UNIVERSITY OF WISCONSIN MADISON

MASTER'S PROJECT REPORT

Optimizing Hierarchical Algorithms for GPGPUs

Author:

ANDREW NERE

Advisor:

MIKKO LIPASTI

APRIL 26, 2010

Optimizing Hierarchical Algorithms for GPGPUs

Abstract

The performance potential of future computer architectures, thanks to Moore's Law, grows linearly with the number of available devices per integrated circuit. Whether these future devices are ultra-small CMOS transistors, nano-tubes, or even individual molecules, it is clearly understood that there will be many of them available to computer architects. If current architecture trends are a good indicator of future designs, likely many of these devices will be allocated as extra cores on chip-multicore systems. However, the nature of highly parallel processors consisting of ultra-small devices brings along with it some inherent difficulties. Between the complexity of programming multiprocessor systems, increased power consumption, and higher fault rate for these tiny devices, future architects will have their work cut out for them. 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-accelerated extension to an intelligent model based on the mammalian neocortex. Our cortical architecture, like the human brain, exhibits massive amounts of processing parallelism, making today's GPGPUs a highly attractive and readily-available hardware accelerator for such a model. Using NVIDIAs CUDA framework, we have achieved up to 330x speedup over an 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. We also investigate applying these techniques to a few CUDA applications that exhibit some structural similarities to our cortical architecture model. Additionally, from our success in extending our model to the GPGPU, we estimate the hardware requirements for simulating the computational abilities of mammalian brains.

1 Introduction

The prospect of future computer architectures bring with them the potential of ever-improving performance. These future computing systems may be made of ultra-small CMOS transistors, nano-tubes, or even individual molecules. Regardless of what device technology is used for these future computing systems, we know two things: they will be ultra-small, and there will be many of them available on a single chip. However, it is not yet known how these devices will be allocated. If current trends in computer architecture continue, likely the increase in devices per chip will mean an increase in the number of cores. Although there is great potential for these future computing devices, they come with some heavy costs. While the number of resources and available cores will continue to increase on future chips, programming these highly parallel devices is not getting easier. While many different parallel programming frameworks have been proposed, writing parallel code is simply not something typical programmers have in their skill set. Other problems exist due to the nature of ultra-small devices. More devices leads to an increase in power consumption, and small devices are more prone to faults [3].

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 [12]. While modern computers can execute a variety of complex mathematical and scientific workloads with peak performance reaching over a petaflop [1], we realize that there are many problems that remain difficult for computers, even though the human brain is quite good at solving them. Tasks such as learning a new game, recognizing a face, or converting audio input into written characters are almost trivial to humans, though programming such tasks takes a massive amount of effort. For these reasons, we propose that the neocortex should be considered as a promising candidate model for future computing devices.

Hashmi et al. [8, 7] propose an intelligent system design inspired by the mammalian neocortex. In [16], we proposed extending this neocortex-inspired architecture to general purpose graphics processing units (GPGPUs) and investigated some optimizations to mitigate inefficiencies with our implementation. In this paper, we expand on the work of [8, 7, 16] to test the performance limits of our cortical architecture on the GPGPU. We also attempt to generalize some of the optimizations we investigated and apply them to a couple similarly structured applications.

The rest of this paper is organized as follows: Section 2 gives a high level description of the cortical architecture described in [8, 7]. We discuss some related work on creating biologically inspired computing devices in Section 3. Section 4 describes the methods used to extend our cortical architectures to the GPGPU using NVIDIA's CUDA framework as well as presents some performance results. Section 5 examines two of the major issues encountered with expanding this cortical architecture to the GPGPU: synchronization for data dependencies and unbalanced resource utilization. Section 6 describes a work-queue implementation used to reduce the overhead associated with such data dependencies and presents some performance results of this solution. Section 7 describes a method to pipeline the training stages of the cortical architecture to improve resource utilization on the GPGPU and presents some performance results. In Section 8, we examine similar inefficiencies for two other applications and attempt to alleviate them with our solutions. Finally, Section 9 discusses our findings, and Section 10 concludes the paper.

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 [24]. For a typical adult, it is estimated the neocortex has around 11.5 billion neurons and 360 trillion synapses, or connections between neurons [19]. Mountcastle observed that the neocortex is structurally very uniform, composed of millions of nearly identical functional units [14]. He identified these functional units as *cortical columns*, later referred to as *hypercolumns*, because of the seemingly column-shaped organizations of neurons exhibiting similar firing patterns for a given stimulus. 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 [15]. Using these connections, a minicolumn is able to alter the synaptic weights of its neighbors to influence learning, typically to identify unique features from the inputs observed in the receptive field.

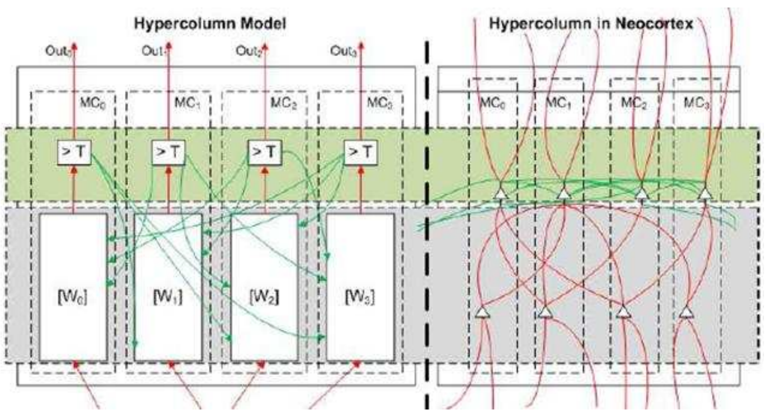

Figure 1: *A hypercolumn is made up of minicolumns. Right: the connectivity of biological minicolumns in a hypercolumn. Left: Our model's interpretation of the hypercolumn structure.*

The cortical architecture model detailed in [8, 7] 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 logic based behavioral models all the way down to the cell-level designs such as artificial neural networks. By using hypercolumns as the level of abstraction, our model can avoid the computational complexity of a neuron-level model while remaining rooted in 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 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 weights W and input vector X:

$$DP = \sum_{i=1}^{N} W_i X_i \tag{1}$$

The result of this dot-product is used as the input to a thresholded activation function. The threshold is calculated by:

$$threshold = \sum_{i=1}^{N} W_i \tag{2}$$

When the activation output DP 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. The most studied and well understood of these hierarchies is the visual cortex, though these hierarchies

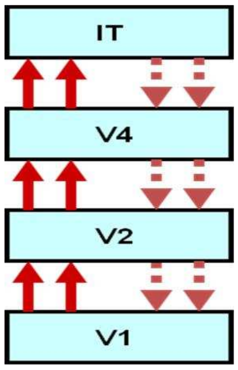

Figure 2: *The hierarchy of the visual cortex.*

are believed to exist for other major parts of the brain such as the auditory cortex and motor control cortex. Figure 2 shows a simplified diagram of the organization of the different levels of the visual cortex. At the lowest level of the visual cortex (V1), unique edges are learned by different minicolumns. Subsequent levels learn to recognize more complex shapes (V2, V4), while upper level of the hierarchy (IT) ultimately recognizes unique full visual scenes [5].

In the same manner, our cortical architecture model also uses a hierarchical design to accomplish complex tasks. Figure 3 shows an example of a three level hierarchical cortical architecture. In the bottom level, each of the hypercolumns has a distinct receptive field 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 input.

For this paper, we consider visual images as the inputs to the cortical architecture. The scale and configuration of the hierarchy depend on the resolution and number of unique inputs. Figure 4 shows a typical visual recognition task we have used for training and testing our cortical architecture: 28x28 (784 total) pixel handwritten characters from the MNIST database (http://yann.lecun.com/exdb/mnist).

Finally, Figure 3 also shows that 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 contextual information from the upper levels of a hierarchy to the lower levels. Using these feedback paths, an invariant representation can be stored in the cortex rather than all the variations of a particular stimulus, reducing unnecessary

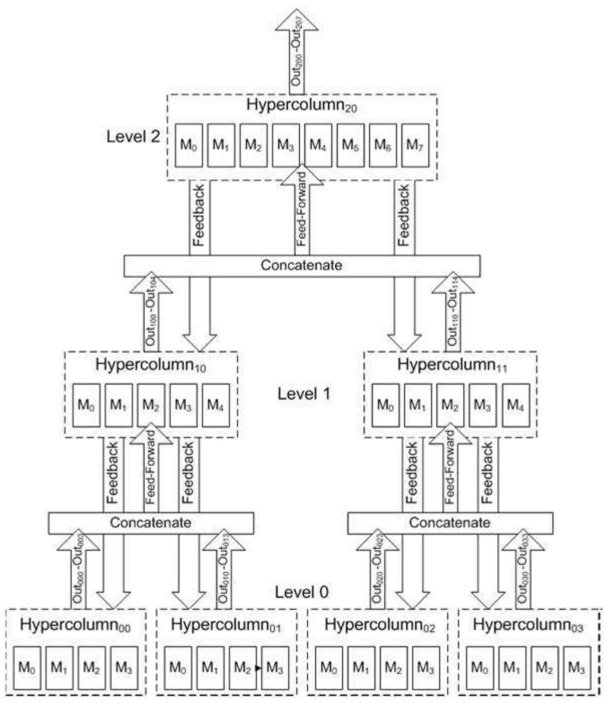

Figure 3: *A hierarchy of hypercolumns.*

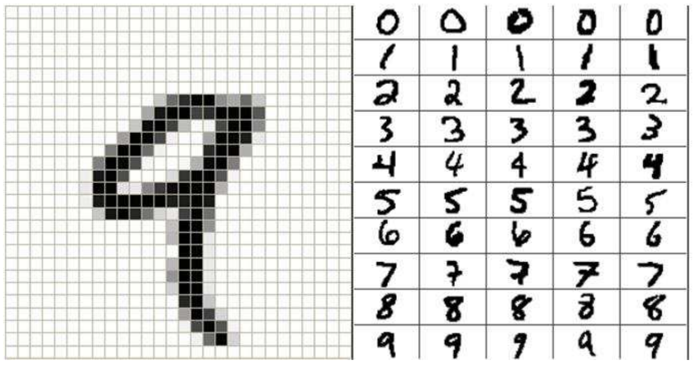

Figure 4: *Samples of handwritten digits from the MNIST database.*

redundancy and making the overall system more robust. These feedback paths are known to exist in biological neural networks for the reasons listed above [23]; we are currently working to extend our model to incorporate their functionality.

3 Related Work

As mentioned above, a wide range of research over the past 50 years has been conducted with the goals of creating an intelligent processing system modeled after the brain. Two models closely related to our cortical architecture are artificial neural networks and, more recently, hierarchical temporal memory (HTM) model.

Multilayer artificial neural networks have historically been a very popular learning model based on the properties of a neuron. In fact, these traditional perceptron-based neural networks have even been ported to the GPGPU [2, 10]. However, traditional artificial neural networks are trained for classification tasks via back-propagation; that is, the correct classification of an object is known and the weights in each layer are adjusted based on this label to minimize the classification error [21]. This form of learning is known as supervised learning. In biology, it is much more likely that learning is accomplished via unsupervised or semi-supervised learning. In unsupervised learning, labels are not provided, but classification is achieved entirely through similarity of features. In semi-supervised learning, only a few of the many objects have labels, and classification is based on similarity to the labeled objects [21]. 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 still maintain biological plausibility.

More recently, the authors of [18] have proposed a neocortex-inspired cognitive model on the Cray XD1 supercomputer. This work, based on a hierarchical Bayesian network model known as HTM proposed in [9], uses advanced software and reconfigurable hardware implementations to scale a model based on the human visual cortex to interesting problems. Like ourselves, the authors of [18] are taking advantage of a massive amount of inherent parallelism in a model based on the neocortex. However, as described above, our implementation of a neocortex-inspired model does not use Bayesian inference. Furthermore, we have opted to use commodity GPGPUs instead of a supercomputer and FPGAs to effectively scale our model toward interesting problems.

4 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. The large amount of extractable parallelism in our design makes the GPGPU an attractive hardware architecture. Particularly, NVIDIA's CUDA framework is a viable option that allows programmers to take advantage of massive amounts of parallel processing units on a commercially available graphics processor. 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 GPGPUs 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 GPGPU hardware, there is 16KB of shared memory per SM. The number of CTAs that can concurrently execute on the GPGPU 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 by each CTA [17]. CUDA applications also can be optimized by loading variables into the shared memory, taking advantage of a read-only texture cache, and optimizing global memory accesses with memory coalescing [22].

4.1 Implementing the Cortical Hierarchy on CUDA

Like the cortical architecture described in this paper, the components of the CUDA framework also are arranged in different levels of a hierarchy. The cortical architecture has minicolumns, hypercolumns, and hierarchies of hypercolumns, 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. For example, if we wanted each hypercolumn to learn 512 unique features, 512 minicolumns must exist in each hypercolumn (or 512 threads per CTA).

Considering the hierarchical structure of the cortical architecture, we realize the inputs of the upper levels depend on the outputs from the lower levels through a producer-consumer relationship. If we consider executing a cortical architecture like the one in Figure 3 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 producer-consumer data dependencies such as these, the typical solution is to execute the structure as separate CUDA-kernels [4]; that is, simply execute one level of the hierarchy on the GPGPU, return control to the CPU, and launch the next level of the hierarchy. Section 5 will detail some of the inefficiencies we discovered using this approach on this hierarchical data structure, and Sections 6 and 7 detail some of the solutions we have explored so far.

4.2 Kernel Configurations

A key design feature of the cortical architecture is that it is easily scalable to any size needed for the particular task of interest. We also make note that as the resolution of the inputs increases, the number of hypercolumns in the network increases likewise. The simplest configuration for the cortical architecture was used for this paper, where hypercolumns are arranged in a binary tree fashion like Figure 3. For the scale of inputs tested, a range of 16 to 32768 CTAs launched to simulate cortical architectures of different scales. As mentioned, the configuration of the number of threads, register file allocation, and shared-memory usage determine the number of CTAs that can concurrently execute on the GPGPU's SMs. Since our cortical architecture is highly configurable, we want to examine both the upper and lower bounds of extractable parallelism for the model. Using different configurations of the network, 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) over a wide range of image sizes.

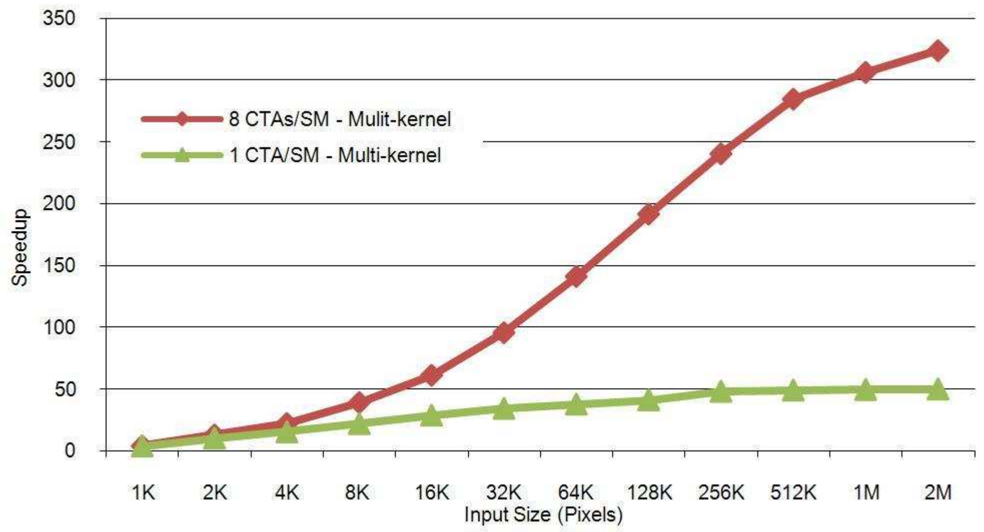

Figure 5: *Speedup achieved using multiple kernel launch CUDA model over original serial implementation.*

4.3 Results of CUDA vs. C++ Model

Figure 5 shows the performance speedups of a highly-optimized CUDA model vs. the 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 training iterations of different scale inputs ranging from 1K to 2M pixels. For reference, the MNIST handwritten database characters are 28x28 (total 784) pixels and have served as the baseline input dataset for training our model. For this paper, we padded the images to 32x32 (total 1024 or 1K) pixels. For larger scale inputs, we simply exposed the cortical architecture to multiple MNIST digits at a time.

In Figure 5, we see that the CUDA version of the cortical architecture provides significant speedups. For the 1 CTA/SM configuration, we see the achievable speedup ranges from 4x to 50x. We also notice that the maximum achievable speedup asymptotically approaches the 50x level when scaled to 256K inputs and continues for larger inputs. For the 8 CTAs/SM configuration, the speedups ranged from 4x to 320x. Once again, we notice an asymptotic trend in the speedup graph indicating that the maximum performance improvement over the serial C++ model is likely near 320x.

For the scales of inputs tested, the maximum achievable speedups range from 50x to 320x for the cortical architecture configurations we considered. While we have not implemented a parallel cortical architecture for the CPU, we believe even with such optimizations our CUDA implementation would still be an order of magnitude faster. If we utilized SSE instructions using 128-bit registers, we could potentially execute the dot-product calculations 4x faster. Furthermore, if we parallelized the C++ model we could also potentially gain a 4x speedup by distributing the cortical architecture across the four cores of the CPU, resulting in a possible 16x speedup. However, even if we consider this overhead-free perfectly optimized C++ model, our CUDA implementation would still exhibit a 3.1x to 20x speedup.

5 Difficulties with Dataflow and Hierarchical Objects 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 have large producer-consumer data dependencies, the typical solution is to separate these dependencies with multiple CUDA-kernel launches. This lock-step method, similar in nature to Bulk Synchronous Processing [25], 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 GPGPU resource utilization. The first inefficiency is that using multiple kernel launches incurs the overhead of transferring control from the GPGPU 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 6 shows the percent of execution time spent for kernel launch overhead for some of the different scale inputs and networks we simulated. We can see that 1.9 - 13% of the total execution time for a hierarchy is spent on kernel launch overhead for a hierarchy with 1 CTA/SM, and 8.1 - 13.5% for a hierarchy with 8 CTAs/SM.

The second inefficiency we observe is poor resource utilization on the GPGPU. The lowest levels of the cortical architecture have receptive fields that divide up a visual input. However, the upper levels must concatenate the information propagated by the lower levels to represent the whole picture, which means that the number of hypercolumns will decrease as we ascend the hierarchy. Using a single kernel launch per level means that the upper layers of the

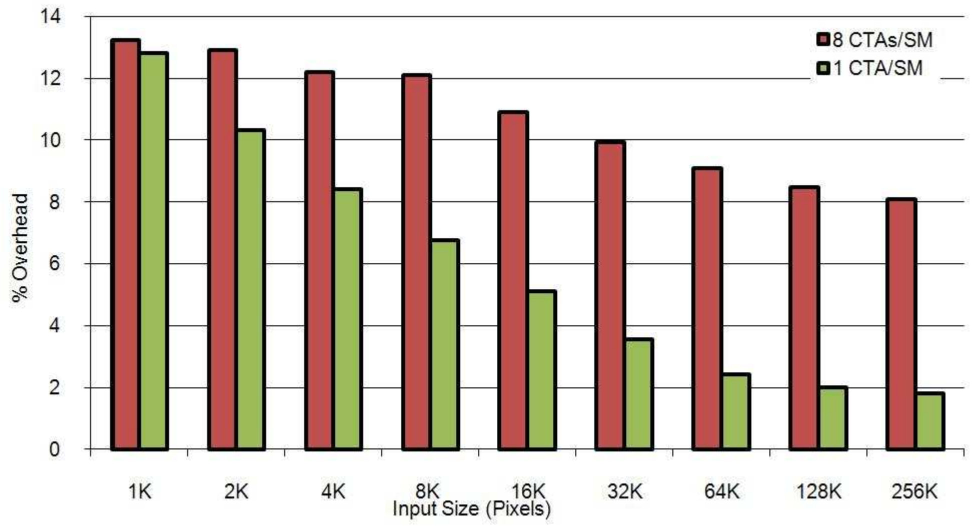

Figure 6: *Overhead of using multiple kernel launches for different scale hierarchies.*

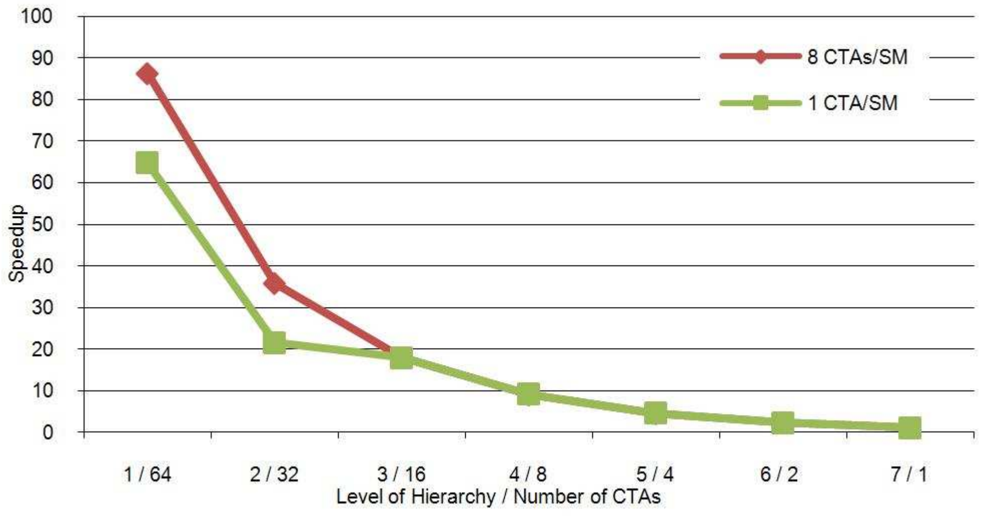

Figure 7: *Speedup of each level in a 7-level hierarchy.*

network, with very few CTAs, will utilize very little of the available resources on the GPGPU. Figure 7 considers a 7-level hierarchy capable of recognizing a 4K pixel image. 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 7 shows the level-by-level breakdown of speedups for such a network. Clearly the majority of the performance benefit is gained when there is much work to do; in our case, when there are many hypercolumns that can evaluate in parallel.

6 Kernel Fusion Using a Queue

From Figure 6 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 GPGPU concurrently, reducing the overhead to a single kernel launch. However, a limitation of the CUDA architecture is that there is no guarantee as to the order in which CTAs are scheduled or finish on the SMs [17]. Furthermore, once a CTA is scheduled on an SM, it will run until completion, which means there are no sleep-wait primitives to allow a hypercolumn to wait until the inputs produced by 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 GPGPU's global memory space, as can be seen in Figure 8. This work-queue method operates as follows: a single CUDA-kernel is launched with only as many CTAs as can concurrently fit across all of the SMs in the GPGPU (Figure 8 shows 8 concurrent CTAs per SM). 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 and the work-queue is empty. Each CTA first uses an atomic primitive to gain a unique index into the work-queue (solid red arrows). Using the index, the CTAs access a unique element of the work queue (dashed blue arrows). 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 (dashed brown arrows). Finally, the CTA atomically 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, it will wait to execute until each of its child hypercolumns have finished execution and indicated so via the set flags. Once the queue is empty, the kernel returns control to the CPU.

We enhanced the work-queue method to reduce the total number of global reads and writes performed during the cortical architectures execution. Specifically, we made the observation that we could reduce the number of global

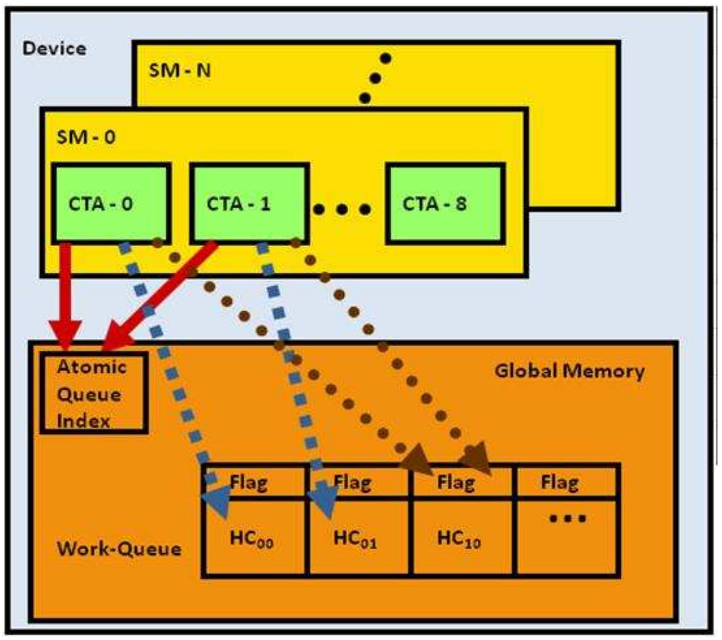

Figure 8: *The implementation of the work-queue. CTAs properly index into the work-queue in global memory using atomic an atomic increment to a single counter.*

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 3. 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 read to the flag (see Figure 8). If the peer hypercolumns have completed, the CTA can simply begin execution of $Hypercolumn_{10}$ rather than fetching a hypercolumn off of the work-queue. Using this optimization only the data produced by $Hypercolumn_{01}$ must be read from the global memory since the data provided by $Hypercolumn_{00}$ is already in the shared memory. The major advantage here is that the overall bandwidth consumption is reduced by limiting the number of transfers to global memory.

Figure 9 and 10 show the speedups of both opt-queue and the standard work-queue normalized to the baseline CUDA (multiple kernel) implementation. We also compare against the speedup of the CUDA baseline minus the

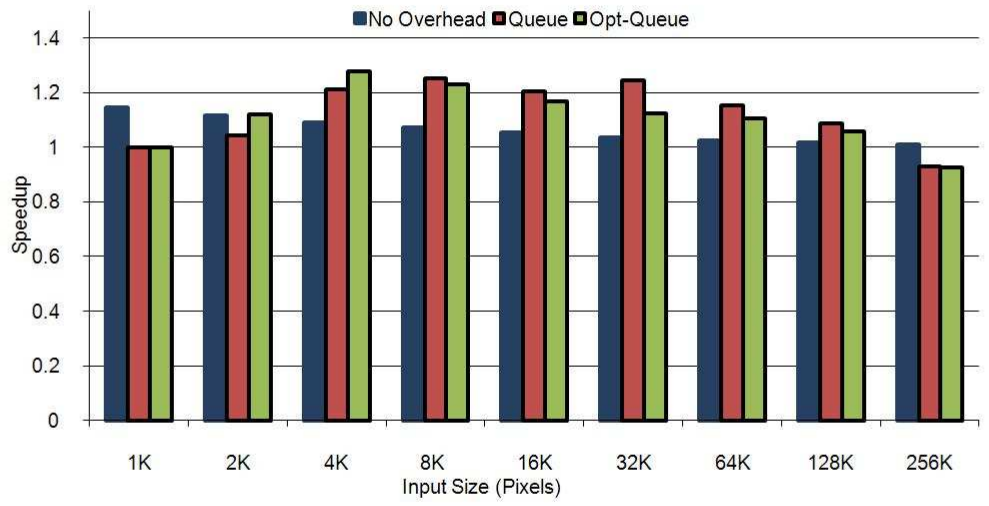

Figure 9: *Queue speedup for 1 CTA/SM.*

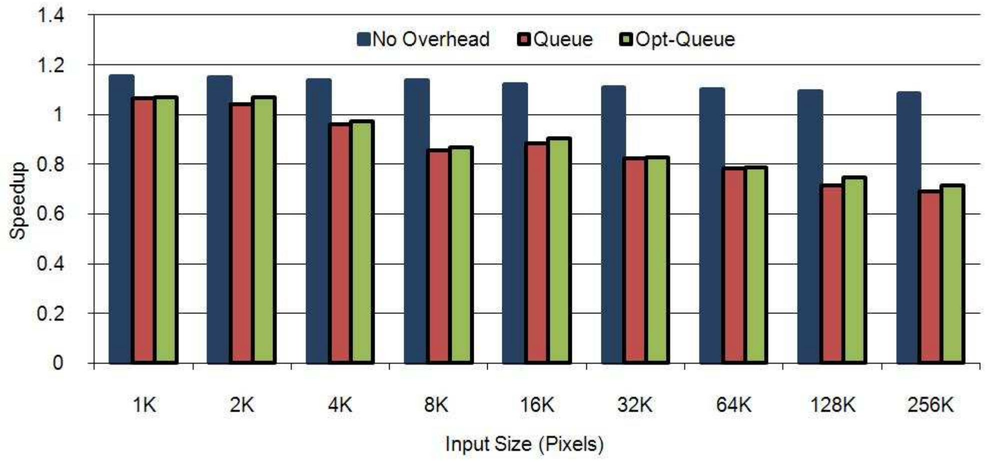

Figure 10: *Queue speedup for 8 CTAs/SM.*

kernel launch overhead (No Overhead) which we observed in Figure 6. For cortical architectures configured to occupy one CTA per SM (Figure 9), we see that the work-queue and opt-queue implementations beat the baseline configuration for cortical architectures scaled between 2K and 128K pixel inputs. They also outperform the baseline minus the kernel-launch overhead for the 4K and 128K pixel input. The queues have both removed the kernel launch overhead and better utilized the GPGPU's resources by executing the entire cortical hierarchy in a single kernel launch. For networks scaled to 256K inputs and beyond, both queue implementations perform worse than the baseline implementation.

Figure 10 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 only perform better than the baseline implementations for the smallest scale networks. However, for all other scale networks the queue implementations perform worse than the baseline. 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 to create a unique index into the queue are actually quite slow, typically on the range of 400-600 cycles [17]. Not only are these memory accesses slow, but we also encounter an effect known as partition camping [20]. Partition 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, with 8 CTAs per SM, we are performing 8x as many concurrent atomic-accesses to the work-queue. The cumulative latencies of the global memory accesses in turn result in longer latencies than the multiple kernel-launch overhead.

7 Pipelining to Increase Resource Utilization

From Figure 7, we are able to see how the hierarchical design of our cortical architecture results in poor utilization of the GPGPU's resources for the upper levels of a tree-like structure. 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 we simulated. However, since upper levels of the hierarchy have fewer hypercolumns to evaluate, it is often the case we have less work than actual resources. When this point is reached, the benefit of using the GPGPU quickly tapers 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 levels of the hierarchy.

Our solution is to simply pipeline the training phase of the cortical architecture. By training, we mean exposing the cortical architecture to a set of training inputs so that it may learn a set of unique features. For example, the MNIST database contains 60,000 handwritten characters for training a particular learning algorithm and 10,000 characters for testing and evaluation. In the pipelined training method, a single kernel-launch executes all hypercolumns

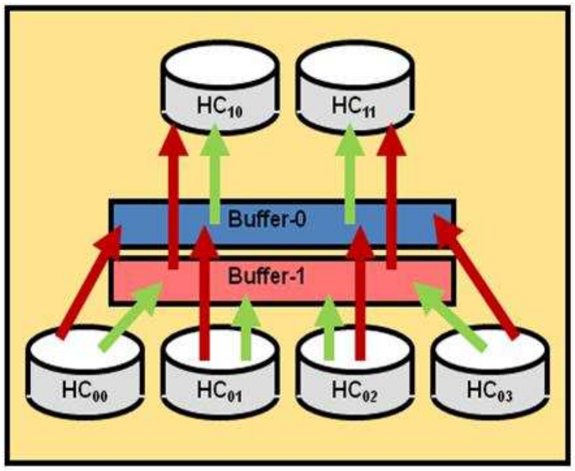

Figure 11: *Separate buffers are read-from and written-to between levels on a particular kernel launch.*

in the hierarchy, and we use multiple buffers between each hierarchy level to guarantee that the producer-consumer relationship is enforced between different levels. Figure 11 shows pipelining between two stages of a cortical architecture hierarchy. On the first kernel launch, the outputs from the lower level hypercolumns will be placed in Buffer-0 (red arrows). On the same kernel launch, the hypercolumns in the upper level will read their synaptic inputs from Buffer-1 (red arrows). On the next kernel launch, the lower level hypercolumns will write to Buffer-1 and the upper level will read from Buffer-0 (green arrows). This buffering allows us to execute all the hypercolumns (CTAs) in a single kernel launch while guaranteeing that the correct flow of data will propagate from the lower levels of a cortical architecture to the upper levels. While this method better utilizes the GPGPU resources and also improves training throughput, it still takes multiple kernel launches for any particular training input to fully propagate the entire way through the hierarchy (seven kernel launches in the case of our example of a 7 level hierarchy). However, considering a training set like the MNIST database with 60,000 training input images, clearly this pipelining can speed up the training phase.

Figure 12 shows the speedups of the pipelined version compared to the multiple kernel-launch implementation. For the cortical architectures scaled for 8 CTAs/SM, the achievable speedup is 330x over the serial C++ implementation. Relative to the multiple-kernel CUDA implementation, the pipelined implementation achieves maximally a 3.5x speedup for some of the scale inputs tested, though again we see the maximal performance asymptotically approaches a 330x speedup over the C++ model. When scaled to 1 CTA/SM, the maximum achievable speedups were 2.55x over the multiple-kernel CUDA implementation. Once again, the pipelined version asymptotically ap-

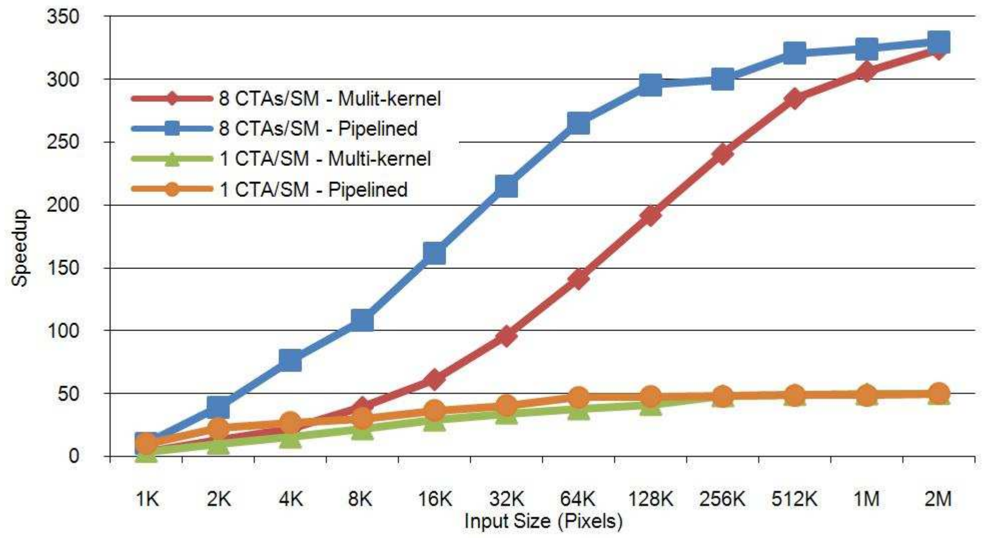

Figure 12: *Speedups achieved by using the pipelined optimization. The same peak speedups are achieved, though pipelined optimization reaches peak performance for much smaller cortical architectures.*

proaches a 51x speedup. For both of the cortical architecture configurations we consider, the maximum performance improvement is comparable between the pipelined and non-pipelined implementations; however, the pipelined implementations simply reach this asymptote earlier (at smaller scale) and better utilize the GPGPU's resources.

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 evaluation phase for the cortical architecture. During the testing phase of the cortical architecture, we may use aforementioned feedback connections from the upper levels of the hierarchy back down to the lower levels for object invariance in the presence of noisy data. 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.

8 Other Applications

For the cortical architecture explored in this paper, the work-queue and pipelining methods were reasonable solutions to alleviate the associated kernel-launch overhead and poor utilization of GPGPU resources. Here we consider two other applications which we investigated that have a similar hierarchical data dependency structure to our cortical architecture: a parallel reduction and a multilevel feedforward neural network.

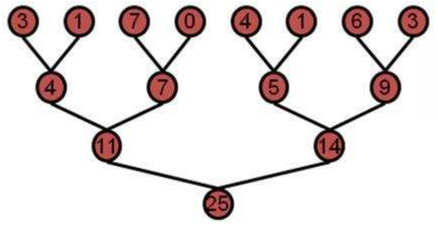

Figure 13: *Parallel reduction exhibits a tree-like structure for data communication.*

8.1 Parallel Reduction using Pipelining

The CUDA SDK features several variations of a parallel reduction algorithm able to process large arrays of elements [6]. This parallel reduction was designed as a benchmark for observing a GPGPU's effective bandwidth. However, a typical method for implementing a parallel reduction is to split the data and perform sub-calculations on each partition. Using a tree-based approach allows the algorithm to continue computing partial results in parallel until the algorithm has completed, like Figure 13. Like our cortical architecture, we again see that the amount of available parallelism reduces as we traverse through the levels of the tree. This application also suffers from underutilizing the available GPGPU computation resources. Here, the pipelining solution described in Section 7 can again help us to increase resource utilization and potentially offer performance benefits. Once again, we use multiple buffers between each level of the parallel reduction to ensure the correct flow of data without. Depending on user-defined inputs such as threads/CTA and input array size, the parallel reductions we tested used three or four kernel launches.

Figure 14 shows the performance comparison of the pipelined and original parallel reduction implementations for different input sizes as well as thread/CTA configurations. We see here that the execution time is reduced by 10 - 21% for the range of inputs and configurations tested. Pipelining this parallel reduction did not result in the same type of speedups achieved for many of the configurations of the cortical architecture. However, if we examine the breakdown of the execution time for the baseline parallel reduction, we see that nearly all of the execution time is spent on the first level of the tree. This particular application has been highly optimized to efficiently perform most of the work in the first kernel launch. However, for our algorithm, it is more important to distribute the computations more evenly between levels. The fan-in to each node in the tree ranged from 64 to 1024 for the configurations we tested, much higher the two node fan-in of the cortical architecture. Intuitively, we realize that our pipelined

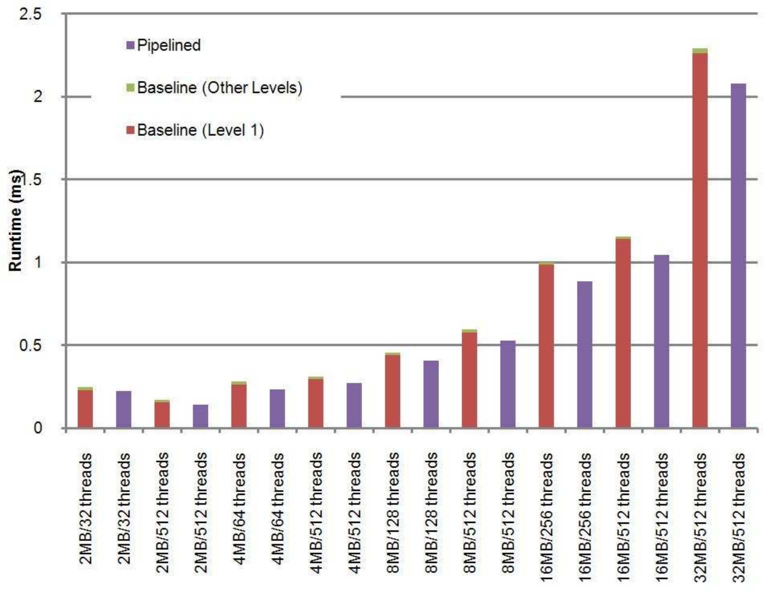

Figure 14: *Performance of pipelining parallel reduction.*

implementation should not be able to execute any faster than the slowest layer of the parallel reduction, minus the multiple kernel launch overheads. With these considerations in mind, we see that the pipelined implementation of the parallel reduction does perform better than the first level of the reduction minus the kernel launch overheads for all of the inputs and configurations we tested.

Although pipelining clearly shows some benefit in terms of performance and resource utilization, the work-queue implementations did not provide any benefit for this application. Again, we hypothesize that the large number of atomic operations needed to properly index the queues are to blame for a 2x to 6x slowdown over the baseline parallel reduction.

8.2 A.Feedforward Neural Network using Pipelining

Another application that exhibits the same kind of producer-consumer hierarchy is a feedforward artificial neural network. In a traditional feedforward neural network, there is an input and an output layer with one or more hidden layers between them. Figure 15 shows a simple four-layer neural network based on the perceptron model. We can clearly see how the hidden layer perceptrons are dependent on the outputs from the input layer, and the output layer perceptrons are dependent on the hidden layers. For this paper, we consider a fully trained four-layer neural network on CUDA from [2] which uses a separate kernel launch for each layer. Furthermore, the number of CTAs and threads in each layer has been optimized specifically for the task of recognizing handwritten digits from the MNIST database. Figure 16 shows the execution time of the baseline, pipelined, and work-queue implementations. The Figure also details the number of CTAs used per level (L1-L4) and shows the breakdown of execution time per layer for the baseline. Layer 4 and Layer 1 execute only 10 and 6 CTAs respectively, far below the possible 30 to 240 concurrent CTAs that can execute on our GPGPU.

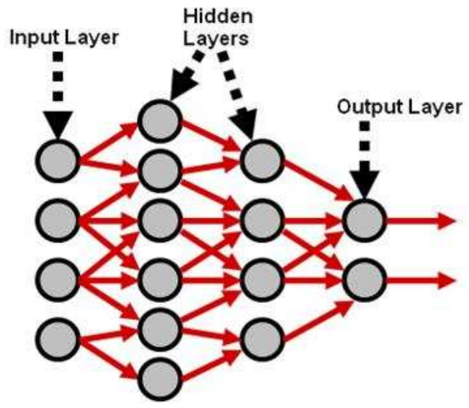

Figure 15: *A multilayer artificial neural network.*

Again, pipelining the neural network has the potential of improving resource utilization as well as performance. However, using the optimizations we have discussed in this paper we again realize an applications execution time can only be reduced to the execution time for the longest kernel. From Figure 16, we see that Layer 3 makes up nearly 80% of the total execution time of the neural network. Once again, our cortical architecture more evenly distributes the execution time between the multiple levels, but this neural network has been optimized to use three relatively quick kernels and a long kernel where most of the computations are performed.

In the pipelined implementations, we examined the execution time for pipelining the entire neural network as well as pipelining just the top three and top two layers. We are able to reduce the execution time 9 - 14% by pipelining these configurations. However, for the work-queue implementation, we again notice a drastic decrease in performance, nearly a 3x slowdown.

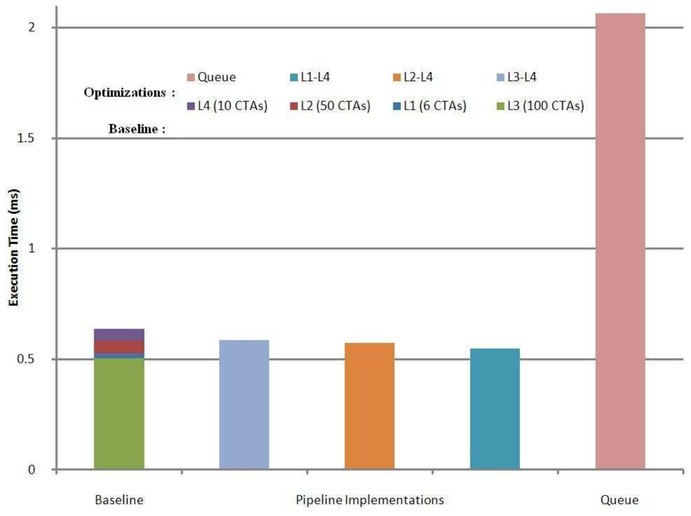

Figure 16: *Execution time of multilayer neural network.*

9 Discussion

While the results in this paper indicate that NVIDIA's CUDA is an excellent framework for our cortical architecture, the optimizations we explored were an attempt to maximize performance and satisfying producer-consumer relationships in a hierarchical structure. Although we explored only a few applications, one could find other instances where a programmer would want to utilize the massive parallelism of CUDA and also have faster synchronization primitives between blocks. Aside from using hand-coded atomic primitives and the global work-queue structures we described in this work, we have not found any other methods for reliably synchronizing CTAs, other than breaking data dependencies into separate kernel launches. Although using atomic global memory accesses allow us to achieve correct synchronization for our data structure, we also see variable performance for this method. Clearly the performance degrades as the number of atomic operations increases, which is intuitive considering these accesses are to a single location in slow global memory. While the CUDA API provides a synchronization primitive between the threads of a CTA (__synchthreads()), we believe that applications such as our own would greatly benefit from similar easy to use synchronization primitives across CTAs.

From the results of our work with these optimizations, we make some generalizations as to the types of applications that fit well to solutions such as the work-queue or pipelining:

- The shape of the application : The types of solutions we presented in this paper are for applications that have strict producer-consumer data dependencies, such as any kind of hierarchy or tree-like structure.

- The homogeneity of the application : In current generation CUDA hardware, only a single concurrent kernel can be executed on the hardware. Therefore, the algorithm used across each level of the applications structure must be homogeneous in terms of the executable kernel code, size of CTAs, number of threads, etc.
- The length of kernel code : For strict data dependencies like those we consider, ordering is vital, and although CUDA doesn't allow explicit order of CTAs, atomic primitives can be used. However, these atomic primitives have long access times, so the amount of computation must be high relative to the cost of using atomic operations to index into a work-queue structure.
- The scale of the application : As the number of CTAs increases in each level of a hierarchy, the overall percentage of time spent in kernel launches decreases and the GPGPU resource utilization increases. The solutions we propose will only result in improvement for applications where the kernel launch overhead or resource underutilization are significant.

$$\#GPGPUs = \frac{\#Neurons \times \#ofMultAdds/Minicolumn \times NeuronFiringRate}{240Cores/GPGPU \times 100Neurons/Minicolumn \times CoreFrequency} \tag{3}$$

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 1 shows our estimates for the number of GPGPUs needed to fully realize the real-time compute capability of different scales of mammalian brains [19]. We consider the current top-of-the-line GPGPU available, the Tesla C1060, with 240 cores @ 1.3 GHz. The estimated number of GPGPUs for real-time performance is given by Equation 3. We assume that biological neurons have a maximum firing rate around 10 ms [13]. We also consider that each of our models minicolumns must perform a multiply-and-add operation per unique synaptic input, estimating 10,000 synapses per minicolumn [11].

Table 1: Estimated number of GPGPUs needed to model mammalian cortexes.

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

10 Conclusion

In this paper we have described a GPGPU parallelization extension to an intelligent system based on the mammalian neocortex. Using CUDA, we achieved 4x to 320x speedups across different scale cortical architectures over 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 330x speedup over the C++ implementation. We then examined using these same solutions two other hierarchy-structured applications. Finally, we discussed some generalizations of applications that would possibly benefit from using similar solutions to alleviate multiple-kernel launch overhead and resource under-utilization.

In the future we plan to further extend this model to include the feedback connections described in Section 2. Emulating these features of the human neocortex, we believe our model will accurately store invariant representations of objects and be capable of recognizing noisy, translated, and rotated images. Furthermore, once these functional connections have been established we plan to test the accuracy of our cortical architecture using the full MNIST database. With these established properties of the neocortex implemented in our model, we believe it will be competitive at this recognition task against other such learning models.

With difficulties surrounding programming of parallel architectures, the likelihood of faulty devices, and excessive power consumption, computer architects may need to consider alternatives to traditional computer architectures. With these concerns in mind, we consider modeling an architecture that has proven to be quite effective in spite of similar limitations: the neocortex. 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 significant step towards realistic intelligent computing machines.

References

[1] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. *SC Conference*, 0:1–11, 2008.

[2] Billconan and Kavinguy. A neural network on gpu.

[3] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, October 2008.

[5] K. Grill-Spector, T. Kushnir, T. Hendler, S. Edelman, Y. Itzchak, and R. Malach. A sequence of object-processing stages revealed by fmri in the human occipital lobe. *Hum. Brain Map.*, 6:316–328, 1998.

[6] M. Harris. Optimizing parallel reduction in cuda. 2007.

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

[8] 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.

[9] J. Hawkins and S. Blakeslee. *On Intelligence*. Henry Holt & Company, Inc., 2005.

[10] H. Jang, A. Park, and K. Jung. Neural network implementation using cuda and openmp. In *DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pages 155–161, Washington, DC, USA, 2008. IEEE Computer Society.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] A. Nere and M. Lipasti. Cortical architectures on a gpgpu. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 12–18, New York, NY, USA, 2010. ACM.

[17] NVIDIA. *CUDA Programming Guide*. NVIDIA Corporation, 2701 San Toman Expressway, Santa Clara, CA 95050, USA, 2007.

[18] K. L. Rice, T. M. Taha, and C. N. Vutsinas. Scaling analysis of a neocortex inspired cognitive model on the cray xd1. *J. Supercomput.*, 47(1):21–43, 2009.

[19] G. Roth and U. Dicke. Evolution of brain and intelligence. *TRENDS in Cognitive Sciences*, 5:250–257, 2005.

[20] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. January 2009.

[21] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[23] A. Sillito, J. Cudeiro, and H. Jones. Always returning: feedback and sensory processing in visual cortex and thalamus. *Trends Neurosci.*, 29(6):307–316, Jun 2006.

[24] L. Swanson. Mapping the human brain: past, present, and future. *Trends in Neurosciences*, 18(11):471 –474, 1995.

[25] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.