of Roadrunner, in *Proc. IEEE/ACM Super Computing,* Austin, TX, Nov. 2008.

[2] Billconan and Kavinguy. A Neural Network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx.x.

[3] CUDA Programming Guide, 2.1, NVIDIA. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf.

[4] K. Grill-Spector, T. Kushnir, T. Hendler, S. Edelman, Y. Itzchak, & R. Malach. A sequence of object processing stages revealed by fMRI in the human occipital lobe. *Human Brain Mapping*, 6, 316-328, 1998.

[5] A. Hashmi and M. Lipasti. Cortical columns: Building blocks for intelligent systems. In *Proceedings of the Symposium Series on Computational Intelligence*, pages 21–28, 2009.

[6] A. Hashmi, H. Berry, O. Temam, and M. Lipasti. December (2009). Leveraging progress in neurobiology for computing systems. In *1st Workshop on New Directions in Computer Architecture (NDCA-1)*. New-York, New-York, USA.

[7] H. Jang, A. Park, and K. Jung. Neural Network Implementation Using CUDA and OpenMP. Proceedings of the 2008 Digital Image Computing: Techniques and Applications-Volume 00, pages 155{161, 2008.

[8] C. Johansson and A. Lansner. Towards cortex sized artificial nervous systems. *Lecture Notes in Computer Science: Knowledge-Based Intelligent Information and Engineering Systems*, 3213:959–966, 2004.

[9] E. Kandel, J. Schwartz, and T. Jessell. *Principles of Neural Science.* McGraw-Hill, 4 edition, 2000.

[10] G. Kreiman, C. Koch, and I. Fried. Category-specific visual responses of single neurons in the human medial temporal lobe. *Nature Neuroscience*, 3:946–953, 2000.

[11] V. Mountcastle. An organizing principle for cerebral function: The unit model and the distributed system. In G. Edelman and V. Mountcastle, editors, *The Mindful Brain*. MIT Press, Cambridge, Mass., 1978.

[12] V. Mountcastle. The columnar organization of the neocortex. *Brain*, 120:701–722, 1997.

[13] K. L. Rice, T. M. Taha, C. N. Vutsinas: Scaling analysis of a neocortex inspired cognitive model on the Cray XD1. The Journal of Supercomputing 47(1): 21-43, 2009.

[14] G. Roth and U. Dicke. Evolution of the brain and intelligence. *Trends in Cognitive Sciences*, 9, 250–257,2005.

[15] S. Ryoo, C. Rodrigues, S. Babhsorkhi, S. Stone, D. Kirk, and W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceesings Symposium on principle and practices of parallel programming, SIGPLAN*, pages 73 –82, 2008.

[16] L. G. Valiant. A Bridging Model for Parallel Computation. Communications of the ACM, 33(8):103{111, 1990.