Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms

Andrew Nere, Atif Hashmi, and Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706, USA
{*nere, ahashmi*}*@wisc.edu*, {*mikko*}*@engr.wisc.edu*

***Abstract*—Recent advances in neuroscientific understanding make parallel computing devices modeled after the human neocortex a plausible, attractive, fault-tolerant, and energy-efficient possibility. Such attributes have once again sparked an interest in creating learning algorithms that aspire to reverse-engineer many of the abilities of the brain.**

In this paper we describe a GPGPU-accelerated extension to an intelligent learning model inspired by the structural and functional properties of the mammalian neocortex. Our cortical network, like the brain, exhibits massive amounts of processing parallelism, making today's GPGPUs a highly attractive and readily-available hardware accelerator for such a model.

Furthermore, we consider two inefficiencies inherent to our initial design: multiple kernel-launch overhead and poor utilization of GPGPU resources. We propose optimizations such as a software work-queue structure and pipelining the hierarchical layers of the cortical network to mitigate such problems. Our analysis provides important insight into the GPU architecture details including the number of cores, the memory system, and the global thread scheduler. Additionally, we create a runtime profiling tool for our parallel learning algorithm which proportionally distributes the cortical network across the host CPU as well as multiple GPUs, whether homogeneous or heterogeneous, that may be 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; GPGPU; profiling systems; CUDA;**

I. 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 models, it becomes relatively straightforward to map them to GPGPUs, which provide massive amounts of parallel hardware at modest expense.

In Hashmi et al. [8], [9], the authors propose an intelligent system design inspired by the mammalian neocortex. One of the interesting aspect of this model is that instead of modeling individual neurons, it models cortical columns [17] as the basic functional unit of the neocortex. The properties incorporated in this learning algorithm enabled creation of a biologically plausible model of the visual cortex without requiring the computational complexity of implementing individual neurons. In [21], the authors extend this neocortex-inspired architecture to a single GPGPU to achieve a significantly faster version of the algorithm.

In this paper, we extend the work of [8], [9], [21] 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 homogeneous GPGPUs as identical CUDA-enabled Nvidia devices, while heterogeneous GPGPUs may span different architecture generations, number of cores, and amount of memory (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.
- To the best of our knowledge, this is also the first work that effectively demonstrates using a profiling tool to automatically distribute an algorithm proportionally across heterogeneous GPUs and a host CPU.

The rest of this paper is organized as follows: Section II details relevant information regarding the neocortex, and Section III 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 IV . Section V describes the methods used to extend our cortical networks to the GPGPU using CUDA and presents some initial performance results. Section VI examines bottlenecks encountered with expanding this cortical architecture to the GPGPU and proposes optimizations we have implemented to mitigate such inefficiencies. In Section VII, we extend our GPGPU implementations to the multi-GPU domain, using online profiling to efficiently distribute a cortical network across multiple heterogeneous or homogeneous GPUs and the host CPU. We examine the results of our optimizations and multi-GPU implementation in Section VIII. Finally, Section IX concludes the paper.

II. 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 [32]. For a typical adult, it is estimated the neocortex has around 11.5 billion neurons and 360 trillion synapses, or connections between neurons [27]. Mountcastle [17] 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 [17] which he termed *cortical columns* because of the seemingly column-shaped organizations of neurons exhibiting similar firing patterns for a given stimulus. Hubel et al. [12] and Mountcastle [18] 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 [13]. 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 [13]. Figure 1 shows a typical arrangement of minicolumns within a hypercolumn.

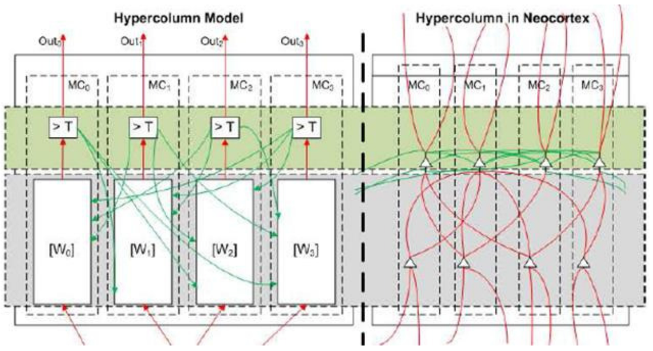

Figure 1. *Right: Biological representation of a hypercolumn, with lateral connections representing local inhibition. Left: Model's representation of a hypercolumn with their corresponding connections and weight vectors W_i.*

III. A Biologically Plausible Model for Cortical Architecture

In this work, we extend the cortically inspired computational model proposed by Hashmi et al. [8], [9]. The traditional approach of Artificial Neural Networks (ANN) is to seek some inspirations 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 neurons, 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 mainly discuss our model in the context of visual cortex for two main reasons. First, the visual cortex is a part of the neocortex that has been extensively studied by the neuroscience community. Second, to test our algorithm, we use images of handwritten digits obtained from MNIST database (http://yann.lecun.com/exdb/mnist).

A. Input

In case of the mammalian visual cortex, the responses of retinal cells is transferred to the Lateral Geniculate Nucleus (LGN) cells [15] via nerve paths. 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 [26], roughly operating like a pixel sensor. Input images are processed using the LGN transform [26] 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.

B. 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. 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 III-D). 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 describe by Equation 1.

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

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

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

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

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

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

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

T in Equation 2 determines the tolerance of a minicolumn to noise, set here to 0.95. 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 [16]. We empirically observed this non-linearity to be necessary for proper functional behavior of our hypercolumn model.

C. Synaptic Weight Update Rule

Hebbian learning [3] 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). This weight modification is only applied to active minicolumns x_i in accordance to Hebbian learning. As a result, at each level, minicolumns will progressively react most strongly to inputs they receive repeatedly, in effect *learning* them. In the visual cortex, these inputs correspond to shapes, which become increasingly more complex in the upper levels.

D. 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, each 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, in our model 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. We empirically observed that this random firing allows a great variety of learnable features to emerge, but stopping random firing is necessary to stabilize columns which have converged.

The biological origins of random firing and the fact it stops after repeated activity are based on several observations. 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 [6]. As a result, the random activity caused by synaptic noise no longer has a significant impact.

E. 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. In case of the visual cortex, at the lowest level (V1), minicolumns learn to identify edges of different orientation. Thereafter, subsequent levels learn to recognize more complex shapes (V2, V4), while upper level of the hierarchy (IT) ultimately recognizes the object under focus [7].

Our cortical network model also uses a hierarchical design to accomplish complex tasks. Figure 2 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 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 3 shows a typical visual recognition task we have used for training and testing our cortical networks.

Finally, Figure 2 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 representation 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 [31]; we are currently working to extend our model to incorporate their functionality. However, in this paper we consider and model the feedforward paths only, which are capable of many unsupervised learning tasks.

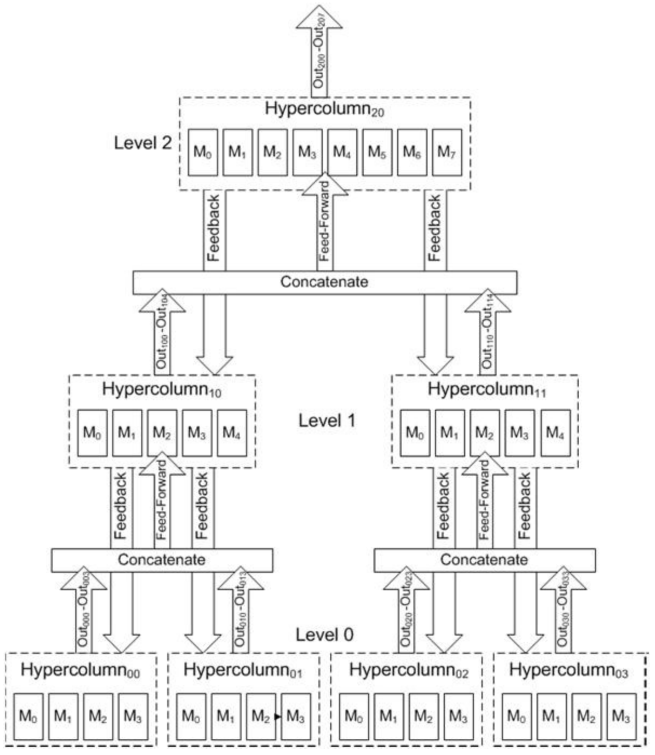

Figure 2. *The cortical network is organized as a hierarchy of hypercolumns with corresponding feedforward and feedback connections.*

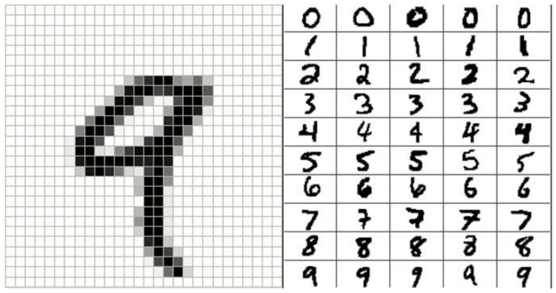

Figure 3. *Left: A sample of a visual recognition task, digits from the MNIST database. Right: Other example handwritten digits (lower resolution).*

IV. 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. Even though these models claim to be biologically plausible, their learning and connectivity rules are quite far from the biological example.

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 [28]. This form of learning is known as supervised learning. In biology, it is much more likely that learning is accomplished via unsupervised or semi-supervised learning. In unsupervised learning, labels are not provided, but classification is achieved entirely through similarity of features. In semi-supervised learning, only a few of the many objects have labels, and classification is based on similarity to the labeled objects [28].

These traditional perceptron-based ANNs have even been ported to the GPU with some success [2], [14]. Other learning algorithms have had similar success achieving significant speedups using GPGPUs. Nages et al. [19], [20] have simulated thousands of spiking neurons on the GPU, taking advantage of such optimizations as memory coalescing and achieving up to a 26x speedup. Finally, Raina et al. [24] have implemented deep unsupervised learning algorithms on a GPU with 5-15x speedups. The cortical network algorithm we consider here is able to learn features from its dataset in an entirely unsupervised fashion. We also consider that the cortical algorithm is able to learn unique features in a distributed manor 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.

Rice et al. [25] have proposed a neocortex-inspired cognitive model on the Cray XD1 supercomputer. The HTM, based on a hierarchical Bayesian network model proposed in [11], 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. [25] 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. Furthermore, we have opted to use commodity GPGPUs instead of a supercomputer and FPGAs to effectively scale our model.

Finally, profiling based runtime systems as StarPU [1] and Harmony [5] 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.

V. 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 such 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.

A. 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 explicitly configure. 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) [23]. 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 between 1 and 8 CTAs concurrently on each SM depending on a number of factors, including the number of threads scheduled per CTA, the number of registers used per thread, and the amount of shared-memory used by each CTA [23]. 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 a read-only texture caches, minimizing synchronization and thread divergence, and optimizing global memory accesses with memory coalescing [29].

B. 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 grids or kernels. Fitting the cortical network to the CUDA software model is achieved by mapping the

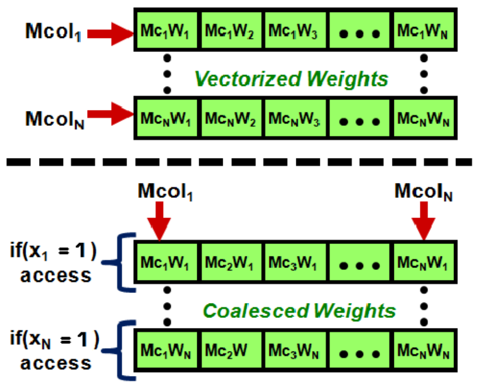
Mcol₁
Mc₁W₁ Mc₁W₂ Mc₁W₃ ••• Mc₁Wₙ
Vectorized Weights
Mcolₙ
McₙW₁ McₙW₂ McₙW₃ ••• McₙWₙ
Mcol₁
Mcolₙ
if(x₁ = 1) access
Mc₁W₁ Mc₂W₁ Mc₃W₁ ••• McₙW₁
Coalesced Weights
if(xₙ = 1) access
Mc₁Wₙ Mc₂W Mc₃Wₙ ••• McₙWₙ

Figure 4. *Top: Naively, 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.*

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 that connect the minicolumns within a hypercolumn. For a hypercolumn to learn more distinct features from a set of inputs, the number of minicolumns can be increased. For example, if we want each hypercolumn to learn 128 unique features, 128 minicolumns must exist in each hypercolumn (or 128 threads per CTA).

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 III, 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. Naively, 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 in $O(\log n)$ 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 4. 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 segment 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 space with one memory transaction. In primary experiments, coalescing these weights contributed over a 2x speedup for 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 4).

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 [4]; 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 VI will detail some of the inefficiencies we discovered using this approach, as well as some solutions we have explored to mitigate them.

C. Experimental Setup

We compare the performance of our parallelized CUDA implementation of the cortical network algorithm with the original single-threaded C++ implementation of the algorithm. The single-threaded implementation was run on an Intel Core i7 @ 2.67 GHz with 12GB of RAM, while the CUDA implementations were executed on a GT200 architecture GeForce GTX 280 and Fermi architecture C2050 (more details in Table I). 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 [23]. While the GTX 280 is compute capability 1.3, we do not explicit utilize any of the additional features and found better performance when compiled as

[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.

	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 I

Configurations of hypercolumns and their resulting occupancy on the GPU.

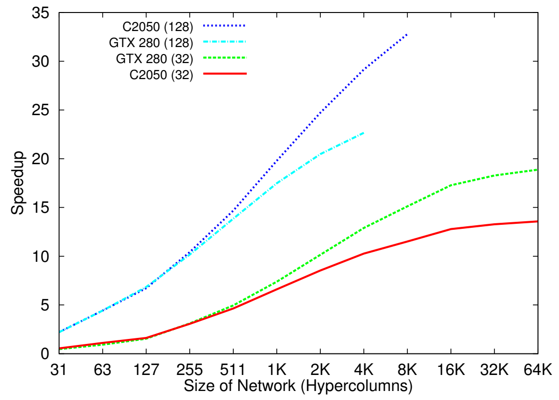

Figure 5. *Speedups of various cortical networks over the single-threaded CPU implementation.*

compute capability 1.1. To measure performance, we examined 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 chose to examine 32-thread and 128-thread implementations simply to examine a couple different configurations of the cortical network algorithm. In biology, hypercolumns typically contain dozens to hundreds of minicolumns as well [17]. In future work, we anticipate the number of minicolumns will be determined by the application or the specific area 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 [10], though this work focuses on the scalability of two different static configurations. Table I details the resulting occupancy of both GPUs for the configurations we tested, obtained by using the CUDA Occupancy Calculator tool [23]. 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.

D. Results of CUDA vs. Serial Implementation

Figure 5 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 I, 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 [23]. 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. Given these considerations, it is reasonable to believe that the configuration of 32-minicolumns (threads) is likely to be memory latency bound, and neither GPU has enough live threads to adequately hide the memory latency (though, the affect seems to be worse for the C2050).

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, but also the total 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. 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 I). 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. There also are more threads available to hide memory latencies, so an increased number of processing cores helps performance. However, we do see when compared to the 32-minicolumn configuration, clearly both GPUs benefit significantly from having a larger number of threads to hide memory latency.

While we have not implemented a multithreaded cortical network for the CPU, we believe our CUDA parallelized implementation of the algorithm will still show a significant speedup. If we 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. Furthermore, if we parallelize the C++ model we can also potentially gain a 4x speedup by distributing the cortical network across the four cores of the CPU. However, even if we consider this overhead-free perfectly optimized CPU model, our CUDA implementation still exhibits up to an 8x speedup.

VI. 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 some observations on a couple of inefficiencies in 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 [33], 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.

A. *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 6 shows the percent of execution time spent on additional kernel launch overhead for the 128-minicolumn configured networks on both the GTX 280 and C2050. 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 observed 1-4% of total execution time spent on this overhead on both GPUs. While such a small portion of the overall execution time may not seem unreasonable, we note that this is is purely synchronization overhead, and optimally we would like to eliminate such overhead entirely.

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, this parallelism diminishes for the upper levels of the network. The cortical network algorithm learns the features of the input in a hierarchically distributed manor; lower levels have a limited receptive field and process simpler features, while upper levels concatenate and combine these features - and ultimately, learn full objects or scenes. However, it is this convergent property of the cortical network 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 7 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.

B. *Pipelining to Increase Resource Utilization*

From Figure 7, we are able to see how the hierarchical design of our cortical architecture results in poor utilization of the GPGPU's resources for the upper levels. 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 architecture, but we are unable to do so due to aforementioned data dependencies between levels.

Our 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 8 shows a simple example of pipelining between two stages of a cortical architecture 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).

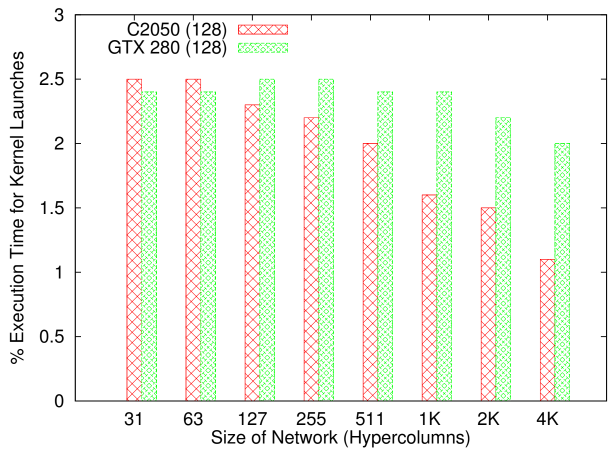

Figure 6. *Overhead of the additional kernel launches needed for different scale cortical networks (128-minicolumn configuration).*

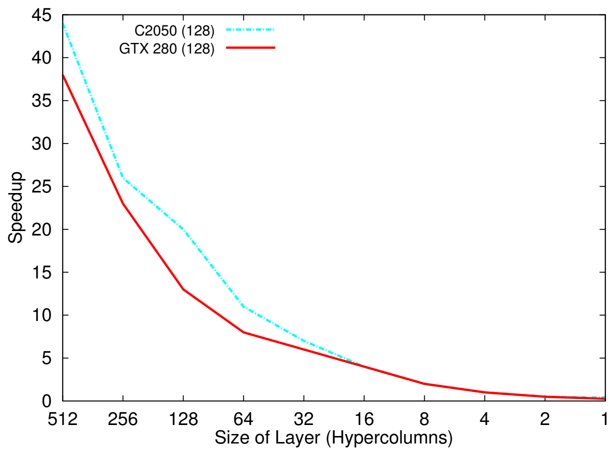

Figure 7. *Level-by-level speedups for a cortical network of 1023 hypercolumns. The lowest level of the cortical network is on the left.*

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. Another disadvantage of this implementation is that the amount of global memory dedicated to input/output activations doubles.

C. *Kernel Fusion Using a Queue*

Ideally we would like to be able to execute the entire cortical architecture on the GPU concurrently, reducing the overhead to a single kernel launch. 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 [23]. 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 in Figure 9. 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 9 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 9, 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 a couple assumptions about the underlying hardware of the CUDA enable 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. 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. In fact, some on-GPU barrier synchronization techniques share the same 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 [34]

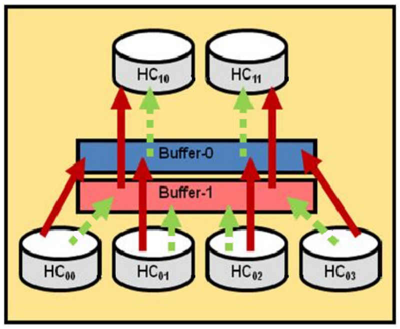

Figure 8. *Separate buffers are read-from and written-to between levels on a particular kernel launch.*

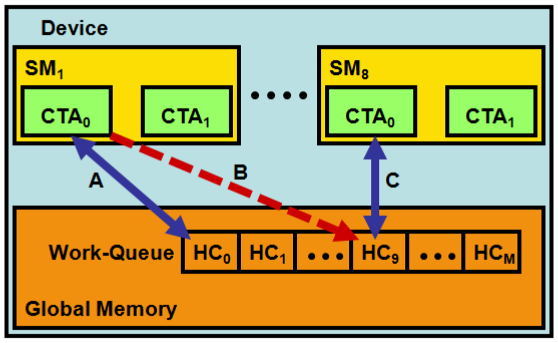

Figure 9. *Software work-queue implementation.*

This work-queue method is quite successful because many of the hypercolumns have no direct interaction with each other. Typically the child nodes have already written their activations to global memory before a parent hypercolumn is scheduled. Only at the uppermost hypercolumns in the network will one CTA need to spin-wait, as a parent and child hypercolumn may now be concurrently scheduled. However, to reduce the amount of time spent waiting for these dependencies, the hypercolumn will write activations to the global memory as soon as they have been calculated, 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 to determine which has the maximum response to the current inputs. 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.

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. Another 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
    $q \leftarrow WorkQueue[\textbf{atomicInc}(qHead)]$ //pop first item
end if
while $q \neq empty$ do
    $s\_stateVars \leftarrow g\_stateVars$ //load some state variables
    if $tid == 0$ then
        while $myFlag \neq ready$ do
            //spin-wait for ready
        end while
        $s\_activeInputs \leftarrow g\_activeInputs$ //load inputs
    end if
    __syncthreads()
    $s\_activation[tid] \leftarrow \textbf{computeActivation}()$
    __syncthreads()
    $s\_activation[tid] \leftarrow \textbf{computeWTA}()$
    $g\_activation \leftarrow s\_activation[tid]$
    __threadfence() //flush activations to memory
    if $tid == 0$ then
        $\textbf{atomicInc}(parentFlag)$
    end if
    updateSynapticWts() //perform local updates
    $g\_state \leftarrow s\_state$
    if $tid == 0$ then
        $q \leftarrow WorkQueue[\textbf{atomicInc}(qHead)]$ //pop first item
    end if
end while
```

VII. 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 which proportionally allocates a given cortical network across the host CPU and one or more homogeneous or heterogeneous GPUs.

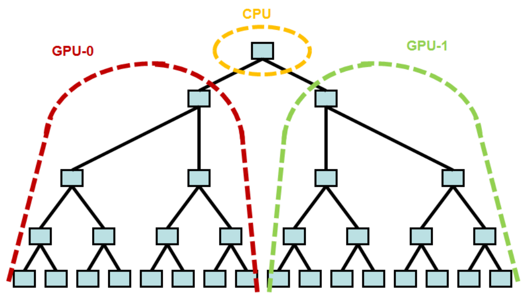

Figure 10. *Naively, the easiest method to split a cortical network across a system of host CPU and multiple GPUs would be to divide it evenly.*

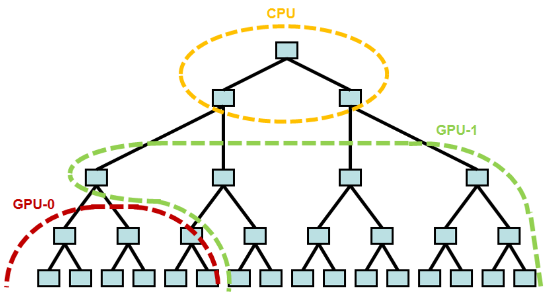

Figure 11. *The online optimizer tool finds the relative performance of a cortical network on the host CPU and one or more heterogeneous GPUs, then proportionally allocates the network to maximize performance.*

A. *Partitioning Cortical Networks Between CPU and GPU*

As seen in the preliminary results of Section V, 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 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 a cortical network hierarchy not utilizing the optimizations outlined in Section VI. When a network is allocated, our online profiler creates a sample cortical 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 it takes 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.

B. *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 5, one configuration of our cortical network exhibited better performance on the GTX 280 GPU, while the other was better on the C2050 GPU. While the simplest solution would be to simply partition the network equally across the available GPUs (see Figure 10), a number of factors would affect the actual execution of each partition. GPUs may have a different number of SMs, clock speeds, 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.

In multi-GPU systems, profiling is performed first 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 11 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 [30] without profiling. However, for our cortical networks, profiling imposes only a minor runtime overhead; does not require 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.

C. *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 upper levels of the cortical network on the host CPU. From experimentation, it was 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 dominant GPU simply takes over to execute the upper levels of the network. The pipelining implementation required no additional complexity here. For simplicity, an additional work-queue is allocated by the profiling tool for the upper levels of the distributed cortical network. Once each GPU has finished executing their proportional work-queue enabled cortical network segments, their input activations are transferred to this final work-queue which executes the top most portion of the cortical network.

VIII. 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.

A. Experimental Setup for Optimizations

Two systems 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 speeups reported are relative to the single-threaded implementation of the cortical network run on the Intel Core i7 processor.

B. Single GPU Optimization Results

Figure 12 shows the speedups of the pipelining and work-queue optimizations over the naive 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. The pipelining optimization slightly outperforms the work-queue, though this is expected as the work-queue utilizes atomic primitives for synchronization. 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 V, this configuration is likely memory latency bound. The performance 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 (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 13 we see the performance results of the cortical network optimizations configured with 32-minicolumns on the GTX 280 GPU, and in Figure 14 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 at 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 15. 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 [22], we see that the GigaThread scheduler of previous architectures managed 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 requires slow atomic operations on global memory for synchronization, they outperform 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. These CTAs still use the double buffer to propagate activations. However, rather than relying on the global CTA scheduler to execute one CTA per hypercolumn, each CTA executes a portion of the overall cortical network until every hypercolumn has executed. In Figures 13, 14, and 15 Pipeline-2 shows the results of this new optimization. As expected, this optimization outperforms the work-queue, as it does not require any atomic synchronization, nor does it suffer from the

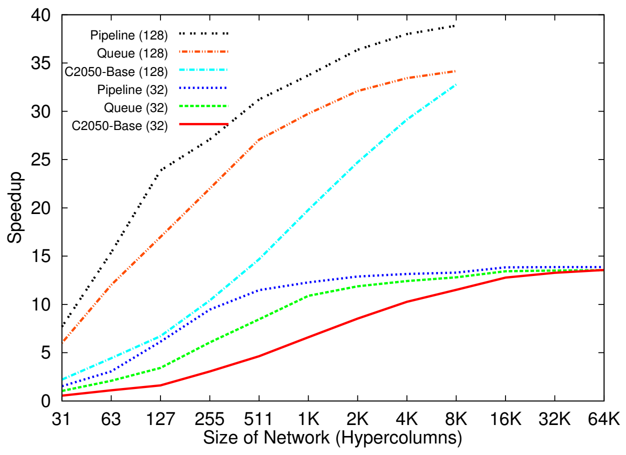

Figure 12. *Speedups achieved on C2050 using pipelining and work-queue optimizations.*

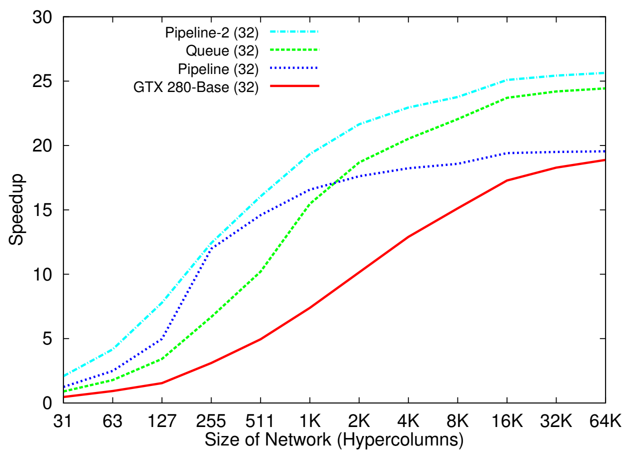

Figure 13. *Speedups achieved on GTX 280 using optimizations for 32-minicolumn cortical networks.*

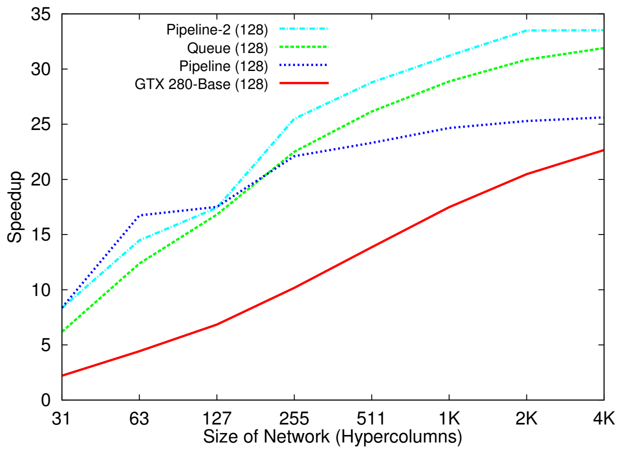

Figure 14. *Speedups achieved on GTX 280 using optimizations for 128-minicolumn cortical networks.*

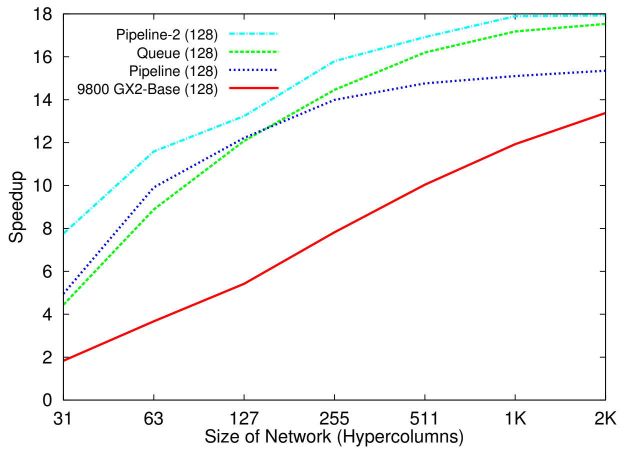

Figure 15. *Speedups achieved on 9800 GX2 using optimizations for 128-minicolumn cortical networks*

possible limitations of the GigaThread scheduler [22]. 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.

C. Profiled Multi-GPU Results

In Figure 16 we examine the performance benefit gained by using our cortical network online profiling tool. We compare a naively distributed cortical network (referred to as "Even" in Figure 16) with a network that has been profiled and proportionally allocated across the host CPU, the GTX 280 and C2050 GPUs ("Profiled" in Figure 16). The naively distributed network executes the top hypercolumn on the CPU and splits the lower levels of the network evenly across the GPUs (see Figure 10). For the cortical network configured with 32-minicolumns, we remember that the GTX 280 performs better (refer to Figure 5), 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 unoptimized network.

The cortical network 128-minicolumn configuration performs better on the 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 unoptimized 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.

Combining the cortical network optimizations with profiling resulted in even better speedups. Again, for both network configurations considered, the pipelining optimization

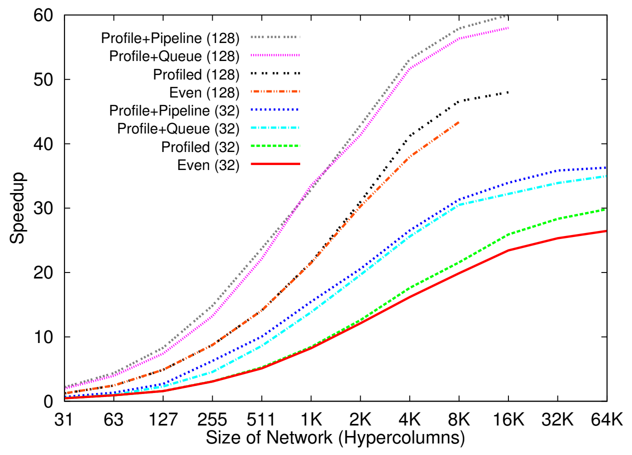

Figure 16. *Speedups achieved using profiling together with execution optimizations on a heterogeneous GPGPU system (C2050 and GTX 280).*

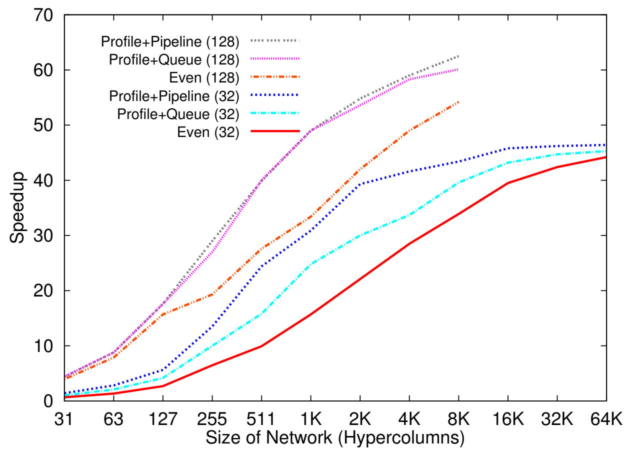

Figure 17. *Speedups achieved using profiling together with execution optimizations on a homogeneous GPGPU system (two 9800 GX2s, for a total of 4 homogeneous GPUs).*

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 17 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.

IX. 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 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 and 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.

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. These optimization techniques were also applied to the multi-GPU cortical networks, resulting in 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 award CCF-0702272, as well as GPU donations from Nvidia.

References

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

[2] Billconan and Kavinguy. A neural network on gpu. *http://www.codeproject.com/KB/graphics/GPUNN.aspx.x*.

[3] R. E. Brown and P. M. Milner. The legacy of Donald O. Hebb: more than the Hebb synapse. *Nat Rev Neurosci*, 4(12):1013–1019, Dec 2003.

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

[5] G. Diamos and 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*, pages 197–200. ACM, 2008.

[6] R. Douglas, C. Koch, M. Mahowald, K. Martin, and H. Suarez. Recurrent excitation in neocortical circuits. *Sciense*, 269(5226):981–985, 1995.

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

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

[9] A. Hashmi and M. Lipasti. Discovering cortical algorithms. In *Proceedings of the International Conference on Neural Computation (ICNC 2010)*, 2010.

[10] A. Hashmi, A. Nere, and 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, New York, NY, USA, 2011. ACM.

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

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

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

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

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

[16] A. Losonczy and J. Magee. Integrative properties of radial oblique dendrites in hippocampal ca1 pyramidal neurons. *Neuron*, 50:291–307, 2006.

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

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

[19] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and 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*, pages 3201–3208, Piscataway, NJ, USA, 2009. IEEE Press.

[20] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5-6):791 – 800, 2009. Advances in Neural Networks Research: IJCNN2009, 2009 International Joint Conference on Neural Networks.

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

[22] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi.

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

[24] R. Raina, A. Madhavan, and A. Y. Ng. Largescale deep unsupervised learning using graphics processors. In *International Conf. on Machine Learning*, 2009.

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

[26] D. Ringach. Haphazard wiring of simple receptive fields and orientation columns in visual cortex. *J. Neurophysiol.*, 92(1):468–476, Jul 2004.

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

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

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

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

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

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

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

[34] S. Xiao and W. chun Feng. Inter-block gpu communication via fast barrier synchronization. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 2010.