2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing

# Accelerating Brain Simulations on Graphical Processing Units

Engin Kayraklioglu
The George Washington University
Washington, DC United States
engin@gwu.edu

Tarek El-Ghazawi
The George Washington University
Washington, DC United States
tarek@gwu.edu

Zeki Bozkus
Kadir Has University
Istanbul, Turkey
zeki.bozkus@khas.edu.tr

***Abstract*—NEural Simulation Tool(NEST) is a large scale spiking neuronal network simulator of the brain. In this work, we present a CUDA® implementation of NEST. We were able to gain a speedup of factor 20 for the computational parts of NEST execution using a different data structure than NEST's default. Our partial implementation shows the potential gains and limitations of such possible port. We discuss possible novel approaches to be able to adapt generic spiking neural network simulators such as NEST to run on commodity or high-end GPGPUs.**

***Keywords*—Brain simulation, CUDA, accelerators**

## I. Introduction

Contemporary neuroscience is evolving into an interdisciplinary science, where more traditional branches such as biology, neurology, genetics and chemistry gets entangled with mathematics, engineering and computer science. Involvement of computational sciences is caused by the fact that gathering experimental data of an intact brain is increasingly difficult[1].

There are various applications implemented to simulate brain on different levels of abstraction and/or approximation. GENESIS[2] is a simulator supporting simulation of neuronal network entities ranging from subcellular processes to neuronal network systems and PGENESIS[3] is an implementation of GENESIS to support running of multiple different simulations on networks of parallel processors. NEURON[4] is a simulator that can be used to simulate and analyze both network of neurons or independent neurons.

NEST[1] is a spiking neuronal network simulator that was designed to support large-scale simulations. Rather than detailed attributes of individual neurons, NEST focuses on dynamics of a neuronal network. NEST is implemented in C++ and has built-in support for MPI and OpenMP. NEST also has PyNN[5] support which provides a Python interface to replace NEST's default command line interpreter, called SLI.

### A. How NEST Works

A NEST network consists of nodes stored in a polymorphic C++ vector. Nodes can be neurons themselves, as well as external modules such as spike generators and voltmeters. There are plethora of neuron models conforming to a class hierarchy that comes bundled with NEST. Moreover, modular structure of NEST allows advanced users to implement their own neuron models.

NEST execution phases are as follows:

1) Create the neuronal network according to user's specifications, and store nodes in the `nodes` vector. Initialize time to 0.
2) Iterate over the `nodes` vector, calling `update` function of every node.
3) Advance time. If current simulation time did not reach user's specification, jump back to step 2
4) End execution. (A typical user would add more instructions to output the results of the execution in various different formats; i.e. text, plot etc.)

While iterating the `nodes` vector as in step 2, nodes can send spikes to the network depending on their state, which in turn is dependent on previous state and possible incoming spikes. `update` function of nodes can vary greatly depending on the type of the node. For instance, a neuron can go through several branches solving differential equations before deciding to send a spike to the network, whereas a poisson generator can use a simple random number generator and choose to send a spike.

NEST supports many different synaptic connection models as well as neuron models. Connections, like neurons, can differ in how they connect senders and receivers(i.e. one-to-one, divergent, convergent etc.) and how they transfer spikes(i.e. weight and delay of the connection). Connections are also stored in a data structure, thus there is no programmatic "connection" between nodes as in linked data structures. Every time a node spikes in step 2, the spike event is passed to it's connector, which transfers the spike to the recipients. In this structure, nodes are not directly aware of their neighbors.

### B. GPGPU Programming

General Purpose Graphical Processing Units(GPGPU) have been widely used in scientific computations as their massively parallel architecture enables executing many instructions at the same time, and scientific computations generally lend themselves to massive parallelization. CUDA[6] and OpenCL[7] are two widely used frameworks for programming GPGPUs. CUDA is a proprietary library developed by NVidia to use on their GPGPUs, whereas OpenCL is open source and

978-1-5090-0154-5/15 $31.00 © 2015 IEEE
DOI 10.1109/CIT/IUCC/DASC/PICOM.2015.79

supports many different devices including digital signal processors(DSPs) and field-programmable gate arrays(FPGAs) as well as GPGPUs.

Regardless of the framework, GPGPU programming has specific architectural limitations. 1) *Cores are more dependent on each other compared to CPU cores.* GPGPU threads are not scheduled independently. Scheduling groups of threads together is a limiting feature as lack of branch independence among threads is a imiting feature. Branching threads differently in the same group causes them to stay idle for many instruction cycles, reducing resource utilization greatly. 2) *Memory bandwidth is a more valuable resource.* Many cores trying to access device memory can cause significant overheads. Memory layout of the data is crucial to be able to serve memory requests from many cores efficiently. 3) *Data offload is costly.* Regardless of the type of computation, execution must start and end in host CPU and memory. Therefore, data needs to be transferred between host and device at least twice during a GPGPU execution cycle. Data transfers in algorithms that require constant host-device communication must be tailored carefully to increase communication/computation overlap and decrease overall execution time.

In this work, we discover ways to leverage many-core architecture of GPGPUs to parallelize NEST. Our contributions can be summarized as follows:

- An initial implementation of NEST in CUDA which focuses on computational part,
- Performance analysis of the implemented version with a previously designed neuronal network[8].
- Discussion of possible optimizations both in default and GPU implementation of NEST.
- Analysis of a projected full implementation of NEST, including its possible limitations. Investigation of the tradeoffs associated with producing a fully parallelized implementation.

Rest of this paper is organized as follows: in section II, we present similar studies in the field that can be roughly categorized as using GPGPUs to simulate generic or non-generic neuronal networks and performance studies of NEST, in section III thorough information about our implementation is given with specific focus on data layout, in section IV we discuss experimental results of our implementation, and finally, section V concludes the paper by briefly discussing possible next steps.

## II. Related Work

HRLSim[9] is a recent simulator that simulates neural networks on GPGPU clusters. Researchers created the simulator so that the network can be distributed to GPU nodes on a cluster. Their primary limitation was the enormous size of the network that is created. As they create the network on a *master* node initially, the size of the network is limited by the memory of the master node, which is 48GB in their setup. They were able to run real-time simulations of size 80K neurons and 800M synapses on such hardware. They also stress that performance and extensibility are conflicting goals while designing a high performance brain simulator especially on GPGPUs.

Other than HRLSim, there are various studies in the literature that develops non-generic neural network simulators that run on GPGPUs. Scorcioni [10] develops a GPGPU simulator that can outperform CPU by 20 times. Scorcioni's simulator bypasses the memory requirements of storing large number of synapses by using a just-in-time computation for some synaptic values. This way, very small amount of data(4 bytes) stored per synapse, and necessary values are computed in every iteration instead of storing them. Another such example is developed by Fidjeland and Shanahan [11]. Researchers developed *nemo*, a platform to simulate specific type of neurons on GPUs. To do that, authors used a special memory organization designed for GPU's memory architecture. They measured their performance in terms of throughput(spikes/sec) and stated that the performance they achieve corresponds to a realtime simulation of 55K neurons with 55M synapses.

Brette and Goodman [12] thoroughly discusses the possible strategies to implement spiking neural networks on GPUs. Along with their conceptual discussion they also give practical examples on CUDA on how to optimize for specific cases. They also pay specific attention to *spike propagation*, which is transferring spikes in the GPU memory and propose several optimized ways to achieve good performance. As with many other researchers they also mention large memory footprint required by very large number of synapses in a non-trivial neural network.

We are not aware of any experiments in the literature that tests NEST's performance on a GPGPU. However, NEST has a very mature implementation in MPI+OpenMP and there is extensive research on the subject matter. In one of the most recent works, Schenk et al. [13] devised a performance model that includes multi-node, multi-thread execution of NEST. Authors also suggested that optimizations regarding the data layout and using GPUs can give more performance for the update step. Kunkel et al. [14] focus on memory limitations of large-scale neural simulators on large clusters. They use NEST in their analysis and also evaluate its performance on more than 10000 processors. Necessity of optimizations in the data structures are also suggested in order to scale on large clusters.

## III. Experiments

### A. CUDA Implementation

As discussed before, GPGPUs are suitable for large scientific computations. However, memory bandwidth is scarce, and applications relying heavily on memory operations(including inter-thread communication) has to be implemented keeping memory limitations in mind[15].

Even though considerable amount of time is spent in computation during execution, NEST can be considered as communication-heavy. Figure 1 shows time spent on communication and computation. It is obvious from the figure that computation has linear time complexity, whereas communication has quadratic time complexity.

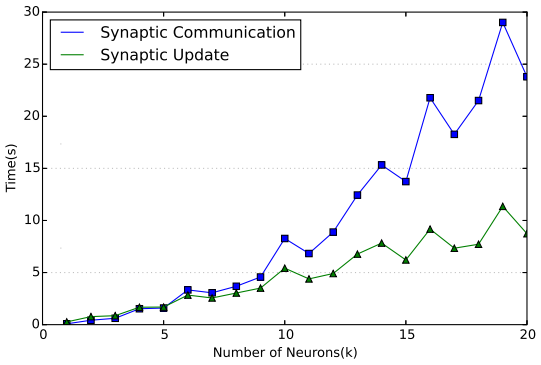
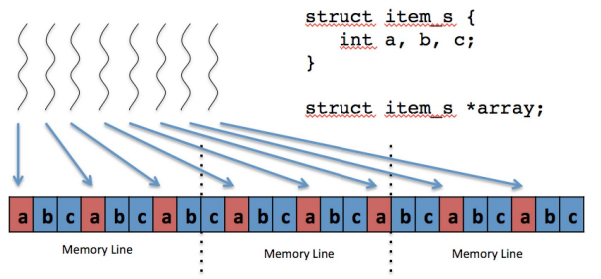
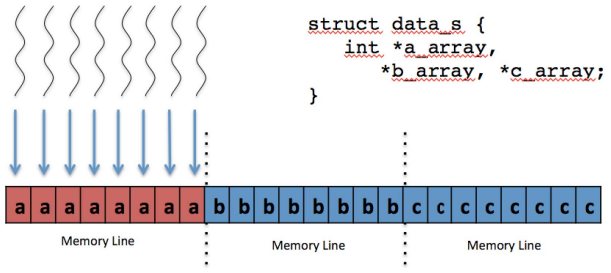



Fig. 1: Time Spent For Interaction Between Neurons(Communication) and Update(Computation) During a Sequential Execution of NEST




(a) Uncoalesced Memory Access




(b) Coalesced Memory Access

Fig. 2: Rearranging Nodes For More Efficient Memory Access

There is a clear distinction between when communication and computation happens in NEST. Considering this clean distinction between logical execution steps and very large source code, we chose an incremental approach to port NEST on CUDA.

*1) Data Layout:* As mentioned earlier, global memory accesses in CUDA must be carefully organized. We realized that NEST's default data structure to store nodes is not suitable for memory accesses in GPGPU. The `nodes` vector stores all NEST nodes in a standard C++ vector, where every node is an object with many attributes(limited to `a`, `b`, `c` in the example in Figure 2a). As CUDA threads are scheduled in groups of 32, which is called *warp*, many threads will be accessing the memory in SIMD fashion, therefore having those locations that are to be accessed at the same time closer can decrease the number of memory accesses significantly[1]. Optimized memory layout is shown in Figure 2b.

*2) Algorithm:* Our current implementation, depicted in Figure 3 runs as follows:

1) Create the network on host memory.
2) Change the memory layout to make it more suitable for GPGPU execution(As will be explained shortly).
3) Offload nodes vector to device memory.
4) Run the simulation on GPU for one time step. Record generated spikes in a sparse table to avoid atomic operations and to achieve better memory utilization.
5) Transfer recorded spike data to host memory.
6) Process generated spikes on CPU, and fill buffers of nodes as necessary.
7) Advance time. If simulation time reached user's specification, stop
8) Transfer node buffers to GPU. Jump back to step 4

A NEST network can consist of many different types of nodes, all of which has different `update` functions. However, some types have only few members in the network. We do not expect to gain much from the update of these nodes, as there are no parallelism to be harnessed. Therefore, we ported only the neuron in the network on CUDA and left the rest of the execution on the CPU. Even though the part on CPU is very minimal, GPU and CPU executions are overlapped.

### B. Characteristics of the Test Simulation

In our work we used a part of the network designed and used by Brunel[8], [16]. PyNEST[17] interface is used to define a neuronal network consisting of 10000 neurons, 8000 of which is "excitatory" and the rest is "inhibitory". Each type of neurons receive connections from 10% of the other population, where communications are made randomly.

To generate some activity in the network, a poisson spike generator is connected to all the neurons in a one-to-all fashion, an important thing to note here is that a poisson generator can generate multiple independent poisson spike trains to all its outgoing connections. In order to observe spikes, 2 spike detectors are attached to 50 neurons from each population. As the connections are uniformly formed among the population, observing only a small portion of the population is enough to make necessary interpretations[16].

## IV. Early Results & Future Work

Experiments were run on George supercomputer at The George Washington University. George is a Cray XE6/XK7 hybrid. XK7 partition consists of 30 NVidia XK40 GPUs.

Breakdown of execution time of different parts of the implementation is given in figure 4. Difference between communication and computation is even more significant, because the speedup over sequential run, which is slightly more than

[1]This is called *memory coalescing*

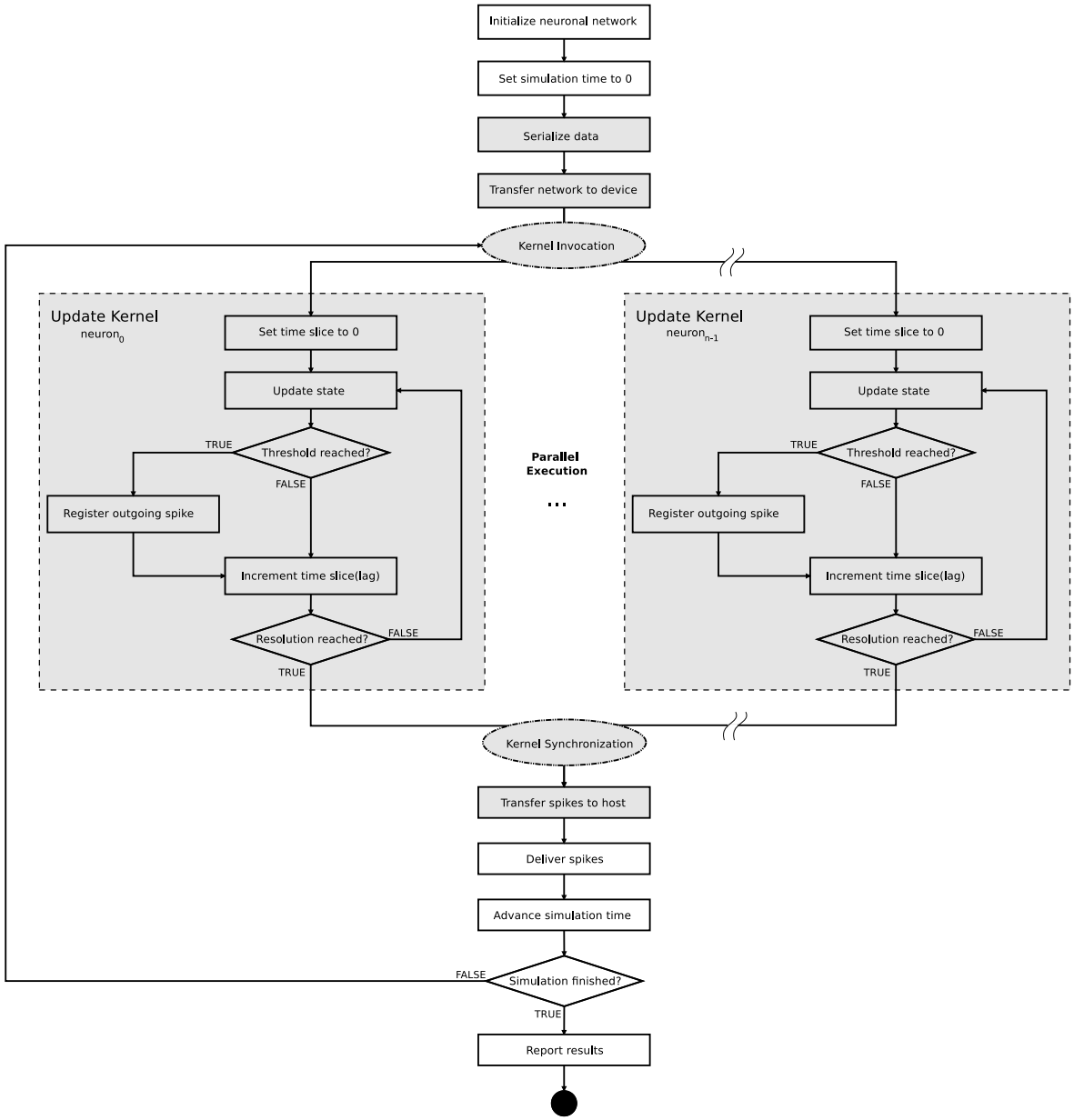



Fig. 3: Flowchart of current CUDA implementation. Shaded parts are CUDA-specific implementations. Update kernel is directly transformed from CPU implementation of the corresponding function. Thread grid is created in a way that each CUDA thread is responsible of a single neuron.

20x, obtained by massive parallelism in GPGPU. However, in terms of overall execution time, our initial port performed virtually same as the sequential execution regardless of the neuronal network size.

As described before, this port currently sends the data between the host and device in every iteration of the execution. Close to 16% of time was spent in these data transfers, where sparse data of spikes or buffers *ping-ponged*. Initial offloading of the data is also taking 5.36% of the overall execution time, which is an inherent overhead and completely dependent on the speed of PCIe bus.

Although in the latest versions of NEST, there has been some upgrades to the data structures they use[18]. Main point of those updates are to minimize memory footprint and we believe that GPGPU execution requires specific design of data structures. Therefore, we plan to analyze the possibility of using a structure that is optimal for GPGPU execution. Design of such structure may conflict with prior studies efforts, because such structure may have sparsity to some extent to provide well-structured and well-defined memory accesses for

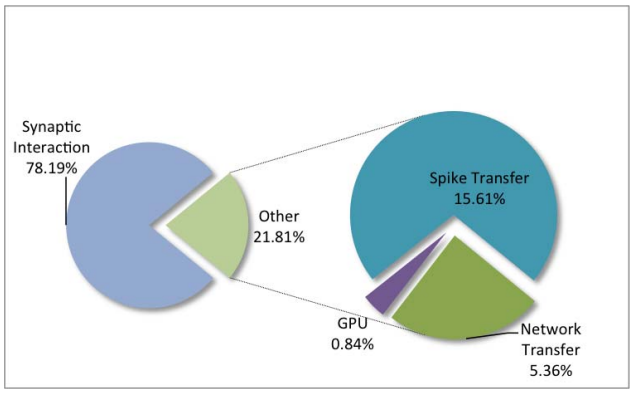



Fig. 4: Time Breakdown of CUDA Execution

GPGPUs. In order to discover the best way to utilize GPU memory we are planning to devise a memory model, keeping GPU architecture in consideration. Then, using our model to identify some possible solutions as feasible or infeasible.

In the current implementation, network is initially created on CPU and then transformed into a different layout suitable for GPU. As our implementation gets more mature, we plan to generate the initial data structure directly suitable for GPU to avoid additional initial overhead. Modular structure of NEST allowed us to port necessary neuron models only. In the future we expect to expand the GPU-supported model base. However, as noted before there are some node models(mainly external tools) that do not have large populations in a network. We do not anticipate porting those kinds of models onto GPU, as there is not much parallelism in small populations of nodes.

We anticipate that, after synapse interaction is efficiently ported on GPU, using Chapel's built in support for MPI, extending our approach to run on cluster of GPUs using message passing is not as complicated.

## V. Conclusion

In this work, we partially implemented NEST, a common spiking neuronal network simulator, on CUDA. We analyzed its current performance and possible limitations that can be faced while finishing the port. Our performance analysis showed that computations in NEST can be made faster by 20 times using GPGPU, compared to single core sequential run. However, we also showed that NEST is communication-intensive and the gap between communication and computation is increases as the network size gets larger. Therefore, we provide a new data layout different than NEST's default layout, to make memory accesses more efficient.

## Acknowledgement

Zeki Bozkus and Tarek El-Ghazawi are funded by Scientific and Technological Research Council of Turkey(TUBITAK; 114E046)

## References

[1] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007.

[2] J. M. Bower and D. Beeman, *The book of GENESIS (2nd ed.): exploring realistic neural models with the GEneral NEural SImulation System*. New York, NY, USA: Springer-Verlag New York, Inc., 1998.

[3] "PGENESIS," http://www.genesis-sim.org/project/pgenesis, accessed: 2015-02-01.

[4] N. T. Carnevale and M. L. Hines, *The NEURON Book*. New York, NY, USA: Cambridge University Press, 2006.

[5] A. P. Davison, D. Bruederle, J. M. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, no. 11, 2009. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/neuro.11.011.2008/abstract

[6] Parallel Programming and Computing Platform — CUDA — NVidia. Accessed: 2015-02-01. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[7] OpenCL - The standard for parallel programming of heterogenous systems. Accessed: 2015-02-01. [Online]. Available: https://www.khronos.org/opencl

[8] N. Brunel, "Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons," *Journal of Computational Neuroscience*, vol. 8, no. 3, pp. 183–208, 2000. [Online]. Available: http://dx.doi.org/10.1023/A%3A1008925309027

[9] K. Minkovich, C. Thibeault, M. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa, "HRLSim: A High Performance Spiking Neural Network Simulator for GPGPU Clusters," *Neural Networks and Learning Systems, IEEE Transactions on*, vol. 25, no. 2, pp. 316–331, Feb 2014.

[10] R. Scorcioni, "GPGPU implementation of a synaptically optimized, anatomically accurate spiking network simulator," in *Biomedical Sciences and Engineering Conference (BSEC), 2010*, May 2010, pp. 1–3.

[11] A. Fidjeland and M. Shanahan, "Accelerated simulation of spiking neural networks using GPUs," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, July 2010, pp. 1–8.

[12] R. Brette and D. F. M. Goodman, "Simulating spiking neural networks on GPU," *Network: Computation in Neural Systems*, vol. 23, no. 4, pp. 167–182, 2012, pMID: 23067314.

[13] W. Schenck, Y. V. Zaytsev, A. Morrison, A. V. Adinetz, and D. Pleiter, "Performance model for largescale neural simulations with nest," Poster, November 2014.

[14] S. Kunkel, T. C. Potjans, J. M. Eppler, H. E. E. Plesser, A. Morrison, and M. Diesmann, "Meeting the memory challenges of brain-scale network simulation," *Frontiers in Neuroinformatics*, vol. 5, no. 35, 2012. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2011.00035/abstract

[15] Best Practices Guide :: CUDA Toolkit Documentation. Accessed: 2015-02-01. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[16] M.-O. Gewaltig, A. Morrison, and H. E. Plesser, *NEST by Example: An Introduction to the Neural Simulation Tool NEST Version 2.2.2*, July 2013.

[17] J. M. Eppler, M. Helias, E. Muller, M. Diesmann, and M.-O. Gewaltig, "PyNEST: a convenient interface to the NEST simulator," *Frontiers in Neuroinformatics*, vol. 2, 2008.

[18] S. Kunkel, M. Schmidt, J. M. Eppler, H. E. Plesser, G. Masumoto, J. Igarashi, S. Ishii, T. Fukai, A. Morrison, M. Diesmann, and M. Helias, "Spiking network simulation code for petascale computers," *Frontiers in Neuroinformatics*, vol. 8, no. 78, 2014. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2014.00078/abstract