Evaluation of GPU Architectures using Spiking Neural Networks

Vivek K. Pallipuram, Mohammad A. Bhuiyan and Melissa C. Smith

Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA
{kpallip, mbhuiya, smithmc}@clemson.edu

***Abstract*— During recent years General-Purpose Graphical Processing Units (GP-GPUs) have entered the field of High-Performance Computing (HPC) as one of the primary architectural focuses for many research groups working with complex scientific applications. Nvidia's Tesla C2050, codenamed Fermi, and AMD's Radeon 5870 are two devices positioned to meet the computationally demanding needs of supercomputing research groups across the globe. Though Nvidia GPUs powered by CUDA have been the frequent choices of the performance centric research groups, the introduction and growth of OpenCL has promoted AMD GP-GPUs as potential accelerator candidates that can challenge Nvidia's stronghold. These architectures not only offer a plethora of features for application developers to explore, but their radically different architectures calls for a detailed study that weighs their merits and evaluates their potential to accelerate complex scientific applications. In this paper, we present our performance analysis research comparing Nvidia's Fermi and AMD's Radeon 5870 using OpenCL as the common programming model. We have chosen four different neuron models for Spiking Neural Networks (SNNs), each with different communication and computation requirements, namely the Izhikevich, Wilson, Morris Lecar (ML), and the Hodgkin Huxley (HH) models. We compare the runtime performance of the Fermi and Radeon GPUs with an implementation that exhausts all optimization techniques available with OpenCL. Several equivalent architectural parameters of the two GPUs are studied and correlated with the application performance. In addition to the comparative study effort, our implementations were able to achieve a speed-up of 857.3x and 658.51x on the Fermi and Radeon architectures respectively for the most compute intensive HH model with a dense network containing 9.72 million neurons. The final outcome of this research is a detailed architectural comparison of the two GPU architectures with a common programming platform.**

I. INTRODUCTION

Various reports [1, 2] predict that without significant technological breakthroughs, we will approach the limits of Moore's Law between 2012 and 2015. Practical considerations, such as power and cooling of large-scale systems [3], are also forcing the exploration of alternative computing architectures. The current trend for processor vendors is to mitigate these limiting factors with multi- and many-core architectures. Leading vendors such as Intel, IBM, AMD and Oracle-Sun are introducing prototypes that can theoretically achieve more than 1 Teraflop [4]. Additionally, General-Purpose Graphical Processing Units (GP-GPUs) are now commonly used in High-Performance Computing (HPC) to accelerate computationally dense algorithms.

The use of GP-GPUs for scientific computing has grown exponentially since the introduction of CUDA by Nvidia Corporation in November 2006 [5]. Several research groups have employed Nvidia's GP-GPUs with CUDA for simulation of large data-parallel algorithms such as the Smith Waterman algorithm [6], NAMD [7], and others of significant concern in the biological and physical sciences community. Given the growing interest in GP-GPUs by researchers, GPU vendors are adding cores and improving the compute capabilities. Nvidia's Fermi and AMD/ATi's Radeon are recent releases that offer significant computational potential via a large number of multiprocessors. While Nvidia's combination of GPUs and CUDA programming model has held a major share of the scientific computing market, Open Computing Language (OpenCL), a standard maintained by the Khronos group, is gaining interest [8]. Unlike CUDA, OpenCL can support a variety of architectures such as GPUs from major vendors (Nvidia and AMD), multi-, and many-core architectures. OpenCL thus provides an opportunity for code portability making it very attractive to the developer community and opens a path for AMD GPUs to enter the HPC community. Further, it allows for interesting comparisons of these architectures.

In this paper, we compare the two leading GPU architectures using four Spiking Neural Network (SNN) models: the Izhikevich model [9], Wilson model [10], Morris Lecar model [11] and Hodgkin Huxley model [12] that have been found to satisfactorily replicate neuronal properties for accurate brain modeling [13]. Each of these models has different computation-to-communication requirements facilitating a broad architectural comparative study.

For the comparative study, we implement a two-level character recognition network based on the aforementioned SNN models that can recognize 48 alpha-numeric characters originally developed in [14]: English characters (A-Z), 10 numerals (0-9), 8 Greek letters and 4 symbols. The SNN models were scaled up to 9.7 million neurons as described in [15]. The acceleration and optimization tasks were used for a detailed comparative study of the Fermi and Radeon architectures, which is the focal contribution of this paper. We analyze the performance differences by studying several architectural parameters such as global memory requests, ALU packing, ALU

978-0-7695-4448-9/11 $26.00 © 2011 IEEE
DOI 10.1109/SAAHPC.2011.20

stalls, global cache hits/misses, etc, and link these parameters to application characteristics such as Flops/Byte ratio. Our OpenCL parallel implementations achieved an application speed-up for the HH model (the most compute intensive) as high as 857.3x on Fermi and 658.51x on Radeon over the serial implementations on a 2.66 GHz Intel Core 2 Quad. Although comparison of the GPU parallel implementations with the multi-core OpenCL implementation will be interesting, our primary focus is to compare the performance of the two GPU architectures, therefore we have limited our studies to a single-threaded serial implementation. The remaining paper is organized as follows: Section II provides a background on the GPU architectures (Fermi and Radeon), a background on the OpenCL programming model, SNN models and the two-level network used for the study. Section III discusses related work. Section IV presents the details of the experimental set-up and describes our implementation of the SNN models that exhausts all typical and relevant optimization techniques available with OpenCL for the mentioned GPU architectures. Section V presents the results and analysis, Section VI summarizes the research results and the paper is concluded in Section VII with conclusions and future work.

II. BACKGROUND

A. GPU Architectures

Originally intended as a fixed many-core device dedicated to transforming 3-D scenes to 2-D images, GPU architectures now provide an established computing accelerator platform for the HPC community. The recent architecture advancements with Nvidia's Fermi and AMD's Radeon have changed the GPU architecture from predominately a graphics device to a more general computing accelerator, as we will explore in the next sub-sections.

A.1 Nvidia's Fermi

Nvidia's previous 10-series architecture views a GPU as an array of streaming multiprocessors (SMPs), each multiprocessor containing in itself, a set of scalar processors, a double-precision (DP) unit, shared memory for thread cooperation, and texture addressing and texture fetch units. A group of threads, called a *thread block*, is executed on the multiprocessor while individual threads are executed on the scalar processors. More recently, the 20-series architecture, codenamed Fermi, has brought much innovation versus previous architectures: 512 CUDA cores organized as 16 SMPs with 32 cores each gathered around an L2 cache. A Gigathread scheduler dispatches thread blocks to the SMP thread schedulers. The GP-GPU has the capability of supporting 6 GB of GDDR 5 DRAM memory. SMPs are provided with an instruction cache, dual warp schedulers and dispatch units. SMPs now have two sets of 16 CUDA cores, 4 special function units for transcendental functions, 16 load/store units, a hefty register file, and most importantly, a configurable 64 KB of shared memory/L1 cache. The SMPs also share a second level L2 cache. More information about the architecture can be found in [16]. For our experiments, we have used a single Nvidia Tesla C2050 which belongs to Compute Capability 2.0 and has 14 multiprocessors (448 cores), 2.6 GB global memory, 64 KB shared memory/L1 cache per multiprocessor, 768 KB L2 cache, 64 KB constant memory, and operates at a clock rate of 1.15 GHz. The system's GDDR interface offers data bandwidth up to 144 GB/sec.

A.2 AMD/ATi Radeon 5870

The AMD/ATi Radeon 5870 used in our study has 1600 ALUs organized in a different fashion compared to the Fermi. The ALUs are grouped into five-ALU Very Long Instruction Word (VLIW) processor units. While all five of the ALUs can issue the basic arithmetic operations, only the fifth ALU can additionally execute transcendental operations. The five-ALU groups along with the branch execution unit and general-purpose registers form another group called the *stream core*. This translates to 320 stream cores in all, which are further grouped into *compute units*. Each compute unit has 16 stream cores, resulting in 20 total compute units in the ATi Radeon 5870. Each ALU can execute a maximum of 2 single-precision Flops: multiply and add instruction per cycle. The clock rate of the Radeon GPU is 850 MHz; for 1600 ALUs this translates to a throughput of 2.72 T Flops/s. The Radeon's memory hierarchy includes a global memory, L1 and L2 cache, shared memory, and registers. The 1 GB global memory has the peak bandwidth of 153 GB/s and is controlled by eight memory controllers. Each compute unit has 8 KB L1 cache and 32 KB of shared memory. Multiple compute units share a 512 KB L2 cache. The 256 KB register space is available per compute unit, totaling 5.1 MB for the entire GPU. More information on the Radeon architecture can be found in [17].

B. The Programming Model: OpenCL

In our research, we have used OpenCL built on the CUDA platform [18]. OpenCL and CUDA are conceptually identical with kernels operating as device functions running on the GPU. When a kernel is launched, a scalar processor executes a CUDA thread for each OpenCL *work-item*. An OpenCL work-item is identified using its *local ID* and *work-group ID*. The work-items are then collected into *work-groups*. The work-groups are further divided into SIMD groups of 32 work-items called *warps,* which are further divided into groups of 16 work-items called *half-warps.* The memory hierarchy of OpenCL comprises of a R/W *private memory* local to a work-item*,* a R/W *local memory* local to a work-group*,* a read-only *constant memory* and R/W *global memory* shared by the entire kernel*.* OpenCL offers three primary optimization categories: *Memory Optimization*, *Instruction Optimization,* and *Execution Configuration Optimization.*

The *memory optimization techniques* for OpenCL used in our work are: *Software Pre-fetching* (SP), *Shared Memory* (SM), and *memory write function* (MW). SP essentially involves the use of fast registers to store model parameters for computation providing significant clock cycle savings. OpenCL SM has been used in our implementations to minimize the communication between the device and the host, which will be discussed further in Section IV. The *clCreateBuffer()* function was used to create a buffer and copy the data from the host to the GPU. Performance improvement can be achieved if the buffer is created in the initialization part and then the function *clMemWrite()* is used inside the execution loop of the host function. We call this optimization *memory write* (MW).

For *instruction level optimization* techniques, OpenCL provides *native math* functions and *unsafe math (UM)* that each reduces the number of clock cycles for an operation. The native math functions are nearly as accurate (error bound information provided in [18]) as the regular math functions and provide a significant performance improvement in some cases. In UM, the compiler optimizes the floating-point arithmetic in the kernel but it may violate IEEE 754 standards and OpenCL numerical compliance requirements. The next instruction-level optimization technique is to *reduce conditional statements* (RCS) whenever possible. If conditional statements are present, the thread execution in a local work group will be serialized, which will eventually slow down the kernel execution. OpenCL additionally offers constructs to exploit *vectorization* with the Radeon architecture. The Fermi architecture does not perform optimally with *vectorization,* hence the use of this technique is limited to the Radeon architecture in this research.

Execution level optimization involves changing the work group size to maximize the *multiprocessor occupancy* The multiprocessor occupancy is defined as the ratio of the number of active warps to the number of warps that can physically run on the multiprocessor. It is worth mentioning that a high multiprocessor occupancy does not directly imply performance, but rather assists in hiding latency in memory bound kernels.

C. Spiking Neural Networks

Several neuron models have been proposed to capture neuronal behavior [19]. According to Izhikevich [13], four SNN models, namely, the Izhikevich model, Wilson model, Morris Lecar (ML) model, and the Hodgkin Huxley (HH) model were found to satisfy the requirements of accurately modeling the neuron dynamics, and hence were used in this research.

The Hodgkin Huxley model is considered to be the most accurate and the most important model in the neuroscience community till date. As mentioned in [13], the model involves 4 equations and ten parameters describing various neuron current activation and deactivation. The model takes 120 flops per 0.1 ms time-step and hence 1200 flops/1ms for the update. In our research, we have used 0.01 ms time-step for the neuron update.

The Morris Lecar model is another biophysically meaningful model, replicating almost all the spiking neuron properties. The relevant equations, found in [11], include evaluating hyperbolic functions making this model more complex than the two mentioned below. The model takes 60 flops per 0.1 ms time-step and hence 600 flops/1ms for the update. For our experiments, we have used a plausible 0.01 ms time-step to simulate a network developed using the ML model.

The Wilson model [10] introduces a few additional conduction channels to the HH model as reported in [13]. With proper tuning of the channel parameters, the Wilson model can mimic all the characteristics of spiking neurons. A time-step of 0.01 ms was used to evaluate the polynomial equations describing neuron dynamics. The model in general takes 45 flops per 0.25 ms time-step and hence takes 180 flops/1ms for the update.

In [9], Izhikevich developed a simple and very computationally efficient spiking neuron model that is almost as plausible as the most accurate Hodgkin Huxley (HH) model. Izhikevich's model requires only 13 flops per neuron update and still sufficiently reproduces a majority of neuronal properties. The equations are found in [9]. In our research, we have used a 1 ms time-step (13 flops/1ms update) for neuronal dynamics update for the Izhikevich model.

We now provide the Flops/Byte ratio information for each of the aforementioned SNN neuron models. The Flops/Byte ratio is an algorithmic specific value that is defined as the ratio of number of floating-point operations required for a complete neuron update to overall bytes requested for all of the neuronal updates [15]. Table 1 gives the relevant Flops/Byte ratio information for each of the models studied.

Table 1. Flops/Byte Ratio for All the Models

Model	FLOPs per neuron update	Bytes per neuron update	Flops/Byte Ratio
HH	246	25	9.84
ML	147	17	8.65
Wilson	38	25	1.52
Izhikevich	13	13	0.9997

D. The Two-Level Network

The SNN used in this research is based on [14] and the overall network used is shown in Figure 1. The task of the network is to detect images from a training data set. The first level neurons act as an input collection layer and the second layer acts as output collection layer. Each neuron in level-1 corresponds to a pixel in the input image; hence the number of neurons in the input level is equal to the total number of pixels in the test image, thereby making it the most compute intensive layer of the network. The number of neurons in the output layer, level-2, is equal to the number of images in the database. When an input image is presented to level-1, each neuron evaluates its membrane potential based on the pixel level presented and the neuron model chosen. If the pixel is "on", a constant current is

supplied to the neuron in order to evaluate its membrane potential. The input current for a level-2 neuron is evaluated as:

$$I_j = \sum_i w(i,j) f(i)$$

Where, I_j is the net input current to the neuron j in level-2, $w(i,j)$ is the weight of the synapse connecting neuron i in level-1 and the neuron j in level-2. A neuron in any level is said to have "fired" if its membrane potential crosses a threshold value that is determined based on the neuron model chosen. The research presented in this paper accelerates the recognition phase of each network by implementing all of the level-1 neurons on the GP-GPU device, while the level-2 neurons are implemented on the host as will be discussed later in Section IV.

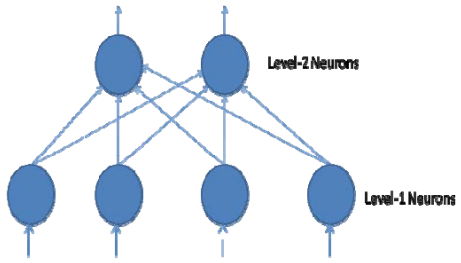

Figure 1. Two-level Character Recognition Network

III. RELATED WORK

Several recent research activities were motivated by the desire to model the neocortex. In [20], the authors have studied the mammalian neocortex in detail and successfully simulated a rat-size cortex in 42% of real-time and a cat-size cortex in 23% of real-time on a 442 node Dell Xeon cluster. They also implemented a two-layer version of the neural network model to detect 128x128 pixel images from the COIL-100 database [21]. Their neuron model fits in the integrate-and-fire (I&F) category, which according to Izhikevich, is insufficient for reproducing neuronal properties [13]. In [22], the authors successfully used Izhikevich's model to simulate a cat-size cortical model with 10^9 neurons and 10^{13} synapses using a BlueGene/P machine with 147,456 processors and 144 TB of main memory. The authors claim that their simulation scale is roughly 1-2 orders of magnitude smaller than the human cortex and 2-3 orders of magnitude slower than real-time.

Alternative computing architectures such as GP-GPUs are now being investigated for biologically realistic simulations. In [23], the authors have implemented Izhikevich's random network on Nvidia's GTX-280 with 1 GB memory and achieved a speed-up of 26x over an equivalent software implementation for a 100K neuron network simulation. Their work discusses mapping strategies on the GP-GPU to efficiently utilize the memory bandwidth and parallelism.

There has been a limited amount of work done to systematically compare GP-GPU architectures. In [24], the authors have studied the performance portability of OpenCL and have concluded that the performance is not portable. They have implemented TRSM and GEMM for their studies on both Fermi and Radeon architectures. However, they do not explicitly study the performance difference between the programming models or the architectures. In [25], the authors have compared GPU architectures from Nvidia for asynchronous communication. They have implemented Cahn-Hilliard equation using several values of field lengths on Tesla (GTX 260) and Fermi (GTX 480) architectures. Their implementations were able to achieve speed-up values as high as 219x with two Fermi GPUs and upto 180x with three Tesla GPUs when compared against the host CPU. Their comparative study focuses on asynchronous/synchronous memory performance of the architectures and does not consider other device parameters that can largely affect application performance such as the magnitude of cache hits/misses, divergent branches, instructions issued/executed, occupancy, etc. Our study not only varies the problem size but experiments with applications having different "communication-to-computation" requirements and correlates these characteristics with device parameters and performance. Additionally, in contrast with existing studies, we compare architectures from different vendors using a common programming model.

IV. EXPERIMENTAL SET-UP AND IMPLEMENTATION

A. Experimental Set-up

One of our single-GPU experimental systems consists of a single Tesla C2050 paired with a 2.66 GHz Intel Core 2 Quad host processor. The OpenCL implementation for the SNN models was developed using CUDA 3.0 SDK on 64-bit Ubuntu 10.04. All of the relevant profiler information was obtained using OpenCL visual profiler supplied with the CUDA SDK. The second experimental system consists of a single ATi Raedon 5870 paired with a 2.8 GHz Intel Core i7. The implementations were developed using the AMD APP SDK 2.4 for OpenCL. While the codes were developed on Ubuntu 10.04, Windows XP was used to collect the relevant AMD profiler data using the AMD APP profiler. The single-GPU OpenCL results on the Tesla C2050 are compared with the single-GPU OpenCL results on the AMD/ATi Raedon 5870. It is worth mentioning that since CUDA does not support AMD GPUs, OpenCL serves as a common programming platform to fairly compare the GPUs from Nvidia and AMD, and hence is chosen by the authors for this comparative study.

B. Implementation

As mention in sub-section II.D, level-1 is the most compute intensive layer of the network and hence the level-1 computations are performed on the GP-GPU device. The GP-GPU device then provides the host processor with the level-1 neuronal firing information, the *global firing vector,* which is

used by the host processor to obtain the level-2 neuron dynamics. The level-2 computations are implemented on the host processor since the level-2 neuron dynamics computation constitutes less than 5% of the total computation overhead and, implementing the level-2 dynamics on the GP-GPU would require the transfer of the weight matrix to the GP-GPU device memory. Hence any computational improvement obtained by implementing level-2 neuron dynamics will be insufficient to amortize the communication overhead involved in transferring the large weight matrix to the GPU device.

The OpenCL implementation of the SNN models on Fermi exhausts all of the *memory optimization* strategies mentioned in sub-section II.B. SP has been used to avoid frequent global memory accesses. SM has been used to avoid transferring the global firing vector in each time-step using *block firing vector.* The block firing vector acts as a collection of flags for the work-groups. Since the work-items are collected in work-groups of size: *blocksize*, the block firing vector is *blocksize* magnitude smaller than the original firing vector. If at any time-step the block firing vector contains information of a firing event, only then will the entire global firing vector be transferred from the device to host and then read by the host. Figure 2 illustrates the idea of the block firing vector where the first half of the global firing vector has all of its flag values equal to 0 resulting in a 0 value for the first flag in the block vector. The second half of the global firing vector has a flag set to 1, which sets the second flag of the block vector to 1. Threads in a block must cooperate with each other via shared memory to update the block firing vector just described. Additionally, MW has been used to provide optimized data transfers as mentioned in sub-section II.B.

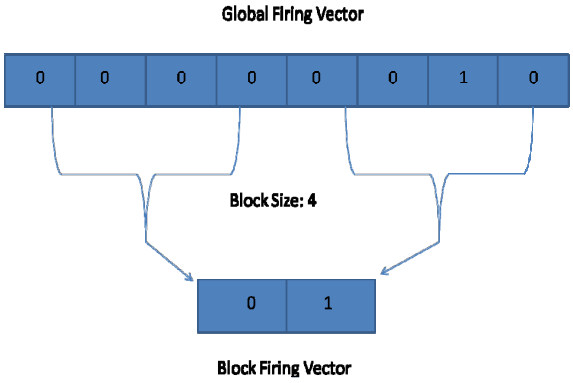

Figure 2. Block Firing Vector Concept Used in the OpenCL Implementation

The OpenCL implementation for Fermi also exploits the *execution configuration optimization* where we have experimented with several work-group sizes and have chosen the work-group size that yields the highest occupancy and performance. Our experiments with several block configurations yielded 256 as the preferred choice both for Fermi and the Radeon. Finally*, instruction level optimization* techniques such as RCS and native/unsafe math functions were incorporated in our Fermi implementation of the SNN models.

The implementation of the SNN models on the AMD GPU using OpenCL [18] is similar to the Fermi implementation. SP has been used to reduce the number of redundant access to the global memory. SM is used to reduce the transfer frequency of the global firing vector. In order to improve the performance, native math functions such as *native_exp()* and the unsafe compiler math optimization called the *cl_unsafe_math_optimization* have been used together with the reduction of conditional statements and the use of the memory write (MW) optimization.

V. RESULTS AND ANALYSIS

Figures 3-10 provides the total runtime, computation and communication time analysis for Nvidia's Fermi and AMD/ATi's Radeon 5870 for HH, ML, Wilson, and the Izhikevich models respectively. In what follows, we analyze the runtime performance of the Fermi and Radeon GPUs and provide an explanation for the observed performance by correlating it to the device parameter values given in Tables 3-11.

A. HH model

We have shown the total application runtime, GPU kernel time (computation time), and host-GPU communication time for the two GPU architectures in Figures 3 and 4. As indicated in these figures, for the largest network size (3120 x 3120), Radeon is 23% slower than Fermi although the theoretical peak vale of computation throughput of Radeon is 59% higher than that of the Fermi (Table 2). For the HH model, the Fermi computation and communication times are observed to be 13.4% and 19.9% faster respectively compared to Radeon for the largest network size. It is also observed that the other overhead time is also less for the Fermi architecture compared to the Radeon architecture. For other image sizes, a similar trend is observed.

The communication time is related to the data transfer time between the host and the GPU. Out of the "clEnqueueWriteBuffer / clEnqueueReadBuffer" and "clCreateBuffer" functions, it is verified that the "clEnqueueWriteBuffer / clEnqueueReadBuffer" function has less overhead. In both the Fermi and Radeon, the same efficient data transfer function was used but the results show that the Radeon GPU data transfer time is 19.9% slower than the Fermi GPU. One explanation for the slower communication is that the implementation of these functions in the AMD's OpenCL driver has more overhead compared to Nvidia's OpenCL driver. Another possible explanation is that the Radeon GPU hardware interface with the PCI-e is slower than that for Nvidia.

The computation time for the Fermi is also faster (13.4%) than that of the Radeon (Figures 3 and 4), although the theoretical computation throughput and the global memory bandwidth are higher for Radeon (Table 2). To explain this

observation, the profiler results were taken and the relevant parameters are reported in Tables 3 and 4. Nvidia provides "openclprof" profiler software whereas AMD supplies "AMD Stream Profiler" [26] for profiling. AMD also provides "Stream KernelAnalyzer" software [27], which was used to analyze the SNN kernel code to get more information about the implementation.

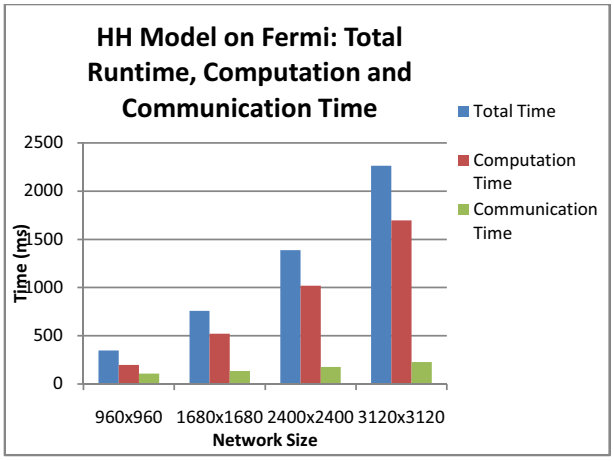

Figure 3. HH Model on Fermi: Total Runtime, Computation and Communication Time

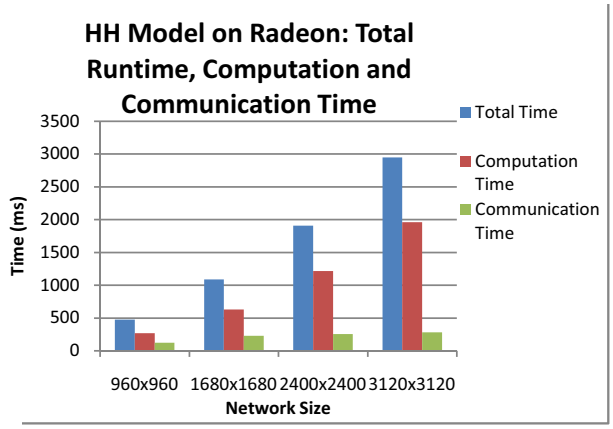

Figure 4. HH Model on Radeon: Total Runtime, Computation and Communication Time

Table 2. Comparison of Fermi and Radeon Architectural Peak Performance

Fermi		Radeon 5870	
Single Precision Throughput (TFlops/s)	**Global Memory Bandwidth (GB/s)**	**Single Precision Throughput (TFlops/s)**	**Global Memory Bandwidth (GB/s)**
1.105	144	2.72	155

The profiler parameters from Nvidia and AMD do not have direct one-to-one correspondence except for two parameters. The two parameters, *General Purpose Registers* (GPRs) per thread and the *Occupancy/ALUpacking* are common to both profiler reports. Nvidia uses the parameter *Occupancy* while AMD uses *ALUPacking*, though both have the same meaning.

We have used two image sizes, 960 x 960 and 3120 x 3120, to produce the profiler reports. The number of general-purpose registers (GPR) used by the Radeon, is a bit higher than the number of GPRs used by Fermi. But the GPRs are on-chip memory and thus do not greatly influence the performance. The number of scratch registers (residing in off-chip memory), which is allocated by the compiler, has a negative effect on the performance of the kernel; therefore minimal scratch register use is favorable. In the HH implementation on Radeon, zero scratch registers are used, meaning, there is no off-chip register communication and thus there will be no performance penalty for scratch registers.

ALU Packing for the Radeon implementation represents the packing of instructions on Radeon ALU, expressed as shown in Table 4. The parameter *ALUPacking* indicates how well the Shader Compiler (on AMD) packs the scalar or vector ALU in the kernel to the 5-way VLIW instructions. Values below 70% indicate that ALU dependency chains may be preventing the full utilization of the processor. The maximum value is 100 and would give the best performance. Here, a value of 87.59 is observed which implies that the ALU is not fully utilized. It is one of the reasons that the Radeon is slower than Fermi.

Table 3. Fermi Profiler Results for the HH Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	18	18
Occupancy	1	1
Gld request	4.2e6	4.35e7
Branch	2.12e6	2.22e7
L1 global hit	0	0
L1 global miss	4.1e6	4.3e7
Divergent Branch	1.14e5	1.01e6

Table 4. Radeon Profiler Results for the HH Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	27	27
ALU Packing	87.59	87.59
Scratch Register	0	0
Path Utilization	100	100
ALU Stalled by LDS	0	0
Fetch Unit Stalled	0.99	1.04
Write Unit Stalled	37.68	46.70

For the Fermi GPU, all of the ALUs are arranged as scalar processors as opposed to vector processors in the Radeon; and, HH model observes maximum occupancy value of 1. The other parameters of Table 3, *Gld request* (global load request), *L1 global miss* (Level 1 cache miss), and *divergent branches* (serialized branch) are all non-zero and increase slightly with the image size. Lower values of these parameters imply better performance. As there are no corresponding parameters for Radeon, we cannot compare these values between the two GPUs.

From Table 4 for the Radeon implementation, it is found that the *FetchUnitStalled* values (the percentage of GPU time the Fetch unit is stalled) of both network sizes are low (about 1), which is desirable. However, the *WriteUnitStalled* values (the percentage of GPU time the write unit is stalled) are high (average value of 42%) as seen in the same table, which is another reason that the Radeon is slower than the Fermi for the HH model. How to reduce these stalls in the user space is unknown to the authors. It is also noted that the *Path Utilization* (the percentage of bytes written through the *FastPath* compared to the total number of bytes transferred over the bus) is 100%, the maximum possible value. We have used the *FastPath* method to transfer data since it has proven to be faster than the *CompletePath* method.

B. *Morris-Lecar Model*

Figures 5 and 6 provide the computation time, and communication time analysis for the ML model implementation on the two GPU architectures. As seen from the figures, both Fermi and Radeon have comparable performance for the ML model. Though the Fermi implementation has better performance on the kernel with lower computation time than the Radeon kernel; Radeon's memory transfers were found to be better than those for the Fermi implementation.

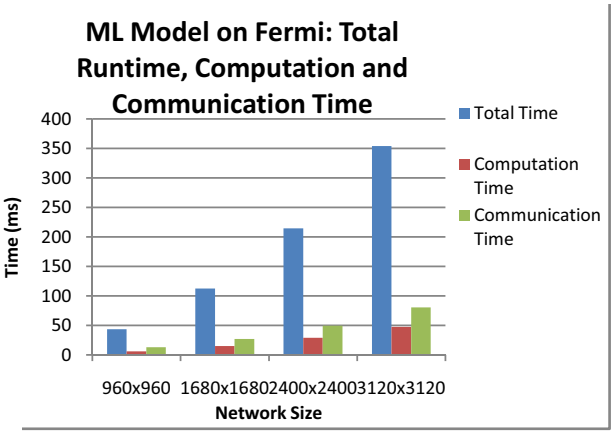

Figure 5. Morris-Lecar Model on Fermi: Total Runtime, Computation and Communication Time

As seen in Table 5, the Fermi architecture has a maximum *Occupancy* of 1, correspondingly from Table 6, the Radeon architecture has a high value of *ALUpacking* (89%), which are desirable values for both architectures. *FetchUnitStalled* remains at a low value for the Radeon implementation. In contrast to the HH model, the Radeon architecture now has lower *WriteUnitStalled* value leading to more optimal memory performance. The observation is primarily due to low memory requirements per neuron update for the ML model. As seen in Table 1, the ML model requires only 17 bytes versus 25 bytes per neuron update for the HH model.

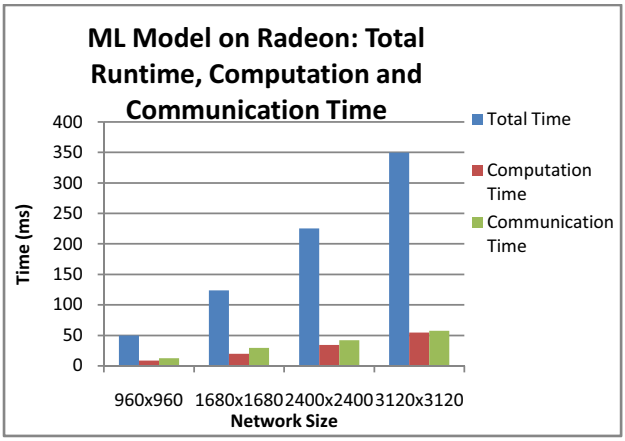

Figure 6. Morris-Lecar Model on Radeon: Total Runtime, Computation and Communication Time

Table 5. Fermi Profiler Results for the Morris-Lecar Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	18	18
Occupancy	1	1
Gld request	1.31e5	1.4e6
Branch	9.96e4	1.05e6
L1 global hit	0	0
L1 global miss	1.3e5	1.32e6
Divergent Branch	4.7e3	4.6e4

Table 6. Radeon Profiler Results for the Morris-Lecar Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	18	18
ALU Packing	89.17	89.17
Scratch Register	0	0
Path Utilization	100	100
ALU Stalled by LDS	0	0
Fetch Unit Stalled	0.80	0.10
Write Unit Stalled	25.70	25.80

C. *Wilson Model*

For the Wilson model (Flops/Byte ratio 1.57), the Fermi implementation outperforms the Radeon implementation with better kernel and memory performance. Figures 7 and 8 provide the relevant timing details for the two architectures. The Radeon kernel calls are (220.14-138.98)/220.14, i.e. 36% slower and memory transfers are (144.11-129.3)/129.3, i.e. 11.45% slower than the Fermi counterparts. Both the Fermi and Radeon architectures fare well with high *occupancy* (1) and *high ALUPacking* (91.45%) values respectively (Tables 7 and 8). However, the *WriteUnitStalls* values are higher compared to the ML model for both network sizes (40% and 47%) respectively, which explains the poor memory performance of the Radeon architecture. The *WriteUnitStall* values for the Wilson model are similar to those for the HH model, given their similar bytes per neuron requirements. We attribute the increased *WriteUnitStall*

values for the Wilson model to the relatively large number of bytes (25) required per neuron update, as seen in Table 1.

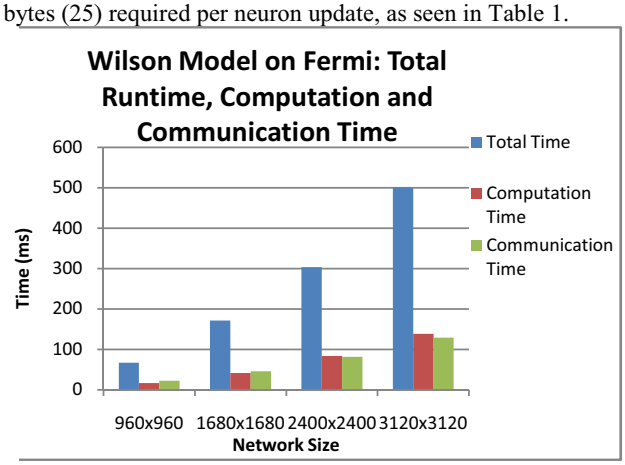

Figure 7. Wilson Model on Fermi: Total Runtime, Computation and Communication Time

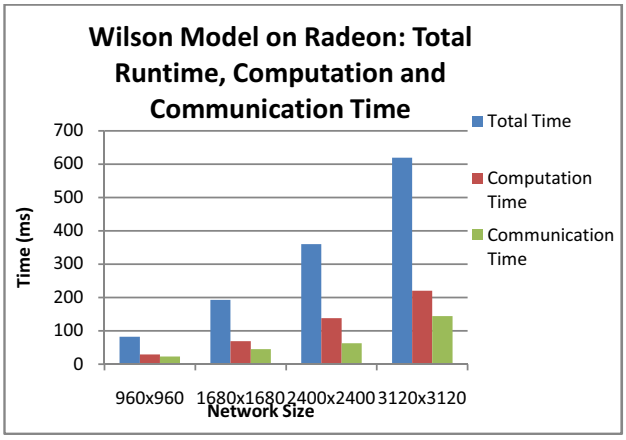

Figure 8. Wilson Model on Radeon: Total Runtime, Computation and Communication Time

Table 7. Fermi Profiler Results for the Wilson Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	19	19
Occupancy	1	1
Gld request	3.21e5	3.4e6
Branch	1.75e5	1.77e6
L1 global hit	0	0
L1 global miss	3.33e5	3.43e6
Divergent Branch	7.2e3	6.44e7

Table 8. Radeon Profiler Results for the Wilson Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	23	23
ALU Packing	91.45	91.45
Scratch Register	0	0
Path Utilization	100	100
ALU Stalled by LDS	0	0
Fetch Unit Stalled	2.91	2.20
Write Unit Stalled	40.00	47.96

D. Izhikevich Model

For the Izhikevich model, as seen in Figures 9 and 10, the results are similar to the HH model; similarly, the experimental performance does not match the theoretically claimed performance as will be explained later in this section.

As mentioned earlier, the parameter *ALUPacking* indicates how well the Shader Compiler (on AMD) packs the scalar or vector ALU in the kernel to the 5-way VLIW instructions. Values below 70% indicate that ALU dependency chains may be preventing the full utilization of the processor. From Table 10, it is found that *ALUpacking* is below 70% for the Izhikevich model but more than 70% for the HH model as seen in Table 4. Since *ALUPacking* is less than 70% for the Izhikevich model, all of the ALUs are not in full utilization. Since the kernel in this case calculates four neurons at a time using four of the five ALUs, the fifth ALU will often be unoccupied. Additionally, for the Izhikevich model, the Flops/Byte ratio is smaller than that of the HH model. Therefore, for the Izhikevich model, over 30% of the ALUs will be stalled. For the HH model, *ALUPacking* is 87.59% and since its Flops/Byte ratio is higher than the Izhikevich model, the fifth ALU is kept mostly busy. This explanation can also be verified by examining the values for *ALUStalledByLDS*. The parameter *ALUStalledByLDS* means the ALU stalls for the Local Data Store (LDS). For the Izhikevich model, the *ALUStalledByLDS* value is 2 as opposed to 0 for the HH model. Therefore, the HH model implementation performs better than the Izhikevich implementation on the Radeon GPU.

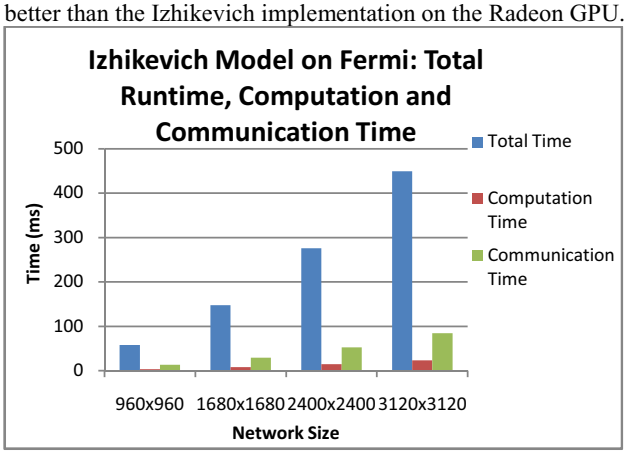

Figure 9. Izhikevich Model on Fermi: Total Runtime, Computation and Communication Time

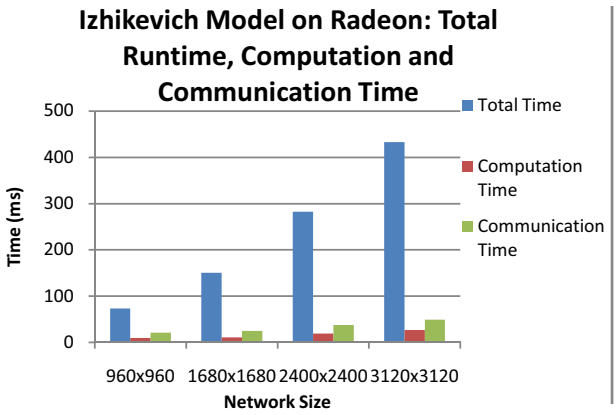

Figure 10. Izhikevich Model on Radeon: Total Runtime, Computation and Communication Time

Table 9. Fermi Profiler Results for the Izhikevich Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	11	11
Occupancy	1	1
Gld request	7.4e4	7.84e5
Branch	5.3e4	5.56e5
L1 global hit	0	0
L1 global miss	7.4e4	7.83e5
Divergent Branch	3.1e3	3.26e4

Table 10. Radeon Profiler Results for the Izhikevich Model

HH Model	Image Size 960x960	Image Size 3120x3120
GPR	12	12
ALU Packing	47.5	47.5
Scratch Register	0	0
Path Utilization	100	100
ALU Stalled by LDS	2	2
Fetch Unit Stalled	0.55	0.02
Write Unit Stalled	79.85	10.15

E. Report from AMD KernelAnalyzer

The AMD APP KenrelAnalyzer is utilized to generate kernel analysis report for the Radeon as shown in Table 11. The *KernelAnalyzer* report does not explain the difference in the performances of Nvidia and Fermi but provides some useful information about the performance differences of the two models (Izhikevich and HH) on Radeon. From this table it is found that the *ALU:Fetch ratio* (balance of ALU and Fetch cost) is the highest with the HH model and the lowest with the Izhikevich model. According to the Radeon user manuals, this value should be more than 1.2, which is maintained in all four SNN models. The throughput shows how many threads can complete their jobs per second. For our results, the values are roughly inversely proportional to the flops/neuron values. As can be seen from the table, this value is the lowest for the HH model since the model has the highest flops/neuron value.

The bottleneck of the Radeon implementation is the ALU operation as see in Table 11. For all four neuron models, the bottleneck is the ALU-ops. Thus if we can increase the computational throughput of the GPU, the kernel could run faster.

Table 11. Radeon KernelAnalyzer Output for All the Models

Model	ALU:Fetch Ratio	Throughput (Million threads/Sec)	Bottleneck
HH	19.17	394	ALU Ops
Morris-Lecar	13.17	898	ALU Ops
Wilson	2.98	1755	ALU Ops
Izhikevich	2.47	5495	ALU Ops

VI. DISCUSSIONS

We have compared the performance of the Nvidia Fermi and AMD Radeon architectures for four different neuron models and compared the runtime performance of the GPU architectures for various problems sizes. The performance improvement for the HH model and the largest problem size, i.e., image size of 3120x3120, was 857.3x for Nvidia and 658.51x for AMD against a serial implementation on Intel Core 2 Quad.

The profiler results for each of the GPU architectures were analyzed and compared highlighting the differences in terms of kernel and memory performance. For all the neuron models and most of the problem sizes, the Fermi architecture performs better than the Radeon. For example, for the HH model, which requires 44 bytes of data and 246 FLOPs per neuron, the Fermi computation and communication times are 13.4% and 19.9% faster respectively compared to the Radeon for the largest network size. From the AMD profiler, it was found that the average WriteUnitStalled is higher than the optimal value (42% vs. 0%), which explains why the Radeon is slower than Fermi for the HH model. Similarly the average WriteUnitStalled is again higher for Wilson model (44%), causing the Radeon's performance to be lower than Fermi.

Conversely for the ML model, which requires only 17 bytes per neuron, the performances of both GPUs are comparable. In this case, the value of WriteUnitStalled for the AMD is lower than that of HH model (26% vs. 42%). Similar results were found for the Izhikevich model, which requires 13 bytes per neuron. The WriteUnitStalled value for the Radeon is smaller in these cases because of the low data access requirement. For the Izhikevich neuron model, the Radeon could perform better than Fermi if the ALUPacking value for the Radeon would have been larger. The ALUPacking value is smaller for the Izhikevich model as it has the lowest Flops:Byte ratio.

VII. CONCLUSIONS

In this study, we have conducted a head-to-head comparison of the state-of-the-art architectures from Nvidia and AMD/ATi (Fermi and Radeon respectively). Our studies have shown that the Fermi implementations outperform those for Radeon for most of the cases. For the less data intensive applications (fewer

bytes per neuron update) such as the Izhikevich and Morris-Lecar models, the Radeon performs better than Fermi for the larger image sizes. For the more data intensive models (more bytes per neuron update) such as the HH and Wilson models, Fermi outperforms Radeon for all of the image sizes. Reasons for these observations are explained with the help of the Nvidia CUDA Proflier and AMD APP Profiler, and application characteristics.

As reflected by the Radeon architectural parameters, although the Radeon architecture appears to be superior to Fermi by merely examining data sheets, our detailed study shows that OpenCL must mature further to extract the maximum performance out of the Radeon architecture. The current OpenCL programming model is better tied with the Nvidia's architecture than Radeon. It is also worth noting that factors such as compiler maturity and driver efficiency may also overshadow architecture features that can subsequently enhance or limit performance. Though Brooks++ is available to the developers to program the AMD/ATi's GPUs and CUDA for Nvidia's GPUs, we have performed a fair comparison of the architectures using a common programming platform. Future work will incorporate applications from other domains and explore methods to allow OpenCL to extract the maximum potential of AMD's architecture. Future work will also incorporate the study of compiler and device driver effects on the GPU architectural performance.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under Grant No. CCF-0916387. The authors gratefully acknowledge vendor equipment and/or tools provided by Nvidia and AMD.

REFERENCES

[1] HECRTF, "Federal plan for high-end computing," Executive Office of the President, Office of Science and Technology Policy, Washington, DC, 2004

[2] D. A. Reed, "Workshop on the roadmap for the revitalization of high-end computing: Computing research association," 2003

[3] J. S. Vetter and B. R. de Supinski, "Evaluation High Performance Computers," *Concurrency and Computation: Practice and Experience,* pp. 1239-1270, August 17, 2005

[4] "Intel's teraflops chip uses mesh architecture to emulate mainframe", http://www.eetimes.com/electronics-products/processors/4091586/Intel-s-teraflops-chip-uses-mesh-architecture-to-emulate-mainframe

[5] "NVIDIA CUDA Programming Guide", http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[6] L. Ligowski and W. Rudnicki, "An Efficient Implementation of Smith Waterman Algorithm on GPU using CUDA, for Massively Parallel Scanning of Sequence Databases." *Proceedings of IPDPS 2009*, Rome Italy, May 2009

[7] J. C. Phillips, J.E. Stone, and K. Schulten, " Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters." *Proceedings of SC 2008,* Austin, TX, November 2008

[8] "OpenCL- Open Standard for Parallel Programming of Heterogeneous Systems", http://www.khronos.org/opencl/

[9] E. M. Izhikevich, "Simple Model to Use for Cortical Spiking Neurons," *IEEE transactions on Neural Networks*, vol. 14, no. 6, pp. 1569-1572, November 2003

[10] H. R. Wilson, "Simplified dynamics of human and mammalian neocortical neurons," *J. Theor. Biol.,* vol. 200, pp. 375-388, 1999

[11] C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophys. J.*, vol. 35, pp. 193-213, 1981

[12] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500-544, 1952

[13] E. Izhikevich, "Which Model to Use for Cortical Spiking Neurons?", *IEEE Transactions on Neural Networks*, vol. 15(5), pp. 1063-1070, 2004

[14] A. Gupta, L. Long, "Character Recognition using Spiking Neural Networks," in *Proc. IJC*NN, pp. 53 – 58, August 2007

[15] Vivek K. Pallipuram, "Acceleration of Spiking Neural Networks on Single-GPU and Multi-GPU systems", Master's Thesis May 2010

[16] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi", http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[17] ATI Mobility Radeon HD 5870 GPU Specifications", http://www.amd.com/us/products/notebook/graphics/ati-mobility-hd-5800/Pages/hd-5870-specs.aspx

[18] "NVIDIA OpenCL Programming Guide", http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf

[19] E. M. Izhikevich, "Dynamical Systems in Neuroscience," *MIT press*, Cambridge, Massachusetts, 2007

[20] C. Johansson and A. Lansner, "Towards Cortex Sized Artificial Neural Systems," *Neural Networks*, 20(1), pp. 48-61, January 2007

[21] Nene, S. A., Nayar, S. K., and Murase, H. (1996). Columbia Object Image Library (COIL-100) (No. CUCS-006-96): Columbia Automated Vision Environment

[22] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The Cat is Out of the Bag: Cortical Simulations with 109 Neurons, 1013 Synapses," *Proceedings of SC '09*, Portland, Oregon, November 2009

[23] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbauma, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," *Special issue of Neural Network, Elsevier*, vol. 22(5-6), pp. 791-800, July 2009

[24] P. Du, R. Weber, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming", Technical Report CS-10-656, Electrical Engineering and Computer Science Department, University of Tennessee, 2010, LAPACK Working note 228

[25] D.P. Playne and K.A. Hawick, "Comparison of GPU Architectures for Asynchronous Communication with Finite-Differencing Applications", *Concurrency and Computation: Practice and Experience*, 23: n/a. doi: 10.1002/cpe.1726, 2011

[26] "ATI Stream Profiler", http://developer.amd.com/gpu/StreamProfiler/Pages/default.aspx

[27] "Stream KernelAnalyzer", http://developer.amd.com/gpu/ska/pages/default.aspx