Simulating Cortical Networks on Heterogeneous Multi-GPU Systems

Andrew Nere, Sean Franey, Atif Hashmi, Mikko Lipasti

Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA

Abstract

Recent advances in neuroscientific understanding have highlighted the highly parallel computation power of the mammalian neocortex. In this paper we describe a GPGPU-accelerated implementation of an intelligent learning model inspired by the structural and functional properties of the neocortex. Furthermore, we consider two inefficiencies inherent to our initial implementation and propose software optimizations to mitigate such problems. Analysis of our application's behavior and performance provides important insights into the GPGPU architecture, including the number of cores, the memory system, atomic operations, and the global thread scheduler. Additionally, we create a runtime profiling tool for the cortical network that proportionally distributes work across the host CPU as well as multiple GPGPUs available to the system. Using the profiling tool with these optimizations on Nvidia's CUDA framework, we achieve up to 60x speedup over a single-threaded CPU implementation of the model.

Keywords: cortical learning algorithms, CUDA, GPGPU, profiling systems

1. Introduction

Computation models based on the structural and functional properties of the human brain have seen some impressive advances over the past several years. As neuroscience and neurobiology have made many significant discoveries about the workings of the mammalian brain, these learning models have benefited from incorporating the properties that make the brain a robust and powerful parallel processing system. One of the major burdens of these biologically plausible models is their massive computational demands. Simulating a large network of neurons, regardless of algorithmic simplicity, may take hours or days of execution time. However, the inherent nature of these biologically plausible computational models makes them quite parallel in structure. Once effort has been placed to parallelize such algorithms, it becomes relatively straightforward to map them to GPGPUs, which provide massive amounts of parallel hardware at modest expense.

Hashmi et al. propose an intelligent system design inspired by the mammalian neocortex [1, 2]. One of the interesting aspects of this model is that instead of modeling individual neurons, it models cortical columns as the basic functional unit of the neocortex [3]. The properties incorporated in this learning algorithm implement a biologically plausible model of the visual cortex without requiring the computational complexity of modeling individual neurons. Nere et al. extend this neocortex-inspired architecture to a single GPGPU to achieve a significantly faster version of the algorithm [4, 5].

In this paper, we extend the work of Hashmi [1, 2] and Nere [4, 5] to benefit from multiple CUDA-enabled GPGPUs. This extended model distributes a hierarchically organized cortical network across a single CPU and one or more heterogeneous or homogeneous GPGPUs. In the context of this paper, we refer to a collection of identical CUDA-enabled Nvidia devices as *homogeneous GPGPUs*, while a collection of *heterogeneous GPGPUs* may span different architecture generations, core counts, and memory capacities (though they still must be CUDA-enabled Nvidia devices). Using intelligent profiling techniques along with a heuristic to estimate the throughput of the available GPGPUs, our model is able to proportionally distribute cortical columns across its available resources to achieve impressive speedups. We also analyze performance limitations encountered in porting this learning algorithm to the GPU framework. To mitigate these limitations, we propose optimizations that prove effective in both the single and multi-GPU domains. As a result, we achieve up to a 60x speedup over a single-threaded CPU implementation of the algorithm.

The main contributions of this paper are as follows.

- We investigate in detail the performance of the cortical network algorithm along with our proposed optimizations.
- These findings provide important insights into the GPU architectures' details, including the number of cores available, the memory system, and the global thread scheduler.

Email addresses: nere@wisc.edu (Andrew Nere), sfraney@wisc.edu (Sean Franey), ahashmi@wisc.edu (Atif Hashmi), mikko@engr.wisc.edu (Mikko Lipasti)

- To the best of our knowledge, this is also the first work that effectively demonstrates using a profiling tool to automatically distribute and concurrently execute an algorithm proportionally across heterogeneous GPUs and a host CPU.

The rest of this paper is organized as follows: Section 2 provides brief but relevant information regarding the neocortex, and Section 3 describes the cortical learning algorithm modeled after it. We discuss some related work on creating biologically inspired computing models, as well as their implementations on GPGPUs, in Section 4. Section 5 describes the methods used to extend our cortical networks to the GPGPU using CUDA and presents some initial performance results. Section 6 examines some of the bottlenecks encountered with our initial GPGPU implementation of the cortical network and proposes two optimizations to mitigate such inefficiencies. In Section 7, we extend our GPGPU implementations to the multi-GPU domain, using online profiling to efficiently distribute a cortical network across a heterogeneous or homogeneous collection of GPUs and the host CPU. We examine the results of our optimizations and multi-GPU implementation in Section 8. In Section 9, we examine two other applications with similar bottlenecks and attempt to alleviate them with our solutions. Finally, Section 10 provides a discussion, and Section 11 concludes the paper.

2. Cortical Structures and Operations

The neocortex is the part of the brain that is unique to mammals and is mostly responsible for executive processing skills such as mathematics, music, language, vision, perception, etc. The neocortex comprises around 77% of the entire human brain [6]. For a typical adult, it is estimated the neocortex has around 11.5 billion neurons and 360 trillion synapses, or connections between neurons [7]. Mountcastle was the first to observe the structural uniformity of the neocortex. He proposed that the neocortex is composed of millions of nearly identical functional units which he termed *cortical columns* because of the seemingly column-shaped organizations of neurons exhibiting similar firing patterns for a given stimulus [3]. Hubel et al. [8] and Mountcastle [9] further classified cortical columns into *hypercolumns* and *minicolumns*. Individual hypercolumns are composed of smaller structures called 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 [10]. Using these connections, a minicolumn is able to alter the synaptic weights of the neighboring minicolumns to influence learning, typically to identify unique features stimulating the receptive field of the hypercolumn [10]. Figure 1 shows a typical arrangement of minicolumns within a hypercolumn.

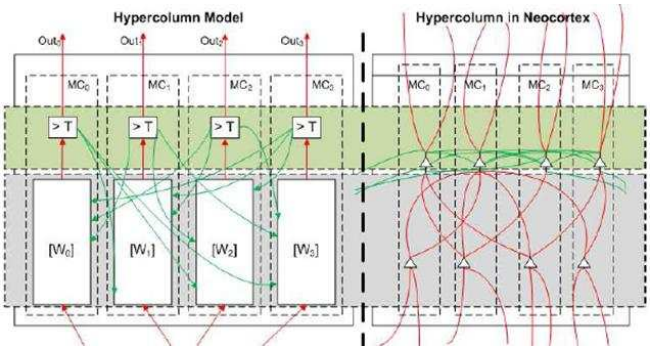

Figure 1: *Right: Biological representation of a hypercolumn, with lateral connections representing local inhibition. Left: Model's representation of a hypercolumn with their corresponding connections and weight vectors W_i.*

3. A Biologically Plausible Model for Cortical Architecture

In this work, we extend the cortically inspired computational model proposed by Hashmi et al. [1, 2]. The traditional approach of Artificial Neural Networks (ANN) is to seek inspiration from biology by modeling neurons, though often such designs depart from biological plausibility due to application requirements. On the other hand, the cortical learning algorithm we investigate reverses this priority: biological plausibility is prioritized to retain its capabilities, even if less natural but more application specific methods can more easily achieve the same task. This motivation anticipates that staying close to biology is the key to developing powerful and robust computational models.

Historically, different levels of abstraction have been used in pursuit of modeling intelligent systems. Some of these models attempt to emulate the brain at a very high level based on behavior and Bayesian inference, while the other end of the spectrum models the brain at the level of highly detailed neuron models, neural conductances, and ion channels. In this paper, we extend a computational model that is highly motivated by the properties and structure of cortical columns. By using cortical columns as the level of modeling abstraction, our model can avoid the computational complexity of a neuron-level model while remaining grounded in biological realism. Furthermore, in this paper, we evaluate and discuss our model in the context of visual cortex for two main reasons. First, the visual cortex is a part of the brain that has been extensively studied by the neuroscience community, and its functionality and organization are described by a large body of literature. Second, to test our algorithm, we use images of handwritten digits obtained from MNIST database (http://yann.lecun.com/exdb/mnist).

3.1. Input

In case of the mammalian visual cortex, the responses of retinal cells are transferred to the Lateral Geniculate Nucleus (LGN) cells via nerve paths [11] . LGN cells detect *contrasts*: they react strongly to an illuminated point

surrounded by darkness (on-off cells) or conversely to a dark point surrounded by light (off-on cells). These LGN cells are spatially distributed with on-off and off-on cells intertwined, roughly operating like a pixel sensor [12]. Input images are processed using the LGN transform before they are fed into the actual model. For the model described in this paper, we consider a regular spatial distribution of LGN cells (one on-off and one off-on per pixel), but we have also experimented with more random distributions without noticeable differences. So far, we have found the most important factor is the spatial density of LGN cells with respect to the image resolution.

3.2. Cortical Column Connectivity and Algorithm

Figure 1 provides an overview of our implementation of a hypercolumn. Within a hypercolumn there are multiple minicolumns that are connected to each other via lateral inhibitory paths. The minicolumns within a hypercolumn are part of a strongly connected competitive learning network. Through the lateral inhibitory connections, the minicolumn with the strongest response inhibits its neighbors from firing for the same input pattern. These lateral inhibitory connections ensure a biologically plausible winner-take-all competition between neighboring minicolumns. Over time each of the minicolumns starts to recognize independent features stimulating the receptive field of the hypercolumn. Activity of a minicolumn depends on two factors: its inputs weighted by the corresponding synaptic weights, or a small probability of random activations (refer to Section 3.4). Formally, the output of a minicolumn with a synaptic weight vector W in response to an input vector x is given by the nonlinear activation function described by Equation 1.

$$f(x) = \frac{1}{1+e^{-g(x)}} \quad (1)$$

$$g(x) = \Omega(W) \times (\Theta(x, W, \tilde{W}) - T) \quad (2)$$

$$\tilde{W} = W/\Omega(W) \quad (3)$$

$$\Omega(W) = \sum_{i=1}^{N} C_i W_i \quad (4)$$

$$C_i = \begin{cases} 1.0, & \text{if } W_i > 0.2 \\ 0.0, & \text{otherwise} \end{cases} \quad (5)$$

$$\Theta(x, W, \tilde{W}) = \sum_{i=1}^{N} \gamma(x_i, W_i, \tilde{W}_i) \quad (6)$$

$$\gamma(x_i, W_i, \tilde{W}_i) = \begin{cases} \text{-2}, & \text{if } x_i = 1.0 \text{ and } W_i < 0.5 \\ x_i \tilde{W}_i, & \text{otherwise} \end{cases} \quad (7)$$

T in Equation 2 determines the tolerance of a minicolumn to noise. Experimentally the value of T was set to 0.95 for the simulations presented in this paper. The weight vector W is initialized to random values close to 0, suggesting that there is no initial feedforward connectivity within the network. Typical ANN models define the input of the activation function simply as $\sum x_i W_i$. However, in our model, Equation 7 can be seen as a reflection of a non-linear activation function. If W_i corresponding to an active input x_i is low, W_i contributes negatively to the input of the activation function. Within the neocortex, these non-linear summation properties have been observed in some dendrites [13]. We empirically observed this non-linearity to be necessary for proper functional behavior of our hypercolumn model.

3.3. Synaptic Weight Update Rule

Hebbian learning [14] is a dominant form of learning in large-scale biological neural networks. With Hebbian learning, if one input of a neuron has strong activation, and that neuron itself has a strong output, then the synapse (synaptic weight) corresponding to that input is reinforced. Intuitively, if the input is strong at the same time as the output, it means that input plays a significant role in the output and should be reinforced. According to this definition, the synaptic weight W_i is increased if the input x_i to the minicolumn is active (emulating long-term potentiation), or decreased if the input x_i to the minicolumn is inactive (emulating long-term depression). It should be noted that these weight modifications are in accordance to Hebbian learning and are applied only to those minicolumns that have strong output activations. As a result, minicolumns will progressively react most strongly to inputs they receive repeatedly, in effect *learning* them. In the visual cortex, these inputs correspond to images or features of images.

3.4. Learning Via Random Firing and Repeated Exposure

Since all minicolumns in a hypercolumn share the same receptive field, the main distinction among these minicolumns rests in their connectivity. Connectivity can be modeled through the value of synaptic weights (as a 0-weight synapse is equivalent to no connection). Initially, there is no specific connectivity among hypercolumns as all the synaptic weighs are initialized to random values that are very close to 0.

We propose that random firing behavior of minicolumns results in establishing initial connectivity between hypercolumns. At each time step, every minicolumn has a small probability to become active, even if its inputs do not justify it. When the random firing coincides with a stable input activation, the synaptic weights corresponding to that activation are reinforced. Thus, over time, connectivity between hypercolumns is established. Instead of having predefined connections between various minicolumns, connectivity is steered by the input patterns stimulating the hierarchical network. The random firing of a minicolumn stops when it has been continuously active for a significant period of time. Empirically we have observed that this random firing behavior allows a great variety of features to be learned by a hypercolumn. However, it is also necessary that such random firing behavior decays as each minicolumn converges on a single unique feature.

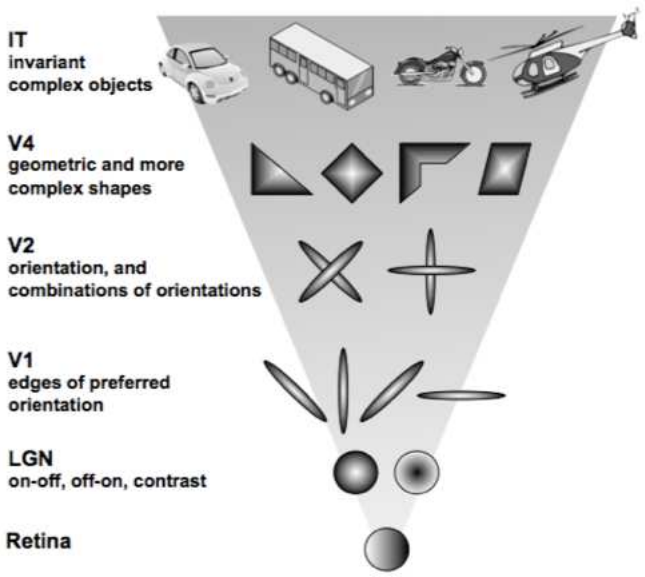

Figure 2: *The visual cortex is known to have a hierarchical organization. Neurons in higher levels respond to more complex and invariant visual stimuli. Feedforward and feedback paths are necessary for communication in the visual cortex hierarchy.*

This random firing behavior is an essential part of our learning model, and we provide the following as a justification of its biological plausibility. Neurons receive synaptic inputs from all types of connections: forward, lateral, feedback. As long as the forward synapses are weak, the combination of these inputs creates a *synaptic noise*, akin to random firing. When the forward connections become strong (because the neuron has *learned* a feature), they become dominant and the neuron output is no longer affected by the remaining synaptic noise [15]. As a result, the random activity caused by synaptic noise no longer has a significant impact.

3.5. Cortical Column Hierarchy

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 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. In case of the visual cortex, at the lowest level (V1), minicolumns learn to identify very simple features, such as edges of a preferred orientation. Thereafter, subsequent levels learn to recognize more complex shapes (V2, V4), while the upper level of the hierarchy (IT) ultimately recognizes the object under focus with invariant representation [16].

Our cortical network model uses this hierarchical design to accomplish complex tasks. Figure 3 shows an example of a three level hierarchical cortical network. 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

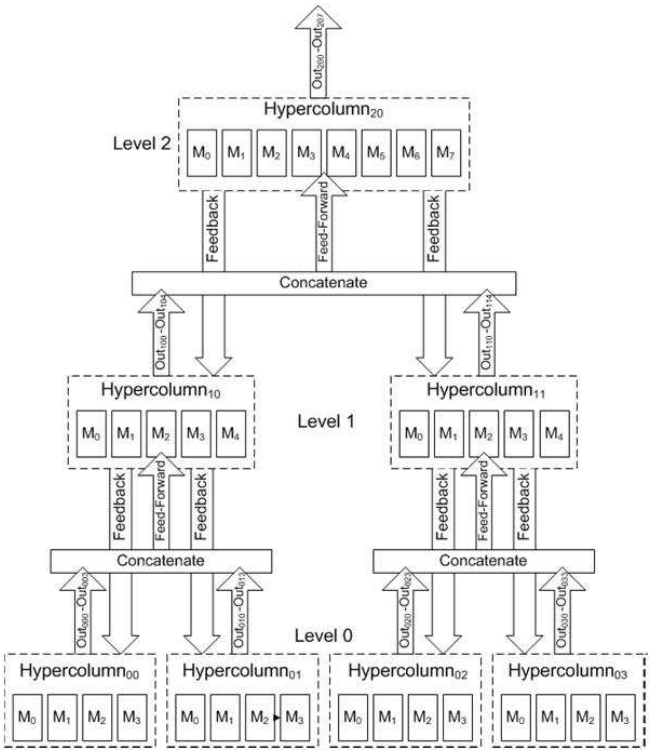

Figure 3: *The cortical network is organized as a hierarchy of hypercolumns with corresponding feedforward and feedback connections.*

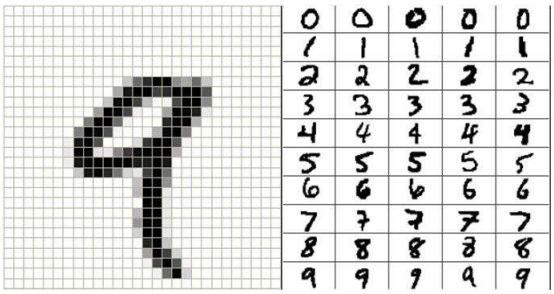

Figure 4: *Left: A sample of a visual recognition task (the digit 9 from the MNIST database). Right: Other example handwritten digits (lower resolution).*

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 network. 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 networks.

Finally, Figure 3 also shows that feedback paths from higher levels of the cortical network 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 represen-

tation can be stored in the cortex rather than all the variations of a particular stimulus, reducing unnecessary redundancy and making the overall system more robust. These feedback paths are known to exist in biological neural networks for the reasons listed above [17]; we are currently working to extend our model to incorporate their functionality. However, in this paper we consider and model the feedforward and lateral paths only, which are capable of many unsupervised learning tasks.

4. Related Work

A wide range of research over the past decades has been conducted with the goals of creating an intelligent processing system modeled after the brain. Some models closely related to our cortical algorithm are ANNs and, more recently, deep unsupervised learning algorithms and the hierarchical temporal memory (HTM) model. While many of these models claim to be biologically plausible, it is often the case that their learning and connectivity rules are quite far from their biological inspiration.

Multilayer ANNs have historically been a very popular learning model based on the properties of a neuron. However, traditional ANNs 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 [18]. This form of learning is known as supervised learning. While evidence exists that a certain degree of supervised learning occurs in biological systems, such back-propagation does not regulate the typical small changes in connectivity and synapse strength in the brain. 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 [18].

These traditional perceptron-based ANNs have even been ported to the GPU with some success [19, 20]. Nages et al. have simulated thousands of spiking neurons on the GPU, taking advantage of such optimizations as memory coalescing and achieving up to a 26x speedup [21, 22]. Raina et al. have implemented deep unsupervised learning algorithms on a GPU with 5-15x speedups [23]. The cortical network algorithm we consider here is able to learn features from its dataset in an entirely unsupervised fashion, though extensions to the model utilize a semi-supervised learning paradigm somewhat similar to self-organizing maps. We also consider that the cortical algorithm is able to learn unique features in a distributed manner without requiring the computational complexity of a spiking neural network. Furthermore, in the future this model may be extended to include semi-supervised learning rules that can make learning more robust and generalizable, yet still maintain biological plausibility.

Other learning algorithms have also shown success at various learning tasks on different hardware substrates. Rice et al. have proposed a neocortex-inspired cognitive model deployed on the Cray XD1 supercomputer [24]. Their learning model is based on HTM, which is a hierarchical Bayesian network model proposed by Hawkins [25]. The learning model proposed by Rice et al. uses advanced software and reconfigurable hardware implementations to scale a model based on the human visual cortex to interesting problems. Like ourselves, Rice et al. take 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, but rather relies on a Hebbian learning paradigm. Furthermore, we have opted to use commodity GPGPUs instead of a supercomputer and FPGAs to effectively scale our model.

Finally, profiling based runtime systems such as StarPU [26] and Harmony [27] have been proposed to take advantage of heterogeneous system architectures. Such models have shown successful scaling on multicore systems equipped with a GPGPU or other hardware accelerators. However, to the best our our knowledge, such profiling based work distribution models have published work utilizing single GPGPU systems. In this work, we present results of our profiling system which considers homogeneous and heterogeneous multi-GPU systems.

5. Cortical Networks on CUDA

While it may be possible to eventually create neuromorphic hardware designs which more closely resemble the physical structures of the brain, we have spent considerable time investigating currently available hardware architectures that are a good match for our existing software model. The goal of the cortical network algorithm 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, many of these tasks depend on real time performance. A major feature of these models is that, like the brain, a large amount of parallelism is inherent to the design of the structure. This extractable parallelism makes the GPGPU an attractive hardware architecture for the cortical network algorithm. 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 GPU.

5.1. The CUDA Framework

The CUDA programming framework has gained considerable favor due to its relative ease of programmability. Using a modest set of extensions to the C programming language, programmers can port their serial programs to parallel ones without any graphics knowledge. The CUDA programming model is built around several layers of components which the programmer can configure explicitly.

The CUDA-thread is the basic unit of execution, and these threads are organized into thread-blocks, or Cooperative Thread-Arrays (CTAs). Within a CTA, threads can communicate and share local data via a fast-access shared memory space.

CUDA-enabled GPUs contain a number of Streaming-Multiprocessors (SMs) on which each CTA executes. Each SM contains shared memory space, which acts as a fast access user managed cache. Previous generations of GPU hardware (G80 and GT200 architectures) include 16KB of shared memory per SM which is shared among 8 shader cores. GPUs based on the newer Fermi architecture include 64KB of combined shared memory and L1 cache. The Fermi architecture gives the programmer the freedom to allocate 16KB or 48KB as shared memory space (with the leftover allocated as an L1 cache) [28]. Several other changes were made with the Fermi architecture, including expanding the number of cores per SM to 32 and adding a 768KB L2 cache shared by all SMs. For both architectures, the threads are grouped into Warps, which are 32 (in current hardware) consecutive threads that will always execute together.

Current and previous generation CUDA enabled devices are capable of executing up to 8 CTAs concurrently on each SM depending on a number of factors, including the number of threads per CTA, the number of registers used per thread, and the amount of shared-memory used by each CTA [28]. These factors are affected both by the CUDA compiler as well as how the programmer has optimized and organized their code. CUDA applications can be optimized by loading often accessed variables into the shared memory space, taking advantage of read-only texture caches, minimizing synchronization and thread divergence, and optimizing global memory accesses with memory coalescing [29].

5.2. Implementing the Cortical Hierarchy on CUDA

Like the cortical network described in this paper, the components of the CUDA framework also are arranged hierarchically. The cortical network has minicolumns, hypercolumns, and hierarchical networks of hypercolumns, whereas CUDA has threads, CTAs, and groups of CTAs known as kernels (or grids). Fitting the cortical network to the CUDA software model is 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 network the basic building block is the hypercolumn. Using the local shared memory space, we are able to model the fast short-range lateral connections between minicolumns within a hypercolumn. For a hypercolumn to learn more distinct features from a set of inputs, the number of minicolumns can be increased. For example, if we want each hypercolumn to learn up to 128 unique features, 128 minicolumns must exist in each hypercolumn (or 128 threads per CTA).

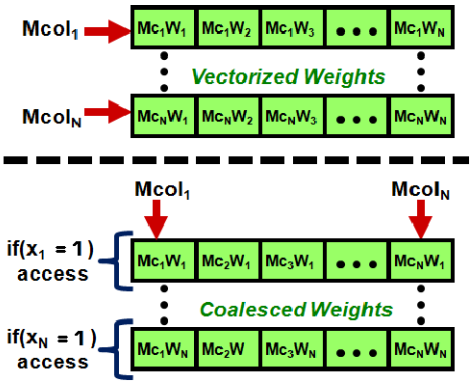

Figure 5: *Top: Naïvely, each minicolumn's weight vector can be allocated in a single vector. Bottom: By allocating a minicolumn's weights in a column, accesses can be coalesced. Depending on activation input x_i, the access of W_i can be skipped altogether.*

We optimize our cortical network algorithm in several ways by understanding the underlying architecture of the CUDA GPU. First of all, as mentioned in Section 3, minicolumns attempt to inhibit their neighbors after performing a winner-take-all competition. Given the combination of random firing, initial randomized weights, and partial weight matches, our learning algorithm favors the minicolumn with the strongest response. Using the shared memory space, all the minicolumns with firing activations compete in a reduction-like nature to determine the maximum response to the input. Naïvely, each minicolumn could compare its activation response to that of its neighbors to determine the minicolumn with the highest activation, which would take $0(n)$ time. However, we optimize this competition and communication by using a reduction-like method in the shared memory space to determine the winning response. For N minicolumns, N/2 determine the highest activation between two minicolumns. Next, N/4 minicolumns determine the highest activation between two winning minicolumns, and so on, until a highest activation is determined. As a result, the winning response can be determined with $O(\log n)$ complexity, and evaluated very quickly in shared memory.

Another method in which we tune the cortical algorithm specifically for CUDA is optimizing access to the minicolumns' weights in global memory. Since each minicolumn has a floating point weight vector the size of its receptive field (or number of inputs), it is not realistic to store the synaptic weights in the shared memory, but rather, optimize their accesses from global memory. To do so, the synaptic weights of the minicolumns within a hypercolumn are striped across separate 128-byte segments in global memory, as seen in the bottom of Figure 5. The first benefit is such an organization *coalesces* memory accesses - that is, a Warp of 32 threads can issue a 128-byte memory transaction of memory in a single cycle. If each thread accessed its weights from different 128-byte segments of memory, each access would issue a separate 128-byte memory transaction. Since all threads need the same weight W_i at any given time, coalescing allows reading or writing 32 synaptic weights to the global memory

	GPU	SMs	Cores	Freq (GHz)	SMem (Bytes)	SMem/CTA (Bytes)	CTAs/SM	Occupancy
32 Minicolumns	GTX 280	30	240	1.49	16384	1136	8	25%
32 Minicolumns	C2050	14	448	1.15	49152	1136	8	17%
128 Minicolumns	GTX 280	30	240	1.49	16384	4208	3	38%
128 Minicolumns	C2050	14	448	1.15	49152	4208	8	67%

Table 1: *Configurations of hypercolumns and their resulting occupancy on the GPU.*

space with one memory transaction. In some initial experiments, coalescing these weights contributed over a 2x speedup (considering the execution time of the entire application) when compared to a non-coalesced version of the cortical network. The second benefit is that, by considering the learning algorithm, we know that any input activation value less than 1.0 will not affect the minicolumn's activation (see Equation 7), nor will it update the corresponding synaptic weights to that input. As such, minicolumns can iterate through their inputs in parallel, and for every input activation that is less than 1.0, the entire group of minicolumns can skip reading the synaptic weights out of global memory (see Figure 5).

Beyond memory coalescing and the optimization of the winner-take-all algorithm, we have spent considerable effort tuning the cortical network algorithm for the GPU. Primarily this meant utilizing shared memory whenever appropriate, as well as balancing register and shared memory usage. For the simulations considered in this paper, both were tuned to maximize SM occupancy and utilization.

Considering the hierarchical structure of the cortical network, we realize the inputs of the upper levels depend on the outputs from the lower levels through a producer-consumer relationship. For producer-consumer data dependencies such as these, the typical solution is to execute the structure as separate CUDA-kernels [30]; 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 6 will detail some of the inefficiencies we discovered using this approach, as well as some solutions we have explored to mitigate them.

5.3. Experimental Setup

We compare the performance of our parallelized CUDA implementation of the cortical network algorithm with the implementation described by Hashmi et al. [1, 2]. We obtained the source code from Hashmi et. al., which was implemented as a serial program written in C++. Additionally, we implemented a parallel version of the cortical network for execution on a multicore machine using the POSIX thread (Pthread) Library. In the Pthread based implementation, each of the Hypercolumns operates as a thread, and barrier synchronization is added between consecutive hierarchical levels. This means that all the hypercolumns within the same level can effectively run in parallel.

The single-threaded and the multi-threaded implementations were run on a four core Intel Quad Core i7 @ 2.67 GHz hyper-threaded system with 12GB of RAM, while the CUDA implementations were executed on a GT200 architecture GeForce GTX 280 and Fermi architecture C2050 (more details are provided in Table 1). The execution times for the single-threaded implementation are shown in Table 2. Kernels were compiled with CUDA 3.1 as both compute capability[1] 1.1 (GTX 280) and 2.0 (C2050) with the host GPU determining the binary to run on the appropriate GPU [28]. While the GTX 280 is compute capability 1.3, we do not explicitly utilize any of the additional features and found slightly better performance when compiled as compute capability 1.1.

To measure performance, we studied the execution time for two configurations of cortical networks. The first configuration allocated 32-minicolumns per hypercolumn (32 threads per CTA) with each minicolumn having a receptive field size of 64 inputs (since the network was configured as a binary converging structure). The second configuration allocated 128-minicolumns per hypercolumn. Though there is an increased amount of parallelism by having more minicolumns per hypercolumn, there is also an increase in the memory usage for this configuration, as each minicolumn now has 256 synaptic weights. We examine two configurations of the cortical network: a 32-thread and a 128-thread implementation. In biology, it has been observed that hypercolumns typically contain dozens to hundreds of minicolumns [3]. In future work, we anticipate the number of minicolumns will be determined by the application or the specific region of the neocortex being modeled. We have also previously investigated using runtime profiling techniques to dynamically reconfigure the number of minicolumns in the cortical network after long-term training epochs, though this work focuses on the scalability of two different static configurations [31]. Table 1 details the resulting occupancy of both GPUs for the configurations we tested, obtained by using the CUDA Occupancy Calculator tool [28]. Occupancy is determined by considering the number of threads per CTA, the number of registers per thread, and the total shared memory used by the CTA.

5.4. Multicore CPU Performance

We first examine the speedups achieved using the parallel Pthread implementation of the cortical network executing on an Intel Core i7 CPU. Figure 6 shows the

[1]Nvidia GPUs have different compute capabilities, which, to the programmer, more or less translates to the extra set of features, such as atomic memory or thread-fence operations.

	31	63	127	255	511	1K	2K	4K	8K	16K	32K	64K
32 Minicolumns	0.32	0.65	1.30	3.13	6.26	12.52	24.90	49.80	99.61	206.31	412.62	825.25
128 Minicolumns	4.51	9.03	18.062	37.61	75.23	150.47	301.70	603.40	1206.81	2413.62	4827.25	9654.50

Table 2: *Single-threaded execution times for the different configured/scaled cortical networks, reported in seconds. Cortical networks were executed on the system described above.*

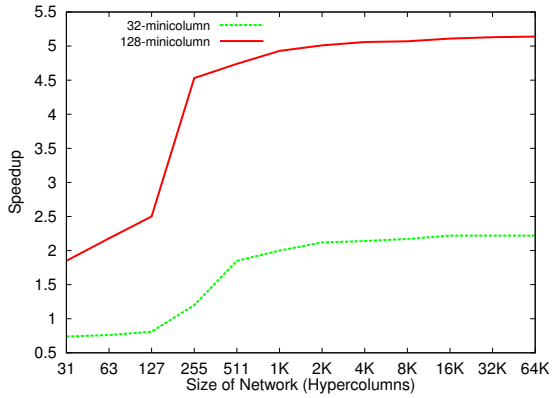

Figure 6: *Speedups of Pthread implementation of cortical network running on Core i7 CPU.*

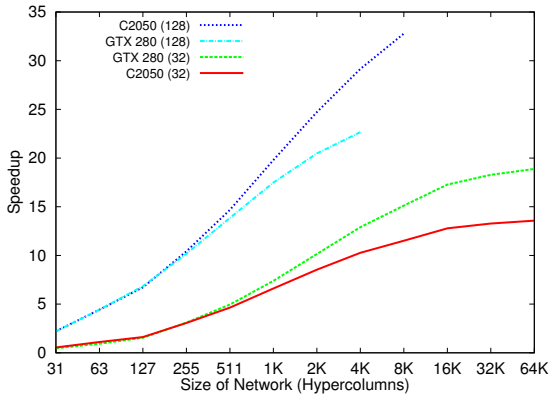

Figure 7: *Speedups of various cortical networks over the single-threaded CPU implementation.*

speedups obtained for the parallel Pthread implementation over the serial C++ implementation. For the 32-minicolumn configuration (dotted line), we see that for the small network sizes (31, 63, and 127 hypercolumn networks) the serial C++ implementation outperforms the parallel version of the program. This slow down can likely be attributed to the fact that, for small networks with less parallelism available, the thread management overhead overshadows the benefits gained from parallel execution. However, as the network size is scaled, we see that the parallel implementation outperforms the serial version and asymptotically approaches a speedup of 2.25x. We note that this is considerably less than the theoretical maximum speedup (8x, since the Core i7 contains four hyper-threaded cores). However, we believe the speedup is limited by the fact each hypercolumn contains only 32 minicolumns. For each thread that is utilized, there is only a small amount of work to perform when executing a hypercolumn. As a result, this configuration does not provide enough opportunity to amortize the costs of thread management and context switching.

However, as we scale the network to 128 minicolumns per hypercolumn, the relative overheads are much smaller. When each hypercolumn is configured to the 128-minicolumn case, each thread now executes 4x as many minicolumns, each with a 4x larger receptive field and synaptic weight vector. For the 128-minicolumn configuration (solid line), the maximum speedup achieved approaches 5.1x.

Considering a best-case Pthreads implementation, it might be possible to achieve an 8x speedup, assuming a doubling of throughput per core from simultaneous multi-threading (SMT). Our result falls somewhat short of that, demonstrating about 25Furthermore,if we were to utilize SSE instructions using 128-bit registers, we can potentially execute the dot-product calculations 4x faster, though this is only a portion of the total execution time for the hypercolumn. However, even if we consider this overhead-free perfectly optimized CPU model, our CUDA implementation still exhibits a significantly higher speedup compared to the hypothetical peak performance of the CPU.

5.5. Results of CUDA vs. Serial Implementation

Figure 7 shows the performance speedups of the CUDA implementation for a range of different scale networks. For the 32-minicolumn configuration, we see the maximum achieved speedups are 14x and 19x for the C2050 and the GTX 280 GPUs respectively. Initially, these results seem counterintuitive, since the C2050 has nearly twice as many cores as the GTX 280. However, consulting Table 1, we first notice that the CUDA Occupancy Calculator estimates that only 17% of the C2050 will be occupied given the specified CTA configuration, while the GTX 280 achieves 25% occupancy. Furthermore, we note that the maximum number of CTAs/SM is bounded by the CUDA compiler to 8 CTAs/SM [28]. Considering that the GTX 280 has 30 SMs, the total number of "live" threads at any given moment is 8192 (32 threads * 8 CTAs * 30 SMs). While the C2050 has a larger number of total cores, it has fewer SMs and is still constrained by the 8 CTA/SM limit. Therefore, the total number of "live" threads on the GPU at any given point is 3584 (32 threads * 8 CTAs * 14 SMs). As such, the restriction of 8 concurrent CTAs/SM seems to limit the C2050 in this configuration, and having a larger number of total cores provides no additional performance benefit.

For the 128-minicolumn configuration, the speedups achieved are 33x and 23x for the C2050 and GTX 280 respectively. We note here that this configuration has quadrupled both the number of minicolumns and the number of synaptic weights that each minicolumn must store.

Therefore, the GTX 280 is only able to store the state of 4K hypercolumns and the C2050 can store 8K hypercolumns. While it is possible to stream each hypercolumn's weights in and out of the GPU to allow simulation of larger scale cortical networks, the overall performance would degrade, and we were interested in testing the achievable performance of a cortical network that could stay resident on the GPU. We leave the investigation and optimization of this feature for future work. Here, we note that the C2050 performs better, as the GPU occupancy has increased to 67%, as compared to 38% on the GTX 280. Furthermore, we see that the amount of shared memory required by each CTA has quadrupled (see Table 1). The GTX 280, with 16KB of total shared memory, can now only support 3 CTAs/SM concurrently, while the C2050, with 48KB of allocated memory, has no problem supporting 8 CTAs/SM.

These simulations provide an interesting comparison between two generations of Nvidia GPUs. While the C2050 (Fermi) GPU boasts a number of architectural improvements, its performance is limited when the parallel workload of each CTA is quite small (as observed in the 32-minicolumn configuration). Furthermore, it can easily be seen that the 32-minicolumn configuration is likely to be memory latency bound, and neither GPU has enough live threads to adequately hide the memory latency (though, the effect seems to be worse for the C2050, which has half as many "live" threads to hide it). For the 128-minicolumn configuration, there are many more threads available to hide memory latencies, resulting in more impressive speedups. Since a larger number of threads are available per CTA, the C2050 shows a clear benefit having a greater number of total cores.

6. Improving Performance through Optimizing Cortical Network Execution

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 observations on some inefficiencies of our implementation. When applications have 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 [32], 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 network hierarchy results two problems: the overhead from multiple kernel launches and poor GPU resource utilization. We examine these inefficiencies in more detail, as well as two solutions we have implemented to mitigate them.

6.1. Difficulties Executing Hierarchical Objects on CUDA

The first inefficiency we consider is that, by using multiple kernel launches to maintain an order between the cortical layers of the network, the overhead of transferring control between the GPU and CPU is incurred multiple times. Figure 8 shows the percentage of execution time spent on additional kernel launch overhead for the 128-minicolumn configured networks on both the GTX 280 and C2050, obtained by executing empty kernels on the GPUs. We can see that 1-2.5% of the total execution time for a hierarchy is spent on the additional kernel launch overhead, with smaller cortical networks suffering from larger overhead. For the 32-minicolumn configuration, we observe 1-4% of total execution time spent on this overhead on both GPUs (not shown in figure). While such a small portion of the overall execution time may seem acceptable, we note that this is pure synchronization overhead which, ideally, should be entirely eliminated.

The second inefficiency we observe is poor resource utilization on the GPGPU. While the cortical networks we have simulated have a large amount of inherent parallelism at the lowest levels of the hierarchy, this parallelism diminishes for the upper levels of the network. The cortical network algorithm learns the features of the input in a hierarchically distributed manner; lower levels have a limited receptive field and process simpler features, while upper levels concatenate and combine these features and ultimately learn to recognize full objects or scenes. However, it is this convergent property of the configurations of cortical networks investigated in this paper that reduces the available parallelism. Using a single kernel launch per level means that the upper layers of the network, with very few CTAs, will under-utilize available resources on the GPGPU. Figure 9 shows the level-by-level breakdown of speedups for a 10-level cortical network hierarchy. At the lowest level, 512 CTAs can be executed in parallel, but at the top level of the hierarchy, only a single CTA is executed. In fact, for both GPUs, when there are 4 or less hypercolumns in a layer, the serial implementation on the host CPU outperforms the CUDA implementation. 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.

6.2. Pipelining to Increase Resource Utilization

From Figure 9, we are able to see how the hierarchical design of our cortical network results in poor utilization of the GPGPU's resources for the upper levels. We see that for the lower levels of the hierarchy we are able to extract a large amount of parallelism, 37x and 44x for the GTX 280 and C2050 GPUs respectively. 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. Ideally, we want to maximize hardware utilization by concurrently executing all hypercolumns across all levels of the cortical network, but we are unable to do so due to the data dependencies between levels.

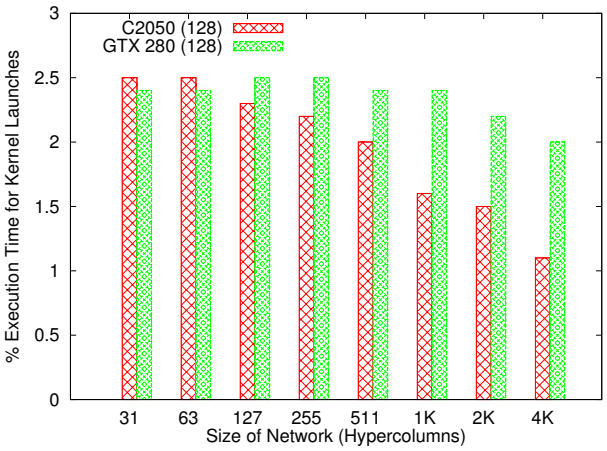

Figure 8: *Overhead of the additional kernel launches needed for different scale cortical networks (128-minicolumn configuration).*

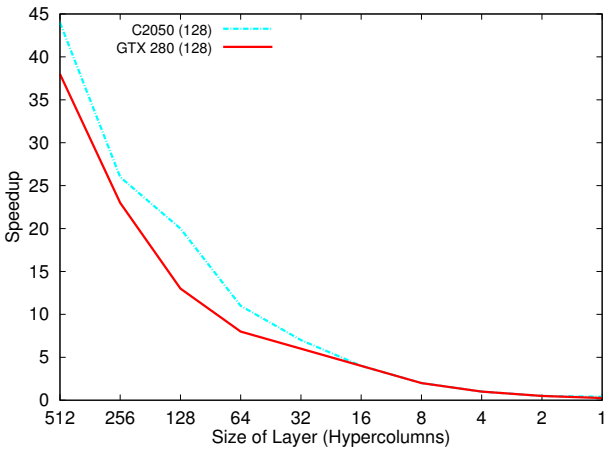

Figure 9: *Level-by-level speedups for a cortical network of 1023 hypercolumns. The lowest level of the cortical network is on the left.*

One solution is to pipeline the propagation of activations between subsequent layers of the cortical network. In the pipelining optimization, a single kernel-launch executes all hypercolumns in the hierarchy, and a double buffer between hierarchy levels guarantees that producer-consumer relationships are enforced. Figure 10 shows a simple example of pipelining between two stages of a cortical network hierarchy. On the first kernel launch, the activations from the lower level hypercolumns will be placed in Buffer-0 (solid red arrows). On the same kernel launch, the hypercolumns in the upper level will read their synaptic inputs from Buffer-1 (solid 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 (dashed green arrows).

While this method better utilizes the GPU resources and also improves training throughput, it should be noted that it still takes multiple kernel launches for any particular bottom level activation to fully propagate to the top of the hierarchy. However, considering that it can take from dozens to thousands of training iterations of an object for the network to converge (depending on learning rates, amount of training data, etc.), clearly this pipelining can speed up the training phase. The clear disadvantage of this implementation is that the amount of global memory dedicated to input/output activations doubles. Furthermore, using this pipelined implementation is feasible when considering feedforward unsupervised learning in the cortical network algorithm. If feedback connections are considered from upper levels of the hierarchy back down to the lower levels, this particular optimization becomes less attractive. Using a pipelined implementation would become increasingly complex as every connection would need to be buffered at each level and evaluation of feedback would possibly induce pipeline bubbles.

6.3. Kernel Fusion Using a Queue

Ideally we would like to be able to execute the entire cortical network on the GPU 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 [28]. For a hierarchical data structure like the cortical network, this means there is no easy way to guarantee that lower level hypercolumns will produce their output activations before the upper level hypercolumns are scheduled and executed.

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, as shown in Figure 11. This work-queue method operates as follows: First, a single CUDA-kernel is launched with only as many CTAs as can concurrently fit across all of the SMs in the GPGPU, as determined by the Occupancy calculator (Figure 11 shows 2 concurrent CTAs per SM). Next, each CTA uses an atomic primitive to gain a unique index into the work-queue (solid blue arrows 'A' and 'C'). The work-queue contains each hypercolumn's ID in the cortical network and is organized to execute hypercolumns in order from the bottom of the hierarchy to the top. If all input activations are available, the hypercolumn can calculate its output activations (in Figure 11, HC_0's inputs are ready, while HC_9 must wait for its inputs to be produced by HC_0). Once a hypercolumn has calculated its output activations, they are written back to the global memory. Afterwards, CUDA's thread-fence function is used to guarantee that prior writes are visible to all other threads, and the hypercolumn atomically increments a flag to indicate to its parent hypercolumn that all activation outputs are available. The dashed red arrow (B) in the figure depicts how HC_0 indicates to HC_9 that all input activations are available via atomic increment of the flag. Finally, the CTA atomically indexes again into the work-queue to execute another hypercolumn until the work-queue is empty.

One should note that this optimization makes some assumptions about the underlying hardware of the CUDA

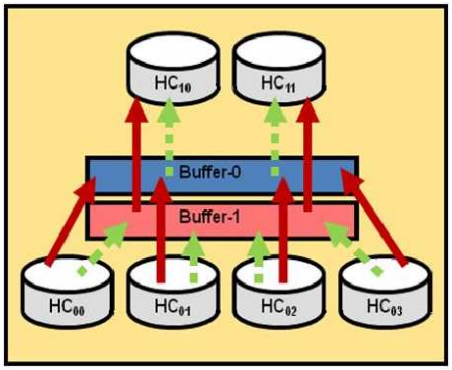

Figure 10: *Separate buffers are read-from and written-to between levels on a particular kernel launch.*

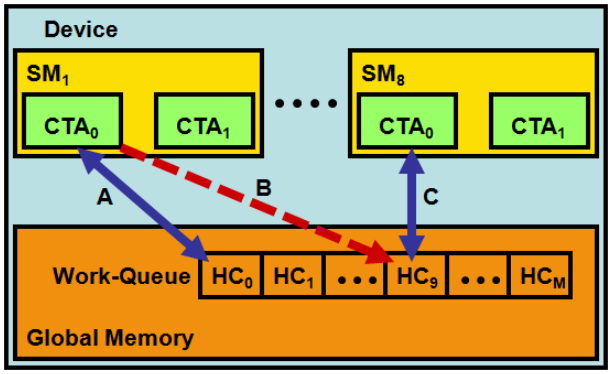

Figure 11: *Software work-queue implementation.*

enabled GPGPU. First, the number of CTAs launched for the work-queue method relies on information from the CUDA Occupancy Calculator tool, which considers how many CTAs will concurrently reside on each SM, given the number of threads, register count, and amount of shared memory. Here, we make the assumption that each SM will concurrently schedule this number of CTAs, and it will not be the case that the global CTA scheduler queues up all CTAs for execution on a single SM. That is, if there are 8 SMs available, and we determine that each SM can concurrently support two CTAs, we take for granted that launching a kernel with 16 CTAs will schedule two per SM, rather than queueing all 16 for execution on a single SM. Furthermore, we assume that the Warps of concurrently scheduled CTAs will not block each other. In practice, this has been quite effective for our application purposes as will be highlighted in the results section, though it should be noted that CUDA makes no definitive claims about how CTAs are scheduled. While it has never been encountered in our implementation, we note that the assumptions made above violate the current rules of the CUDA programming model and could possibly result in deadlock. However, other on-GPU barrier synchronization techniques share similar ideas and assumptions of our work-queue optimization and have been used to facilitate CTA to CTA communication without returning to the host CPU for synchronization [33]

This work-queue method is quite successful because many of the hypercolumns have no direct interaction with each other. Typically the producer-consumer dependencies have been met before the "consumer" hypercolumns are even scheduled. However the uppermost hypercolumns of a cortical network will require CTAs to spin-wait, as a "consumer" hypercolumn may be concurrently executing with the "producer" hypercolumn it is depending on. To reduce the amount of time spent waiting for these dependencies, we organize our code as seen in Algorithm 1. In the CUDA code, each hypercolumn first loads all the necessary state variables into the shared memory space. If its input activations are ready, each thread computes the output activation level for a minicolumn within the hypercolumn. After synchronizing the threads via the the *__synchthreads()* API call, the minicolumns compete in a winner-take-all fashion to determine which has the maximum response to the current inputs. The hypercolumns write activations to the global memory as soon as they have been calculated. Since these activations will propagate to the next level, the *__threadfence()* API call is used to guarantee that all prior writes are visible to all other threads, after which the hypercolumn can indicate to its parent that the activations are available. Afterwards, the hypercolumn can now perform local updates on its synaptic weights, write state variables back to the global memory, and pop the next hypercolumn from the work-queue. The major benefit of this code organization is that even when parent/child hypercolumns are scheduled at the same time on the GPU, their executions can partially overlap with useful work.

The work-queue optimization allows the execution of an entire cortical network from a single kernel launch and better utilizes the GPU resources. Furthermore, the memory overhead to maintain the work-queue is much smaller than the double buffer used by the pipelining optimization. The major hindrance of the work-queue is that it depends on slow atomic operations to the global memory for proper synchronization, as well as the additional overhead for storing, updating, and modifying the work-queue structure. However, an additional benefit of the work-queue optimization is that hypercolumns can be dynamically rescheduled and re-evaluated without needing another kernel launch. While the role of top-down feedback connections has not been considered for this work, in the future we anticipate their role to heavily influence the success of our model. As such, top-down and bottom-up activations may require several iterations before convergence, and the work-queue optimization fits nicely with such behavior. Under the context of strong feedback activations, a higher level hypercolumn could simply reschedule lower level hypercolumns to re-evaluate in the context of top-down processing information.

Algorithm 1 Pseudocode for Cortical Algorithm with Work-Queue.

```
if tid == 0 then
  q ← WorkQueue[atomicInc(qHead)] //pop first item
end if
while q ≠ empty do
  s_stateVars ← g_stateVars //load some state variables
  if tid == 0 then
    while myFlag ≠ ready do
      //spin-wait for ready
    end while
    s_activeInputs ← g_activeInputs //load inputs
  end if
  __syncthreads()
  s_activation[tid] ← computeActivation()
  __syncthreads()
  s_activation[tid] ← computeWTA()
  g_activation ← s_activation[tid]
  __threadfence() //flush activations to memory
  __syncthreads()
  if tid == 0 then
    atomicInc(parentFlag)
  end if
  updateSynapticWts() //perform local updates
  g_state ← s_state
  if tid == 0 then
    q ← WorkQueue[atomicInc(qHead)] //pop first item
  end if
end while
```

7. Utilizing Multiple Heterogeneous GPUs

In the previous sections, we have clearly shown the performance benefit of implementing the cortical learning algorithm on a GPGPU. However, systems today may have multiple GPGPUs at their disposal. We describe an online profiling tool that proportionally allocates a given cortical network across the host CPU and one or more homogeneous or heterogeneous GPUs.

7.1. Partitioning Cortical Networks Between CPU and GPU

As seen in the preliminary results of Section 5, cortical network layers with many parallel hypercolumns benefit from GPU execution, while layers with few hypercolumns result in a performance degradation. To combat this performance hindrance, we have designed an online cortical network profiler to determine the point at which the algorithm will gain a performance benefit from execution on the GPGPU and where it is better suited for the host CPU. From our experimentation, this point is typically the top few layers of the cortical network hierarchy. When a network is allocated, our online profiler creates a sample cortical network on both the GPU and the host CPU. Each network is executed in a level by level fashion (from the top down), collecting execution time information to determine the point at which the GPU is able to actually execute faster than the host CPU. This profiling also takes into account the PCIe transfer time to communicate activation outputs between the portion of the cortical network resident on the GPU and CPU. After profiling, the actual cortical hierarchy is allocated proportionally between the CPU and GPU. We note that profiling and distributing the cortical network between the CPU and GPU is only evaluated in the context of the baseline implementation, as we have not implemented the work-queue and pipelining optimizations for the CPU.

7.2. Partitioning Across Heterogeneous GPUs

Since a system may be made up of heterogeneous GPUs, our online profiling tool determines the relative performance between the GPUs available as well. As seen in Figure 7, one configuration of our cortical network exhibits better performance on the GTX 280 GPU, while the other is better on the C2050 GPU. While the simplest solution would be to naïvely partition the network equally across the available GPUs (see Figure 12), a number of factors would affect the actual execution of each partition. GPUs may have a different number of SMs, clock speeds, DRAM capacities, or additional features such as a cache hierarchy.

Considering these factors, the goal should then be to proportionally allocate the network across the GPUs so that they are all active the same amount of time, improving throughput and minimizing the synchronization time between GPUs. To do so, again the profiler executes a sample cortical network on the GPUs available. Afterwards, the profiling tool allocates and initializes proportional amounts of the network across the GPUs, depending on their relative performance.

Furthermore, the profiling tool also considers the total amount of DRAM capacity of the available GPU devices. If a particularly large cortical network is being simulated, device capacity must also be considered, in addition to relative performance. In our current profiling scheme, the cortical network will be partitioned with performance as a first priority, and DRAM capacity second. That is, once the higher-performance GPU reaches its memory capacity limit, the remaining portion will be allocated on the other GPU or GPUs. Alternatively, a large cortical network partition could be time-multiplexed on the GPGPU, and saved state could be transferred to and from the host; however, we leave investigation of this alternative to future work.

In multi-GPU systems, profiling is first performed among the available GPUs. The best performing GPU is then profiled against the host CPU to determine the number of upper levels that will execute on the CPU. Figure 13 shows an example of how the profiler may distribute the network across the available hardware resources. In its current implementation, the profiler attempts to minimize communication between GPUs. As a result, the first point at which GPU to GPU communication takes place, the best performing GPU will execute the higher layers of the cortical network until control is passed on to the host CPU.

Prior work has shown that analytic models can predict application performance accurately enough to effectively distribute work across multiple GPGPUs without profiling [34]. However, for our cortical networks, profiling imposes only a minor runtime overhead, does not require

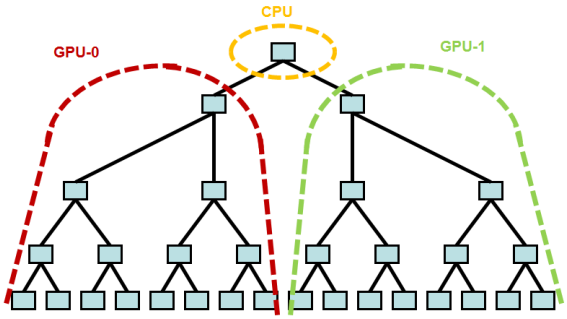

Figure 12: *Naïvely, the easiest method to split a cortical network across a system of host CPU and multiple GPUs would be to divide it evenly.*

careful selection of representative inputs since performance is insensitive to input values, and enables accurate predictions across heterogeneous computer resources (CPU and multiple generations of GPGPUs) for network configurations that can be either compute bound or memory latency bound, depending on platform. Hence, it is an appropriate and attractive approach for our environment. While an analytic approach appears promising and could be applicable here, we opted to rely on profiling in our initial implementation and leave investigation of analytic performance models to future work.

7.3. Using Optimizations on Multi-GPU

We also extended the pipelining and work-queue optimizations to the multi-GPU domain. Since both of these optimizations attempt to "flatten" the cortical network hierarchy for parallel execution, it is no longer necessary to execute the upper levels of the cortical network on the host CPU. Through experimentation, we found that the additional complexity of applying these optimizations in conjunction with CPU-GPGPU partitioning was not justified by an improvement in performance. Rather, the profiler partitions the network only across the available GPGPUs. Again, at the first point where communication is required to propagate activations between GPUs, the better performing GPU simply takes over to execute the upper levels of the network. The pipelining implementation requires no additional complexity in the multi-GPU domain. The work-queue optimization, on the other hand, requires an additional work-queue structure to execute these upper levels of the network. Initially, the lower levels of the cortical network are proportionally divided onto the available GPUs, each with their own work-queue. Once each GPU has finished executing their proportional cortical network segments, their input activations are transferred to this final work-queue. Again, we note that our profiler distributes the cortical network with performance as the first priority, and DRAM capacity second. In this work, we do not consider cortical networks that scale beyond the capacity of the total number of devices available to the system.

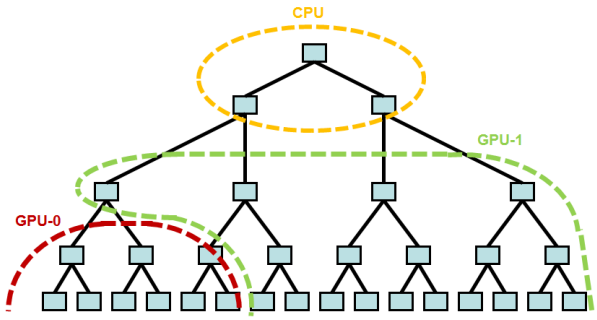

Figure 13: *The online optimizer tool finds the relative performance of a cortical network on the host CPU and one or more heterogeneous GPUs, then proportionally allocates the network to maximize performance.*

8. Online Profiler and Optimization Results

In the following section, we examine the performance results of the various optimizations and profiling techniques described earlier. We examine the performance on a system with two heterogeneous GPUs, and a system with four homogeneous GPUs.

8.1. Experimental Setup for Optimizations

Two systems with Nvidia GPGPUs were used in the following experiments. The first system had an Intel Core i7 @ 2.67 GHz with 12GB of RAM, a GTX 280 with 1GB of on board memory, and a Fermi C2050 with 3GB of on board memory. Each GPGPU was connected via its own 16x PCI-e bus. The second system had an Intel Core2 Duo @ 3.0 GHz with 4GB of RAM and two GeForce 9800 GX2 GPGPUs, each with 1GB of on board memory and connected via a 16x PCI-e bus. Each of the GeForce 9800 GX2s is composed of two GPUs, so the entire system contains four GPGPUs (sharing two PCI-e busses). Again, all speedups reported are relative to the single-threaded implementation of the cortical network run on the Intel Core i7 processor.

8.2. Single GPU Optimization Results

Figure 14 shows the speedups of the pipelining and work-queue optimizations achieved compared to the naïve multi-kernel launch approach on the C2050 GPU. Again, the speedups presented here are relative to the serial CPU algorithm. For the 32-minicolumn configuration, the performance results of the work-queue and pipelining optimizations are fairly close, and both provide a considerable boost for the smaller scale cortical networks where multiple kernel launch overhead and GPGPU resource underutilization are more significant. The pipelining optimization slightly outperforms the work queue, though this is expected as there is additional overhead required to manage the work-queue structure, and many of such operations use high latency atomic primitives. However, this overhead does not seem to be significant. Both optimizations asymptotically approach the same performance limit near 14x speedup since, as mentioned in Section 5, this configuration is likely memory latency bound. The performance

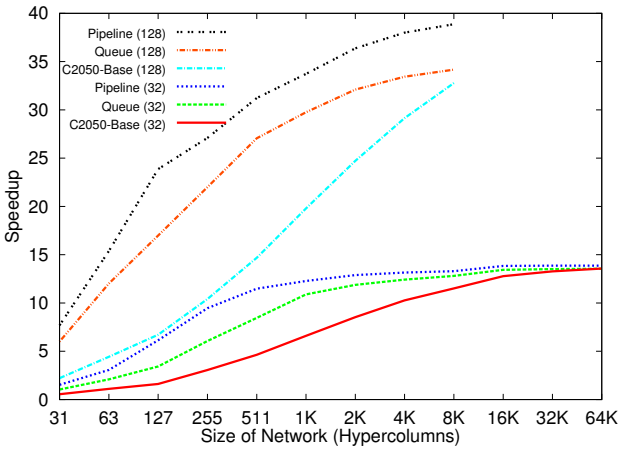

Figure 14: *Speedups achieved on C2050 using pipelining and work-queue optimizations.*

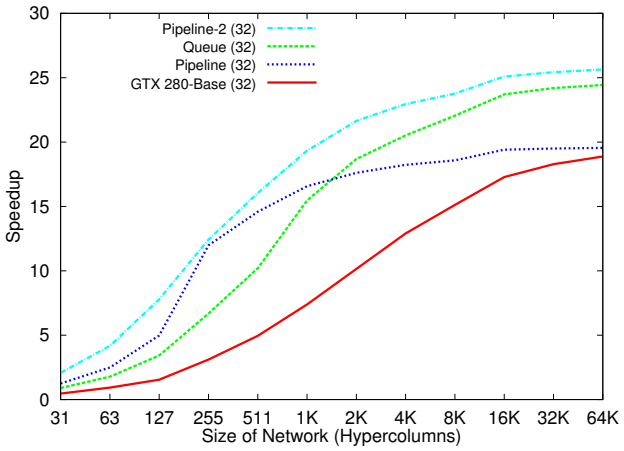

Figure 15: *Speedups achieved on GTX 280 using optimizations for 32-minicolumn cortical networks.*

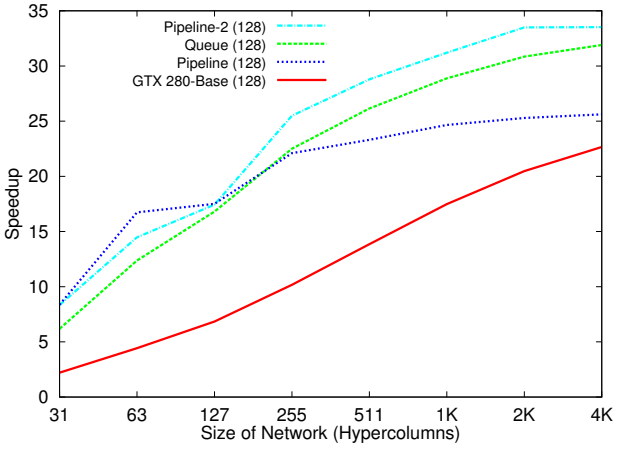

Figure 16: *Speedups achieved on GTX 280 using optimizations for 128-minicolumn cortical networks.*

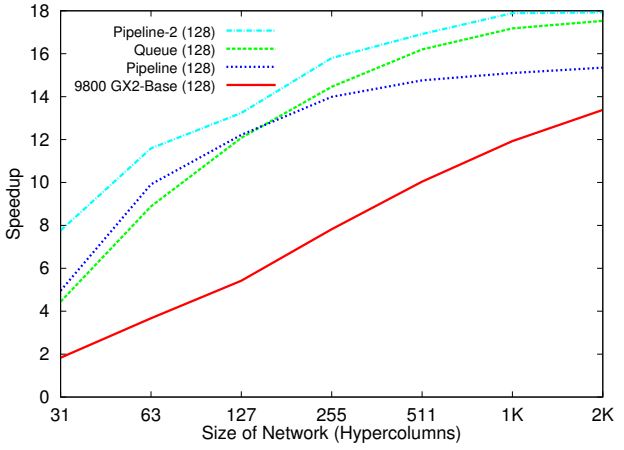

Figure 17: *Speedups achieved on 9800 GX2 using optimizations for 128-minicolumn cortical networks*

results for the 128-minicolumn configuration optimizations are similar, though here we see a maximum speedup of 39x for pipelining and 34x for the work-queue. The tradeoff between these optimizations is that pipelining provides a better speedup at the cost of double-buffering every input activation (and thus increasing memory utilization), while the work queue uses less memory overhead and is able to propagate the activations from the input layer to the top hypercolumn in a single kernel launch.

In Figure 15 we see the performance results of the cortical network optimizations configured with 32-minicolumns on the GTX 280 GPU, and in Figure 16 we see the results for the 128-minicolumn configuration. Again we see the performance benefits of utilizing both the pipelining and work-queue optimizations. While the pipelining implementation initially outperforms the work queue, interestingly enough in both configurations, a point is reached where the work queue shows better speedups. Considering that the work queue is dependent on synchronizing CTAs through slow atomic operations in global memory, these results appear counterintuitive. However, we note a major difference between the pipelining and work-queue optimizations. In the work queue, the kernel is launched with only as many CTAs as can concurrently reside on the GPU, and these CTAs loop until every hypercolumn in the work queue has been executed. The pipelining optimization simply launches a kernel with as many CTAs as there are hypercolumns, meaning as soon as one CTA is finished, the GPU's block scheduler must switch in the next CTA to the SM. For the 32-minicolumn configuration, the performance crossover point occurs around 1K hypercolumns (32 threads * 1K blocks = 32K threads), and for the 128-minicolumn case, the crossover is near 255 hypercolumns (128 threads * 255 blocks = 32K threads). Furthermore, a similar trend is evident for the 9800 GX2 GPU, as seen in Figure 17. The pipelining optimization initially outperforms the work queue, but performs worse at networks larger than 127 hypercolumns (128 threads * 127 blocks = 16K threads).

Consulting the Fermi Architecture Whitepaper [35], we see that the GigaThread scheduler of previous architectures manages up to 12,288 threads at a time, while the Fermi architecture provides improved block scheduling. We believe that this crossover point means that, although the work-queue structure requires additional overhead and utilizes slow atomic operations on global mem-

ory for synchronization, the work queue outperforms the CTA scheduling required for the large number of CTAs launched by the pipelining optimization. To test this theory, we implement a second pipelining optimization which only launches as many CTAs as can concurrently reside on the GPU (similar to the kernel launch of the work-queue optimization). These CTAs still use the double buffer to propagate activations. However, rather than relying on the global CTA scheduler to schedule each hypercolumn as a separate CTA, each CTA executes a predetermined portion of the overall cortical network until every hypercolumn has executed. In Figures 15, 16, and 17, Pipeline-2 shows the results of this new optimization. As expected, this optimization outperforms the work queue as it does not require the overhead associated with accessing and updating the work-queue structure, nor does it suffer from the possible limitations of the GigaThread scheduler [35]. We note that the C2050 GPU results do not show this crossover point between the work queue and pipelining optimizations, as one may expect due to Nvidia's improvements to the scheduler.

8.3. Investigation of Work-Queue Overheads

As shown previously, the pipeline implementation consistently outperforms the work queue and we believe this is largely due to slow atomics. We could implement a similar optimization to avoid atomics, but before undertaking that task believed it would be beneficial to quantify their contribution to the overhead. In order to accomplish this, we systematically removed the atomics to determine the execution time overheads of various components. There are two places in the work-queue code where we can perform this without fundamentally changing the execution path of the application: the spin-wait and the atomic update of the parent (or consumer) hypercolumns' ready flags. It should be noted that by removing such elements of the work-queue optimization, we sacrifice program correctness. That is, without the spin-wait, a hypercolumn may begin execution before its data dependencies have been written to the global memory space (either global DRAM or the L2 cache). We believe this sacrifice to be acceptable as we are just performing a preliminary experiment to determine the potential value of a more complete solution.

While we were willing to sacrifice program correctness, we chose not to remove the atomic access of the work-queue index. If this particular location is accessed non-atomically, multiple CTAs could operate on the same work item and impact runtime in ways that could not be strictly attributed to the work-queue overhead. Though the modifications described could affect program correctness, the amount of work done (in terms of number of hypercolumns executed) remains the same. At best, these modifications give some insight into the lower bound of the execution time using the work-queue optimization, less the overheads associated with actually managing the queue and ensuring data dependencies were met.

The evaluation setup involves two modified versions of the original work-queue implementation. In the first revision, we removed the spin-wait element used while a hypercolumn waits for its children (or producer) hypercolumns to update the activation information (indicated by increments of its ready-flag, see Section 6.3). The second revision of the code involved removing both the spin-wait element as well as the atomic increment of the aforementioned ready-flag. Rather, the update of the ready-flag is performed with a standard non-atomic increment of the memory location. This change required us to remove the spin-wait because, without it, racing children-hypercolumns could increment the flag to '1' simultaneously and the parent would never see the requisite '2'. Since the two changes could not be made in isolation, we determined that the contribution of the atomics is revealed by comparing the results of the first revision to the second revision. An unintended benefit from this approach is the revelation of the spin-wait's contribution to the overhead associated with the work-queue optimization.

Figures 18 through 21 show the results of these comparison runs. Here, we plot the relative speedups of the modified work-queue implementations of the cortical network, as normalized to the fully-functional work-queue implementation. The first trend we note is that the speedup provided by removing these elements from the work-queue structure diminishes with increasing network size. This decreasing speedup can be attributed to the relatively decreasing likelihood that a CTA begins work on a hypercolumn that still has unmet dependencies in the baseline case. That is, with large networks, the work queue spends more time executing hypercolumns that do not directly have producer-consumer dependencies, and therefore spend less time spin-waiting. However, when the network size is small, there is greater likelihood that a parent hypercolumn and one of its children hypercolumns will be executed at the same time. In such cases, the parent hypercolumn must spin-wait, delaying forward progress until all activation dependencies are met.

The second trend we notice is that removing both the spin-wait and the atomic increment of the ready-flag performs nearly the same as removing the spin-wait alone, with the notable exception being the 32-thread case on the GTX 280. While small networks see a slight advantage in removing both of these elements from the work-queue structure, the relative performance appears negligible for larger networks. The small incremental improvement achieved by removing the atomic increment of the ready-flag can only lead to the conclusion that the atomics, in fact, are not large contributors to the overhead of the work queue, and that the bulk of the overhead is actually due to spin-waiting. Therefore, our investigation can conclude that the effort required to create an optimization that would avoid atomics is unlikely to be worthwhile.

Finally, we make note of the noticeable dip in performance for the 32-thread work queue executed on the GTX 280 (Figure 20) when both the spin-wait and parent atomic

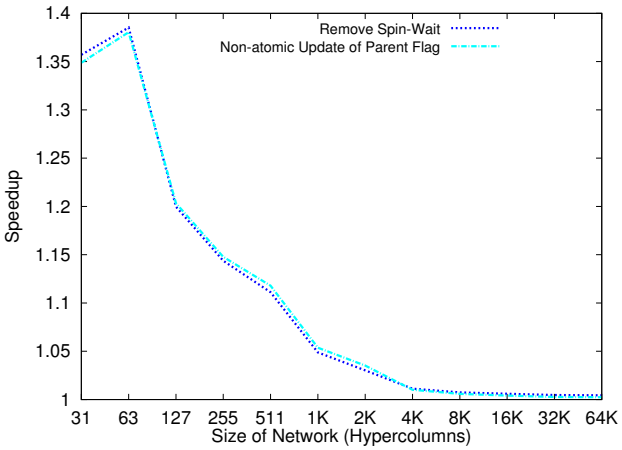

Figure 18: *Speedup achieved on a C2050 with 32 threads as atomic/spin-wait operations are removed from the application.*

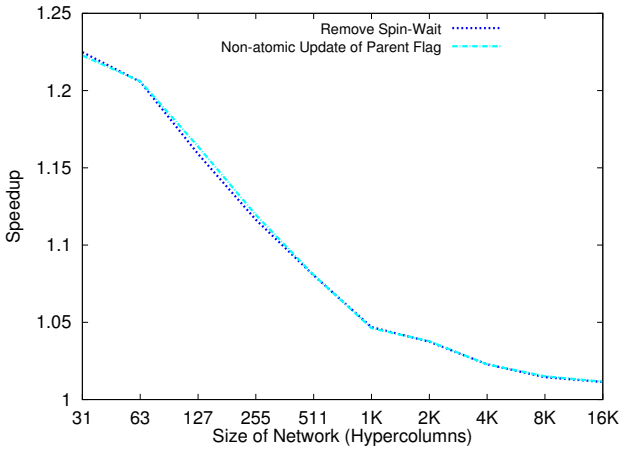

Figure 19: *Speedup achieved on a C2050 with 128 threads as atomic/spin-wait operations are removed from the application.*

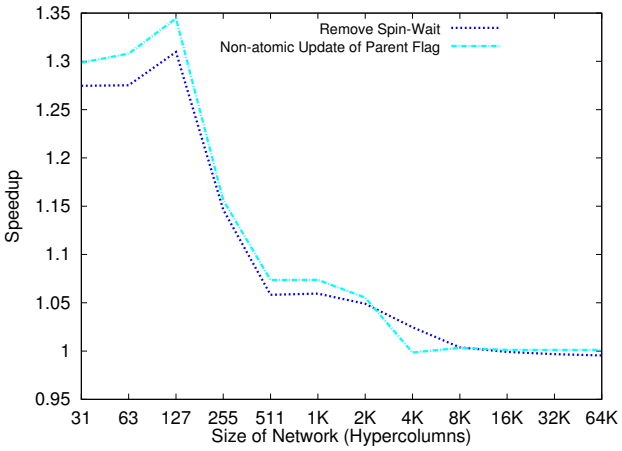

Figure 20: *Speedup achieved on a GTX 280 with 32 threads as atomic/spin-wait operations are removed from the application.*

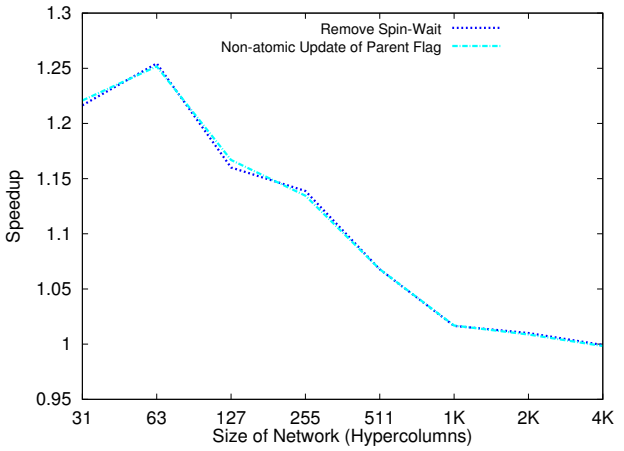

Figure 21: *Speedup achieved on a GTX 280 with 128 threads as atomic/spin-wait operations are removed from the application.*

updates are removed on a network of 4K hypercolumns. During experimentation, the same affect is a consistently occurring anomaly in this particular configuration. While we suspect the cause may be related to changes in memory interleaving at that particular network size, we leave further investigation to future work where we intend to examine the memory interleaving via GPGPU-Sim [36].

8.4. Comparing Shared Memory with Register and L1 Cache Utilization

Our work on porting the original implementation of the cortical learning algorithm to CUDA began on a compute capability 1.1 device (GeForce 9800 GX2). Since that time, many advances have been made to CUDA and the underlying hardware. Two changes of particular interest were the addition of a hardware-managed L1 cache with the Fermi architecture [35] and ever-increasing register file size. Where previous generations simply had a software-managed shared memory space, Fermi incorporated a hardware-managed L1 within the shared memory space. Furthermore, Fermi devices can configure the division between the user-managed shared memory and hardware-managed L1, giving 48KB to the preferred partition and 16KB to the other [28].

Since our initial deployment of the cortical network on CUDA was on a compute capability 1.1 device with a much smaller register file, we finely tuned our code to utilize as much of the 16KB shared memory space as possible. With older CUDA devices, the amount of register space available to the programmer is more limited, and writing an application that uses too many registers can greatly impact performance by limiting occupancy of the SMs - hence it is worth the extra programming effort [37]. Here, we seek to evaluate how important this extra work is in the context of much more powerful CUDA devices with larger register files and a hardware-managed L1. Considering the hardware improvements that have occurred since the introduction of compute capability 1.1 devices, it would appear that the shared memory/register usage break point has shifted.

In order to evaluate both the impact of utilizing the hardware-managed L1 and increased registers, we create an alternate version of the application that removes the finely tuned usage of the shared memory space in favor of simply using standard variable declarations for each hy-

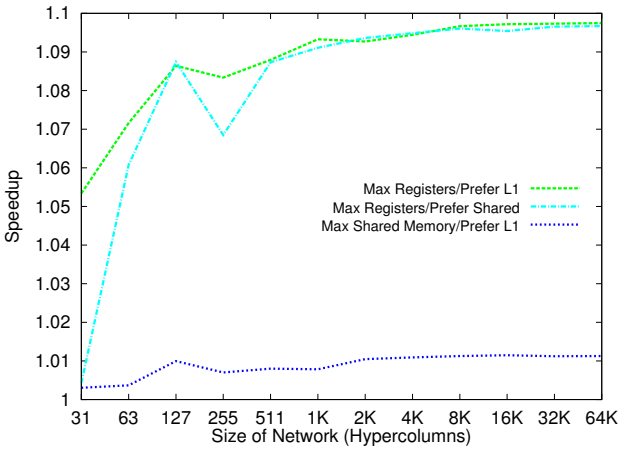

Figure 22: *Speedup relative to original (Max Shared Memory/Prefer Shared) implementation of the work queue for 32 threads.*

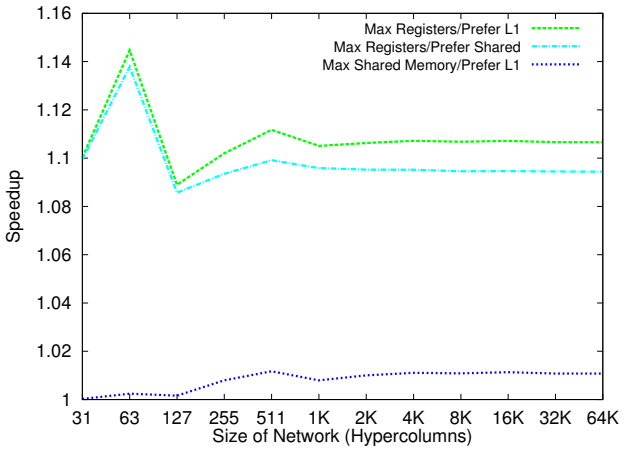

Figure 23: *Speedup relative to original (Max Shared Memory/Prefer Shared) implementation of the pipeline for 32 threads.*

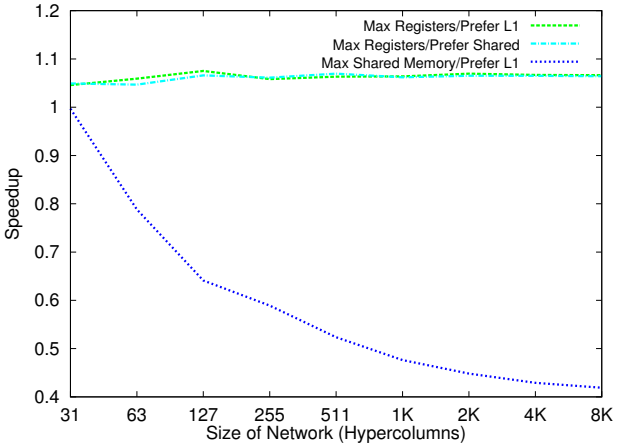

Figure 24: *Speedup relative to original (Max Shared Memory/Prefer Shared) implementation of the work queue for 128 threads.*

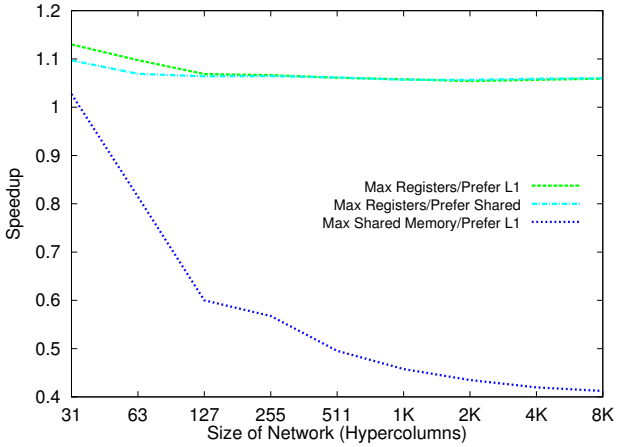

Figure 25: *Speedup relative to original (Max Shared Memory/Prefer Shared) implementation of the pipeline for 128 threads.*

			Max Registers			Max Shared Memory		
	Impl	**Prefer**	**# registers**	**smem (bytes)**	**occupancy**	**# registers**	**smem (bytes)**	**occupancy**
32 Minicolumns	Work-Queue	Shared	24	496	8	17	1166	8
		L1	24	496	8	17	1166	8
	Pipeline	Shared	21	488	8	15	1152	8
		L1	21	488	8	15	1152	8
128 Minicolumns	Work-Queue	Shared	24	1648	8	17	4236	8
		L1	24	1648	8	17	4236	**3**
	Pipeline	Shared	21	1640	8	15	4224	8
		L1	21	1640	8	15	4224	**3**

Table 3: *Configurations of hypercolumns and their resulting occupancy on the GPU.*

percolumn. With this new implementation, we not only reduce the use of shared memory, but we generally increase the register usage of the threads. By utilizing switches to the CUDA nvcc compiler, we are able to find the shared memory usage per CTA as well as the number of registers per thread (shown in Table 3) to determine the occupancy of the device. With this knowledge we are able to evaluate the speedups achieved for different configurations as shown in Figures 22 through 25. These figures show the performance of three different configurations relative to the original baseline code (which would be categorized as Max Shared Memory/Prefer Shared). We also note that these results were collected using the Fermi C2050 GPGPU. The "Prefer Shared" and "Prefer L1" descriptors refer to hardware configuration options used to set the size of the shared memory and L1. When the "Prefer Shared" directive is used, the SM is configured to have 48KB of shared memory and 16KB of L1, while the "Pre-

fer L1" directive means the opposite (16KB shared, 48KB L1). The "Max Shared Memory" and "Max Registers" refer to the code implementation of the algorithm with "Max Shared Memory" referring to the original implementation and "Max Registers" referring to the new implementation that removes the finely tuned use of the shared memory.

As the results for the 32-thread (or minicolumns per hypercolumn) configurations (Figures 22 and 23) show for both the work-queue and pipeline optimizations, the "Max Registers " code performs rather well with speedups in the 10% range over the baseline after some variability with small network sizes. We attribute this improvement primarily to the increased usage of registers. In each implementation, 6 more registers are used per thread without impacting the overall occupancy of the SMs (see Table 3). We note that for all of the 32-thread configurations, the maximum of 8 CTAs can be concurrently scheduled per SM. While the CUDA Programming Guide states latency of accesses to shared memory is similar to registers [28] and investigation has shown that shared memory can be used as an instruction operand [38], Volkov and Demmel found that throughput is negatively impacted when shared memory is used instead of registers [39]. Our results would seem to corroborate this. Another interesting feature of the 32-thread configuration is the modest gain when the original code is given a larger L1 cache. Since the SM occupancy is unaffected, these configurations would seem to benefit from the increased caching of temporary variables.

When one looks at the results of the 128-thread configuration of the cortical network, new trends appear. Now, instead of improvement across the board, some configurations of the hardware and preferred code implementation are actually penalized. The dramatic slowdowns seen by the configuration using the "Max Shared Memory" code with the hardware configuration preferring the larger L1 cache - in both the pipeline (Figure 25) and work-queue (Figure 24) implementations - are rather easily explained by a reduced occupancy relative to the baseline configuration (see Table 3). With the "Max Shared Memory" code, the shared memory usage per CTA is large enough that constraining the shared memory space to 16KB becomes the limiting factor. This effectively reduces the number of CTAs an SM can schedule at once, and instead of 8 blocks per SM, only 3 can reside concurrently.

For the remaining cases involving the 128-thread implementations of both the software work queue and pipeline, where the code emphasizes register usage, we again see an interesting trend. Similar to the 32-thread cases using the same approach, there is steady improvement over the baseline, though a bit more modest. Instead of the speedups approaching 10%, these are generally in the 5% - 6% range, yet still clear improvements. It should be noted however, that finding the right balance between what should be shared and what should be allocated to registers was more challenging than expected. While the 32-thread configurations were were generally insensitive to which variables were removed from explicit declaration in shared memory, naïvely removing all possible values from shared memory in the 128-thread configurations didn't provide us with benefits in all cases, with some penalizing us more than 30% over the results seen here. Instead an iterative approach needed to be taken to determine which variables had access patterns that lent themselves to shared memory and which ones to registers. Experimentally, we determined that whenever a variable was shared amongst threads, it was best to keep it in shared memory as duplicating it across all threads' registers showed no advantage. On the other hand, variables that were truly thread-private benefitted from being promoted to registers. Therefore, while the hardware-managed cache seems like a more hands-off approach for the programmer, it is not truly so and can require just as much analysis, iteration, and trade-off as those previously experienced when shared memory was the only option.

Overall, it can been seen that advances in the hardware over the years have shifted the optimal design points for various applications. Our cortical network application is a clear example of this. In previous hardware, where register files were much smaller and hardware caching was non existent, the shared memory space provided a convenient scratch pad for a dedicated programmer to keep frequently used variables close at hand. However, when considering newer generation devices with the large register files and hardware-managed L1's that appear to perform fairly well, it may make more sense to allow the compiler to allocate more variables to registers. In such cases, it may not be worth it for the additional programming effort to organize and optimize the use of shared memory variables. However, the old caveat still applies: no one way works better in all cases. Even in our limited set of configurations, it can be seen that removing as much of the shared memory as possible isn't always beneficial. Also, even though advances in the hardware can on the surface appear to remove much of the programming effort, the fact is that careful consideration still needs to be taken when allocating these resources.

8.5. Profiled Multi-GPU Results

In Figure 26 we examine the performance benefit gained by using our cortical network online profiling tool. We compare a naïvely distributed cortical network (referred to as "Even" in Figure 26) with a network that has been profiled and proportionally allocated across the host CPU, the GTX 280 and C2050 GPUs ("Profiled" in Figure 26). The naïvely distributed network executes the top hypercolumn on the CPU and splits the lower levels of the network evenly across the GPUs (see Figure 12). For the cortical network configured with 32 minicolumns, we remember that the GTX 280 performs better (refer to Figure 7), so the profiling tool will favor it with a larger portion of the cortical network. We see that the profiled cortical network achieves up to a 30x speedup here, compared with a 26x speedup of the naïvely partitioned network.

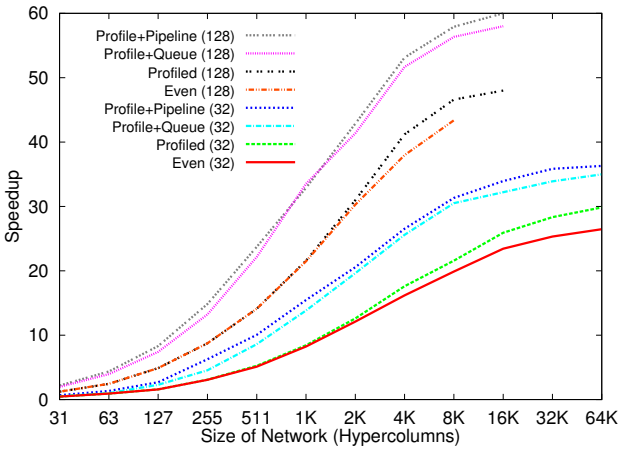

Figure 26: *Speedups achieved using profiling together with execution optimizations on a heterogeneous GPGPU system (C2050 and GTX 280).*

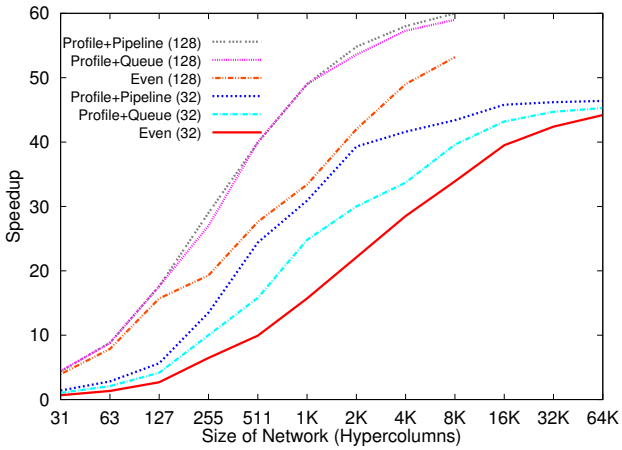

Figure 27: *Speedups achieved using profiling together with execution optimizations on a homogeneous GPGPU system (two 9800 GX2s, for a total of 4 homogeneous GPUs).*

As was discussed in our results above, the cortical network 128-minicolumn configuration performs better on the Fermi C2050, and here we see that the optimizer tool has allocated a larger portion of the network to it. The profiled network shows a maximum of a 48x speedup compared to an 42x speedup on the naïvely partitioned network. We also see that the profiler is able to execute larger networks than the simple evenly distributed network. Since the C2050 has 3GB of global memory but the GTX 280 has only 1GB, the largest evenly distributed network that can be allocated is 8K hypercolumns. However, the profiler recognizes that there is still available memory on the C2050, and thus can successfully allocate a 16K hypercolumn network across both GPUs. At this point we see the speedup trend has literally levelled off, as now the C2050 is executing 3/4ths of the network. However, our runtime profiling technique allows us to take advantage of the extra DRAM memory resources, which would not have been possible if we simply divide our application across the available GPGPUs.

Combining the cortical network optimizations with profiling resulted in even better speedups. Again, for both network configurations considered, the pipelining optimization slightly outperforms the work queue. As a result, we see up to a 36x speedup for the 32-minicolumn configuration, and an impressive 60x speedup for the 128-minicolumn network.

Finally, we examine the performance of our multi-GPU optimizations on a system of homogeneous GPUs. Figure 27 shows the speedups achieved on a system containing two 9800 GX2 GPUs, containing in total four identical GPUs. Again, "Even" provides a baseline of the cortical network being evenly distributed across all four GPUs. Since the GPUs are identical, profiling the system results in the exact same distribution. However, we see that adding the additional optimizations we examined, a maximum speedup of 60x can again be achieved on this four GPU system.

9. Other Applications

For the cortical network 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 similar optimization techniques for two other applications that have a similar hierarchical data dependency structure: a parallel reduction and a multilevel feedforward neural network. All results in this section were obtained using the Fermi C2050 GPGPU.

9.1. Parallel Reduction using Pipelining

The CUDA software development kit features several variations of a parallel reduction algorithm able to process large arrays of elements [40]. 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 28. Like the cortical networks, we again see that the amount of available parallelism reduces as we traverse through the levels of the tree, resulting in an underutilization of the GPGPU computation resources. Here, the pipelining solution described in Section 6.2 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 throughout. 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 29 shows the performance comparison of the pipelined and original parallel reduction implementations for different input sizes as well as thread/CTA configurations. For each configuration tested, we show the normalized breakdown of execution time. For simplicity, the bar graph is broken into the percent of execution time spent in

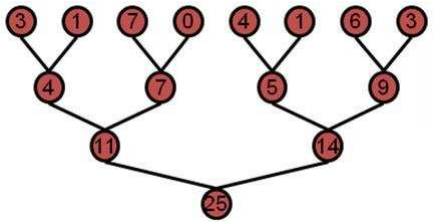

Figure 28: *Parallel reduction exhibits a tree-like structure for data communication, and utilizes one kernel launch per level.*

the first (and largest) kernel launch, and the subsequent kernel launches. By pipelining the reduction and utilizing a single kernel launch, the total execution time can be reduced up to 6%. While pipelining the parallel reduction offers some speedup over the baseline implementation, these results are less impressive than the dramatic speedups provided by pipelining the cortical network. 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 reduction tree (or first kernel launch). 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 reduction tree ranged from 64 to 1024 for the configurations we tested, much higher than the two node fan-in of the cortical network. Intuitively, the pipelined implementation cannot execute any faster than the longest kernel (or layer) of the parallel reduction (minus the multiple kernel launch overheads). Referring back to the figure, we see exactly this delivered performance for the pipelining optimization. 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. As before, 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 (results are not shown).

9.2. A Feedforward Neural Network using Pipelining

Another application that exhibits a similar 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 30 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 which uses a separate kernel launch for each layer [19]. With this particular application, the number of CTAs and threads in each layer has been optimized specifically for the task of recognizing handwritten digits from the MNIST database. The CTA and thread counts can also be seen in Figure 30. Furthermore, the application is parameterized to execute

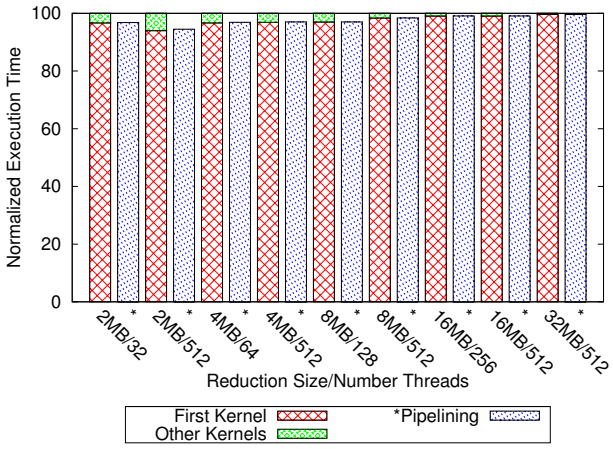

Figure 29: *Performance of pipelining parallel reduction. Baseline execution time is normalized to 1, and the breakdown of the histogram details the percent time spent executing the first kernel and the subsequent kernels. The X-axis label is Reduction Size/thread count.*

N neural networks in parallel, each evaluating a different input. While this improves GPU resource utilization for very small layers, this technique simply creates multiple copies of the exact same neural network and still requires four kernel launches to execute the application.

We use the pipelining optimization as well as the work-queue optimization in an attempt to better utilize GPU resources and reduce the amount of kernel launch overhead. Implementing the work-queue optimization in the neural network is much coarser-grained than the cortical network application. In particular, the neural network exhibits a much denser connectivity between layers (e.g. some layers connect all-nodes-to-all-nodes), which essentially requires that an entire layer must finish before the next can proceed. The cortical network, on the other hand, exhibited a much sparser and topological connectivity, allowing the work-queue to schedule nodes in different layers much more flexibly.

Figure 31 shows the normalized execution time of the work-queue and pipelining optimizations for the neural network. The figure also details a breakdown of the percent of time spent executing each layer for the baseline implementation. We note that in this figure, we present the results for a single neural network (that is, $N = 1$). When executing multiple copies of the neural network in parallel (that is, N *greater than 1*), the percent of time the application spends executing Layer3 approaches 99% in the baseline. We are less interested in such cases, since executing many copies of a single application in parallel will clearly more fully utilize GPU resources and mitigate kernel launch overheads. Again, we realize that even with the optimizations presented in this paper, an application's execution time can only be reduced to the execution time for the longest kernel. From Figure 31, we see that Layer3 makes up nearly 75% of the total execution time of the neural network. Once again, our cortical network application sees more benefit since it more evenly distributes the

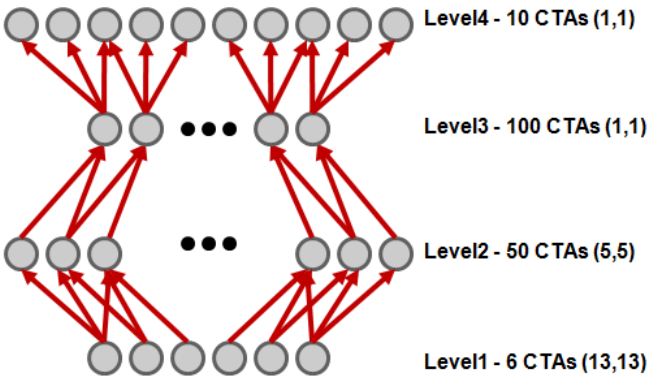

Figure 30: *A multilayer artificial neural network. Thread configurations are in parentheses.*

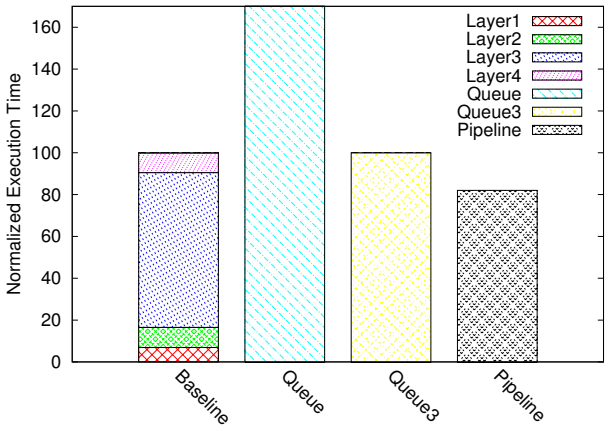

Figure 31: *Performance of optimizing a neural network.*

execution time between the multiple levels; in contrast, this neural network has been optimized to use three relatively quick kernels and a long kernel where the majority of the work is performed.

Figure 31 shows that using the pipelining optimization reduces the execution time by 18%. We note that this approaches the execution time for Layer3 in the baseline, which is essentially the best performance we can achieve without altering the underlying code for the neural network application. However, for the work-queue implementation ('Queue' in the figure), we notice a 70% increase in the overall execution time. We suspected that part of this was due to the different thread requirements of each layer, in particular, the fact Layer1 requires 169 (i.e. 13x13) threads/CTA. Queue3 uses the work-queue on only Layers2-4 (with Layer1 still executed alone) to avoid this problem, but the overall execution time is still the same as the baseline. Again we note that the high level of connectivity between layers requires many atomic operations which contribute to the poor performance of the work-queue optimization. For example, Layer1, with 100 CTAs, must perform 100 atomic operations to signal to Layer2 that all dependencies have been met. As a result of the lock-step nature of this particular application, we see that simply utilizing multiple kernels outperforms the use of our work-queue structure for synchronization.

10. Discussion

While the results in this paper indicate that Nvidia's CUDA is an excellent framework for accelerating cortical networks, the optimizations we explored were an attempt to maximize performance and satisfy 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 and communicating between CTAs, other than breaking data dependencies across 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, depending on the application structure. Clearly the overhead associated with using the work-queue structure limits the performance from achieving the same speedups as the pipelining optimization. However, the work-queue still performs much better than the multiple kernel implementation of our cortical network. 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.

As was noted earlier, the cortical network model considered in this paper utilize only feedforward and lateral connectivity. However, in biological cortical networks, it is apparent that feedback (or top-down) connections are just as numerous as feedforward connections [17]. The role these feedback connections play still requires extensive investigation, both by the neuroscientific community and neural network modelers. Previous models have looked at the role of feedback for attention and learning [41], consolidating learning [2], pattern completion, and stimulus prediction [42]. We are presently expanding our cortical network models to include such features.

Our preliminary experiments show that including such feedback connections does not hinder the performance improvement granted from execution on the GPGPU. We consider two possible ways to model feedback in the cortical network, and find that they have no effect on the performance improvement gained by execution on GPGPUs. First, if feedback must be generated in the context of each input, we simply execute the network from the bottom-up, and then again from top-down. If it is necessary that the network converge on a stable state, this process can be repeated until such a state is reached. However, our GPGPU implementation of this algorithm still exhibits significant speedups over a serialized implementation due

to the parallel organization of the cortical network model. Second, if feedback from the previous time step modulates the current time step, no significant change to the network is required. The hypercolumn and minicolumn models will include feedback state from the previous time step, but the overall execution of the model is still highly parallel and can benefit from execution on the GPGPU. Considering the pipeline optimization, including feedback paths will likely increase the complexity of buffering between cortical levels, but the performance still improves over serial code. The work-queue optimization is a bit more amenable to to the inclusion of feedback paths. Hypercolumns that must be re-executed due to feedback signals can simply be added again to the work-queue. This may mean a small increase in the complexity of the work-queue scheme, but will not affect the performance.

From the results of our work with these optimizations, as well as the performance results of the cortical network, neural network, and parallel reduction, we can draw some general conclusions 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 previous generation CUDA hardware (G80, GT200), only a single concurrent kernel can be executed on the hardware. Fermi based GPUs allow for multiple kernels to be executed on hardware, though there still are not methods to communicate or synchronize between concurrently running kernels. Therefore, the optimizations we investigated in this paper work within a single kernel launch and the application's 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 ordering of CTAs, atomic primitives can be used. However, these atomic primitives have long execution latencies, 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 and/or resource underutilization are significant.

11. Conclusion

In this paper we described a GPGPU-parallelized extension to an intelligent system based on the neocortex. Using CUDA, a cortical network executing on a single GPGPU achieved a 33x speedup over a single-threaded CPU implementation. We investigated inefficiencies with this initial implementation and described optimizations to mitigate their affect on performance. By using atomic operations or using a double-buffering scheme to maintain producer-consumer dependencies, we were able to reduce kernel launch overhead and improve GPGPU resource utilization.

Studying the behavior of our initial implementation, its inefficiencies, and the behavior of our optimizations, we learned several key insights about the different underlying architectures of the G80, GT200, and Fermi generation GPUs:

- Synchronization and workload imbalance bottlenecks inherent to the CUDA bulk synchronous processing model can be overcome with algorithmic changes (work-queues, pipelining with double-buffering).
- Performance is highly sensitive to cortical network configuration, since the same network can be either memory- or compute-bound on different GPGPU generations, while changes in configuration can invert the relative performance for these generations of GPGPUs.
- Improvements in thread scheduling in the Fermi generation can reduce or even eliminate the need for algorithmic modifications to moderate the number of threads in a kernel launch.
- The larger register file and L1 cache available in Fermi devices may eliminate rigorous use of the programmer managed shared memory space in some applications.

We also extended the GPU implementation of the cortical algorithm to the multi-GPU domain. By creating an online profiling tool, we were able to even further improve performance by proportionally allocating a cortical network across the host CPU and available homogeneous or heterogeneous GPGPUs. Applying our optimization techniques to the multi-GPU cortical networks, we were able to achieve an overall 60x speedup over the serial CPU implementation of the algorithm.

Acknowledgment

We wish to thank our collaborators Olivier Temam and Hugues Berry for many fruitful discussions on cortical models, as well as the paper's anonymous reviewers for their helpful comments. This work was supported in part by National Science Foundation awards CCF-0702272 and CCF-1116450, as well as donations from Nvidia and Google.

References

[1] A. Hashmi, M. Lipasti, Cortical columns: Building blocks for intelligent systems, in: Proceedings of the Symposium Series on Computational Intelligence, pp. 21–28.

[2] A. Hashmi, M. Lipasti, Discovering cortical algorithms, in: Proceedings of the International Conference on Neural Computation (ICNC 2010).

[3] V. Mountcastle, An organizing principle for cerebral function: The unit model and the distributed system, in: G. Edelman, V. Mountcastle (Eds.), The Mindful Brain, MIT Press, Cambridge, Mass., 1978.

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

[5] A. Nere, A. Hashmi, M. Lipasti, Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).

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

[7] G. Roth, U. Dicke, Evolution of brain and intelligence, TRENDS in Cognitive Sciences 5 (2005) 250–257.

[8] D. Hubel, T. Wiesel, Receptive fields, binocular interactions and functional architecture in cat's visual cortex, Journal of Physiology 160 (1962) 106–154.

[9] V. Mountcastle, The columnar organization of the neocortex, Brain 120 (1997) 701–722.

[10] D. Hubel, T. Wiesel, Receptive fields and functional architecture of monkey striate cortex, Journal of Physiology 195 (1968) 215–243.

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

[12] D. Ringach, Haphazard wiring of simple receptive fields and orientation columns in visual cortex., J. Neurophysiol. 92 (2004) 468–476.

[13] A. Losonczy, J. Magee, Integrative properties of radial oblique dendrites in hippocampal ca1 pyramidal neurons, Neuron 50 (2006) 291–307.

[14] R. E. Brown, P. M. Milner, The legacy of Donald O. Hebb: more than the Hebb synapse., Nat Rev Neurosci 4 (2003) 1013–1019.

[15] R. Douglas, C. Koch, M. Mahowald, K. Martin, H. Suarez, Recurrent excitation in neocortical circuits, Sciense 269 (1995) 981–985.

[16] 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, Hum. Brain Map. 6 (1998) 316–328.

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

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

[19] Billconan, Kavinguy, A neural network on gpu, http://www.codeproject.com/KB/graphics/GPUNN.aspx, 2008.

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

[21] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. Veidenbaum, Efficient simulation of large-scale spiking neural networks using cuda graphics processors, in: IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks, IEEE Press, Piscataway, NJ, USA, 2009, pp. 3201–3208.

[22] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A. V. Veidenbaum, A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors, Neural Networks 22 (2009) 791 – 800. Advances in Neural Networks Research: IJCNN2009, 2009 International Joint Conference on Neural Networks.

[23] R. Raina, A. Madhavan, A. Y. Ng, Largescale deep unsupervised learning using graphics processors, in: International Conf. on Machine Learning.

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

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

[26] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, cheaper, better a hybridization methodology to develop linear algebra software for gpus, GPU Computing Gems 2 (2010).

[27] G. Diamos, S. Yalamanchili, Harmony: an execution model and runtime for heterogeneous many core systems, in: Proceedings of the 17th international symposium on High performance distributed computing, ACM, 2008, pp. 197–200.

[28] NVIDIA, CUDA 3.1 Programming Guide, NVIDIA Corporation, 2701 San Toman Expressway, Santa Clara, CA 95050, USA, 2010.

[29] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, 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, ACM, New York, NY, USA, 2008, pp. 73–82.

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

[31] A. Hashmi, A. Nere, M. Lipasti, A case for neuromorphic isas, in: Proceedings of the sixteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS '11, ACM, New York, NY, USA, 2011.

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

[33] S. Xiao, W. chun Feng, Inter-block gpu communication via fast barrier synchronization, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1 –12.

[34] D. Schaa, D. Kaeli, Exploring the multiple-gpu design space, in: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12.

[35] NVIDIA, Nvidia's next generation cuda compute architecture: Fermi, Queue1322 (2010).

[36] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator, in: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, p. 163174.

[37] NVIDIA, CUDA occupancy calculator, 2010.

[38] W. J. van der Laan, Decuda, https://github.com/laanwj/decuda/wiki, 2010.

[39] V. Volkov, J. W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, p. 111.

[40] M. Harris, Optimizing parallel reduction in cuda, http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf, 2007.

[41] P. R. Roelfsema, A. R. Van Ooyen, Attention-gated reinforcement learning of internal representations for classification, Neural Comput. 17 (2005) 2176–2214.

[42] J. Hawkins, D. George, Hierarchical temporal memory, 2006.