Using Parallel GPU Architecture for Simulation of Planar I/F Networks

Jan-Phillip Tiesel and Anthony S. Maida

Abstract— **Our work describes the simulation of a planar network of spiking I/F neurons on graphics processing hardware. The described approach adds to the fast-growing field of general-purpose computation on GPUs (GPGPU). We provide an in-depth explanation of the steps involved in implementing the network using programmable shading hardware. We replicated simulation results by Hopfield et al. [1] and Maida et al. [2] and give qualitative and quantitative measures of our implementation.**

I. Introduction

MODERN day graphics processing units (GPU) exploit the inherently parallelizable nature of the standard graphics pipeline [3] by employing massive multi-threading and accumulating dozens or hundreds of ALUs in the processing hardware. In recent years, the rapid market growth in high-performance graphics cards has led to an exponential increase in their computing power; in terms of floating point operations per second, they now outperform many multi-core CPUs.

Although GPUs are optimized for efficient implementation of the specialized graphics pipeline model, we can use them for parallel computational tasks other than real-time rendering of 3D graphics. In fact, many general-purpose computations which can be (partially) performed in parallel may be processed on GPU architecture with significant gains in execution speed.

We investigate the simulation of a locally connected, planar network of computational units on GPU architecture. The individual units implement the leaky integrate-and-fire (I/F) neuron model described by Hopfield and Herz [1]. As our work focuses on a practical implementation and evaluation of the framework, we establish our simulations using some well-known parameters for both network topology and numerical integration techniques as described in [2].

Simulations were performed using two different graphics cards and a conventional CPU implementation in MatLab. While all network simulations converged to oscillating global firing, we identified discrepancies in the observed network behavior which we attribute to precision differences between the employed GPUs. In addition, the presented mechanism for instantaneous propagation of action potentials was found to be a performance bottleneck and requires further investigation. Results regarding performance and accuracy are given in section IV.

Jan-Phillip Tiesel and Anthony S. Maida are with the Center for Advanced Computer Studies at the University of Louisiana at Lafayette (e-mail: {jpt4246,maida}@cacs.louisiana.edu). Anthony S. Maida is also with the Institute of Cognitive Science at the University of Louisiana at Lafayette.

II. Related work

A. Simulation of spiking neurons

Many researchers have proposed using networks of spiking neurons to study the role of spike timing and synchronized firing in brain computations. Cortical neurons are known to show synchronized firing but how synchronous firing patterns emerge is not fully understood yet. Several models with widely varying biological realism exist for simulating the membrane potential and related spiking behavior of neurons. One such model that exhibits synchronous oscillations and rapid convergence is described by Hopfield and Herz [1]. It uses integrate-and-fire (I/F) neurons and assumes instantaneous propagation of action potentials across the network. Maida et al. [2] showed that the degree of synchrony decreases steadily when propagation delays are introduced (which is more biologically plausible since the transmission speed of the axon is limited).

B. General-purpose computation on the GPU

Many recent publications discuss the applicability of the GPU architecture to general-purpose computations. Owens et al. [4] provide a thorough introduction to the GPU programming model and a good overview of algorithms and applications that have been adapted to work on the GPU. Meuth and Wunsch [5] give some examples of neural computation problems that have been addressed by GPU implementations. Luo et al. [6] present design rules and performance improvements for two different kinds of artificial neural networks – self-organizing maps and multi layer perceptron – when implemented on graphics processing architecture. A planar grid of spiking I/F neurons was simulated on the GPU for image segmentation tasks by Bernhard and Keriven [7]. Our work extends their implementation techniques for global inhibition in order to address the problem of global instantaneous propagation of spikes.

Our main contributions are

- detailed description of a framework for simulating synchronizing I/F networks on GPU architecture,
- validation and evaluation of our simulation by comparison to results acquired by Maida et al. [2].

We show that the specific type of network described in detail in [2] can be simulated on GPU hardware and that high computational accuracy can be achieved using state-of-the-art graphics boards. In addition, we point out some limitations of our approach and make suggestions on further investigations of simulating networks of spiking neurons on GPU architecture.

978-1-4244-3553-1/09/$25.00 ©2009 IEEE

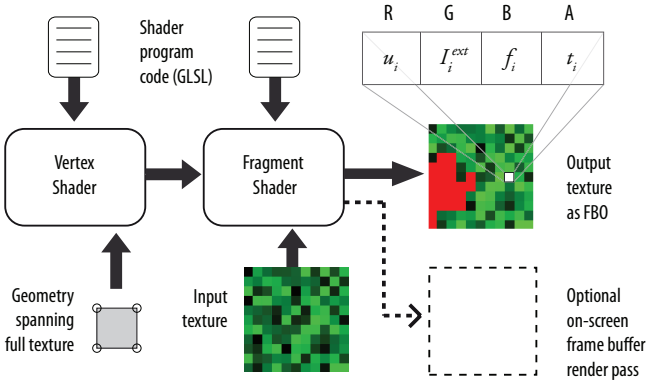

Fig. 1. Overview of components of the GPU pipeline. Additional data structures are shown to depict the way a single update pass over the neuronal grid is rendered.

III. Methods

A. Simulation

Our network model follows closely the instantaneous propagation model as described by Hopfield and Herz [1] and subsequently by Maida et al. [2]. A grid of 40x40 I/F neurons is simulated for comparative results while square grids of arbitrary size are possible (limited by the maximum texture size supported by the graphics card installed on the host machine – typical values on modern graphics accelerators are 2048 / 4096). All neurons have local connections to their four closest neighbors (north, east, south, west); each of the connections has a weight of 0.24. As we are only interested in observing spiking behavior of the network, the connection weights remain fixed throughout the simulation; no training of the network is performed. Both periodic and open boundary conditions can be established. Instead of restating the dynamics of the membrane potential and the rules of the instantaneous spike transmission, we refer to the detailed description given in [2]. Our simulation follows their instantaneous propagation model exactly unless stated otherwise.

Our simulation is deterministic except for the random initialization of membrane potentials. Identical random number sequences were used for all comparisons.

We used two different graphics cards to perform the simulations: the first card – ATI FireGL V5100 – has been on the market for several years and provides 12 parallel fragment processing pipelines. The second unit – NVIDIA GeForce 9800 GX2 – is a latest generation board containing two GPUs providing a total of 256 parallel stream processors.

B. Data structures

Large data structures accessible to the shading units of the graphics card have to be stored as textures. The textures used for our purposes are square arrays of vectors with four components. The vector components are used in the context of graphics synthesis to represent the red, green, and blue components of a certain color along with an alpha value used for color blending. In our implementation, the components – which are floating point scalars with their precision depending on the GPU architecture – fully represent the state of a single neuron:

$$\vec{x_i} = (R_i, G_i, B_i, A_i) = (u_i, I_i^{ext}, f_i, t_i) \qquad (1)$$

where u_i denotes the membrane potential of unit i, I_i^{ext} is the external input to the unit, f_i is its instantaneous action potential (which can be either 0 or 1), and t_i is the remaining time before an action potential is propagated. We have not implemented delayed propagation but mention it here to hint at the extensibility of our approach. While Bernhard and Keriven [7] use multiple textures to store information about the network (as they also store connectivity which is local and uniform in our network), we decided to use a single texture to hold the complete state of the network.

Two textures of identical size are needed for our implementation: one serves as input to the program executed on the GPU's shading units, the other one is used for writing. This is necessary because of a major difference between CPU and GPU architectures described in the next section.

C. Gather and scatter operations on the GPU

An important and critical limitation that allows the heavy parallelization of computation on GPUs is its missing *scatter* property. Shading units can concurrently read from multiple input data structures (*gather*), but may only write to a single output structure: the frame buffer (which may actually consist of multiple buffers such as stencil, depth, or color buffer; we will only use the color buffer for our purposes). The contents of the frame buffer are eventually displayed on the screen once rendering of a frame is complete.

Instead of writing output to the on-screen frame buffer, we attach the texture currently selected to be the output buffer as a frame buffer object (FBO) using an OpenGL extension available for that purpose. This lets us write neuron state vectors to the output texture instead of the screen's frame buffer. While we can update each neurons' membrane potential per cycle this way, we will not be able to propagate action potentials across the grid in one render pass.

D. Implementation

We implemented the simulation as a C++ application using the OpenGL graphics API, the cross-platform windowing toolkit GLUT, and the OpenGL extension manager GLEW. The actual computation performed on the GPU is described as a shader program written in OpenGL Shading Language (GLSL), consisting of two code artifacts – one program to process incoming vertices in the vertex shading unit and a second one that is executed for each fragment about to be written to the frame buffer (compare Figure 1 for an overview of the involved elements).

After creating a texture data structure holding the initial state of the network (as described in section III-B), its data is copied to a texture in GPU memory and is now accessible by our shader program. As we want internal state updates of individual neurons to be computed by the GPU, we have to make sure that each neuron in the planar grid maps to

one fragment being processed in the fragment shading unit. We accomplish this by setting up the projection and viewing matrix of the graphics pipeline in a way that if we initiate rendering properly, each fragment shader program execution reads the state vector $\vec{x_i}(t)$ of a single neuron i from the input texture, performs a state update, and writes the new state $\vec{x_i}(t+1)$ to the correct position in the output texture.

The actual geometry information we send to the GPU describes a simple quadrilateral with position and texture coordinates of its four vertices as shown in Figure 2. This geometry description is compiled to a display list and thereby cached in GPU memory which reduces CPU workload per iteration.

In addition, any optional multi-sampling techniques of the graphics card have to be deactivated. These techniques – e.g. full-scene antialiasing (FSAA) – evaluate fragment color values on a subpixel level and are therefore not only irrelevant in our context but would distort the computed neuron states.

A single update step then consists of $w*w = n$ executions of our fragment shader program, where w is the side length of the grid and n is the total number of simulated neurons. After each step, we swap the role of the input and output buffer so that the texture containing the computed update is used to read the neurons' states in the next iteration. More details on the employed techniques can be found in the excellent GPGPU tutorials by Dominik Göddeke [8].

Once a neuron's membrane potential reaches the fixed threshold $u_{th} = 1$, the shader program will set the action potential of the neuron to 1 and the progression of time has to be suspended until this and all other remaining action potentials are propagated across the entire network. Due to the lack of *scatter* operations during the execution of the fragment shader, we can not achieve instantaneous propagation in a single render pass. While it is possible for the fragment shader program to update the neuron's state according to the action potentials of its neighbors, it may not alter the state of adjacent neurons in case the neuron fires. We therefore split the problem up into multiple render passes:

1) Update membrane potentials, if $u_i > u_{th}$ set f_i to 1 (Pass 1)
2) Detect number of action potentials in network n_f (Pass 2a – indication)

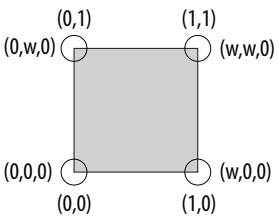

Fig. 2. Quadrilateral geometry rendered per pass to achieve one-time execution of fragment shader program for each neuron. w denotes the width of the square planar grid. Vectors shown denote position (x, y, z) and texture coordinates (u, v) of the respective vertices.

3) Copy states from input to output texture if $n_f > 0$ (Pass 2b)
4) Propagate action potentials if $n_f > 0$, otherwise exit (Pass 3 – propagation)
5) Detect number of action potentials in network n_f (Pass 2a – indication), if $n_f > 0$ continue with 4.

We detect the number of action potentials by using a function provided by modern graphics cards called *occlusion query*. This query is generally used as a technique for object visibility determination. It lets us set a start and an end point for the query and returns the total number of fragments that passed the fragment shader. This is defined as fragments for which the `discard` keyword was not encountered during execution of the fragment shader program. A fragment that is discarded does not cause a write operation to the attached frame buffer. The shader implementation for Pass 2a therefore only passes fragments for which $f_i = 1$. The result of the occlusion query is therefore the number of neurons which have an action potential that still needs to be propagated across the network.

The actual spike propagation is done in Pass 3: every neuron examines the firing status of their connected neighbors and updates its membrane potential, if necessary. In case this drives the potential above threshold, f_i is set to 1 and the next execution of Pass 2a will indicate more action potentials to be propagated. We continue this iteration until all spikes are propagated across the grid. For the sake of clarity, additional pseudo-code for the described steps is given below.

```
// Update membrane potentials
renderPass_1()
swapBuffers()
// Are any neurons firing?
firingNeurons = renderPass_2a()
// If so, copy input to output texture
// to match network state at t+1
if (firingNeurons > 0)
  renderPass_2b()
totalNeuronsFired = 0
// Propagate action potentials
while (firingNeurons > 0)
  totalNeuronsFired += firingNeurons
  renderPass_3()
  swapBuffers()
  firingNeurons = renderPass_2a()
```

Our implementation contains three test cases: simulation of a single I/F neuron for precision evaluation and two different simulations of the planar 40x40 grid as described in [1], [2]. One simulation does not use a regular frame buffer to visually communicate the current network state to the user. This provides a performance increase as additional passes for rendering the network state to the on-screen frame buffer can be avoided. Optionally, an additional render pass is added to the 40x40 grid simulation to show the state of the network in real-time. Figure 5 shows a sequence of screenshots of our application. For improved performance, we do not update the on-screen frame buffer at every time step, but after a fixed step interval (as the number of update iterations per second might be as high as 5000).

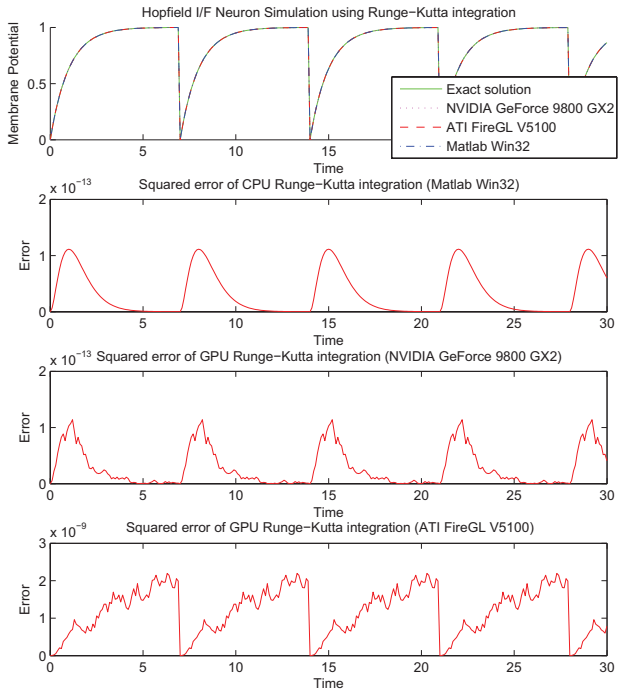

Fig. 3. Comparison of membrane potential integration error for three different architectures ($\Delta t = 0.1$).

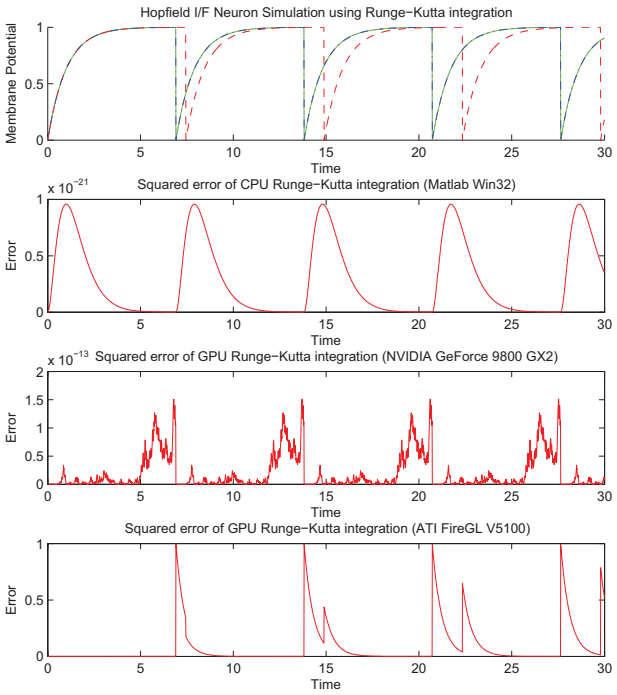

Fig. 4. Comparison of membrane potential integration error for three different architectures ($\Delta t = 0.01$).

IV. Results

A. Runge-Kutta integration precision

We performed a precision test using fourth-order Runge-Kutta integration of the differential equation describing the membrane potential of an integrate-and-fire neuron

$$\frac{du_i}{dt} = -\alpha u_i(t) + I^{ext}(t).$$

Assuming a constant value for I^{ext} and a known initial membrane potential $u(t_0)$, we can find a closed-form solution to express the membrane potential at time t as

$$u(t) = \frac{I^{ext}}{\alpha} + (u(t_0) - \frac{I^{ext}}{\alpha})e^{-\alpha t}. \qquad (2)$$

The precision test was performed initially in MatLab to compare the Runge-Kutta results to the exact solution. In a second step, the GPU shader program implementation of the Runge-Kutta integration was used to determine the precision we can expect for the simulation of an I/F neuron on the GPU. Simulation parameters were $\alpha = 1.0, I^{ext} = 1.001$. $\Delta t = 0.1$ was used for results shown in Figure 3. We computed error in Matlab as the squared difference between the exact solution obtained using equation (2) and the results of numerical integration performed on the respective architecture. The test was repeated on the three different architectures using $\Delta t = 0.01$. Figure 4 gives plots of the second simulation.

Our results show that the number of integration steps before the simulated units reach threshold is identical across architectures for the relatively large time step $\Delta t = 0.1$. The maximum squared error of the GPU solution compared to the reference value is in the order of $2\text{x}10^{-9}$. The difference in error magnitude between the two tested GPUs results from the respective floating point precisions used. While the latest generation NVIDIA card offers 128-bit floating point accuracy, the older ATI card provides only 32-bit single precision accuracy per RGBA component.

If the size of the time step is decreased to $\Delta t = 0.01$, the importance of this difference becomes more apparent: the spike times of the units are now not synchronized across the test cases anymore. As the membrane potential nears saturation, the diminishing increment per time unit (depicted by the flattening part of the I/F membrane potential curve) causes an underflow effect if insufficient accuracy in the

Fig. 5. Sequence of (non-adjacent) network states showing the characteristic growth of synchronously spiking regions. Red squares indicate spiking neurons, values between black and saturated green indicate increasing membrane potentials. The color mappings were chosen for visual clarity and differ from the ones given in equation (1).

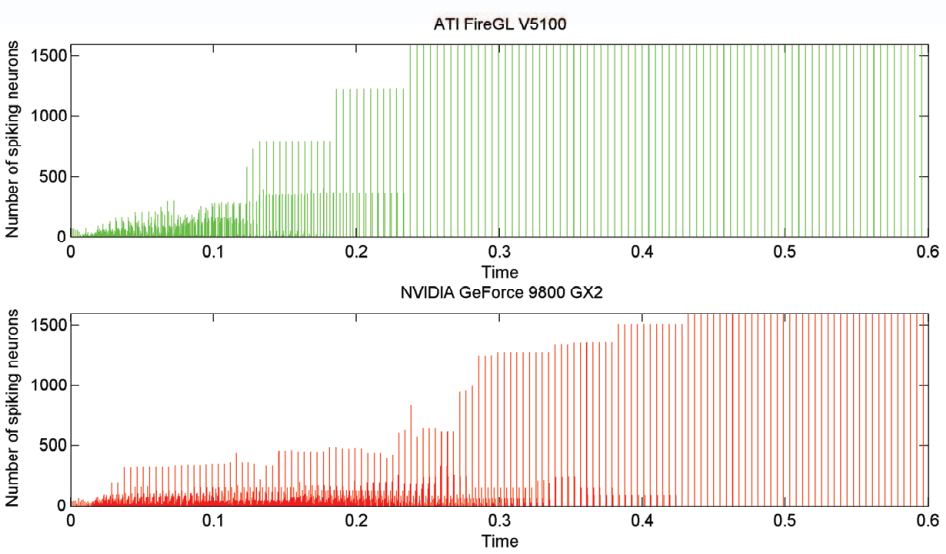

Fig. 6. Comparison of spiking behavior between simulations on two different GPU architectures.

floating point representation is used. Running the simulation with $\Delta t = 0.001$, the membrane potential never reached threshold on the ATI card.

B. Network simulation

In our network simulation a higher value of $I^{ext} = 10$ was chosen to be able to compare results to Maida et al. paper. We used periodic boundary conditions for the simulations although our implementation allows for open boundary conditions, as well. Initially, all membrane potentials were set to a random value $0 \leq u_i \leq 1$. In all of our simulations, the network converged to global synchronous spiking with a fixed period between instants where global firing occurs. As expected, this period differed between the two graphics cards due to different precision capabilities as described in the previous section.

The predicted period P_{pred} for given simulation parameters can be calculated precisely [2]; in our case, it is given by $P_{pred} = 0.00443$. We compare this value to the observed period on the two GPU architectures by calculating

$$P_{obs} = n\Delta t.$$

Using $\Delta t = 0.00001$, our simulation yielded $n = 477$ for the ATI FireGL V5100 and $n = 444$ for the NVIDIA 9800 GX2. Therefore, $P_{obs} = 0.00444$ for the simulation on the NVIDIA card, which does not exactly match the predicted value. We were not able to determine the cause of this inconsistency as the provided computational precision seems sufficient on the NVIDIA card.

The time until convergence was $\approx 25,000\Delta t$ on the ATI card (which is significantly less than the result given for the CPU implementation in [2]). It took $\approx 45,000\Delta t$ on the NVIDIA card for the network to synchronize. This number matches the result given by Maida et al. for their simulation. Figure 6 plots the number of synchronously firing neurons in the network over time. It can easily be seen that the simulation on the ATI card reaches global spiking earlier than the second simulation. This deviation can be attributed to the difference in floating point precision between the two cards as mentioned in section IV-A.

C. Bottleneck

Further performance measurements and run-time profiling showed that the major bottleneck of the described system is the overhead caused by additional render passes necessary to propagate action potentials across the grid. Every pass that causes new action potentials to arise (possibly that of a single neuron) also triggers at least one more render pass for the current time step. The time needed for binding the shader output to a frame buffer object and running an indication and propagation pass for any remaining action potentials in the network slows down the update rate of the simulation drastically.

The significance of the slowdown becomes apparent when comparing the network update times for a 40x40 grid without spike propagation (0.087 ms) and with full spike propagation (0.36 ms).

By contrast, the performance of the network scales very well if the propagation of action potentials is omitted and only the computation necessary for the numerical integration step is taken into account. Table I contrasts the scaling behavior of a corresponding *serial* CPU implementation with the *parallel* execution on a graphics processing unit as described previously. It states the computation time for a single iteration of the Runge-Kutta numerical integration step over the whole grid of neurons.

The computation times given in Table I indicate that for the GPU implementation, increasing the number of simulated neurons by a factor of 100 only causes a doubling (0.087 to 0.18) of the time needed to perform the numerical integration

TABLE I
PERFORMANCE RESULTS FOR CPU AND GPU ARCHITECTURE

	Simulation grid size		Factor
	40x40	400x400	
CPU (Intel Core 2 Quad 2.4GHz)	0.016 ms	2.66 ms	166.25
GPU (NVIDIA 9800 GX2)	0.087 ms	0.18 ms	2.069

step. We used a second implementation of the Runge-Kutta integration in C to execute the computation on a CPU. Here increasing the number of neurons by a factor of 100 increased the computation time by a factor of 166 (0.016 to 2.66).

V. DISCUSSION AND FURTHER RESEARCH

We showed that a planar, locally connected grid of I/F neurons can be simulated using GPU architecture and that results of the simulation are comparable to earlier simulations on conventional CPU architectures. Our results also underline the scalability of the approach with respect to parallel per-neuron computations that can be efficiently and precisely performed on conventional GPU hardware. The scaling behavior of the simulation in terms of per-neuron computations proposes experimenting with neuron models that provide more biological plausibility than the I/F neuron. One such example is the simple spiking model of Izhikevich [9], which creates much more realistic spiking and bursting behavior of cortical neurons while keeping the computational complexity low enough so that a large network of units can be simulated in real-time.

While our comparison to earlier simulation results confirmed the validity of the implementation, the slow down caused by the propagation technique used in our implementation is significant and has to be addressed in the future to allow for better performance of comparable network simulations on the GPU.

In order to make simulation results comparable to CPU simulations, additional work might investigate ways to achieve arbitrary precision for computations on graphics processing units. Göddeke et al. [10] presented first advances in this direction which suggest that this can be achieved in the future.

Recently, NVIDIA introduced a scalable parallel programming model (CUDA) that allows for flexible GPU programming while bypassing the graphics pipeline model and eliminating the need to access GPU functionality through a graphics API like OpenGL. Instead, it follows the single-program multiple data (SPMD) paradigm and lets programmers write C/C++ thread programs that are instanced and executed in parallel on the multiple processors of modern graphics processing hardware. For a thourough overview of the CUDA paradigm and state-of-the-art GPU architecture see [11].

We recommend further research on exploring the potential of this parallel programming model for incorporating scalable neural network simulations on multiprocessor architectures and for overcoming the limitations and performance bottlenecks depicted in this paper.

REFERENCES

[1] J. J. Hopfield and A. V. M. Herz, "Rapid local synchronization of action potentials: Toward computation with coupled integrate-and-fire neurons," *Proceedings of the National Academy of Sciences*, vol. 92, pp. 6655–6662, 1995.

[2] A. S. Maida, B. A. Rowland, and C. Günay, "Simulation of planar I/F networks with delayed connections," in *Proceedings of the International Joint Conference on Neural Networks*, 2001, pp. 302–307.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008.

[4] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[5] R. Meuth and D. Wunsch, "A survey of neural computation on graphics processing hardware," in *Proceedings of IEEE 22nd International Symposium on Intelligent Control*, 2007, pp. 524–527.

[6] Z. Luo, H. Liu, and X. Wu, "Artificial neural network computation on graphic process unit," in *Proceedings of IEEE International Joint Conference on Neural Networks*, vol. 1, 2005, pp. 622–626.

[7] F. Bernhard and R. Keriven, "Spiking neurons on GPUs," in *Proceedings of 6th International Conference on Computational Science*, 2006, pp. 236–243.

[8] D. Göddeke, "GPGPU tutorials," http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html, accessed 10/18/08.

[9] E. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[10] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating double precision FEM simulations with GPUs," Sep. 2005, ASIM 2005 - 18th Symposium on Simulation Technique Workshop Parallel Computing and Graphics Processors, Erlangen, Germany, September 12th-15th 2005.

[11] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface (Fourth Edition)*. Morgan Kaufmann, 2008.