Accelerating complex brain-model simulations on GPU platforms

H.A. Du Nguyen, Zaid Al-Ars
Laboratory of Computer Engineering
Faculty of EE, Mathematics and CS
Delft University of Technology
Delft, The Netherlands
{H.A.DuNguyen, Z.Al-Ars}@tudelft.nl

Georgios Smaragdos, Christos Strydis
Neuroscience Department
Erasmus Medical Center
Rotterdam, The Netherlands
{g.smaragdos, c.strydis}@erasmusmc.nl

Abstract—**The Inferior Olive (IO) in the brain, in conjunction with the cerebellum, is responsible for crucial sensorimotor-integration functions in humans. In this paper, we simulate a computationally challenging IO neuron model consisting of three compartments per neuron in a network arrangement on GPU platforms. Several GPU platforms of the two latest NVIDIA GPU architectures (Fermi, Kepler) have been used to simulate large-scale IO-neuron networks. These networks have been ported on 4 diverse GPU platforms and implementation has been optimized, scoring 3x speedups compared to its unoptimized version. The effect of GPU L1-cache and thread block size as well as the impact of numerical precision of the application on performance have been evaluated and best configurations have been chosen. In effect, a maximum speedup of 160x has been achieved with respect to a reference CPU platform.**

I. Introduction

The human brain is a complex organ with a 100 billion neurons [1], which constantly interact with each other to create cognition and consciousness. The human brain includes plenty of interesting characteristics such as rapid processing, low energy consumption, large memory storage, etc. Scientists and engineers are attempting to understand these impressive brain characteristics for medical purposes (e.g., brain-related diseases and brain rescue [2]–[4]) and to build brain-inspired computing systems encompassing similar traits [5]. Neuroscience intends to understand brain behavior by creating simulation models replicating neuron behavior in various degrees of complexity.

Spiking-neural-network (SNN) models are one such model category. A SNN model typically simulates neural networks that include multiple interconnected neurons. Efficient simulation of such networks is the first step towards creating a functional simulation of the brain. SNN models have been well-studied and their simulation can provide significant insights on brain operation. Depending on the information needed to extract from the simulation, different SNN model complexities can be used [6].

The Inferior-Olive (IO) cell model we consider here [7] is one such SNN model. It is a mathematical representation of the IO cell which is very important to the cerebellum's operations. The IO cell provides the main input signal to the cerebellar system [8]. Hence, if successfully simulated, it can play a major role in understanding and treating motor-coordination problems caused by corruption between the cerebellum and IO signals. Simulating this model successfully can also have an impact on engineering by exploring the cerebellum's fault-tolerant mechanisms and using the human body's motor-coordination methods in state-of-the-art robotics applications.

However, there are a number of challenges such simulations face. There are two distinct tracks that such a brain simulation can follow. One track that an experiment can seek is to study the model behaviour in real-time, for example in an artificial motor-coordination experiment. Such setups require real-time performance in the computation of the model (simulated brain time equals actual computation execution time). The other track is the study of large network dynamics in sizes that are comparable to the biological systems. Even though there is no real-time constraint here, the size of the networks require significant computational power. Typical CPUs have limitations in both cases as they are unable to provide real-time performance for models as complex as the IO model, while large scale simulations could require several days or even weeks to conclude.

This paper is focused on simulations of large-scale networks and encompasses the following contributions: (i) We simulate successfully the IO-cell network, for a relatively large number of cells while providing significant performance benefits compared the CPUs execution; (ii) We present a comparison among multiple GPU platforms in order of effectiveness for simulating this model in particular, and complex neuron-model simulation in general.

This paper is organized as follows: Section II discusses related works in the field. Section III presents detailed information of the IO model in a network setting. Section IV and V discuss implementation and optimization of the model on various GPU platforms. Section VI presents the performance results of the implementation, while Section VII provides an analyses of the results. Section VIII concludes the paper.

II. Related work

Complex neuron models which carry more biological significance often require large computational resources, hence simulating activities of thousands of nerve cells (in real time or not) is inefficient even with the support of supercomputers [9], [10]. Graphics Processing Units (GPUs) are a promising platform efficient neural-network simulations [11]. Using a simplified Izhikevich model of a random network with a 100,000 neurons, a GPU-based implementation has been reported to achieve 26x speed-up over a CPU-based one [12]. With a more complex model, e.g., a simple compartment HH model, the achieved speed-up came up to 857.3x for NVIDIA's Tesla C2050 over the serial implementation on the Intel CPU Core 2 Quad [13].

The model simulated in this paper is a model of the IO as developed in [7]. The C-language simulation of the IO

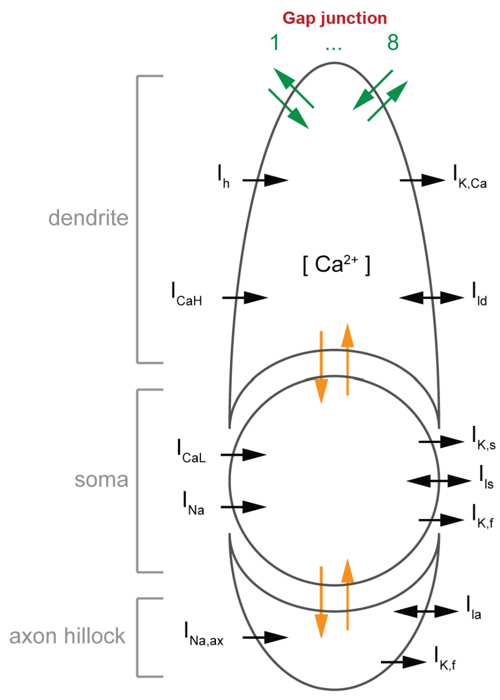

Fig. 1: Three-compartment dynamics of the IO cell [7]

model was ported by the Neuroscience dept., Erasmus MC, Netherlands and was simulated on CPU platforms with very low speed due to its high complexity. In order to achieve real-time simulation, a FPGA-based implementation [14] of the model was carried out. However, the requirement for performing all model operations in floating-point format has impacted FPGA area utilization resulting in limited network sizes for both real-time (96 cells) and the maximum networks size (14,440 cells) simulations. The intrinsic capability for floating-point arithmetic of the GPUs as well as the highly regular structure of the modeled IO network (to be discussed later) hint towards the use of a GPU platform for accelerating such model simulations. Additional advantages of the GPU platforms are their accessibility and availability provided by the relatively low cost of GPUs, multitude of different hardware options and the mostly open libraries that are used in GPU programming.

For scientific applications this is specifically important as it allows for easy and quick validation of scientific conclusions and results by multiple stakeholders. Due to the novelty and complexity of this model, there has not been any existing simulation of the model on a GPU platform in the past. In this paper, we present the implementation results of this model on various GPU platforms as well as a detailed performance comparison among them.

III. The Inferior-Olive cell model

In this IO model, the Hodgkin-Huxley paradigm is applied on three compartments: a *dendrite*, a *soma* and an *axon hillock*, as shown in Figure 1. Each compartment is modelled based on three parameters of conductance: leakage, sodium and potassium conductance. Each conductance depends on a number of current parameters. Inside an IO cell, the three compartments interact with each other, while multiple IO cells interact with each other through bi-directional electrical channels, called gap junctions.

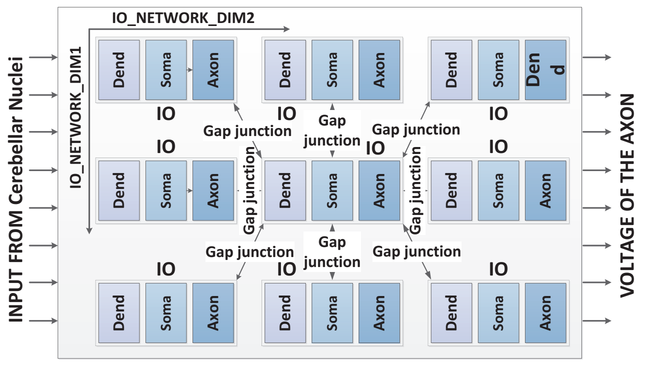

Fig. 2: The network of IO cells

Multiple IO cells are connected to create an IO network. In order to simplify the calculation on the IO network, the network is considered as a 2D mesh [4]. The calculation of each IO cell is involved in computing the three compartment properties such as currents and voltages. Figure 2 is an illustration of the IO network. An external input current (I_app) is fed into the dendritic compartment of every IO cell, which represents the input from the deep Cerebellar Nuclei (CN). The interconnection among cells in an IO network occurs via gap junctions interconnecting each cell with its 8 immediate neighbors (in this particular model).

The interconnection among cells is represented by collecting all the dendritic voltages of neighbor cells to compute the dendritic voltage of each cell. The modeling equation for the gap junction is discussed in [7]. As the IO network is a 2D cell mesh, cells at the corners have only 3 neighbors, and other cells at the borders have 5 neighbors. Though this interconnection pattern appears to be simple, this pattern was used in many meaningful experiments [4].

IV. Implementation of the model

The program in Figure 3a starts by initializing all cell parameters with a predefined value. After initializing all cell states and feeding inputs, three for-loops are employed to compute the model of the cells one by one and update new states at each time-step. The C implementation in Figure 3a includes one loop for 120,000 time-steps (in this particular modeling experiment) and two loops of visiting all elements of the network rows and columns. The compute-intensive part of the program is located within the two inner loops of simulation. A function in the third loop computes all cell parameters such as dendrite, soma and axon voltages.

Thus, the CUDA implementation (as shown in Figure 3b) focuses on resolving this critical part. Firstly, the two inner loops (2^{nd} and 3^{rd} loop) are mapped onto a 2-dimensional grid of CUDA threads. With this setting, every CUDA thread corresponds to an IO cell. Each thread computes parameters of the dendrite, soma and axon for every time step. Each time step corresponds to one iteration of the 1^{st} loop. The three compartments interact tightly with each other as the parameters of one compartment are used to compute other compartment parameters in the next iteration. Inside one iteration, the

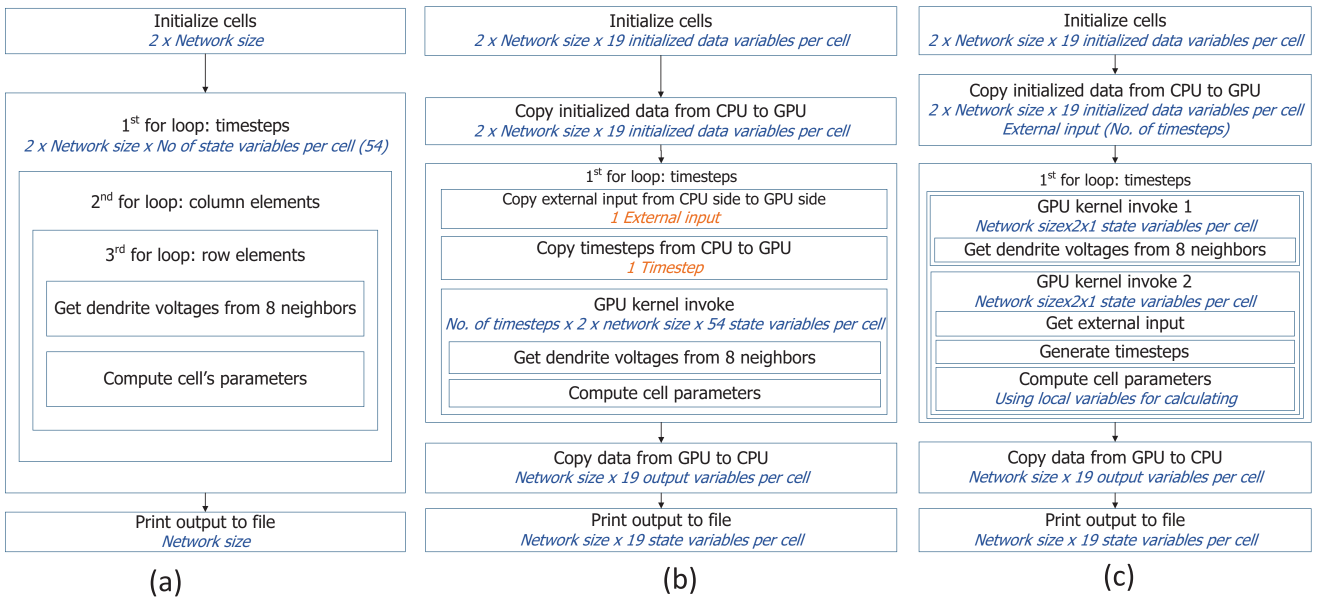
Initialize cells
2 x Network size
1st for loop: timesteps
2 x Network size x No of state variables per cell (54)
2nd for loop: column elements
3rd for loop: row elements
Get dendrite voltages from 8 neighbors
Compute cell's parameters
Print output to file
Network size
(a)
Initialize cells
2 x Network size x 19 initialized data variables per cell
Copy initialized data from CPU to GPU
2 x Network size x 19 initialized data variables per cell
1st for loop: timesteps
Copy external input from CPU side to GPU side
1 External input
Copy timesteps from CPU to GPU
1 Timestep
GPU kernel invoke
No. of timesteps x 2 x network size x 54 state variables per cell
Get dendrite voltages from 8 neighbors
Compute cell parameters
Copy data from GPU to CPU
Network size x 19 output variables per cell
Print output to file
Network size x 19 state variables per cell
(b)
Initialize cells
2 x Network size x 19 initialized data variables per cell
Copy initialized data from CPU to GPU
2 x Network size x 19 initialized data variables per cell
External input (No. of timesteps)
1st for loop: timesteps
GPU kernel invoke 1
Network sizex2x1 state variables per cell
Get dendrite voltages from 8 neighbors
GPU kernel invoke 2
Network sizex2x1 state variables per cell
Get external input
Generate timesteps
Compute cell parameters
Using local variables for calculating
Copy data from GPU to CPU
Network size x 19 output variables per cell
Print output to file
Network size x 19 state variables per cell
(c)

Fig. 3: Original implementation: (a) C implementation; (b) Un-optimized CUDA implementation; (c) Optimized CUDA implementation

three compartment parameters can be computed in parallel. The synchronization across steps is carried out on the CPU side (host) while computing cell parameters at each step is performed on the GPU side (device).

This implementation reveals a memory bottleneck. As the input size scales with the number of simulated neuron cells, the device (GPU) memory is not sufficient for a large input size. In addition, the bigger the amount of memory used, the more time-consuming memory fetches becomes. Hence, various optimizations should be applied on the implementation to reduce memory usage and access time.

V. Optimization of CUDA implementation

Figure 3c shows the optimizations applied to the CUDA implementation as compared to Figure 3b. We will go through the various optimizations in detail next.

A. *Coalesced global-memory accesses*

The external input current I_app is required for each time step, and therefore, it is transferred from the CPU to the GPU at the beginning of each time step. Since – for this experiment – all cells in our network receive a fixed input current at each time step, we transfer the external inputs for all time steps as a single fixed array in the improved implementation. Each external input value corresponds to a *#timesteps*. As the whole array of size *#timesteps* is transferred, this host-to-device communication can be moved outside of the first loop as shown in Figure 3c, hence reduce vastly the transfer time, avoid unnecessary data-transfer overhead at every time step and also help increase the global memory coalescing. The index of the time step is generated as a global variable by a specific thread on the device side. This global variable is updated by only one thread at the end of each iteration to make sure that the index is increased by one with to each iteration. This index is used to determine the proper access for the external input values.

The *global memory* – used to store cell states – is reorganized into an array of multiple sets of 27 double variables (needed to be stored per cell). A set of 27 double variables includes 19 state variables, together with 8 variables reserved for 8 values originating from a cell's neighbors. The number of needed variables was reduced by removing the variables that are used for storing cell states. Indeed, only the neighbor's dendrite voltage should be stored across the iterations. Hence, a half of the variables can be eliminated. With this organization of data, the kernel can be split into two separate kernels (shown as GPU kernel invoke 1 and 2 in Figure 3c) and allows explicit synchronization (between internal cell parameters and neighbor cell voltages) through global variables without affecting the robustness of the program. This optimization also permits a part of the state variables to be defined as local variables. Furthermore, the local memory is handled by the compiler, hence more systematic optimization is done automatically by the compiler such as coalescing, dynamic allocation, etc.

B. *Eliminating branching divergence*

Another optimization is to use the *texture memory*. The array of neighbor dendrite voltages only changes across time steps (i.e., the array is considered as read-only in one time step). In addition, the kernel is reloaded on every time step, along with the contents of the texture memory. These conditions allow the texture memory to be used to load the neighbor dendrite voltages. The texture memory is useful in this situation because it is cached by the geographic allocation of memory instead of by standard cache lines (as shown in Figure 4). For the implementation of this IO network, every cell needs its 8 neighbor voltages which fit the caching style of the texture memory. Therefore, the texture memory is very effective in reducing the memory access time.

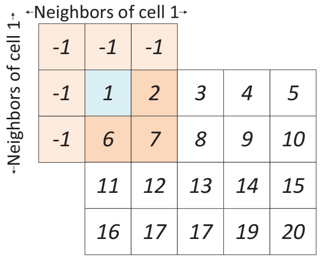

Fig. 4: Texture memory helps eliminate border conditions

Furthermore, it helps remove the branch divergence. As explained above, each cell is connected to eight neighbors. In this implementation, the connection is represented by getting dendrite voltages of neighbors stored in a specific array. To ease the task of filling those values in an array, the number of neighbors should be all equal to each other. In the original source code, each cell needs to check if it lies at the border. In that case, some neighbor values will be missing, hence it has to take its own dendrite voltage value instead of the neighbor voltage values. For example, in Figure 4, the cell number 1 has to take 3 neighbor voltage values from cell 2, 6, and 7. As it is at the border, the voltage values of the neighbors marked -1 are filled by the cell's dendrite voltage value. Using this scheme, all the cells have eight neighbor voltage values and avoid the complexity of dealing with different numbers of neighbors.

VI. Experimental Setup and Results

In this section, the experimental framework, the exploration dimensions, and various measured results will be presented.

A. Simulation environment

A CPU platform (Intel Core i5-2450M (2.5 GHz) with 4 GBs of RAM) is used as the baseline platform to run the simulation of the sequential implementation. The performance of the implementation is used to compare with that of the parallel implementation on GPU platforms. The GPU platforms chosen to investigate simulation speed-ups are four NVIDIA GPU platforms: Tesla K20c (Kepler) [15], Tesla C2075 (Fermi) [16], GeForce GTX480 (Fermi) [17] and GeForce GT640 (Kepler) [18]. These platforms represent two NVIDIA families in which Kepler has higher processing capabilities and energy efficiency than the Fermi architecture [19]. The IO network model is ported onto each GPU both in single-floating-point and in double-floating-point precision (referred to as single- and double-precision simulation, respectively).

Two of our exploration dimensions are the L1 cache usage and the optimal *thread block size*, which represents partly the GPU processing capability. The thread block size is the number of threads which are processed on the same GPU streaming-processor. According to the GPU specifications, it should be a multiple of 32 for best benefits in memory transfers and fine-grain execution of all the threads [19]. The idea behind choosing the thread block size is to maximize the occupancy of the GPU processors when executing the application. The bigger the thread block size is, the higher occupancy the application should get. Block size of 16 is chosen to represent an inefficient block size.

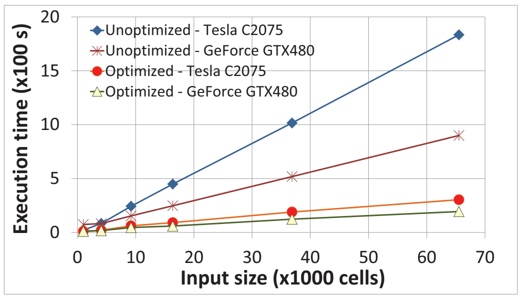

Fig. 5: Performance comparison between unoptimized vs optimized implementation (Fermi platforms - Block size 64)

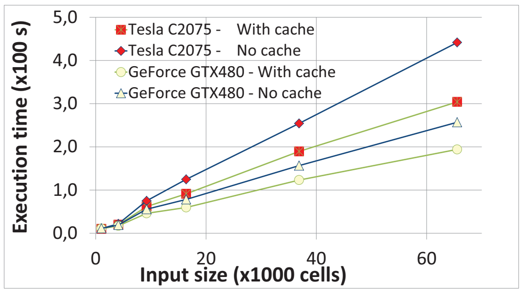

Fig. 6: Cache usage performance on Fermi architecture with double precision and 64 threads/block

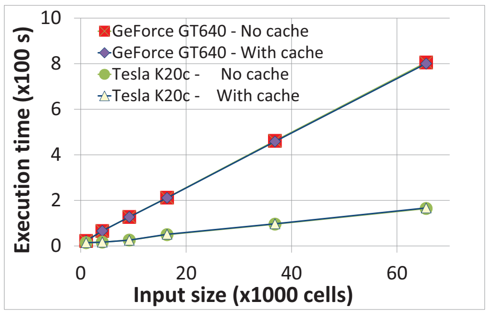

Fig. 7: Cache usage performance on Kepler architecture with double precision and 64 threads/block

B. Performance comparison

1) Comparison with unoptimized implementation: In order to evaluate the optimization methods used in this paper, we simulated the unoptimized implementation on the Fermi platforms (Tesla C2075 and GeForce GTX480). Figure 5 reveals at least a 3x speedup of the optimized implementation as compared to the unoptimized one. The speed-up in comparison with CPU-based implementation reaches 9.2 times for the Tesla C2075 and 10.5 times for the GeForce GTX480. Apart from the high execution time, the unoptimized implementation also has drawbacks in term of high memory usage due to the large number of variables per cell. Hence, the simulation only facilitates small input sizes.

2) L1 cache impact: The L1-cache usage might improve application performance if data can be cached efficiently. In this paper, we verify the L1-cache impact on application performance on two different architectures.

On the Fermi architectures (Tesla C2075 and GeForce GTX480), the execution time of the simulation without L1-

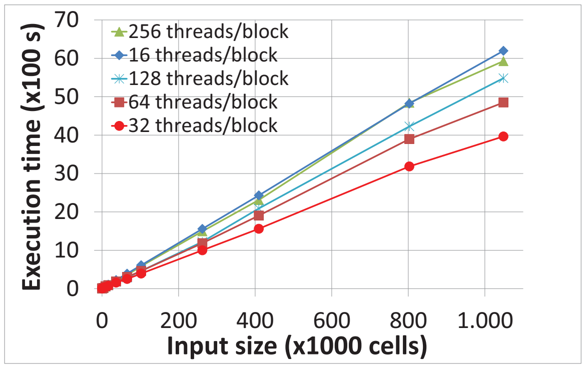

Fig. 8: Execution-time comparison for different thread block sizes (double precision with cache on Tesla C2075)

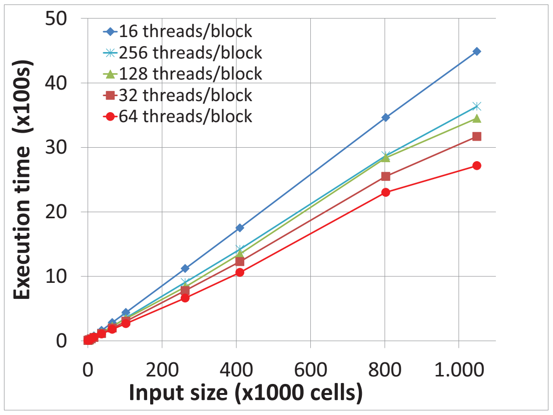

Fig. 9: Execution-time comparison for different thread block sizes (single precision simulation with cache on Tesla C2075)

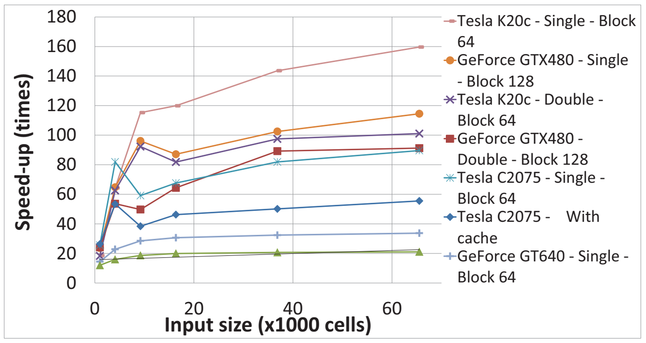

Fig. 10: Performance comparison among all 4 GPU platforms

cache usage is much higher than that with L1-cache usage (as shown in Figure 6). The performance improvement with L1 cache partly shows that the global-memory data distribution has spatial locality. If the data in the global memory is scattered, L1 cache cannot help increase the application performance.

On the Kepler architectures (Tesla K20c and GeForce GT640), application performance appears to remain unaffected by cache presence (as shown in Figure 7). The reason is that L1 cache in Kepler architectures is reserved for local memory accesses, such as register spills and stack data. The global-memory accesses are only cached in L2 cache or read-only data cache. As the implementation relies significantly on the global-memory access time, the configuration with maximum L1 cache cannot improve performance.

L2 cache and read-only data cache are controlled implicitly by the compiler, hence, the impact of those caches was not studied in this paper.

3) Thread-block-size optimization: We have also optimized the *thread block size* on the GPU platforms to achieve higher performance, which affected application performance significantly (as shown in Figure 8 and 9).

With double precision, the thread block size of 32 has the lowest execution time. With the single precision, the same memory throughput can facilitate double amount of data because one single precision variable (of type float) needs only 4 bytes of data instead of 8 bytes of data needed by one double precision variable (of type double). In addition, the instruction throughput for single precision operations is also increased since it requires only half of the computing units of the double precision. Hence, the block size can increase up to 64 to get the best acquired performance of the application with single precision (as shown in Figure 9).

With a small input size, the difference in performance for different thread block sizes is small. The difference becomes more dramatic when increasing the input size. The results for larger input sizes reflect the GPU processor occupancy. The GPU processors are occupied less for small input sizes since the number of inputs is not large enough to fill in all GPU processors. When increasing the input size, more and more workloads are available to distribute to all GPU processors. Therefore, the GPU processor occupancy increases linearly. Once all the GPU processors are occupied, the performance of the application reaches a saturation point and stays constant for all bigger input sizes.

4) Speed-up comparison: Figure 10 shows a comparison of maximum speed-ups achieved by the four platforms with respect to the reference CPU platform. The figure shows that speed-up is low for small input sizes and increases with increasing input size until it reaches a saturation point beyond the input size of 40,000 cells (except for Tesla K20c, which has the saturation point at a larger input size and was not shown in this figure). After reaching the saturation point, speed-up on different platforms stays constant. As expected previously, the performance of the single precision is always better than that of the double precision on the same platform. The Tesla K20c achieves the best performance for both single and double precision. The GeForce GTX480 performs the simulation significantly faster than the Tesla C2075. Even the double precision performance on the GeForce GTX480 is better than the single-precision performance on Tesla C2075. GeForce GT640 scores the worst performance, however, it still achieves speed-up (up to 20x) in comparison with the CPU platform.

5) Maximum achievable network size: Table I shows the maximum network size that can be simulated on each platform. This number depends on the global memory size on the GPU side. The optimized implementation achieves 3.3x larger network size than the unoptimized one by optimizing the number of required variables. Thus, the optimized version is

TABLE I: Maxium achievable network size (cells) on different platforms of unoptimized and optimized implementation

	K20c	C2075	GT640	GTX480
Unoptimized	7,225,344	5,760,000	2,876,416	2,166,784
Optimized	23,658,496	18,939,904	9,437,184	7,054,336

able to simulate a realistic human IO network size that is approximately 1,000,000 cells. The number of simulated cells in all the GPU platforms is much higher than any biological IO network such as in rat (53,000 cells), chicken (21,600 cells), or cat (146,000 cells) [20].In addition, the FPGA platform can simulate only 14,400 cells [14] that is not large enough for any biological case. Hence, GPUs are shown to be promising for simulating the human-brain neuronal network.

VII. Discussion

The purpose of executing simulation on the GeForce GT640 and GeForce GTX480 is to evaluate the possibility of using a low-cost graphics card for a high-performance computing application. Consider, that the maximum single-precision simulation speed-up of the Tesla K20c platform and of the GeForce GT640 platform is 159x and 34x, respectively. However, the Tesla K20c card costs 2,700 US dollars, while the GeForce GT640 costs only 79 US dollars. Therefore, the cost-per-unit of the speed-up on the two platforms is 17 and 2.3 (US dollars/unit), respectively. Hence, the *cost efficiency* of the Tesla K20c platform is 7.3 times less than the GeForce GT640 platform. Calculation for double precision revealed a similar result. Due to the correlation in the application data across iterations, the application is difficult to be split to map on multiple platforms. Considering that fact, even with multiple separate GeForce GT640 platforms, we might not achieve the performance that can be achieved on the other platforms. However, the GeForce GT640 and GTX480 platforms can offer a cheap and flexible solution in order to reduce the simulation time of neuro-modelling experiments.

VIII. Conclusions

In this paper, the implementation of an Inferior-Olive network application was explored on multiple GPU platforms. Our implementation was simulated on two NVIDIA GPU Fermi platforms (Tesla K20c, Tesla C2075) and two Kepler platforms (GeForce GTX480, GeForce GT640). On all considered platforms, our implementation achieved high speed-ups over the serial implementation executed on the CPU platform, ranging from 10x to 160x for the various GPUs evaluated. A large number of IO cells (up to 23,658,496) can be simulated on all experiment platforms. The achieved speed-ups relied on various optimizations per platform, such as optimal thread block size and L1-cache utilization. An investigation of optimal cost efficiency indicated that the lower performance GeForce GT640 platform has a higher cost efficiency than the other platforms. Although a lot of neuron models have been implemented on GPUs, the currently ported IO model in a network setting is crucial for simulating biologically realistic brain structures. The achieved high number of cells can lead to valuable experiments for further unrevealing brain functionality as well as for discovering next-generation bio-inspired computing machines, storage devices and so on.

Acknowledgement

The authors would like to thank Kees Lemmens, from the Applied Mathematics Department and Rahul Kukreja, from the Computer Engineering laboratory, the Delft University of Technology, for their support on simulation.

References

[1] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in Human Neuroscience*, vol. 3, p. 31, 2009.

[2] B. Neild, "Scientists to simulate human brain inside a supercomputer," 2012.

[3] M. Knight, "Mapping out a new era in brain research," 2012.

[4] B. Torben-Nielsen, I. Segev, and Y. Yarom, "The generation of phase differences and frequency changes in a network model of inferior olive subthreshold oscillations," *PLoS Comput Biol*, vol. 8, no. 7, p. e1002580, 07 2012.

[5] M. Djurfeldt, M. Lundqvist, C. Johansson, M. Rehn, O. Ekeberg, and A. Lansner, "Brain-scale simulation of the neocortex on the ibm blue gene/l supercomputer," *IBM Journal of Research and Development*, vol. 52, no. 1.2, pp. 31–41, Jan 2008.

[6] E. Izhikevich, "Which model to use for cortical spiking neurons," *Neural Networks, IEEE Transactions on*, vol. 15, no. 5, pp. 1063 – 1070, sept. 2004.

[7] J. R. De Gruijl, P. Bazzigaluppi, M. T. G. de Jeu, and C. I. De Zeeuw, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," *PLoS Comput Biol*, vol. 8, no. 12, p. e1002814, 2012.

[8] D. Purves, D. Fitzpatrick, L. Katz, A. Lamantia, J. McNamara, S. Williams, and G. Augustine, *Neuroscience*. Sinauer Associates, 2001.

[9] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: Cortical simulations with 109 neurons, 1013 synapses," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 63:1–63:12.

[10] EPFL, "The blue brain project," 2011.

[11] T. Nowotny, "Flexible neuronal network simulation framework using code generation for nvidia cuda," *BMC Neuroscience*, vol. 12, no. 1, pp. 1–2, 2011.

[12] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," in *Proceedings of the 2009 International Joint Conference on Neural Networks*, ser. IJCNN'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 3201–3208.

[13] V. Pallipuram, M. Bhuiyan, and M. Smith, "Evaluation of gpu architectures using spiking neural networks," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, july 2011, pp. 93 –102.

[14] G. Smaragdos, S. Isaza, M. F. van Eijk, I. Sourdis, and C. Strydis, "Fpga-based biophysically-meaningful modeling of olivocerebellar neurons," in *FPGA*, 2014, pp. 89–98.

[15] NVIDIA, "Tesla k20 gpu active accelerator," 2014.

[16] NVIDIA, "Nvidia tesla c2075 companion processor," 2014.

[17] NVIDIA, "Geforce gtx480 specification," 2014.

[18] NVIDIA, "Geforce gtx 640 specification," 2014.

[19] NVIDIA, "Tuning cuda applications for kepler," July 2013.

[20] M. E. Scheibel and A. B. Scheibel, "The inferior olive. a golgi study," *The Journal of Comparative Neurology*, vol. 102, no. 1, pp. 77–131, 1955.