J Supercomput (2012) 61:673–718
DOI 10.1007/s11227-011-0631-3

A comparative study of GPU programming models and architectures using neural networks

Vivek K. Pallipuram · Mohammad Bhuiyan · Melissa C. Smith

Published online: 31 May 2011

Abstract Recently, General Purpose Graphical Processing Units (GP-GPUs) have been identified as an intriguing technology to accelerate numerous data-parallel algorithms. Several GPU architectures and programming models are beginning to emerge and establish their niche in the High-Performance Computing (HPC) community. New massively parallel architectures such as the Nvidia's Fermi and AMD/ATi's Radeon pack tremendous computing power in their large number of multiprocessors. Their performance is unleashed using one of the two GP-GPU programming models: Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). Both of them offer constructs and features that have direct bearing on the application runtime performance. In this paper, we compare the two GP-GPU architectures and the two programming models using a two-level character recognition network. The two-level network is developed using four different Spiking Neural Network (SNN) models, each with different ratios of computation-to-communication requirements. To compare the architectures, we have chosen the two extremes of the SNN models for implementation of the aforementioned two-level network. An architectural performance comparison of the SNN application running on Nvidia's Fermi and AMD/ATi's Radeon is done using the OpenCL programming model exhausting all of the optimization strategies plausible for the two architectures. To compare the programming models, we implement the two-level network on Nvidia's Tesla C2050 based on the Fermi architecture. We present a hierarchy of implementations, where we successively add optimization techniques associated with the two programming models. We then compare the two programming models at these different levels of

V.K. Pallipuram · M. Bhuiyan · M.C. Smith (✉)
Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA
e-mail: smithmc@clemson.edu

V.K. Pallipuram
e-mail: kpallip@clemson.edu

M. Bhuiyan
e-mail: mbhuiya@clemson.edu

implementation and also present the effect of the network size (problem size) on the performance. We report significant application speed-up, as high as 1095× for the most computation intensive SNN neuron model, against a serial implementation on the Intel Core 2 Quad host. A comprehensive study presented in this paper establishes connections between programming models, architectures and applications.

Keywords CUDA · OpenCL · Fermi · Radeon · Spiking neural network · Programming models · Architectures · Speed-up · Profiler counters

1 Introduction

The era of relying on the principles of Moore's Law for increasing performance from a single core processor is rapidly closing due to various limiting factors including the memory wall and clock wall. To overcome these limitations, industry trends have moved to multicore and many-core processors. Currently, some vendors such as IBM, AMD and Oracle-SUN are presenting prototypes with as many as 80 cores that can theoretically achieve more than 1 Teraflops of computing performance [1]. The current state-of-the-art is a 100-core processor available for general-purpose and high-performance computing [2]. However, amongst these advancements, the General Purpose Graphical Processing Units (GP-GPUs) have been very effective and popular in High-Performance Computing (HPC) for the last four years. GP-GPU computing uses the powerful heterogeneous co-processing environment where computationally intensive tasks are performed on the GP-GPU while the host commodity processor usually manages small computations, data movement, and management.

GP-GPUs have attracted researchers from the HPC community to accelerate numerous data parallel algorithms. The introduction of CUDA by the Nvidia Corporation in November 2006 [3] has completely changed the face of the graphical computing. Several research groups have explored Nvidia GP-GPUs and their clusters with CUDA in order to simulate large data-parallel algorithms such as the Smith Waterman algorithm [4], NAMD [5], and others of significant concern in the biological and physical sciences community. Given the growing interest of researchers in GP-GPUs, several specialized architectures and programming models have emerged, establishing their niche in the HPC community. In response to this growing interest, GPU vendors continue to add more cores and Compute capabilities to their device families. Nvidia's Fermi and AMD/ATi's Radeon are such examples that pack enormous computing power in their large number of multiprocessors. While CUDA coupled with Nvidia GPUs has had a major share of the HPC graphical computing market, Open Computing Language (OpenCL), maintained by the Khronos group, is a growing standard that is gaining interest [6]. Unlike CUDA, which is solely dedicated for computing with Nvidia GPUs, OpenCL was conceived to support a variety of architectures such as GPUs from different vendors, DSPs, Cell processors and multi-core architectures.

Graphical computing has reached a phase where it is interesting to compare the architectures and programming models, and establish a definite link between the GP-GPU architectures, programming models and applications that can be accelerated using this technology. In this spirit, we have chosen the Spiking Neural Networks (SNN) for a comparative study of two leading GP-GPU architectures and programming models.

SNNs are of particular interest, given their popularity in the neuroscience community for modeling the mammalian brain. In [7], the author has listed neuronal properties expected for a neuron model to accurately reproduce characteristics of the neo-cortex. Four SNN models, namely the Izhikevich model [8], Wilson model [9], Morris–Lecar model [10] and the Hodgkin–Huxley model [11], were found to satisfy these requirements. Each of the above listed models has different ratios of computation-to-communication requirements and hence is suitable for a broad comparative study. The Izhikevich model requires the least computation and is sufficiently accurate, whereas the HH model is computationally dense and is considered to be the most accurate model. The Wilson and ML models lie between these two extremes. Details of the Flops/Byte requirements for each of the models are provided later in Sect. 2.3.1.

The models mentioned above are highly data parallel and hence are better suited for data parallel architectures such as GP-GPUs [12]. In this paper, we accelerate a two-level character recognition network that can recognize 48 alpha-numeric characters originally developed in [13]: 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. The acceleration and optimization tasks were used to accomplish the prime focal contributions of this journal, which are:

1. Comparison of the Nvidia Fermi and AMD/ATi Radeon architectures using a common programming model.
2. Comparison of CUDA and OpenCL programming models at three different levels of algorithm implementation on the same GPU platform.
3. A broad comparison of the programming models using algorithms with different Flops/Byte ratio.
4. Network scalability and speed-up analysis.

Our parallel implementations were successful in attaining application speed-up for the most computation intensive HH model as high as 1095× for CUDA and 1074× for OpenCL on Nvidia's Tesla C2050, and 588× using OpenCL on AMD/ATi's Radeon 5870. Application speed-up values were found to be dependent on the communication and computation requirements of each of the models, and the way they were mapped on the architecture. Section 2 of this paper provides an overview of the GP-GPU architectures (Nvidia Fermi and AMD/ATi Radeon), background on the CUDA and OpenCL programming models, SNN models and the two-level network used for the study. Section 3 discusses related work. Section 4 presents the details of the experimental setup and the hierarchy of implementations with successively added optimization techniques. Section 5 presents the results and analysis, and the paper is concluded in Sect. 6 with conclusions and suggestions for future work.

2 Background

2.1 The GPU architecture

The GPU architecture has a rich and fascinating history. Initially intended as a fixed many-core processor dedicated to transforming 3-D scenes to a 2-D image composed of pixels, the GPU architecture has undergone several innovations to meet the computationally demanding needs of supercomputing research groups across the globe. The traditional GPU pipeline designed to serve its original purpose came with several disadvantages. Shortcomings such as the limited data reuse in the pipeline, excessive variations in hardware usage, and lack of integer instructions coupled with weak floating-point precision rendered the traditional GPU a weak candidate for HPC. In November 2006 [14], NVIDIA introduced the GeForce 8800 GTX with a novel unified pipeline and shader architecture. In addition to overcoming the limitations of the traditional GPU pipeline, the GeForce 8800 GTX architecture added the concept of *streaming processor (SMP) architecture* that is highly pertinent to current GP-GPU programming. SMPs can work together in close proximity with extremely high parallel processing power. The outputs produced can be stored in fast cache and can be used by other SMPs. SMPs have instruction decoder units and execution logic performing similar operations on the data. This architecture allows SIMD instructions to be efficiently mapped across groups of SMPs. The streaming processors are accompanied by units for texture fetch (TF), texture addressing (TA), and caches. The structure is maintained and scaled up to 128 SMPs in GeForce 8800 GTX. The SMPs operate at 2.35 GHz in the GeForce 8800 GTX, which is separate from core clock operating at 575 MHz. Several GP-GPUs used thus far for HPC applications have architectures that are concurrent with the GeForce 8800 GTX architecture. However, the introduction of the Fermi by Nvidia in September 2009 [15] has radically changed the contours of the GP-GPU architecture, as we will explore in the next subsection.

2.1.1 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 a lot of 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 have a new look 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 share a second level L2 cache. More information about the architecture can be found in [15]. For our experiments, we have used a single Nvidia Tesla C2050

which belongs to the 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 Tesla system's GDDR interface offers data bandwidth up to 144 GB/s.

2.1.2 AMD/ATi Radeon 5870

The AMD/ATi's Radeon 5870 architecture [16] is very different compared to NVIDIA's Fermi architecture. 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. One thread can be executed on one stream core, thus 16 threads can be run on a single compute unit. In order to hide the memory latency, 64 threads are assigned to a single compute unit. When one 16-thread group accesses memory, the other 16-thread group executes on the ALU. Therefore theoretically, a throughput of 16 threads per cycle is possible on the Radeon architecture. Each ALU can execute a maximum of 2 single-precision Flops: multiply and add instructions per cycle. The clock rate of the Radeon GPU is 850 MHz; for 1600 ALUs this translates to a throughput of 2.72 TFlops/s.

The Radeon 5870 has a memory hierarchy that is similar to the Fermi's memory hierarchy. The 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 having an aggregate bandwidth of 1 TB/s. Multiple compute units share a 512 KB L2 cache with 435 GB/s of bandwidth between L1 and L2 cache. Each compute unit also has a 32 KB of shared memory, providing a total 2 TB/s aggregate bandwidth. The registers have the highest bandwidth, 48 bytes per cycle in each stream core (aggregate bandwidth of $48 * 320 * 850$ MB/s, i.e., 13 TB/s). The 256 KB register space is available per compute unit, totaling 5.1 MB for the entire GPU.

2.2 The programming models

In this subsection, we will discuss the two popular programming models used for the GP-GPU computing, namely the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL). While CUDA was introduced by the Nvidia Corporation in November 2006 and has a stable release since its inception, OpenCL is a recent standard maintained by the Khronos group created to support multiple HPC architectures such as the GPUs, multi-core CPUs, cell processors, DSPs, etc.

2.2.1 CUDA

In CUDA for C, the GPU functionality is defined by writing device functions in C, which are called *kernels.* Typically, only one kernel can be executed on the GPU at a time. A *thread*, which is a sequence of instructions, is instantiated several thousands of times. When a kernel is called, N threads execute the kernel in parallel. Threads are accessed inside kernels using built-in variables: *threadIdx, blockIdx,* and *blockDim.* Collections of threads, called thread blocks are executed on the SMPs. The blocks are further divided into SIMD groups of 32 threads called *warps,* which are further divided into groups of 16 threads called *half-warps.* The memory hierarchy in CUDA comprises a set of *registers* (on-chip) and *local memory* (residing in an off-chip DRAM) for each thread, a private shared memory for thread blocks, a *global memory* for all threads launched, and read-only *texture cache* and *constant memory.* CUDA offers three primary optimization strategies, namely the *Memory Optimization, Execution Configuration Optimization,* and *Instruction Optimization*.

Several memory optimization strategies can be found in [17], here we discuss the prominent ones used in this research. One memory optimization strategy is to reduce the frequent transfers between the host and the device since the host-to-device bandwidth is usually many orders of magnitude lower than the device-to-device bandwidth. It is highly beneficial to transfer all of the relevant data to the device memory for processing and later transfer the data back to the host memory once all of the operations are finished. The device–host bandwidth can be most efficiently utilized by overlapping the kernel execution with the data transfers using *Zero Copy* (Z). This feature is available only in the devices with Compute capability greater than or equal to 1.1. In this technique, the data transfers are performed implicitly as needed by the device kernel code. For the operation described, it is required that the device should support the host mapped memory. Earlier Compute capability cards analyze the global memory performance in terms of *coalesced accesses* where a *half-warp* can complete memory accesses in single or a couple of transactions.

Nvidia's Fermi architecture (Compute capability 2.0) has a significantly different memory structure with the introduction of L1 and L2 caches. Hence the global memory access rules are slightly modified for the Fermi architecture, and we will analyze the global memory performance more often in terms of *global misses*, *global hits*, and *hit rate* instead of *uncoalesced accesses*. The Fermi architecture also allows the user to configure the amount of L1 cache and shared memory used. From the 64 KB of on-chip memory, 48 KB can be configured either as L1 cache or shared memory. The user is also allowed to cache the global memory either in L2 cache alone, or both in L1 and L2 caches [3]. Caching the global memory can promote performance improvement in applications that involve frequent global memory data access or those that suffer from register pressure.

Software Pre-fetching (SP) is another useful memory optimization technique for avoiding frequent accesses to the device global memory. The technique involves the use of on-chip *Registers* and/or *Shared Memory* (SM) to cache and operate on the data. Once all of the operations are finished, the data is transferred back to the device memory. *Registers* are more commonly used for such scenarios since they do not involve *bank conflicts* that can occur with shared memory accesses. Bank conflicts

occur when threads in a half-warp access the same shared memory bank. These conflicting accesses are serialized and therefore negatively impact the performance. SM has been used in our research to minimize the communication between the device and the host as will be seen in Sect. 4.

Execution Configuration Optimization is an effective method for hiding latency on the memory bound kernels. Execution configuration is related to the number of threads per block. Varying the number of threads per block changes the *multiprocessor occupancy*: the ratio of the number of warps running on the multiprocessor to the maximum number of warps that can physically run on the multiprocessor. The CUDA profiler provides information about the multiprocessor occupancy. The number of threads per block should also remain a multiple of 32 and sufficiently large, typically greater than or equal to 192. Keeping the number of threads per block a multiple of 32 facilitates coalescing. Specifying the number of threads per block shall henceforth be referred to as specifying a *block configuration.*

The *Instruction-level Optimization* technique examined in our work with CUDA involves the use of fast math functions and *Reduced Conditional Statements* (RCS). The use of fast math results in fewer clock cycles for the instruction at the expense of reduced accuracy. The compiler optimization *–use_fast_math* forces compiling arithmetic functions as fast math functions. RCS reduces divergent paths taken within a warp. Divergent paths are serialized which results in reduced performance.

2.2.2 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*. Similar to CUDA, an OpenCL work-item is identified using its *local ID* and *work-group ID*. Analogous to blocks in CUDA, OpenCL collects work-items into *work-groups.* The concept of warps and half-warps is identical to that of CUDA. The memory model is identical to that of CUDA, with minor differences in the naming convention in OpenCL: CUDA shared memory is referred to as the *OpenCL local memory*.

The optimization techniques used for the OpenCL implementations are similar to that of CUDA. 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. Parameters are loaded from the global memory to the registers, where they are operated upon, and then the final result is written back to the global memory. SP reduces repeated accesses to the global memory thus saving a significant number of clock cycles. Similar to the CUDA implementations, OpenCL SM has also been used to minimize the communication between the device and the host. While sending data arrays from the host to the GPU, the *clCreateBuffer()* function was used to create a buffer and copy the data from the host to the GPU. But it is found that additional performance improvements are possible if the buffer is created in the initialization part and then a function, *clMemWrite()*, is used inside the execution loop of the host function. We call this optimization *memory write* (MW).

We now discuss the *instruction level optimization* techniques available in OpenCL. Analogous to the fast math functions in CUDA, OpenCL offers *native math* functions. Another math optimization technique that OpenCL offers is called the *unsafe math (UM)*. In this optimization, the compiler optimizes the floating-point arithmetic in the kernel but it may violate IEEE 754 standards and OpenCL numerical compliance requirements. The last optimization 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.

Execution level optimization, similar to the *execution configuration optimization* in CUDA, involves changing the work group size to maximize the multiprocessor occupancy. It is worth mentioning here, and as shall be seen in our implementation results, that a high multiprocessor occupancy does not directly imply performance but assists in hiding latency in memory bound kernels.

Structural differences between CUDA and OpenCL are elucidated in [19].

2.3 Spiking neural networks

In this subsection, we discuss the Spiking Neural Network (SNN) models and two-level character recognition network used as the case study in our research to study and compare several features of the two GP-GPU architectures and the two programming models. SNNs constitute the "Third Generation" of the neural networks and are highly biologically accurate. A "spiking" neuron fires an electric pulse, commonly referred to as "spike" at certain time intervals. The amplitude of the spike is irrespective of the input, but the time of spike is a function of the input. This type of time encoding is useful for many signal-processing applications. Several models have been proposed for SNNs, some being very computationally efficient and moderately accurate, and some being computation intensive and hence, highly accurate. In [7], Izhikevich lists the 20 most prominent features of biological neurons and ranked several models based on their ability to mimic these neuron features. Four 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.

2.3.1 Four models

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 [7], the model involves 4 equations and 10 parameters describing various neuron current activation and deactivation. The model takes 120 flops per 0.1 ms time-step and hence 1200 flops/1 ms 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 [10], involve 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/1 ms 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.

Table 1 Flops/byte ratio for all the models

Model	FLOPs required for the complete neuron update	Flops/byte ratio
HH	246	9.84
ML	147	8.65
Wilson	38	1.52
Izhikevich	13	0.9997

Wilson [9] attempted to model cortical neurons with a system of polynomial equations. This model introduces a few additional conduction channels to the HH model as reported in [7]. 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/1 ms for the update.

In [8], Izhikevich developed a simple and very computation efficient spiking neuron model that is almost as plausible as the most accurate Hodgkin–Huxley (HH) model. Izhikevich was successful in reducing the complex HH model equations to a 2-D system of ordinary equations. Izhikevich's model requires only 13 flops per neuron update and still sufficiently reproduces a majority of neuronal properties. The equations are found in [8]. In our research, we have used a 1 ms time-step (13 flops/1 ms update) for neuronal dynamics update for the Izhikevich model.

The time-steps used in our research for the models discussed are in the valid range of time-steps that are deemed sufficient for reproducing biologically relevant neuron dynamics [7].

We now provide the Flops/Byte ratio of the four SNN models, which provides an algorithmic analysis of the aforementioned SNN models. The Flops/Byte ratio is an algorithm specific value and is defined as the ratio of the number of floating-point operations required for a complete neuron update (level-1 and level-2) to the overall bytes requested (all model parameters, firing vector and block firing vector) for all of the neuron updates [20]. Table 1 gives the relevant Flops/Byte ratio information for each of the models studied.

2.4 The two-level network

The SNN used in this research is based on [13] and the network used to test the models is shown in Fig. 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. 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

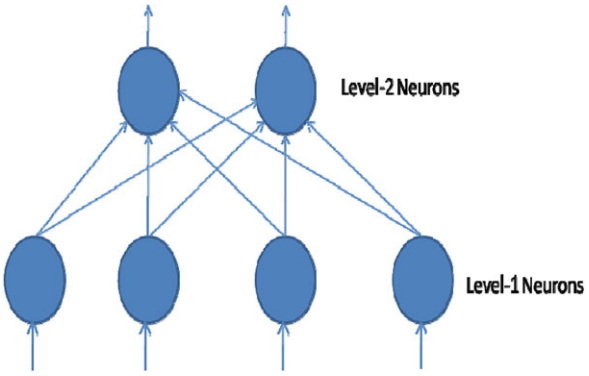

Fig. 1 Two-level character recognition network

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 Sect. 4.

3 Related work

Several research activities have been motivated by the idea of modeling the neocortex. In [21], the authors have studied the mammalian neocortex in detail and implemented an abstract and scalable neural network model of the neocortex. The authors were successful in simulating 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 model to detect 128 × 128 pixel images from the COIL-100 database [22]. Their neuron model fits in the integrate-and-fire (I&F) category, which according to Izhikevich, is insufficient for reproducing neuronal properties [7]. In [23], 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. EPFL's blue brain project is trying to reverse engineer the brain using the Hodgkin–Huxley [11] and Wilfred Rall [24] models for simulating 100,000 neurons on the BlueGene/L supercomputer.

Alternative computing architectures such as GP-GPUs are now being investigated for biologically realistic simulations. In [25], the authors have implemented Izhikevich's random network on Nvidia's GTX-280 with 1 GB memory and achieved a

speed-up of 26× over an equivalent software implementation for a 100 K neuron network simulation. Their work discusses mapping strategies on the GP-GPU to efficiently utilize the memory bandwidth and parallelism.

A comparison of application performance on different vendor GPU architectures was not found in the current literature; however, there has been a limited amount of work done on systematic comparison of OpenCL and CUDA. In [26], the authors have accelerated an EMRI modeling application using Nvidia's C1060 as one of the accelerators and have achieved a similar performance factor for both CUDA and OpenCL. In [27], the authors have used the Adiabatic Quantum Algorithms (AQUA), which are Monte Carlo simulations, to compare CUDA and OpenCL on Nvidia's GTX-260 (Compute capability 1.3). They have compared the programming models for data transfer time, kernel execution time and end-to-end runtime. They have concluded that CUDA implementations perform consistently better than the OpenCL implementations. In [19], 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. They do not explicitly study the performance difference between the programming models or the architectures. Contrary to the existing studies, our research varies the problem size and experiments with the applications of different "communication to computation" requirements for a well-rounded comparison between the two programming models. GPU architectures offer maximum performance for larger problem sizes, given their highly data parallel architecture. Their behavior is best studied when the problem size is varied from a relatively small value, for instance, neural network size equal to 9000, to a large value, a network of size 9 million [20]. Additionally, we compare application performance on different vendor architectures. The final outcome of this research is the establishment of connections between GPU architectures, programming models and applications.

4 Experimental setup and implementation

4.1 Experimental setup

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 CUDA implementations for the Nvidia GPU were developed using CUDA 3.0 installed on the host system running 32-bit Ubuntu 9.04. The CUDA visual profiler was used to obtain the useful information for optimizations, such as the number of *global hits* and *misses*, *divergent branches*, *instructions issued, instructions executed* and the *multiprocessor occupancy*. The same physical system is used for the OpenCL implementations on the Tesla C2050 and relevant profiling information was found using the OpenCL visual profiler. The second experimental system consists of a single ATi Radeon 5870 paired with a 2.8 GHz Intel Core i7. The implementations were developed using the ATi stream SDK 2.1 for OpenCL. While the codes were developed on Ubuntu 9.04, Windows XP was used to collect the relevant profiler data. The single-GPU OpenCL results on the Tesla C2050 are compared with the single-GPU OpenCL results on the

AMD/ATi Radeon 5870. It is worth mentioning that since CUDA does not support AMD GPUs, OpenCL serves as a common programming platform to fairly compare the graphics cards from Nvidia and AMD, and hence is chosen by the authors for this comparative study.

4.2 Implementations

We first provide the hierarchy details of the three implementations used in this research to compare CUDA and OpenCL. As discussed in Sect. 4.2.4, the best OpenCL implementation used to compare the GP-GPU architectures corresponds to the final implementation in the hierarchy, which will be described in Sect. 4.2.3. These implementations are similar to the ones found in [20]. As we proceed through the hierarchy of the implementations, we successively add optimization techniques available with the programming models. In the first two implementations, we successively add memory optimization techniques. The last implementation in the hierarchy adds instruction-level optimization. Each implementation involves the use of execution configuration optimization.

As introduced in Sect. 2.4, level-1 is the most computation intensive layer of the network since the number of neurons is equal to the total number of pixels in the input image therefore these 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. For example, for the largest network size 3120×3120, the transfer will incur a communication overhead of 1.08 seconds for transferring 1.74 GB data ($3120 * 3120 * 48 * 4$ Byte) from the host to device (host-to-device bandwidth of 1.61 GB/s obtained from the *bandwidthTest* application of the CUDA SDK). This data transfer overhead leads to an increase in runtime by 304% and 54% for the Izhikevich model and HH models, respectively. Hence, any computational improvement achieved by implementing level-2 neuron dynamics (only 48 neurons) on the GP-GPU will be insufficient to amortize the communication overhead just described.

Table 2 provides the number of neurons at each level for the two-level networks corresponding to the various input image sizes. The initial SNN with 576 neurons in level-1 and 48 neurons in level-2 was developed, trained and tested in MATLAB before converting them to C [28]. While the research published in [12] using these models scaled up to 5.7 million neurons, the implementations discussed in this paper were scaled up to 9.7 million neurons [20].

4.2.1 Implementation 1

Implementation 1 is the most basic implementation of the hierarchy. The implementation uses direct global memory accesses and uses the execution configuration optimization. *Software Prefetching* (SP) is also used which involves the use of fast on-chip registers to store the data from the device memory. *Implementation 1* involves a

Table 2 Network configurations for different image sizes

Input image size	Level-1 neurons	Level-2 neurons	Total neurons
96 × 96	9216	48	9264
192 × 192	36864	48	36912
240 × 240	57600	48	57648
384 × 384	147456	48	147504
480 × 480	230400	48	230448
720 × 720	518400	48	518448
960 × 960	921600	48	921648
1200 × 1200	1440000	48	1440048
1680 × 1680	2822400	48	2822448
2160 × 2160	4665600	48	4665648
2400 × 2400	5760000	48	5760048
2592 × 2592	6718464	48	6718512
2800 × 2800	7840000	48	7840048
3120 × 3120	9734400	48	9734448

single host-to-device transfer of all the parameters pertaining to the level-1 neurons before the start of simulation. Once the simulation begins, the *global firing vector*, which is the level-1 neuron fire status vector, is transferred from the GP-GPU device to the host at each time-step. This transfer leads to a significant amount of overhead in highly accurate neuron models, such as the HH model, that require several time-step evaluations.

4.2.2 Implementation 2

In [20], the author introduces the *block firing vector*, which is similar to the global firing vector, but instead acts as a collection of flags for the blocks. Since the threads are collected in thread blocks 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 (an improvement over *Implementation 1* in data transfer overhead). This technique avoids unnecessary reads of the global firing vector at the host-end thereby reducing the host-time. For instance, in the HH model, out of 373 time-steps, only one time-step involves a level-1 neuron firing for a particular image in the database. The block firing vector, with its nominal size can afford to be transferred and read on each time-step. The complete firing vector then is transferred and read only once versus 373 times offering a significant performance improvement. 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.

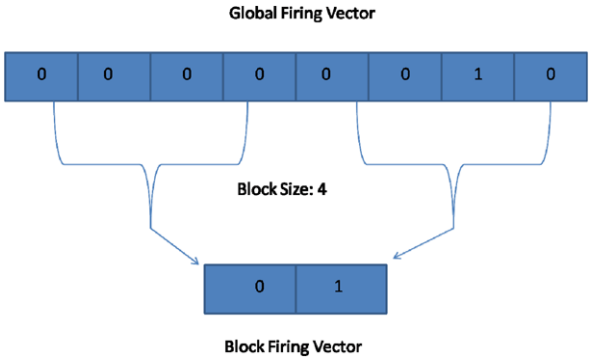

Fig. 2 Block firing vector concept used in Implementation 2

Implementation 2 for CUDA and OpenCL differ in the following ways. The Zero Copy (Z) optimization available in CUDA is used to overlap block firing vector transfer with the kernel execution; OpenCL does not offer the Z memory optimization. Additionally, with CUDA, we have also experimented with the cache system. As mentioned in Sect. 2.2.1, CUDA allows us to configure the amount of L1 cache used (16 KB L1 and 48 KB shared memory, default configuration, or 48 KB L1 and 16 KB shared memory) and also provides us the option of caching the global memory either in L1 and L2 (cache load) or only in L2 (Non-caching load). Non-caching load is suitable where accesses to the device global memory are scattered. Since our applications access the global memory uniformly, we have used the cache load configuration. OpenCL does not offer constructs to exploit these new features introduced by the Fermi architecture.

4.2.3 Implementation 3

Implementation 3 is the final implementation of the hierarchy that adds instruction level optimization as discussed in Sect. 2.2.1. The results obtained for each of the implementations discussed are detailed in Sect. 5.

Though CUDA also offers the use of fast texture cache that can potentially increase the performance for our applications [12], our studies with the texture cache for the Fermi architecture did not perform any better than simple global memory accesses. It is worth re-iterating here that global memory accesses in the Fermi architecture are substantially different than previous Compute capability cards. The Fermi architecture introduces the cache for the global memory which, based on the application, can potentially perform similar to a texture cache.

"*Concurrent kernels*" (CK) is another feature available with the Fermi architecture, where multiple streams concurrently execute the kernels. However, our SNN implementations performed better with single kernel-single stream structure. Another important feature offered by the Fermi architecture is the ECC support for data integrity. Switching the ECC support off, using the command: *nvidia-smi −g 1 −e 0*, can further reduce the kernel time. We have kept the ECC on in our experiments, though some performance improvement was seen with ECC off.

4.2.4 AMD/ATi Radeon 5870 implementation

The implementation of the SNN models on the AMD GPU using OpenCL [29] is similar to *Implementation 3*. Software pre-fetching has been used to reduce the number of redundant access to the global memory. Shared memory 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. Results obtained for each of the implementations discussed are detailed in Sect. 5.

5 Results and analysis

In Sect. 5.1, we compare the best Tesla C2050 (Fermi) implementation developed using OpenCL and the AMD/ATi Radeon implementation also developed using OpenCL. It is re-iterated here that since CUDA does not support AMD GPUs and is specific to GPU technologies only from Nvidia, we have chosen OpenCL as the base programming model for a fair comparison between the two architectures. We analyze and compare the performance of the architectures for the two extremes of the SNN models, namely the HH model and Izhikevich model. We observe several equivalent architectural parameters such as the global memory requests, occupancy, ALU Packing, ALU stalls, global cache hit/miss, etc., and also link these parameters to the given application characteristics such as the Flops/Byte ratio introduced in Sect. 2.3.1.

In Sects. 5.2 through 5.5, we compare and analyze the CUDA and OpenCL programming models at different levels of implementation on the Fermi architecture. We discuss the most computation intensive HH model first and then proceed to the lowest complexity Izhikevich model, as most of the features connecting the GPU architecture to the algorithm are more apparent in the higher computation density models. In order to perform a fair comparison of the two programming models, the kernel structure, memory transfers and the host code have been kept similar. We analyze several important profiler counter values. We specifically observe: *L1 global hits, L1 global misses, divergent branches, instructions executed,* and *instructions issued*. In the process, we define two additional terms: *hit-rate* and the *instruction performance ratio*. The *hit-rate* is defined as the percentage hits to the L1 cache. In order to define the *instruction performance ratio*, it is imperative to understand the profiler counters: *instructions executed* and *issued*. *Instructions executed* are those where threads in a warp execute simultaneously. *Instructions issued*, in addition to *instructions executed*, involve serial instructions where threads in a warp execute instructions serially. If the number of *instructions issued* is significantly larger than the *instructions executed*, it implies a greater degree of serialization. Serialization can be due to divergent branches, instruction cache misses, etc. The *instruction performance ratio* is defined as the ratio of the number *instructions executed* and the number of *instructions issued*. More information on the instruction counters can be found in [30].

The speed-up values provided in the following subsections are based on the comparison against a serial implementation on a 2.66 GHz Intel Core 2 Quad processor.

All of the compiler optimizations are used on the serial implementation; however, SSE and POSIX threadings are not used.

5.1 HH and Izhikevich models Fermi vs. AMD ATi

Tables 3 through 6 provide the runtime details for Nvidia's Fermi and AMD/ATi's Radeon 5870 for the two extremes of the SNN models: the HH and the Izhikevich models. As evident from Tables 3 and 4 for the HH model and the largest network size (3120×3120), Radeon is 49% slower than Fermi. But from Table 7, it is evident that the peak value of the computation throughput of Radeon is 169% higher than that of the Fermi, and the global memory bandwidth of the Radeon is 108% higher than that of Fermi. We have also shown the runtime breakdown for the two implementations to aid the comparative study. We have reported the GPU kernel time (computation time),

Table 3 HH model on Fermi: total runtime, computation and communication time

	Fermi (blocksize = 256)			
Image size	Total time (ms)	Computation time (ms)	Communication time (ms)	Other overhead (ms)
960	248.91	188.66	24.73	35.52
1680	669.02	530.09	49.52	89.41
2400	1322.60	1062.58	88.34	171.68
3120	2209.26	1779.36	140.88	289.02

Table 4 HH model on Radeon: total runtime, computation and communication time

	Radeon (blocksize = 128)			
Image size	Total time (ms)	Computation time (ms)	Communication time (ms)	Other overhead (ms)
960	502.86	330.57	97.25	75.03
1680	1103.61	769.60	139.03	194.99
2400	2048.21	1451.99	207.99	388.22
3120	3297.66	2284.98	371.82	640.86

Table 5 Izhikevich model on Fermi: total runtime, computation and communication time

	Fermi (blocksize = 256)			
Image size	Total time (ms)	Computation time (ms)	Communication time (ms)	Other overhead (ms)
960	46.18	3.17	9.68	33.33
1680	131.57	8.04	24.92	98.61
2400	268.15	16.01	52.51	199.63
3120	448.72	26.45	86.94	335.33

Table 6 Izhikevich model on Radeon: total runtime, computation and communication time

	Radeon (blocksize = 128)			
Image size	Total time (ms)	Computation time (ms)	Communication time (ms)	Other overhead (ms)
960	80.27	7.73	18.29	54.25
1680	189.30	12.34	39.64	137.32
2400	368.70	19.96	74.47	274.27
3120	552.47	27.57	118.51	406.38

Table 7 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

host-to-GPU communication time, and other overhead time (which includes host calculation, host-GPU synchronization, etc.) in Tables 3–6. For the HH model, the Fermi computation and communication time are $(2284.98 - 1779.36) * 100/1779.36$, i.e., 28% and $(371.82 - 140.88) * 100/140.88$, i.e., 163% faster, respectively, 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. For the Izhikevich model, as seen in Tables 5 and 6, results similar to the HH model are observed. Thus the experimental performance does not match the theoretically claimed performance as will be explained later in this section.

The communication time is related to the data transfer time between the host and the GPU. There are two types of functions that are used to transfer data using buffers between the host and the GPU, "clEnqueueWriteBuffer/clEnqueueReadBuffer" and "clCreateBuffer." Out of these two types, it can be verified that the "clEnqueueWriteBuffer/clEnqueueReadBuffer" function has less overhead and thus is faster. 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 163% slower than the Fermi GPU. One of the plausible explanations of this observation is that the implementation of these functions in the AMD OpenCL driver requires more overhead than that for Nvidia. Another 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 (28%) than that of the Radeon (Tables 3 and 4), although the theoretical computation throughput and the global memory bandwidth are higher for Radeon (Table 7). To explain this observation, the profiler results are taken and the relevant parameters are reported in Tables 8 and 9. Nvidia provides the "openclprof" profiler software for Linux OS, which was used to profile the SNN implementation on Fermi. On the other hand, AMD provides "ATI Stream Profiler" [31] which can only be used with Windows Visual Studio and

Table 8 Fermi profiler results for the Izhikevich and HH models

Model	Image size	GPR	Gld request	Occupancy	Branch	L1 global hit	L1 global miss	Divergent branch
Izhikevich	960 × 960	9	73776	1	52258	0	73776	3065
	3120 × 3120	9	779640	1	552245	0	779640	32479
HH	960 × 960	18	4.59926e+06	1	2.38208e+06	0	4.59926e+06	126201
	3120 × 3120	18	4.85597e+07	1	2.50378e+07	0	4.85597e+07	1.12786e+06

Table 9 Radeon profiler results for the Izhikevich and HH models

Model	Image size	GPR	Scratch reg.	ALUPacking	Path utilization	ALU stalled by LDS	Fetch unit stalled	Write unit stalled
Izhikevich	960 × 960	12	0	47.5	100	2	0.55	79.85
	3120 × 3120	12	0	47.5	100	2	0.02	10.15
HH	960 × 960	21	0	87.59	100	0	0.99	37.68
	3120 × 3120	21	0	87.59	100	0	1.04	46.7

Windows OS. Thus to use the AMD profiler, the Linux implementation was converted to Windows and the code was profiled. AMD also provides "Stream KernelAnalyzer" software [32], which was also used to analyze the SNN kernel code to get more information about the implementation.

The summary of the profiler reports for Fermi and Radeon are shown in Tables 8 and 9, respectively. The profiler parameters from Nvidia and AMD do not have 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, 960×960 and 3120×3120, to produce the profiler reports. Since the difference in the number of GPRs used by the Fermi and Radeon is not significant (see Tables 8 and 9), the number of GPRs does not have much influence on the performance. *Occupancy* is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU and is helpful in determining how efficient the kernel will be on the Nvidia GPU. A low value of 0.375, for instance, implies inefficient multiprocessor utilization. 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 9, it is found that *ALUpacking* is below 70% for the Izhikevich model but more than 70% for the HH model. 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, 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.

For the Fermi GPU, all of the ALUs are arranged as scalar processors as opposed to vector processors in the Radeon; and, both the SNN models show the maximum occupancy value of 1. From the other parameters of Table 8, *gld request*, *L1 global miss*, and *divergent branches* increase with the image size. From Table 9, it is found that the *FetchUnitStalled* values (the percentage of GPU time the Fetch unit is stalled) for the Radeon implementation of both network sizes and both models are low (about 1), which is desirable. Thus the fetch units are not responsible for the lower performance of the Radeon GPU. However, the *WriteUnitStalled* values (the percentage of GPU time the write unit is stalled) are very high (about 80) as seen in the same table. A large value of the *WriteUnitStalled* is another reason that the Radeon is slower than the Fermi for the SNN models. How to reduce these stalls 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.

Table 10 Radeon KernelAnalyzer output for the Izhikevich and HH models

Model	ALU:Fetch ratio	Bottleneck
Izhikevich	2.47	ALU Ops
HH	6.87	ALU Ops

The kernel analyzer report for the Radeon is shown in Table 10. 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 on Radeon. From this table it is found that the ALU:Fetch ratio (balance of ALU and Fetch cost) is higher with the HH model than that of the Izhikevich model. According to the Radeon user manuals, this value should be more than 1.2, which is maintained in this case. The bottleneck of the Radeon implementation is the ALU operation as seen in Table 10.

5.2 HH model CUDA vs. OpenCL

We now discuss and present the results from the comparative study of the two programming models: CUDA and OpenCL. As mentioned earlier, we present our findings for the HH model first as it is the most computation expensive model and features connecting algorithm to architecture are more apparent in this model. Section 5.2.1 discusses the performance results obtained for the two programming models for the first implementation in the hierarchy: *Implementation 1*. As discussed in Sect. 4.2.1, *Implementation 1* is the most basic implementation that uses direct global memory accesses, SP, and execution configuration optimization as primary optimization techniques. Section 5.2.2 presents the results for *Implementation 2* that reduces the frequency of the global firing vector transfer using the block firing vector as introduced in Sect. 4.2.2. The CUDA implementation additionally involves the *Z* optimization technique and the use of cache loading technique for *Implementation 2*; OpenCL, being more generic, does not offer these additional device specific techniques. The comparative results and discussion for *Implementation 3* that adds instruction level optimization is presented in Sect. 5.2.3. Section 5.2.4 summarizes our results for the set of experiments conducted for the HH model.

5.2.1 HH model CUDA vs. OpenCL: Implementation 1

Figure 3 provides the application speed-up results for the CUDA and OpenCL implementations. A maximum speed-up of 116.45× was observed for the CUDA implementation for the largest network size, while a speed-up of 140.9× was observed for the OpenCL implementation.

Table 11 provides the CUDA and OpenCL kernel and memory transfer times. As evident from the table, the OpenCL kernel calls are more efficient than the CUDA kernel calls. The memory transfer time for the programming models are similar but CUDA performs marginally better for most of the network sizes. Tables 12 and 13 provide the CUDA and OpenCL profiler results.

As seen in Tables 12 and 13, the CUDA kernel hits the cache more often than the OpenCL kernel, though the CUDA kernel incurs more cache misses. The cache

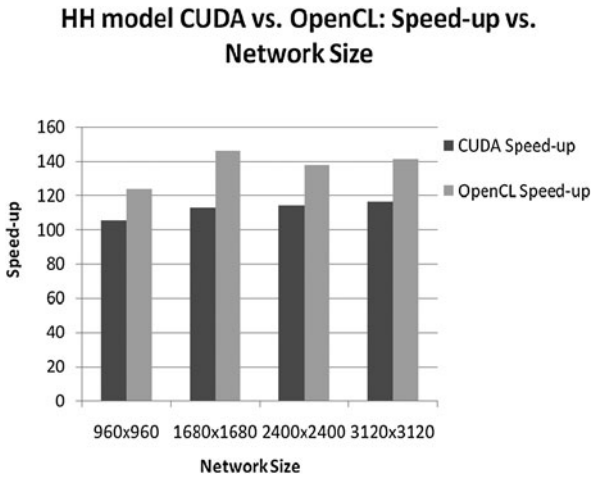

Fig. 3 HH model CUDA vs. OpenCL: Implementation 1

Table 11 HH model CUDA and OpenCL kernel and memory transfer time: Implementation 1

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	415.76	338.6	221.87	357
1680	1241.71	766.6	480.42	667.9
2400	2515.17	1447.17	1249.26	1499.56
3120	4230.65	2352.7	2089.012	2425.56

line requests to L1 or L2 are serviced at the throughput of L1 or L2 cache on a hit, whereas misses are serviced at the throughput of the device global memory. Though the *instruction performance ratio* for CUDA is slightly higher compared to that of OpenCL, a larger number of instructions are issued and executed for the CUDA kernel. CUDA observes some *divergent branches* while OpenCL observes none. All of the above factors account for better OpenCL kernel operation compared to CUDA. A block configuration of 192 performed the best for the CUDA implementation and yielded an occupancy of 0.875, whereas the block configuration of 192 yielded an occupancy of 1 for the OpenCL implementation.

5.2.2 HH model CUDA vs. OpenCL: Implementation 2

Implementation 2 (built on *Implementation 1*) further enhances the performance by reducing the number of host processor reads of the global firing vector, thus conserving memory bandwidth. CUDA provides *Zero Copy* (Z) where the kernel execution and block firing vector transfers are overlapped. We also explore the cache loading scheme where both L1 and L2 caches can be used for caching the global memory. *Implementation 2* exhausts all of the memory optimization strategies described in Sect. 2.2.1.

Table 12 HH model CUDA profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	5.27e6	1e5	1.8	1.91e4	4.01e8	3.99e8	0.997
1680	1.61e7	2.99e5	1.8	3.95e4	1.23e9	1.22e9	0.991
2400	3.29e7	7.24e5	2.1	5.1e4	2.5e9	2.49e9	0.996
3120	5.56e7	1.06e6	1.8	7.3e4	4.23e9	4.21e9	0.995

Table 13 HH model OpenCL profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	4.59e6	0	0	0	1.94e8	1.85e8	0.95
1680	1.4e7	0	0	0	5.93e8	5.68e8	0.96
2400	2.82e7	0	0	0	1.21e9	1.19e9	0.98
3120	4.74e7	0	0	0	2.02e9	2.02e9	1

Figure 4 summarizes the application speed-up results obtained with the two programming models. As seen in Fig. 4, OpenCL takes a larger leap in the performance compared to CUDA. A maximum speed-up of 378.9× was observed for the CUDA implementation for the largest network size, whereas OpenCL observed a higher speed-up of 715.4×. Table 14 provides the CUDA and OpenCL kernel and memory transfer times.

As evident from Table 14, the CUDA memory transfers are more efficient than the OpenCL memory transfers. CUDA takes the advantage of Zero Copy, which reduces the time spent in data transfers. However, the OpenCL kernel call performs better than the CUDA counterpart. Analysis of the ratios of the CUDA and OpenCL kernel times reveals that they are similar for *Implementations 1* and *2*, yet OpenCL significantly outperforms CUDA for *Implementation 2*. As explained in Sect. 4.2.1, *Implementation 1* requires the device (GPU) to provide the host processor with the global firing vector in each time-step. Therefore, the host processor is required to read the global firing vector in each time-step which is very costly operation, especially for a large image size 3120 × 3120. For the network size 3120 × 3120, the aforementioned operation results in a software time nearly equal to 10 seconds for both programming models. Adding this software time to the kernel time and memory transfer time (see Table 11) approximately equals the total runtime for the two programming models. The ratio of the estimated runtimes for the programming models for *Implementation 1* is 1.15, which is nearly equal to the observed speed-up ratio of the two programming models (140.9/116.45 equals 1.20). However, as explained in Sect. 4.2.2, *Implementation 2* serves to reduce the global firing vector transfer frequency, and hence the global firing vector reads at the host side. This optimization results in a significantly

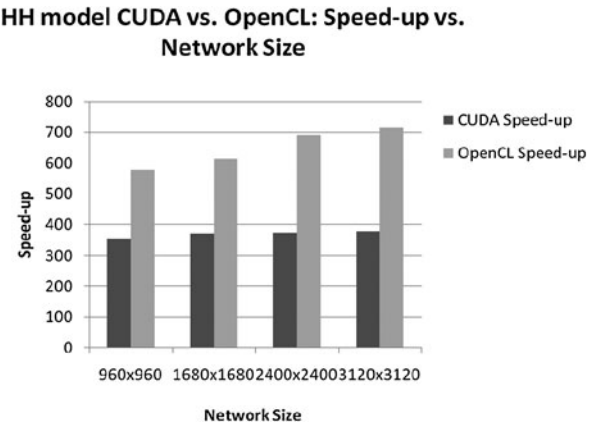

Fig. 4 HH model CUDA vs. OpenCL: Implementation 2

lower software time, approximately 0.3 seconds for both programming models for the network size 3120 × 3120.

Performing a similar analysis for *Implementation 2* with reference to Table 14, the ratio of the runtime calculations is 1.85, which is close to the ratio of the observed speed-up values (715.4/378.9 equals 1.88). Therefore, for *Implementation 2*, the contribution of greatly reduced software time and memory transfer time is overshadowed by a large value of the kernel time for both programming models. Therefore, OpenCL has a significant performance improvement compared to CUDA, given its significantly better performing (∼twice) kernel. Tables 15 and 16 provide the CUDA and OpenCL profiler results.

It is observed from Tables 15 and 16 that both CUDA and OpenCL kernels experience a similar number of *global misses* and *divergent branches*; however, the CUDA kernel attempts to hit the cache more often than the OpenCL kernel. The cache hits are larger than the previous implementation due to the cache loading scheme adopted in *Implementation 2*. Though the CUDA kernel involves less serialization due to higher *instruction performance ratio*, it issues almost twice as many instructions as the OpenCL kernel, resulting in CUDA's greater application time compared to OpenCL.

Finally, both CUDA and OpenCL performed the best with the block configuration of 256, which yielded an occupancy of 0.67 for CUDA, whereas OpenCL observed an occupancy of 0.833. Again we observe different occupancies yielded by the programming models for a similar block configuration.

5.2.3 HH model CUDA vs. OpenCL: Implementation 3

Implementation 3 (built on *Implementation 2*) involves the use of instruction optimization. Instruction optimization, as described in Sect. 2.2.1, involves the use of fast math and the reduction of conditional statements. Figure 5 summarizes the application speed-up results for the two programming models.

The CUDA application performance is found to be better than the OpenCL application speed-up for the final implementation of the implementation hierarchy. A maximum speed-up of 976.2× was observed for the largest network size (3120 × 3120)

Table 14 HH model CUDA and OpenCL kernel and memory transfer time: Implementation 2

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	455.92	18.5	231.46	41.35
1680	1364.43	45.38	495.84	86.64
2400	2767.17	86.23	1288.5	156.53
3120	4664.67	141.17	2158.05	230.3

Table 15 HH model CUDA profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	4.84e6	5.4e5	10	1.52e5	4.74e8	4.73e8	0.997
1680	1.44e7	2.05e6	12.5	4.2e5	1.45e9	1.45e9	1
2400	2.94e7	4.14e6	12.3	7.95e5	2.96e9	2.96e9	1
3120	4.97e7	7.07e6	12.5	1.31e6	7.05e8	7.02e8	0.995

Table 16 HH model OpenCL profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	4.6e6	0	0	1.5e5	2.26e8	2.13e8	0.94
1680	1.41e7	0	0	4.2e5	6.84e8	6.51e8	0.951
2400	2.87e7	0	0	8.1e5	1.4e9	1.32e9	0.94
3120	4.86e7	0	0	1.34e6	2.36e9	2.25e9	0.95

for the CUDA implementation, whereas OpenCL observed an application speed-up of 878.4× for the same network size. Turning off ECC yielded a speed-up of 1095× for the CUDA implementation and 1074× for the OpenCL implementation, for the largest network size (3120 × 3120).

Table 17 provides the CUDA and OpenCL kernel and memory transfer times. As seen in Table 17, both kernel calls and memory transfers are more efficient for the CUDA implementation than the OpenCL counterparts.

Tables 18 and 19 provide the CUDA and OpenCL profiler results used to analyze the difference in performance. While CUDA observes fewer *global misses* and a cache *hit-rate* of nearly 16%, OpenCL observes more *global misses* and no cache hits. OpenCL experiences fewer *divergent branches* and issues fewer instructions compared to CUDA; however, CUDA has significantly better *instruction performance ratio* (0.97 vs. 0.72), which leads to less CUDA kernel time. While CUDA observed

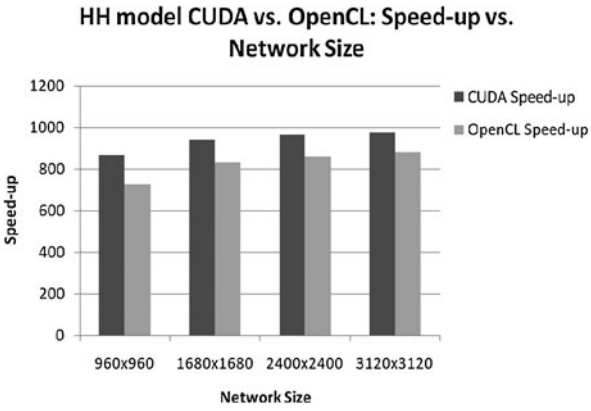

Fig. 5 HH model CUDA vs. OpenCL: Implementation 3

Table 17 HH model CUDA and OpenCL kernel and memory transfer time: Implementation 3

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	159.73	18.47	188.66	24.73
1680	462.186	45.29	530.1	49.5
2400	929.3	86.5	1062.6	88.33
3120	1564.5	141.67	1779.362	140.88

Table 18 HH model CUDA profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.9e6	7.17e5	15.5	2.4e5	1.78e8	1.73e8	0.971
1680	1.18e7	2.28e6	16.1	6.85e5	5.38e8	5.298	0.983
2400	2.4e7	4.75e6	16.5	1.32e6	1.1e9	1.08e9	0.981
3120	4.06e7	8.02e6	16.5	2.19e6	1.85e9	1.82e9	0.983

the highest occupancy of 1 for the block configuration of 192, a block configuration of 256 yielded the highest occupancy of 1 for the OpenCL implementation.

5.2.4 HH model CUDA vs. OpenCL: summary

OpenCL has higher performance compared to CUDA for the early implementations in the hierarchy. For *Implementations 1* and *2*, the CUDA kernel time has been observed to be twice that of the OpenCL kernel time. We attribute a relatively large number of *divergent branches* and *instructions issued* to the increased kernel time for the CUDA implementations. The performance of CUDA *Implementations 1* and *2* for

Table 19 HH model OpenCL profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	4.6e6	0	0	1.26e5	1.37e8	9.85e7	0.72
1680	1.4e7	0	0	3.56e5	4.17e8	3.01e8	0.72
2400	2.87e7	0	0	6.8e5	8.52e8	6.15e8	0.72
3120	4.86e7	0	0	1.23e6	1.44e9	1.04e9	0.722

the HH model, a dense computing model with Flops/Byte ratio equal to 9.87, is also penalized due to the sheer bulk of the serialized instructions. Additionally, CUDA observes lower occupancy compared to OpenCL for *Implementations 1* and *2*, 0.875 vs. 1 and 0.67 vs. 0.833, respectively. A low value of occupancy implies fewer active warps and hence a pronounced effect on an application with higher Flops/Byte ratio such as the HH model [20]. These reasons contribute to a lower performance for CUDA compared to OpenCL, even though instruction performances are in favor of CUDA.

The HH model, with Flops/Byte ratio of 9.84, has the best performance with CUDA for the final implementation in the hierarchy yielding the highest speed-up of 976× (1095× with ECC off) for the largest network size. *Implementation 3* exhausts all of the optimizations as described in Sect. 4.2.3. While CUDA takes a greater leap in performance for *Implementation 3*, any additional performance that can be achieved with OpenCL after the application of instruction level optimizations is thwarted by the fall in the *instruction performance ratio* coupled with a large number of *global misses* with no cache hits. These reasons contribute to only a slight improvement in kernel time and memory transfer time, and hence only a slight improvement in the overall speed-up. Although CUDA outperforms OpenCL only when all of the optimizations are employed, overall CUDA observes better memory utilization in terms of memory time and cache hits. CUDA's *instruction performance ratio* has also been observed to be better than that of OpenCL for most of the cases.

5.3 ML model CUDA vs. OpenCL

This subsection details the results and analysis for the three implementations of the second-most computation intensive model, the ML model with an 8.65 Flops/Byte ratio. *Implementation 1* results are presented in Sect. 5.3.1. *Implementation 2* results and discussion follow in Sect. 5.3.2. The results for the final implementation of the hierarchy, *Implementation 3,* is presented and discussed in Sect. 5.3.3. A summary and discussion of the results obtained for the implementations is presented in Sect. 5.3.4.

5.3.1 ML model CUDA vs. OpenCL: Implementation 1

Figure 6 summarizes the performance results for the two programming paradigms for the ML model. The CUDA application outperforms the OpenCL's performance

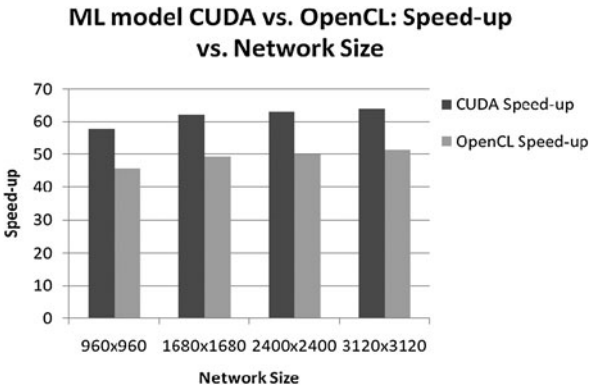

Fig. 6 ML model CUDA vs. OpenCL: Implementation 1

Table 20 ML model CUDA and OpenCL kernel and memory transfer time: Implementation 1

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	10.21	23.73	19.72	27.76
1680	30.12	57.99	51.3	73.41
2400	60.8	111.41	99.75	145
3120	101.94	183.7	165.2	226.33

for the first implementation in the hierarchy. CUDA has a speed-up of 64× for the largest network size, 3120 × 3120, whereas OpenCL has a speed-up of 51.5× for *Implementation 1* for the same network size.

Table 20 provides the CUDA and OpenCL kernel and memory transfer times for the intermediate network sizes. CUDA kernel calls and memory transfers are more efficient than those of OpenCL. For the ML model with Flops/Byte ratio of 8.65, this is in contrast with that of *Implementation 1* of the HH model where OpenCL's kernel time was observed to be almost half the CUDA kernel time.

Tables 21 and 22 provide the profiler results. Both CUDA and OpenCL experience a similar number of cache misses and no cache hits. CUDA kernel calls experience more *divergent branches* than OpenCL kernel calls. Though the CUDA kernel issues slightly more instructions compared to the OpenCL kernel for a similar kernel structure, the CUDA implementation has marginally better *instruction performance ratio* compared to OpenCL. Both the CUDA and OpenCL implementations observe the maximum occupancy of 1 with the block configuration size of 192.

5.3.2 ML model CUDA vs. OpenCL: Implementation 2

Implementation 2 exhausts all of the memory optimization techniques available with the two programming models, including the extra optimization techniques for CUDA, Z and cache loading. Figure 7 summarizes the performance results for the program-

Table 21 ML model CUDA profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.3e5	0	0	1.2e3	9.23e6	9.2e6	0.996
1680	4e5	0	0	2.4e3	2.82e7	2.81e7	0.996
2400	8.23e5	0	0	3e3	5.74e7	5.73e7	9.998
3120	1.39e6	0	0	4.6e3	9.7e7	9.7e7	1

Table 22 ML model OpenCL profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.31e5	0	0	5.8e2	5e6	4.9e6	0.98
1680	4.03e5	0	0	1.25e3	1.55e7	1.5e7	0.96
2400	8.24e5	0	0	1.5e3	3.16e7	3.07e7	0.97
3120	1.38e6	0	0	2.2e3	5.32e7	5.16e7	0.97

ming models where CUDA outperforms OpenCL with a speed-up of 159× versus 90× for the largest network size.

Table 23 provides the CUDA and OpenCL kernel and memory transfer times for the implementation. The profiler results are helpful for analyzing the performance of the two programming models and are provided in Tables 24 and 25.

As expected for *Implementation 2*, the memory transfer time for both CUDA and OpenCL has dropped due to the reduction in the frequency of the transfers of the global firing vector from the GP-GPU to the host with the introduction of the block firing vector. As observed in the previous implementation, CUDA has better performance than OpenCL for kernel calls and memory transfers, but CUDA involves additional memory optimization techniques which further reduces the memory transfer time.

Though the CUDA kernel issues more instructions, it still performs better than OpenCL for similar kernel structure with a better *instruction performance ratio* because it results in less instruction serialization and hence promotes concurrency amongst the threads in a warp. CUDA also has fewer *divergent branches* compared to OpenCL. OpenCL, in general has fewer *global misses*. However, for the ML neuron model, CUDA has no cache hits although the cache loading technique is turned on. A block configuration of 512 was observed to perform the best for CUDA implementation and yielded an occupancy of 1. OpenCL observed the best performance and maximum occupancy of 1 with a block configuration of 256.

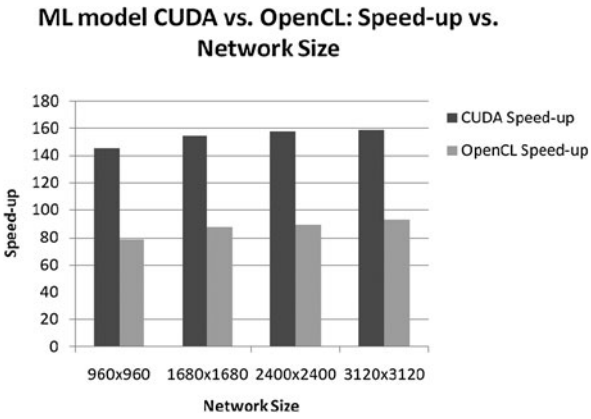

Fig. 7 ML model CUDA vs. OpenCL: Implementation 2

Table 23 ML model CUDA and OpenCL kernel and memory transfer time: Implementation 2

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	10.5	10.7	20	15.13
1680	30.78	28.88	51.85	44.53
2400	62.2	56.73	100.78	91.1
3120	104.6	94.26	167.26	137.61

Table 24 ML model CUDA profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.31e5	0	0	4.25e3	1.125e7	1.122e7	0.995
1680	7.56e5	0	0	1.18e4	3.45e7	3.44e7	0.997
2400	8.23e5	0	0	2.1e4	7.04e7	7.02e7	0.997
3120	4.84e6	0	0	3.4e4	1.19e8	1.18e7	0.991

5.3.3 ML model CUDA vs. OpenCL: Implementation 3

Implementation 3 extends *Implementation 2* to accommodate instruction optimization as described in Sect. 4.2.3. Figure 8 summarizes the performance results for the two programming models. CUDA outperforms OpenCL with a speed-up of 190.67× versus 144.5× for the largest network size. Table 26 provides the kernel and memory transfer times for the two programming models. Tables 27 and 28 provide the relevant profiler results. Both CUDA and OpenCL have similar kernel performance for similar kernel structures. However, the OpenCL memory transfers are more efficient than the CUDA counterpart. The CUDA profiler counters are larger than those

Table 25 ML model OpenCL profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.32e5	0	0	7e3	6.25e6	6.05e6	0.97
1680	4.04e5	0	0	2e4	1.9e7	1.86e7	0.98
2400	8.2e5	0	0	3.77e4	3.9e7	3.77e7	0.967
3120	1.38e6	0	0	6.13e4	6.57e7	6.4e7	0.97

Table 26 ML model CUDA and OpenCL kernel and memory transfer time: Implementation 3

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	5.48	10.78	5.6	8.9
1680	15.55	29.08	14.83	23.13
2400	31.27	56.73	30.95	49.06
3120	52.58	94.34	51.69	81.81

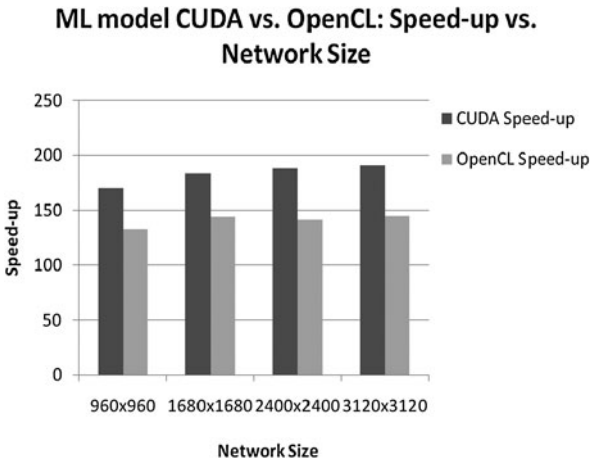

Fig. 8 ML model CUDA vs. OpenCL: Implementation 3

of OpenCL; however, CUDA experiences a larger *hit-rate* and slightly better *instruction performance ratio* compared to OpenCL. A block configuration of 192 yielded an occupancy of 1 for the CUDA implementation, whereas a block configuration of 256 yielded maximum occupancy of 1 for the OpenCL implementation.

5.3.4 ML model CUDA vs. OpenCL: summary

Unlike the HH model, CUDA performs better for the first two implementations in the hierarchy of the ML model both in terms of kernel and memory transfer times.

Table 27 ML model CUDA profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.44e5	2.14e4	13	1e4	6.4e6	5.72e6	0.89
1680	4.4e5	6.62e4	13	3e4	1.95e7	1.75e7	0.89
2400	8.94e5	1.36e5	13	6.1e4	3.97e7	3.6e7	0.90
3120	1.5e6	2.3e5	13	9.86e4	6.72e7	6.03e7	0.89

Table 28 ML model OpenCL profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	1.31e5	0	0	4.7e3	4.42e6	3.83e6	0.87
1680	4e5	0	0	1.4e4	1.36e7	1.17e7	0.86
2400	8.2e5	0	0	2.7e4	2.77e7	2.4e7	0.87
3120	1.38e6	0	0	4.5e5	4.7e7	4.04e7	0.86

Though the CUDA kernel calls experience slightly more *divergent branches* and *instructions issued,* the first two CUDA implementations have a better *instruction performance ratio* compared to OpenCL. Additionally, in contrast to all the CUDA implementations of the HH model, all CUDA implementations of the ML model experience fewer serialized instructions which has a milder effect on the performance. *Implementations 1* and *2* for the ML model have better multi-processor occupancy compared to corresponding implementations for the HH model (1 vs. 0.875 and 1 vs. 0.67, respectively). These factors contribute to the higher performance of CUDA over OpenCL for the first two implementations in the hierarchy. For *Implementation 3*, CUDA performs better in terms of application speed-up although OpenCL matches CUDA in terms of kernel calls and observes slightly better memory performance even though the *cache hit rate* and *instruction performance ratio* remain in favor of the CUDA implementation.

5.4 Wilson model CUDA vs. OpenCL

The following subsections discuss the results for the three implementations in the hierarchy using the Wilson neuron model. The Wilson model has a relatively low Flops/Byte ratio of 1.52 compared to the previous models discussed. The most naïve implementation in the hierarchy, *Implementation 1* is discussed in Sect. 5.4.1. *Implementation 2* reduces the global firing vector frequency leading to a better memory performance and is presented in Sect. 5.4.2. *Implementation 2* is extended to *Implementation 3* with the addition of instruction-level optimizations as discussed in Sect. 4.2.3. *Implementation 3* results are presented in Sect. 5.4.3. Finally, Sect. 5.4.4 summarizes the findings for all of the implementations for the Wilson model.

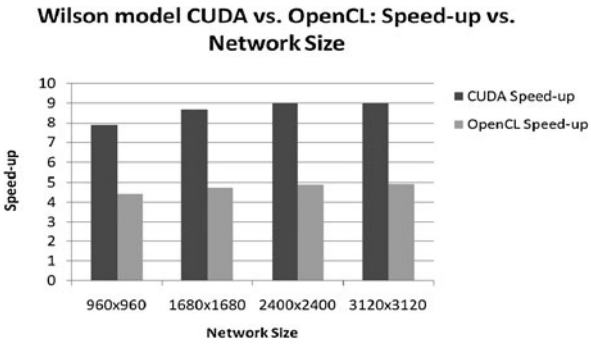

Fig. 9 Wilson model CUDA vs. OpenCL: Implementation 1

Table 29 Wilson model CUDA and OpenCL kernel and memory transfer time: Implementation 1

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	15.33	40.5	35.75	47.13
1680	45.15	98.7	96.68	121.52
2400	91.18	189.32	190.66	236.92
3120	153.4	312.06	318.62	372.66

5.4.1 Wilson model CUDA vs. OpenCL: Implementation 1

Implementation 1 results are given in Fig. 9. It is observed that the CUDA kernel consistently performs better than OpenCL for all of the network sizes with a highest speed-up of 9× for the largest network size, whereas OpenCL has a maximum speed-up of 4.9× for the same network size. Table 29 provides the kernel and memory transfer times for the two programming models. As seen in Table 29, both the CUDA kernel calls and memory transfers are more efficient than those of OpenCL. The CUDA kernel calls are almost twice as efficient as the OpenCL kernel calls, which is in contrast with *Implementation 1* of the HH model.

Tables 30 and 31 provide the profiler counter values for the two programming paradigms. While both the CUDA and OpenCL implementations have a similar number of *global misses*, OpenCL has no *divergent branches* and issues relatively fewer instructions than CUDA. However, CUDA has larger *instruction performance ratio* compared to OpenCL, leading to less instruction serialization and hence promoting concurrency amongst the threads in a warp.

5.4.2 Wilson model CUDA vs. OpenCL: Implementation 2

Implementation 2 uses all of the memory optimization techniques presented in Sect. 2.2.1 in order to improve the performance. Figure 10 provides the CUDA and OpenCL application speed-up results. CUDA outperforms OpenCL with a maximum speed-up of 20.5× for the largest network size versus OpenCL's application speed-up of 10.5× for the same network size. Table 32 provides the CUDA and OpenCL

Table 30 Wilson model CUDA profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	6.9e2	1.1e7	5.95e6	0.54
1680	1.13e6	0	0	1.5e3	3.3e7	1.8e7	0.54
2400	2.3e6	0	0	1.85e3	6.6e7	3.7e7	0.56
3120	3.9e6	0	0	2.6e3	1.12e8	6.23e7	0.556

Table 31 Wilson model OpenCL profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	0	9.72e6	4.32e6	0.44
1680	1.13e6	0	0	0	2.97e7	1.32e7	0.44
2400	2.3e6	0	0	0	6.1e7	2.7e7	0.44
3120	3.9e6	0	0	0	1.04e8	4.5e8	0.43

kernel and memory transfer times for the intermediate network sizes. It is again observed that for a similar kernel implementation, the CUDA kernel calls are more efficient than the OpenCL kernel calls. The CUDA memory transfers, with the addition of Zero Copy, perform better than the OpenCL memory transfers.

Tables 33 and 34 provide the profiler counter values for the two programming models. CUDA and OpenCL experience a similar number of *global misses*. While CUDA experiences fewer *divergent branches*, OpenCL issues fewer instructions for a similar kernel structure; however, *instruction performance ratio* for CUDA is significantly better than that of OpenCL. CUDA has on an average an *instruction performance ratio* of 0.75, whereas OpenCL has a lower average value of 0.5. A higher value of *instruction performance ratio* implies a lower degree of instruction serialization. CUDA has an occupancy of 0.67 with the best performing block configuration of 512, OpenCL has an occupancy of 1 with its best performing block configuration of 256.

5.4.3 Wilson model CUDA vs. OpenCL: Implementation 3

Implementation 3, which is the final implementation of the hierarchy, adds instruction level optimization to *Implementation 2*. Figure 11 provides the application speed-up results for CUDA and OpenCL.

While CUDA does not have any significant performance improvement with the instruction optimization, OpenCL has a noticeable improvement in speed-up. For the largest network size, OpenCL has a speed-up of 15.9×. Nonetheless, CUDA outperforms OpenCL with a speed-up of 20.1× for the same network size. Table 35

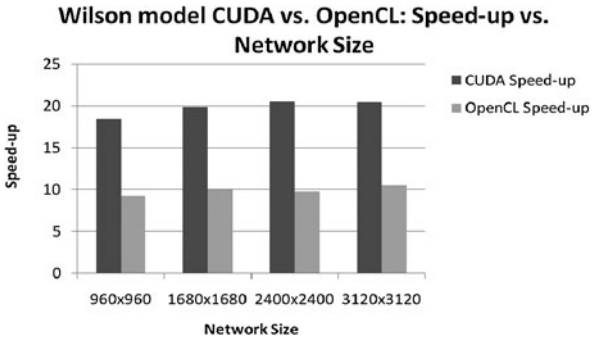

Fig. 10 Wilson model CUDA vs. OpenCL: Implementation 2

Table 32 Wilson model CUDA and OpenCL kernel and memory transfer time: Implementation 2

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	15.17	15.50	35.36	22.4
1680	44.33	42.07	95.73	65.82
2400	89.37	83.04	189.84	131.7
3120	150.45	138.67	317.39	200.6

Table 33 Wilson model CUDA profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	5.5e3	1.26e7	9.7e6	0.77
1680	1.13e6	0	0	1.65e4	3.8e7	2.95e7	0.77
2400	2.31e6	0	0	3.4e4	7.9e7	6e7	0.76
3120	3.9e6	0	0	5.7e4	1.32e8	1e8	0.76

provides the kernel and memory transfer times, while Tables 36 and 37 provide the profiler counter values for CUDA and OpenCL.

Though the CUDA kernel calls seem to be better than those for OpenCL, the OpenCL memory transfers have proved to be more efficient than that of CUDA for this implementation. Both programming models have a similar number of *global misses* and *instructions issued*; however, CUDA has a significantly better *instruction performance ratio*. While a best configuration of 256 provided an occupancy of 0.67 for CUDA, the same block configuration yielded an occupancy of 1 for OpenCL.

5.4.4 Wilson model CUDA vs. OpenCL: summary

CUDA performs better than OpenCL for the very first implementation in the hierarchy, *Implementation 1*. The CUDA kernel calls and memory transfers are found

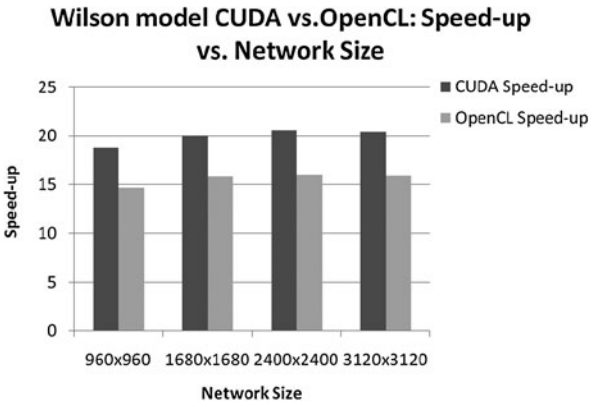

Fig. 11 Wilson model CUDA vs. OpenCL: Implementation 3

Table 34 Wilson model OpenCL profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	1e4	1.14e7	5.78e6	0.5
1680	1.13e6	0	0	2.9e4	3.52e7	1.77e7	0.5
2400	2.3e6	0	0	5.9e4	7.24e7	3.61e7	0.49
3120	3.9e6	0	0	9.9e4	1.21e8	6.1e7	0.504

Table 35 Wilson model CUDA and OpenCL kernel and memory transfer time: Implementation 3

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	14.6	15.52	15.31	14.35
1680	42.57	42.1	43.06	37.51
2400	86.16	83.13	87.62	79.2
3120	145.05	138.02	146.97	132.5

to be more efficient than OpenCL. While OpenCL maintains comparable figures for most of the profiler counters, CUDA has a better *instruction performance ratio* and similar results are found in *Implementation 2*. While both CUDA and OpenCL take advantage of the reduced global firing vector frequency due to the use of block firing vector, the CUDA kernel calls are found to be more efficient than those for OpenCL with fewer *divergent branches* and a significantly better *instruction performance ratio*. With the addition of instruction-level optimizations, OpenCL has significantly better performance improvement for *Implementation 3* compared to CUDA. Though OpenCL memory transfers are more efficient than those for CUDA, the CUDA kernel calls are more efficient with a significantly better *instruction performance ratio*.

Table 36 Wilson model CUDA profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	1.1e4	1.01e7	9.7e6	0.96
1680	1.13e6	0	0	3.3e4	3.303e7	2.95e7	0.90
2400	2.3e6	0	0	6.67e4	6.2e7	6e7	0.96
3120	3.9e6	0	0	1.13e5	1.04e8	1.01e8	0.97

Table 37 Wilson model OpenCL profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	3.7e5	0	0	8.4e3	9.6e6	4.61e6	0.48
1680	1.13e6	0	0	2.5e4	2.9e7	1.42e7	0.489
2400	2.3e6	0	0	5e4	6e7	2.87e7	0.478
3120	3.88e6	0	0	8.36e4	1e8	4.88e7	0.488

Unlike previous SNN models with a higher Flops/Byte ratio, the Wilson model has a wide gap in the *instruction performance ratios* for the two programming models. Additionally, the *instruction performance ratio* is observed to follow the Flops/Byte ratio.

5.5 Izhikevich model CUDA vs. OpenCL

The next series of experiments using the implementation hierarchy explores the behavior of CUDA and OpenCL for the most computation friendly Izhikevich model with lowest Flops/Byte ratio of 0.9997. *Implementation 1* results are presented and discussed in Sect. 5.5.1 followed by *Implementation 2* results in Sect. 5.5.2. *Implementation 3* exhausts all of the optimizations and is studied for Izhikevich model in Sect. 5.5.3. Finally, Sect. 5.5.4 summarizes the results obtained for the three implementations using the Izhikevich model.

5.5.1 Izhikevich model CUDA vs. OpenCL: Implementation 1

Figure 12 provides the application speed-up results for the two programming models. CUDA again outperforms OpenCL with a maximum speed-up of 7.83× for the largest network size versus 5.88× for the OpenCL implementation for the same network size. Table 38 provides the CUDA and OpenCL kernel and memory transfer times.

Tables 39 and 40 provide the important CUDA and OpenCL profiler counter values for the intermediate network sizes, respectively. As seen in the previous SNN

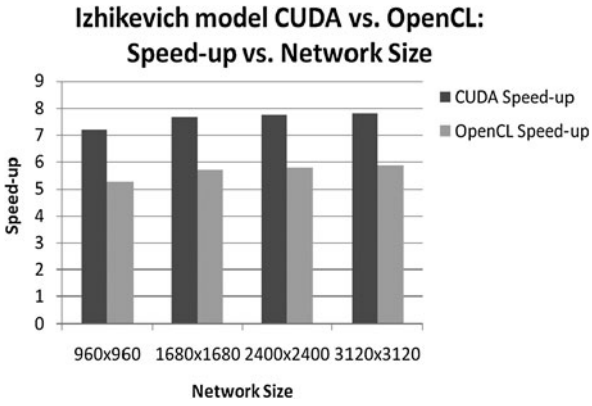

Fig. 12 Izhikevich model CUDA vs. OpenCL: Implementation 1

Table 38 Izhikevich model CUDA and OpenCL kernel and memory transfer time: Implementation 1

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	2.87	20.36	14.35	20.877
1680	7.982	46.63	36.18	55.54
2400	15.87	86.69	70.12	109.17
3120	26.5	140.73	115.5	173.22

Table 39 Izhikevich model CUDA profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	7.4e4	0	0	0	1.08e6	9.4e5	0.87
1680	2.35e5	0	0	0	3.27e6	2.87e6	0.87
2400	4.62e5	0	0	0	6.71e6	5.85e6	0.87
3120	7.8e5	0	0	0	1.13e7	9.9e6	0.87

neuron models, both the CUDA kernel calls and memory transfers were found to be more efficient than that of OpenCL. While the *global miss* and *divergent branch* values were comparable for the two programming paradigms, CUDA issues slightly more instructions than OpenCL. As seen in previous implementations, CUDA involves less instruction serialization compared to OpenCL, given its larger *instruction performance ratio*, resulting in better overall performance. Both CUDA and OpenCL have the highest occupancy of 1 for the best block configuration of 256.

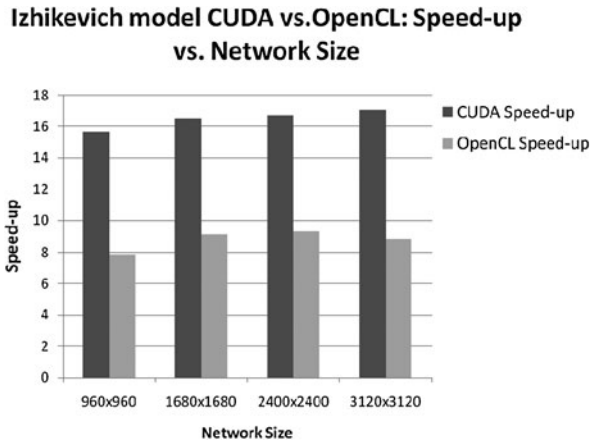

Fig. 13 Izhikevich model CUDA vs. OpenCL: Implementation 2

Table 40 Izhikevich model OpenCL profiler results: Implementation 1

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	7.4e4	0	0	0	1.02e6	8.5e5	0.83
1680	2.25e5	0	0	0	3.1e6	2.65e6	0.85
2400	4.6e5	0	0	0	6.34e6	5.37e6	0.85
3120	7.77e5	0	0	0	1.07e7	9.1e6	0.85

5.5.2 Izhikevich model CUDA vs. OpenCL: Implementation 2

Implementation 2 results are summarized in Fig. 13 where it is again observed that CUDA provides superior results compared to OpenCL. For the largest network size 3120 × 3120, CUDA has a speed-up of 17.07× whereas OpenCL has a speed-up of 8.87×. Table 41 presents the kernel and memory transfer times for CUDA and OpenCL.

Tables 42 and 43 provide the profiler counter values. As seen previously, CUDA performs better than OpenCL both in kernel calls and memory transfers. The CUDA kernel hits the cache with a rate of 18.5% whereas OpenCL does not hit the cache. While both CUDA and OpenCL have a similar number of *instructions issued*, CUDA's *instruction performance ratio* is superior to that of OpenCL. Both programming models performed the best with the block configuration of 256 which yielded an occupancy of 1.

5.5.3 Izhikevich model CUDA vs. OpenCL: Implementation 3

Figure 14 summarizes the results for the final implementation of the hierarchy. CUDA has the highest speed-up of 17.1× for the largest network size, whereas the OpenCL application speed-up lags behind with a speed-up of 12.22×. Table 44 provides the

Table 41 Izhikevich model CUDA and OpenCL kernel and memory transfer time: Implementation 2

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	3.3	11.57	14.35	12.53
1680	7.99	27.23	32.46	27.08
2400	15.8	50.13	62.02	52.57
3120	26.31	80.92	108.72	116.8

Table 42 Izhikevich model CUDA profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	8.17e4	1.85e4	18.5	5e3	1.95e6	1.83e6	0.94
1680	2.5e5	5.7e4	18.5	1.4e4	5.97e6	5.7e6	0.95
2400	5.1e5	1.16e5	18.5	2.86e4	1.22e7	1.15e7	0.94
3120	8.65e5	1.96e5	18.5	4.87e4	2.07e7	1.95e7	0.94

Table 43 Izhikevich model OpenCL profiler results: Implementation 2

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	7.4e3	0	0	4.3e3	1.56e6	1.4e6	0.89
1680	2.25e5	0	0	1.4e4	4.73e6	4.3e6	0.90
2400	4.6e5	0	0	2.7e4	9.7e6	8.8e6	0.90
3120	7.8e5	0	0	4.6e4	1.6e7	1.5e7	0.94

kernel and memory transfer times for both of the programming models. Both programming models perform similarly as far as kernel calls and memory transfers are concerned with CUDA having a slight edge over the OpenCL implementation.

Tables 45 and 46 provide profiler counter values for the CUDA and OpenCL implementations respectively. Both CUDA and OpenCL have comparable profiler values; however, CUDA has cache hits whereas OpenCL has none. Unlike previous cases, OpenCL has a higher *instruction performance ratio* compared to CUDA leading to its slightly lower kernel time. Both CUDA and OpenCL performed the best with block configuration of 256 which yielded the maximum occupancy of 1.

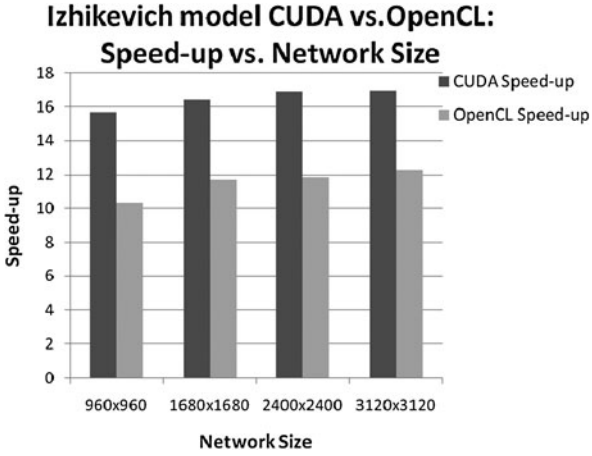

Fig. 14 Izhikevich model CUDA vs. OpenCL: Implementation 3

Table 44 Izhikevich model CUDA and OpenCL kernel time and memory transfer time: Implementation 3

Network size	CUDA		OpenCL	
	Kernel time (ms)	Memory transfer time (ms)	Kernel time (ms)	Memory transfer time (ms)
960	2.89	11.7	3.53	13.38
1680	7.6	23.81	7.97	29.6
2400	14.8	49.97	14.73	54.55
3120	24.64	81.53	23.88	87.86

Table 45 Izhikevich model CUDA profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	8e4	1.85e4	18.8	3.1e3	1.65e6	1.43e6	0.87
1680	2.45e5	5.64e4	18.7	9.5e3	5.1e6	4.4e6	0.86
2400	5e5	1.15e5	18.7	1.94e4	1.03e7	8.9e6	0.86
3120	8.45e5	1.95e5	18.8	3.27e4	1.74e7	1.51e7	0.87

5.5.4 Izhikevich model CUDA vs. OpenCL: summary

Implementation 1 for the Izhikevich model did not have any significant differences in performance for the two programming paradigms. However, as seen previously, the CUDA kernel calls and memory transfers are relatively more efficient than that of OpenCL. While most of the profiler counters were similar for both of the programming models, CUDA has a better *instruction performance ratio*. *Implementation 2* provides additional performance gain with reduced global firing vector frequency for both CUDA and OpenCL; however, both the CUDA kernel calls and memory trans-

Table 46 Izhikevich model OpenCL profiler results: Implementation 3

Network size	L1 global miss	L1 global hit	L1 hit rate%	Divergent branch	Instructions issued	Instructions executed	Instruction performance ratio
960	7.4e4	0	0	3e3	1.12e6	9.93e5	0.88
1680	2.26e5	0	0	9.4e3	3.45e6	3.04e6	0.88
2400	4.6e5	0	0	2e4	6.97e6	6.24e6	0.89
3120	7.8e5	0	0	3.2e4	1.17e7	1.05e7	0.89

fers were more efficient than the OpenCL counterparts. CUDA continues to outpace OpenCL in terms of the *cache hit ratio* and *instruction performance ratio.*

In *Implementation 3*, both CUDA and OpenCL had near similar kernel performance with similar profiler counter values. CUDA maintains its *cache hit ratio* for *Implementation 3* but OpenCL observes no cache hits. CUDA memory transfers were also found to be slightly more efficient than the OpenCL counterpart; however, larger than expected differences in speed-up values were observed for the two programming models. A similar outlier was observed in the case of ML model *Implementation 3* where most of the profiler values were in favor of OpenCL; however, speed-up values favored the CUDA implementation. The authors observed longer software times for OpenCL *Implementation 3* compared to the CUDA counterparts. The reasons for the increased software time are under further investigation by the authors. Currently, the authors attribute the behavior of these outliers observed in some of the implementations to variations and randomness in the operating system and in future work plan to explore the operating system effects on programming models and co-processors. For the Wilson and Izhikevich models, the *instruction performance ratio* falls with the Flops/Byte ratio; however, the fall is not as pronounced as in the case of Wilson model. It should be noted that though the Wilson model has a small edge over Izhikevich model as far as Flops/Byte ratio is concerned, it involves larger memory transactions (almost twice) and more computations (almost thrice) compared to Izhikevich model. Hence the results presented in above subsections also depend on the amount of memory and computation together with their ratio.

5.6 Analysis: application to architecture to programming model links

In this subsection, we summarize and analyze the results for CUDA vs. OpenCL comparative study conducted on the Fermi architecture using four different SNN neuron models, each with different computation and communication requirements. We establish a triangular relationship between the device parameters (profiler parameters, such as instructions issued/executed, global hits/misses, divergent branches, and instruction performance ratio), programming models and applications.

Tables 47 and 48 summarize the speed-up improvement ratios between successive implementations in the hierarchy for each of the SNN neuron models. The speed-up improvement ratios are found to vary from a model to another.

First, we establish the "programming model to device parameters link." A majority of the experimental results were found to be in favor of the CUDA kernel call

Table 47 CUDA vs. OpenCL speed-up improvement ratios: HH model and ML model

	HH model		ML model	
	Impl. 1 to Impl. 2	Impl. 2 to Impl. 3	Impl. 1 to Impl. 2	Impl. 2 to to Impl. 3
CUDA	3.25×	2.57×	2.5×	1.1×
OpenCL	5.1×	1.22×	1.8×	1.6×

Table 48 CUDA vs. OpenCL speed-up improvement ratios: Wilson model and Izhikevich model

	Wilson model		Izhikevich model	
	Impl. 1 to Impl. 2	Impl. 2 to Impl. 3	Impl. 1 to Impl. 2	Impl. 2 to to Impl. 3
CUDA	2×	1×	2.2×	1.0×
OpenCL	2×	1.5×	1.5×	1.4×

and memory transfer performance. Although the CUDA kernel issued more instructions per kernel call, its *instruction performance ratio* was found to be better than OpenCL's *instruction performance ratio*. CUDA observed in general more *divergent branches* and *global misses*, although the CUDA kernel calls hit the cache more often than that for OpenCL thereby yielding a better *cache hit ratio*.

In many cases for the two programming models, better occupancies were observed for different block configurations. The differences in profiler counter values observed are primarily attributed to the different compilers used by the programming models. While CUDA uses *open64*, OpenCL uses *llvm* as its compiler; both result in the generation of different ptx assembly files, which result in different profiler parameters. The parameters such as the *instruction performance ratio* and *cache hit rate* have larger effect on the kernel call performance. These parameters were found to be in favor of CUDA, which is expected since CUDA is tightly coupled with Nvidia's architecture and hence is able to better exploit these device parameters.

Next we establish the "application to device parameters link." We specifically observe the effects of application parameters on the device parameters. The profiler counters are observed to have a stronger correlation with the performance of the models having a higher Flops/Byte ratio (HH and ML models). The effects of the profiler counters on the performance improvement ratios were found to be stronger in the case of HH model, an example application with a high Flops/Byte ratio, though the effects were more or less uniform for the other models. Nonetheless, the authors strongly emphasize on the role of Flops/Byte ratio in magnifying the effects of the profiler counters on the application performance. Higher Flops/Byte ratio models also observed higher instruction performance ratio in general.

In addition to the Flops/Byte ratio, the number of FLOPs performed and memory transactions also must be considered individually as they affect the device parameters and hence the application performance as demonstrated in the Wilson and Izhikevich model cases. Though the Wilson model has an edge over the Izhikevich model

in terms of the Flops/Byte ratio, the Wilson model has larger communication and computation requirements and therefore has a significantly lower *instruction performance ratio* than the Izhikevich model. Further, the overall application performance was similar for the two SNN neuron models, though one would expect a model with a higher Flops/Byte ratio to be capable of extracting more performance from the GP-GPU architecture. Therefore, the degree to which the application parameters affect the device parameters also depends on the magnitude of communication and computation overheads, along with their ratio. Consequently, the application parameters, namely FLOPs, bytes transferred, and Flops/Byte ratio that vary from application to application, influence the device parameters as measured with the profiler. Since the application parameters can strongly affect the device parameters, and hence the overall application performance, we observe some variations in performance and improvement ratios from one SNN neuron model to the next.

Finally, we establish the "programming model to application link." The results presented identify the CUDA memory transfer strategies (SP, SM, and Z) to be more efficient in general than the OpenCL memory transfer strategies (SP, SM and SW), particularly for the applications with higher Flops/Byte ratio (i.e., the HH and ML neuron models). Additionally, CUDA allows the user to configure the cache for an application as either *cache load scheme* or *non-cache load scheme* for those applications that involve frequent high latency global memory accesses or those that suffer from register pressure. OpenCL, however, with its generic structure does not offer these additional optimizations that an application can use to gain better performance with the Fermi architecture. With the speed-up gained by CUDA over OpenCL in the HH neuron model's *Implementation 3*, it is also safe to establish that CUDA's instruction level optimization (fast math and RCS) are superior to those provided in OpenCL (native and unsafe math) for a highly computation intensive application.

Therefore, we conclude for the Nvidia architecture that the programming model to application link is stronger for CUDA compared to OpenCL. Exhaustive studies conducted in Sects. 5.2 through 5.4 establish CUDA as the overall choice to extract the best performance from Nvidia's architecture. CUDA has a stronger device parameter-programming model link and stronger optimization techniques that establish a tighter link between the programming model and application compared to OpenCL. Though some instances were observed where OpenCL outperformed CUDA, particularly in the earlier *Implementations 1* and *2* of the HH model, CUDA overtakes OpenCL when all of the optimizations are exhausted. Nonetheless, this indicates at this time, using OpenCL for such applications will restrict the developer's ability to maximize optimizations and extract maximum performance from Nvidia's architecture.

6 Conclusion and future work

In this paper, we have surveyed the two most popular GP-GPU programming models (CUDA and OpenCL) and two GP-GPU architectures from different vendors (Nvidia's Fermi and AMD/ATi's Radeon 5870). The programming models and architectures were compared for performance using a two-level character recognition network developed employing the four most accurate SNN models (i.e., the HH,

ML, Wilson and the Izhikevich models), which possess different computation-to-communication requirements. A hierarchy of implementations was developed by successively adding optimization strategies available in the two programming models. We have also compared the programming models by varying the problem size.

With a broad survey conducted using a wide range of application complexity, multiple implementation hierarchy and varying problem size, it has been found that the CUDA programming model can extract more performance out of Nvidia's GPU architecture compared to OpenCL. Both the kernel calls and memory transfers were found to be more efficient using CUDA for most of the cases. The difference in performance is mainly attributed to the different compilers used by the two programming models. While CUDA uses the *open64* compiler and is solely dedicated for the Nvidia GPUs, OpenCL uses *llvm* as its compiler and is required to preserve its general structure in order to accommodate other architectures. This artifact prevents OpenCL from extracting maximum performance out of Nvidia's architecture.

Our study satisfactorily establishes CUDA as the preferred choice for developing applications using Nvidia's GPUs, provided portability is not an issue. However, our study also shows that there can be instances where OpenCL can potentially outperform CUDA hence we cannot strictly establish CUDA as the preferred model for Nvidia's devices. Our intermediate implementations of the HH model affirm our argument, where it was found that OpenCL performed better than CUDA for the *Implementations 1* and *2*. There can be applications similar to HH model (similar Flops/Byte ratio) that potentially restrict the developer's ability to perform some optimizations beyond those in *Implementations 1* and *2* and these could potentially achieve maximum performance with the OpenCL programming model. Nonetheless, OpenCL continues to be an important programming paradigm due to its portability across architectures, an important attribute that CUDA does not possess.

In our second study, we conduct a head-to-head of the state-of-the-art architectures from Nvidia and AMD/ATi (Fermi and Radeon, respectively). Though 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 maximum performance out of the Radeon architecture. Though Brooks++ is available to the developers to program the AMD/ATi'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 include a detailed study of the operating system effects on programming models and co-processors.

Acknowledgements 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. 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

2. Tilera's homepage. http://www.tilera.com/products/processors
3. NVIDIA CUDA programming guide. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf
4. Ligowski L, Rudnicki W (2009) An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In: Proceedings of IPDPS 2009, Rome, Italy, May 2009
5. Phillips JC, Stone JE, Schulten K (2008) Adapting a message-driven parallel application to GPU-accelerated clusters. In: Proceedings of SC 2008, Austin, TX, November 2008
6. OpenCL-open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/
7. Izhikevich E (2004) Which model to use for cortical spiking neurons? IEEE Trans Neural Netw 15(5):1063–1070
8. Izhikevich EM (2003) Simple model to use for cortical spiking neurons. IEEE Trans Neural Netw 14(6):1569–1572
9. Wilson HR (1999) Simplified dynamics of human and mammalian neocortical neurons. J Theor Biol 200:375–388
10. Morris C, Lecar H (1981) Voltage oscillations in the barnacle giant muscle fiber. Biophys J 35:193–213
11. Hodgkin AL, Huxley AF (1952) A quantitative description of membrane current and application to conduction and excitation in nerve. J Physiol 117:500–544
12. Bhuiyan MA, Pallipuram, VK, Smith MC (2010) Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In: HiCOMB 2010, a workshop in IPDPS 2010, Atlanta, GA, April 2010
13. Gupta A, Long L (2007) Character recognition using spiking neural networks. In: Proc. IJCNN, pp. 53–58, August 2007
14. Technical Brief: NVIDIA GeForce 8800 GPU architecture overview. www.nvidia.com
15. NVIDIA's next generation CUDA compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf
16. ATI Mobility Radeon HD 5870 GPU specifications. http://www.amd.com/us/products/notebook/graphics/ati-mobility-hd-5800/Pages/hd-5870-specs.aspx
17. NVIDIA CUDA C programming best practices guide. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf
18. NVIDIA OpenCL programming guide. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf
19. Du P, Weber R, Tomov S, Peterson G, Dongarra J (2010) 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
20. Pallipuram VK (2010) Acceleration of spiking neural networks on single-GPU and multi-GPU systems. Master's thesis, May 2010
21. Johansson C, Lansner A (2007) Towards cortex sized artificial neural systems. Neural Netw 20(1), 48–61
22. Nene SA, Nayar SK, Murase H (1996) Columbia object image library (COIL-100) (No. CUCS-006-96): Columbia Automated Vision Environment
23. Ananthanarayanan R, Esser SK, Simon HD, Modha DS (2009) The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses. In: Proceedings of SC '09, Portland, Oregon, November 2009
24. Rall W (1959 Branching dendritic trees and motoneuron membrane resistivity. Exp Neurol 1, 503–532
25. Nageswaran JM, Dutt N, Krichmar JL, Nicolau A, Veidenbauma AV (2009) A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Netw 22(5–6), 791–800. Special issue
26. Khanna G., McKennon J. (2010) Numerical modeling of gravitational wave sources accelerated by OpenCL. Comput Phys Commun 181(9), 1605–1611
27. Karimi K, Dickson NG, Hamze F (2010) A performance comparison of CUDA and OpenCL. The Computing Research Repository (CoRR), arXiv:1005.2581
28. Bhuiyan MA, Taha TM, Jalasutram R (2009) Character recognition with two spiking neural network models on multi-core architectures. In: Proceedings of IEEE symposium on CIMSVP, Nashville, TN, March 2009, pp. 29–34

29. ATI stream computing OpenCL. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf
30. CUDA visual profiler release notes. http://developer.download.nvidia.com/compute/cuda/3_0/sdk/docs/OpenCL_release_notes.txt
31. ATI stream profiler. http://developer.amd.com/gpu/StreamProfiler/Pages/default.aspx
32. Stream KernelAnalyzer. http://developer.amd.com/gpu/ska/pages/default.aspx