Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments

MICHIEL A. VAN DER VLAG and GEORGIOS SMARAGDOS, Erasmus Medical Centre - Department of Neuroscience, The Netherlands
ZAID AL-ARS, Delft University of Technology - Quantum & Computing Engineering, The Netherlands
CHRISTOS STRYDIS, Erasmus Medical Centre - Department of Neuroscience, The Netherlands

In-silico brain simulations are the de-facto tools computational neuroscientists use to understand large-scale and complex brain-function dynamics. Current brain simulators do not scale efficiently enough to large-scale problem sizes (e.g., >100,000 neurons) when simulating biophysically complex neuron models. The goal of this work is to explore the use of true multi-GPU acceleration through NVIDIA’s GPUDirect technology on computationally challenging brain models and to assess their scalability. The brain model used is a state-of-the-art, extended Hodgkin-Huxley, biophysically meaningful, three-compartmental model of the inferior-olivary nucleus. The Hodgkin-Huxley model is the most widely adopted conductance-based neuron representation, and thus the results from simulating this representative workload are relevant for many other brain experiments. Not only the actual network-simulation times but also the network-setup times were taken into account when designing and benchmarking the multi-GPU version, an aspect often ignored in similar previous work. Network sizes varying from 65K to 2M cells, with 10 and 1,000 synapses per neuron were executed on 8, 16, 24, and 32 GPUs. Without loss of generality, simulations were run for 100 ms of biological time. Findings indicate that communication overheads do not dominate overall execution while scaling the network size up is computationally tractable. This scalable design proves that large-network simulations of complex neural models are possible using a multi-GPU design with GPUDirect.

CCS Concepts: • Computing methodologies → Simulation evaluation; • Computer systems organization → Interconnection architectures; Multicore architectures; • Applied computing → Computational biology;

Additional Key Words and Phrases: Neural networks, multi-node, multi-GPU

ACM Reference format:
Michiel A. van der Vlag, Georgios Smaragdos, Zaid al-Ars, and Christos Strydis. 2019. Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments. ACM Trans. Archit. Code Optim. 16, 4, Article 53 (December 2019), 25 pages. https://doi.org/10.1145/3371235

We gratefully acknowledge the NVIDIA Corporation for their support during the development of this work.
Authors’ addresses: M. A. van der Vlag, G. Smaragdos, and C. Strydis, Erasmus Medical Centre, Department of Neuroscience, Rotterdam, The Netherlands; emails: michielvandervlag@gmail.com, g.smaragdos@erasmusmc.nl, cstrydis@erasmusmc.nl; Z. al-Ars, Delft University of Technology, Quantum & Computing Engineering, Rotterdam, The Netherlands; email: Z.Al-Ars@tudelft.nl.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1544-3566/2019/12-ART53
https://doi.org/10.1145/3371235
1 INTRODUCTION

Spiking Neural Network (SNN) [8, 19, 20] simulations are commonly used tools to help understand the dynamics of biological brains. The larger and more complex a simulation becomes, the more computationally intensive it will be. Complex SNN simulations do not scale linearly with large-scale problem sizes (e.g., >100,000 neurons) [1, 7, 29], which makes them impractical for researchers. High-Performance Computing (HPC) technology such as Graphics-Processing Units (GPUs) has often been used to cope with this limitation. Most modern GPU simulators take the naive approach of using single or multiple (but isolated) GPUs to perform large-scale simulations, squeezing each individual experiment into one GPU device. Even though such solutions can provide considerable speedup for a wide variety of (typically small-scale) problems, large-scale experiments of complex models cannot be tackled by such an approach. Performance aside, with increasing problem (i.e., network) size and model complexity, large-scale experiments cannot even fit within a single GPU [7], thus requiring partitioning to multiple GPUs in a way that information can be efficiently exchanged among devices. There is much to be gained by performing true multi-GPU simulations, made possible by the more recent advances in GPU technology, such as GPUDirect™.

GPUDirect [25] is a technology that permits the disaggregation of (GPUDirect-enabled) GPUs from CPUs and enables a direct path for data exchange among GPUs on, for instance, independent compute nodes in a cluster, with the purpose of keeping the latency to a minimum. Then, true multi-GPU simulations whereby inter-GPU data exchange occurs with minimal-latency technology, instead of having to stage the data exchange by various memory copies, of which [2, 17] are examples. For instance, in molecular dynamics, the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [27] or the Amber Package [28] are software for performing large-scale trajectory simulations, which GPUDirect has reduced from days to hours.

The goal of this article is to explore the use of true multi-GPU acceleration on very computationally demanding brain models and to assess the performance and scalability of such models given sufficient access to multi-node acceleration platforms. As a representative workload, an in-house developed, extended Hodgkin-Huxley (eHH), three-compartmental model of the Inferior-olivary nucleus (ION) of the brain is used in this work. The ION is a critically studied structure (e.g., see References [24, 34, 36]) found under the medulla oblongata which, jointly with the cerebellum, is responsible for crucial functionality involving motor control, sensorimotor integration, dexterity, and more. Furthermore, the ION workload is based on the eHH formalism, which captures a widely accepted and very compute-intensive type of biophysically meaningful SNNs. As a result, observations on such a representative workload have large applicability to any other simulation using the eHH representation [9, 12, 31]. Not only the cell-simulation (i.e., kernel) time but also the network-setup time is taken into account when designing our multi-GPU version. The network-setup overhead is often ignored, as most previous work focuses solely on improving kernel times, yet on very large-scale networks the former can introduce comparable delay and complexity as the useful simulation itself.

In Reference [15], the effect of the network-setup time is also taken into account; however, in contrast to the current article, the performance for a single GPU is only studied. In this article, different network-connectivity distributions are selected, which still match realistic connectivities encountered in the ION, to test the limits of the multi-GPU design. Based on the above, the contributions of this article are as follows:

1. A comprehensive survey of existing (multi-)GPU simulators, platforms, and libraries.
2. Design, implementation, and deployment of a true multi-GPU design based on GPUDirect.
3. In-depth evaluation of performance, scalability, and energy-consumption aspects of the design.
General recommendations for efficient simulation of future large-scale SNNs with multiple GPUs.

This article is structured as follows: Section 2 introduces essential background knowledge on neuroscientific modelling, the equations of the ION model and multi-node GPUDirect technology. Section 3 gives an overview of the related work of GPU-based SNN simulators, taxonomized according to supported model features and performance. Section 4 delineates the main architecture—consisting of two setup and two computation phases—of our proposed multi-GPU implementation for scaling neural networks to and beyond large-scale problem sizes. Section 5 provides a performance evaluation of the developed multi-node GPU design in terms of speedup for all four phases separately and in total and energy evaluation. Also, a comparison with a similar design implementing high-performance multi-threading that uses Intel Xeon Phi (Knights Landing) hardware [1] and a roofline model analysis is presented. In Section 6, the results from the performance evaluation are discussed and final conclusions are drawn in Section 7.

2 BACKGROUND

2.1 Spiking Neural Networks

There are three main categories of SNN models: Leaky Integrate and Fire (LIF), Izhikevich (IZH), and conductance-based models of which the Hodgkin-Huxley (HH) is perhaps the most prominent. All categories have scientific merit depending on the topic the computational neuroscientist aims to study.

LIF models are simple models re-creating the most basic characteristics of neuron behavior. The model controls the neuron membrane potential. As inputs are coming into the neuron, the membrane potential increases until it reaches a certain threshold value. The moment the threshold is reached the neuron fires a spike communicating it to the rest of the network. The membrane potential is then reset to its initial value. This model is quite useful for studying large-scale networks and the most basic neural dynamics [14]. IZH models, on the other hand, can display most biologically plausible spiking patterns and with similar complexity to the LIF models. However, even though it is a much more advanced model, the IZH still treats the neuron as a black box [14]. If the subject of the study is to research the biophysical characteristics of neurons, then the neuroscientist needs to use conductance-based models. This is the reason this model type has been selected for our simulations. The HH is the most common variant used in the field. It consists of four equations and tens of parameters describing membrane potentials and activation or inactivation of ion channels representing them as RC circuits [14]. As a result, this level of modelling begins to accurately emulate neuron biology, alas at the cost of orders-of-magnitude higher computational demands. Adaptive exponential (ADEX) models are a popular variant of integrate-and-fire models that can better capture the biochemical processes inside neurons but not as closely as HH models.

The mathematical description of the HH model is shown in Equations (1) to (14). The derivative of the membrane potential $V$ equals the sum of currents which go through the membrane. $I_{app}$ is the membrane current and can be modelled by any function. $I_{leak}$ is the leakage current which has a constant conductance. $I_{channels}$ is the sum of the currents generated by the ion channels (the Na and K channel in the case of the standard HH model). The conductances are dependent on the so-called gate-activation variables ($n$, $m$, and $h$ in the case of the HH model). The gate-activation variables define the proportion of ion gates in the total population which are open. Depending on the values of those variables, the ionic currents $g_K n^4 (G_K)$ and $g_Na m^3 h (G_{Na})$ change. How the gate-activation variables change is described by Equations (2) and (4) [22].
2.2 Numerical Methods

To simulate the HH neural model is to solve Ordinary Differential Equations (ODEs) and thus numerical methods need to be used [22]. Due to the high computational cost of a root-finding algorithm required by implicit methods, only explicit numerical methods are typically considered for HH models. An explicit method calculates the next step in time based only on the current state of the model [21] and can mathematically be represented by Equation (15). The first-order forward-Euler method, used to solve the ODEs, is described by Equation (16) and a fixed time step is then used (typical value: 1 ms), which makes HH simulations time driven.

\[
\frac{dV}{dt} = \frac{I_{app} - I_{channels} - I_{\text{leak}}}{C_M}, \quad (1) \\
\frac{dn}{dt} = \alpha_n (1 - n) - \beta_n n, \quad (2) \\
\frac{dm}{dt} = \alpha_m (1 - m) - \beta_m m, \quad (3) \\
\frac{dh}{dt} = \alpha_h (1 - h) - \beta_h h, \quad (4) \\
I_{\text{channels}} = I_K + I_{Na}, \quad (11)
\]

where

\[
\alpha_n = \frac{0.01(V + 10)}{\exp\left(\frac{V + 10}{10}\right) - 1}, \quad (5) \text{ where:}
\]

\[
\beta_n = 0.125 \exp(V/80), \quad (6) \\
\alpha_m = \frac{0.1(V + 25)}{\exp\left(\frac{V + 25}{10}\right) - 1}, \quad (7)
\]

\[
X_{n+1} = F(X_n), \quad (15) \\
I_{\text{gap},i} = \sum_{j=0}^{N_{\text{cells}}-1} \left( w_{i,j} (0.8 \exp(-0.01 \cdot V_{i,j}^2) + 0.2) V_{i,j} \right). \quad (17)
\]

2.3 The Inferior-Olivary Nucleus

The Olivocerebellar system [6] generates precise patterns of complex and simple spiking during the learning process of motor skills and is a relatively well-charted region of the brain. Its structure is highly repetitive and essentially consists of the granule-cell layer, Purkinje-cell layer, deep cerebellar nuclei, and inferior-olivary nucleus (ION). The model used to simulate ION cells is an extended-HH (eHH) model and is being developed for many years within the neuroscience department of the Erasmus Medical Center in Rotterdam [5, 24].

The ION is one of the most compacted brain regions, and the neurons therein are densely interconnected through so-called gap junctions (GJs). Gap junctions are electrical synapses that allow for rapid communication in the form of electrical current between two or more neurons. The model has only electrical and no chemical synapses, which are complex connections considered to be associated with important aspects of cell behavior. GJ connectivity density ranges from 10% to 100%, which poses significant computational challenges. The mathematical representation of
the GJs is shown in Equation (17), which is then added to the sum of currents in the nominator of Equation (1). \( w_{i,j} \) is a weight that dictates the strength of the connection between two cells \( i \) and \( j \).

There is no significant difference between younger (7.1 × 10^5) and older (6.5 × 10^5) subjects, when it comes to the number of neurons in the ION [18]. All cells may be connected to all other cells, meaning that the ION could comprise up to 5.0 × 10^11 interconnected cells. To simulate such a model, high-throughput computational as well as low-latency communication technologies are essential.

The neurons in this model consist of three compartments: a dendrite, a soma, and an axon [5]. These compartments have electro-chemical characteristics which influence the overall state of the neuron. The dendritic compartments are interconnected through the GJs, making the cells dependent not only on their own compartments but also on those of other cells. This real interdependence among neurons is what makes the ION unique in exhibiting constant neuronal oscillations at rest. Because every neuron needs to be informed of its neighbors’ results at each simulation step, this further enforces the use of time-driven (cf. event-driven) simulations and makes the ODEs interdependent in time. Due to this high timing sensitivity, the ION ODEs become very stiff, enforcing the use of a rather small fixed time step of 0.05 ms to prohibit equations breaking. The multiple compartments per neuron as well as the GJs are, therefore, extensions to the standard HH model, and thus its characterization is “extended,” eHH, and they are both highly relevant for modern computational neuroscience.

2.4 Multi-node Accelerator Technology

Profiling the ION model reveals the relation among network size, connectivity density, and the number of floating-point operations needed to be executed, resulting in a high computational complexity. It is shown that when density grows, even more floating-point operations need to be performed to simulate the model. It also reveals that the model is compute bound for larger densities [29]. This makes it an excellent target for HPC fabrics such as the GPU. Since the application is compute bound, adding more computational devices will reduce overall execution times. To take advantage of a multi-node GPU (or multi-GPU for short) implementation, the neural network will have to be partitioned across multiple GPU devices, which implies exchanging a lot of data across the GPUs, data exchange effectively becoming a major performance bottleneck.

GPUDirect [25] is a software technology providing high-bandwidth and low-latency communications directly among multiple (disaggregated) GPUs in a cluster. The main two technologies under this umbrella are Peer-to-Peer (P2P) for intra-node and Remote DMA (RDMA) for inter-node communication. P2P transfer technology allows for fast intra-node transfers but does nothing for inter-node memory transfers. It allows for buffers to be copied directly between the memories of GPUs. It employs Unified Virtual Addressing (UVA), which enables the host memory and memory of all GPUs to be combined into one large virtual address space [16]. A prerequisite for P2P is that the source and destination device need to be attached to the same PCIe root complex, the application needs to be 64 bit, and the operating system must enable the Tesla Compute Cluster option. RDMA allows memory transfers between GPUs in a cluster over PCIe or Infiniband. The GPU is enabled to send data directly to a remote system without any intervention from the CPU. This technology is available for the Tesla accelerator cards, starting from the Kepler architecture onward. In Figure 1, the memory-transfer paths are shown with and without GPUDirect.

CUDA-aware MPI [16] (CAM) libraries are the result of the effort of combining the two parallel programming approaches Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). OpenMPI (not to be confused with OpenMP [3], the API for multiprocessing programming) and MVAPICH2 are CAM libraries that implement GPUDirect. These libraries
Fig. 1. Memory-transfer paths for the RDMA mode [33]. The orange and red arrows represent data being transferred from a GPU to another GPU on another node. Without GPUDirect on the top right side: The same data needed to be copied 3 times. First, the GPU writes to the pinned system memory 1 (SM); second, the CPU copies from SM1 to SM2; and third, the Infiniband driver copies the data from SM2. When the data are received on the other end, the same memory copies need to be done once more. With GPUDirect, shown in the bottom half, the network adapter can directly read/write data from/to the GPU device memory.

enable the direct passing of the GPU buffer pointers to MPI, instead of needing to stage GPU buffers through the host memory using cudaMemcpy, resulting in a direct write from one device to another.

The cluster selected for benchmarking is Cartesius from SURFsara, which consists of many heterogeneous nodes [32]. The GPU accelerator island consists of 66 Bullx B515 GP-GPU accelerated nodes. Each one of these nodes consists of 2 × 8-core 2.5 GHz Intel Xeon E5-2450 v2 (Ivy Bridge) CPUs/node, 2 × NVIDIA Tesla K40m GPUs/node, and 96 GB of memory/node. The Tesla K40m is a server-grade PCIe accelerator board consisting of a single GK110B GPU with NVIDIA Kepler architecture, which fully supports RDMA. From the node topology, it became clear that the GPUs are not connected to the same PCI root complex. Each node contains two CPUs which have their own root complex. Each CPU has a GPU connected to it, separating the memory address space. Intra-node communication between nodes using the improved RDMA technology on Cartesius is, thus, not possible. Luckily, this does not affect our core experiments which are focused on RDMA inter-node communication. Also, from reported benchmarks on Cartesius, it is observed that, even for larger messages, OpenMPI has the lowest latency and could acquire the highest bandwidth.

3 RELATED WORK: SURVEY OF NEURAL-NETWORK SIMULATORS

The ION is not a widely simulated model; we have no knowledge of simulators that focus on this model, apart from our own work. The goal of our work is BrainFrame, which is a heterogeneous acceleration platform that incorporates three distinct acceleration technologies, Intel Xeon-Phi CPUs, NVIDIA GP-GPUs, and Maxeler Dataflow Engines (DFEs). So far, we have simulated the ION model on a single-node GPU [7], Xeon-Phi [29], and DFE [30] setup as well as on a multi-node (eight-way) Xeon-Phi [1] setup. Eventually, BrainFrame will move toward multi-node heterogeneity and into the Cloud for all to access. In this section, parallelizing technology is surveyed, with the
focus on true multi-GPU architectures. We will show that such technology has not been utilized to the fullest to create a design that scales efficiently enough with large-scale problem sizes.

For instance, the Brian simulator is written in Python and is available on almost all platforms [10]. Custom models are coded as equations and it supports multi-compartmental LIF, IZH, and HH models. Brian offers single-GPU support in the form of Brian2CUDA, which generates C++/CUDA code to run on NVIDIA GPUs [31]. Another back-end that can be used to execute Brian-generated models on GPUs is simulator/code generator GeNN [37]. GeNN consists of a C++ source library that generates CUDA kernels and runtime code according to a user-specified network model. This is similar to ANNarchy; a high-level Python interface to facilitate the creation of neural networks [35]. Neurons and synapses can be implemented with an equation-orientated syntax close to Brian. These definitions are used to generate an entire C++ library optimized for parallel frameworks such as OpenMP or CUDA for a GPU. GeNN, Brian2CUDA, and ANNarchy; however, all are single-GPU oriented.

The Cognitive Anteater Robotics Laboratory simulator (CARLsim), is a GPU-accelerated library for simulating large-scale SNNs [2]. Parameter details at the synapse, neuron, or network levels can be specified in a PyNN-like [4] programming interface in C/C++. Currently, the four- and nine-parameter IZH with either current-based or conductance-based synapses and LIF models are supported. Neurons can be extended to include multiple compartments, and it has support for chemical but not for electrical synapses (GJs). CARLsim 4.0.0 beta allows for concurrent simulation on up to eight GPUs, which reside on the same machine. It does not support MPI and needs extra memory copies between host and devices to transfer data. There is no support for GPUDirect.

NEURON provides a powerful and flexible environment for implementing biologically realistic models of electrical and chemical signalling in neurons and networks of neurons [12]. It makes use of the database ModelDB, which contains over 1,000 neural models and many variants of LIF, IZH, and HH models can be downloaded for simulation. NEURON supports multiple neuron compartments called sections (and referred to as unbranched cables). CoreNEURON is a simplified engine for the NEURON simulator optimized for both memory usage and computational speed [17]. Its goal is to simulate larger network models on modern supercomputing platforms and to reduce memory footprints. It can target both CPU and GPU architectures. The reports on its performance are unclear. They mention that a network of 5,000 cells with 1.2 million synapses is simulated \( \times 1.2 \) faster than NEURON. The number of GPUs and the underlying communication technology used are not mentioned.

HRLSim is a high-performance simulator for GPU clusters [23]. Its goal is to build a cognitive computer with similar form, function, and architecture to the mammalian brain. HRLSim supports LIF and a simple IZH model and allows for excitatory and inhibitory synapse modelling but makes no explicit statements about GJ modelling. For intra-node P2P and for inter-node communication, MPI is used; however, there is no explicit statement about making use of RDMA. This project is closed-source and not available for others to use.

Neural Cortical Simulator (NCS6) is an extensible real-time neural simulator for heterogeneous clusters of both CPUs and CUDA-capable GPUs [13]. Advantages of NCS6 are its computational power, biological-modeling capabilities at multiple abstraction levels and its minimum programming effort. NCS6 has built-in support for the LIF and IZH models and both excitatory and inhibitory synapses are supported. Limitations are the lack of biophysical parameters due to lack of direct HH-model support, yet its sub-threshold dynamics are determined by channels that follow the HH formalism.

In Table 1, an overview of the surveyed simulators and their characteristics are shown. Most of the simulators support the common neuron models, GJ and multi-compartmental modelling. The “performance indication” field reports wall-clock time normalized to 1 s of brain time, which serves...
It portrays the landscape of the brain simulators that utilize GPUs. From the cited studies, information about their best-effort performance has been extracted. The performance indication reports execution times for 1 s of brain time.

as a comparison for the current work. For Brian2CUDA, no official performance results have been published, nor are results found for the multi-GPU design of CARLsim. NEURON reports that the displayed execution times are acquired using 128k nodes of the BlueGene/Q cluster \[11\]. Besides GeNN, none of the simulators reports the setup time of the connectivity network. For GeNN, the total network-setup time is 1 s on the GPUs, next to 4 s for simulating 1 s of brain time. There are four known simulators which have a multi-GPU implementation: CARLsim, CoreNEURON, HRLSim, and NCS6, but none of them are true multi-GPU simulators.

### THE MULTI-NODE GPU DESIGN

In Figure 2(a), a schematic view of the ION network acting as our simulation benchmark is shown, which consists in this case of six fully interconnected neurons (via GJs). As we shall show next, it is these dense GJs that impose large data moves across neurons and, subsequently, across GPUs. Figure 2(b) zooms in on a single neuron and illustrates the three compartments and the GJs.

Our multi-GPU implementation, running the ION model, performs the setup of the connectivity network and the computation of the cells. To avoid confusion with the neural networks being simulated, the network of issued GPUs for the simulation shall be referred to as the GPU-world. The setup process consists of two phases, namely the connectivity-network generation phase, in which the network is created, and the connectivity-network dispersal phase, in which the connectivity information is dispersed throughout the GPU-world. After setup, the simulation process begins, which breaks down into two further phases, namely the cell-computation phase and the dendrite-communication phase. In the cell-computation phase, the computations of the dendrite, soma, and axon compartments take place. The dendrite-computation phase will be shown to have a huge impact on the multi-GPU design. A GJ connects cells through their dendritic compartments. Whenever interconnected cells need to be computed, the dendritic voltages need to be gathered and dispersed throughout the GPU-world. Figure 3 shows a flowchart of the four phases of the design, the platform on which each phase runs and a high-level, pseudocode-based overview of the complexity. Implementation details of each of these phases will be discussed in the next sections.

The GPUs are used to speedup the connectivity-network generation for which the cuRAND library for random-number generation is utilized. The OpenMPI library, used to interconnect the GPUs over RDMA, facilitates inter-node dendrite-voltage communication. The OpenMP library is employed to speedup the packing and unpacking of the dendrite data during the
Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments

Fig. 2. Graphical representation of the inferior-olivary network model. (a) Six-neuron all-to-all connectivity network; (b) single-neuron model in detail [29]. From the three compartments, only the dendrite is connected with GJs. The bidirectional arrows represent the GJ. In case of an all-to-all network, this cell is connected to another cell, which in turn is connected to the GJs by its dendrite compartment.

Fig. 3. Main flowchart with the four phases of simulation and their complexity. The connectivity-network generation is executed on the GPUs. In this phase, every cell (thread), considers every other cell to be its neighbor. The connectivity-network dispersal is executed on a CPU with OpenMP and the MPI functions with OpenMPI. The cell computation is executed on the GPUs. In this phase, every cell (thread) takes all its neighbors into consideration for each simulation step. The dendrite communication is executed on a CPU with OpenMP and the MPI function with OpenMPI.

dendrite-communication phase. The result is a hybrid version using both GPUs and CPU cores for parallelization, supporting both network generation and inter-node communication.

When multiple GPUs are enlisted, each GPU is assigned a portion of the total neural network. This means that the generation and computation of the network cells—that is, the workload—is properly distributed across the per-case available GPUs. The more GPUs enlisted, the more memory becomes available and the fewer cells per GPU need to be stored, assuming a fixed network size. A higher GPU count, however, also means that the communication between nodes increases. In Figure 4, an overview of the multi-node GPU implementation is shown.

4.1 Connectivity-network Generation
To analyse the impact of network topology on performance, two naturally occurring topologies are studied, following a uniform or a 3D-Gaussian distribution [1]. In the latter, the likelihood of two cells forming an electrical connection between them is based on their proximity in three-dimensional (3D) space. Since the ION model does not capture geometric properties of the cell topology, the above distributions are used to decide on efficient ways of distributing cells across
Fig. 4. System overview for multi-node GPU design. At the beginning of the application the connectivity-network generation kernel, divided over the SMs, is executed for an equal portion of the entire cell network to determine the neighbors of each cell. In case of a GPU-world of 2, every GPU takes care of half the network. After dispersal of the neighbor information to both GPUs via GPUDirect over Infiniband, the computation kernel, divided over the SMs, computes the new states of the neurons. It stores the axon and soma in global and the dendrite voltages in texture memory. The dendrite voltages are then communicated via GPUDirect over Infiniband to the other GPU, after which the next cell-computation phase can commence.

GPUs based on how many and which connections individual cells form; all directly connected cells have equal spatial distance independent of their place in the neural grid.

The uniform distribution creates uniformly spread-out connections within the network, while the Gaussian distribution creates clusters of densely packed neurons internally which, however, are only lightly interconnected to each other. Connectivity-network generators of both types have been implemented as multi-GPU kernels working independently at the onset and being invoked in this simulation process to significantly accelerate connectivity-network creation. The Gaussian generator is more computationally complex than the uniform generator, which accounts for a longer execution time in the case of Gaussian-distributed networks (to be discussed later). The uniform generator is less complex, as it is based on comparing a fixed probability against the outcome of a random generator. The Gaussian distribution is not only more complex, it also bears a third loop, which prolongs the effort of finding a possible neighbor. The pseudocode for generating the Gaussian distribution is shown in Algorithm 1.

The density setting for the network is exposed and configurable by the user. For the uniform distribution, the probability of a connection forming is inversely proportional to the network size: probability = density / network size. The larger the network size, the smaller the probability for a potential connection, thus ensuring a constant number of connections per cell when the network size varies.

Random-number generation is a vital part of both network generators. Instead of the library function rand(), the cuRAND library is available for GPUs and provides facilities that focus on the simple and efficient generation of high-quality pseudo- and quasi-random numbers. This library invokes the XORWOW generator, which is a variation on the XOR shifting generators. They generate the next number in their sequence by repeatedly taking the exclusive OR of a number with a bit-shifted version of itself.
4.2 Connectivity-network Dispersal

The target cell for which neighbor cells are selected always resides on a local GPU. Neighbor cells may reside on the local GPU or on remote GPUs. Once the connectivity network with the neighbor cells connected through GJs has been set up, the remote connections need to be dispersed among the GPU-world in such a way that the computed dendritic values in each time step are sent to the correct GPUs. The local GPU only knows of the cell neighbors it requires; it does not know anything about which neighbors other GPUs require. After dispersal, it has informed remote GPUs which cells it requires the dendrite voltages of for use during the computation phase. Afterward, the local GPU thus has a list of successive unique identification numbers, called global-cell-IDs. When this phase is complete, all GPUs have lists of global-cell-IDs. From these global-cell-IDs the local GPU can decode the remote GPU to which it has to send the dendrite voltages, since each GPU gets allocated a fixed number of global-cell-IDs beforehand. Each GPU most likely has a different set of neighbor connections that it requires from the other GPUs.

The dispersal of the connections is twofold: First, the total neighbor count is communicated across the GPU-world enabling, at a second step, the actual connectivity-network dispersing. The pseudocode for this phase is shown in Algorithm 2. In between these MPI calls, the size of the to be received global-cell-IDs array and the send and receive displacements are calculated. The displacements determine which part of the global-cell-ID array is sent to which GPUs.

In this phase, the GPUDirect technology is invoked for the connectivity-network dispersal to reduce latency and increase bandwidth. Because the communication between GPUs is most likely to become the performance bottleneck, an effective communication should only transport the exact amount of cell data to the demanding remote GPUs. For the cell communication, the collective function MPI_Alltoallv is used, in which the suffix \textit{v} indicates that packets with variable sizes can be sent.

After the first step, every GPU knows the amount of cell data that need to be sent to each remote GPU. Because the GPUs know the cell count, memory can be allocated on the GPUs for the to be received neighbor-cells, after which the second MPI_Alltoallv function is executed, sending the actual connectivity information to the demanding GPUs. The GPUs now know which dendrite voltages they have to collect and send back to the demanding GPUs. This concludes the setup process.
ALGORITHM 2: Pseudocode for connectivity-network dispersal.

1: procedure CONNECTIVITY-NETWORK DISPERAL
2: MPI_Alltoall(xsize_local, 1, xsize_glob, 1) ▷ Spread total amount of global-cell-IDs
3: for (each neuron/GPU: i) do
4: xcell_size+ = cellno_perGPUglob[i] ▷ Determine size for malloc of retrieved cells
5: sd[i] = sd[i-1] + xsize_local[i-1] ▷ Determine send displacement
6: rd[i] = rd[i-1] + xsize_glob[i-1] ▷ Determine receive displacement
7: end for
8: MPI_Alltoall(xcell, xsize_local, sd, retrieveCells, xsize_glob, rd) ▷ Spread global-cell-IDs
9: end procedure

ALGORITHM 3: Pseudocode for Gap-Junction calculation.

1: procedure CALCULATEGJ(previousVolt) ▷ Parallelized by the GPU. Every thread is a target cell
2: for (each target-cell) do
3: for (each neighbor_cell) do
4: V = previousVolt - neighborVolt ▷ Compute GJs
5: f = 0.8 * exp(-1 * V^2 / 100) + 0.2
6: Ic = Ic + (neighborConductance * f * V)
7: end for
8: end for
9: return Ic
10: end procedure

4.3 Cell Computation

When all the cell connections have been dispersed, we can enter the actual simulation, which is a transient process. Now, the axon, soma, and dendrite voltages of each neuron can finally be computed. In this cell-computation phase, the system of ODEs representing the ION model are solved using the forward-Euler numerical method described in Section 2. The computation of the dendrite voltages is the biggest bottleneck in performance, because it includes the gathering of these voltages from each of the neighbors of the target cell. The pseudocode for this phase can be seen in Algorithm 3. For each cell, which means for every GPU thread, every neighbor needs to be considered. Essentially, this means that every thread is stalled for a number of iterations equal to the number of neighbors the particular cell has.

When calculating the new dendrite voltages, the neighboring dendrite voltages are fetched from 2D texture memory, where they were stored at the end of the previous simulation step, according to their global-cell-IDs. The difference of these voltages is then multiplied with the conductance value to result in the new dendrite voltage of the target cell. The texture memory is cached on chip and can provide a higher effective bandwidth by reducing off-chip memory requests. When an application has a memory access pattern with high spatial locality, texture memory can be very effective. A thread is likely to read from an address nearby addresses read by neighboring threads. A Gaussian connectivity network will display a greater deal of spatial locality then a uniform connectivity network, but both cases could benefit from using texture memory.

The dimensions of the texture memory are equal to those of the neural network and do not scale with the GPU-world size. All GPUs have an equally sized texture memory according to the size of the cell network. The population stored in this memory is different for every GPU, since the required neighboring cells vary for different GPUs. Only after each cell-computation phase can the update process of the texture memory be conducted, such that when a new cell-computation phase starts, the new neighboring voltages can be fetched.
Next to the computation of the new voltages for the dendrite compartment, also the next voltages for the soma and the axon compartments are computed in each cell-computation phase. None of the soma or axon computations need voltages from other cells. These computations only need their own voltages from the previous simulation step and an external input and, thus, do not require any data from remote GPUs.

### 4.4 Dendrite Communication

In every simulation step, the dendrite voltages of the neighboring cells on remote GPUs are required by the local GPU to perform the neuron computations. Thus, after each cell-computation step, the newly calculated dendrite voltages are communicated across the GPU-world.

This phase consists of two loops and an `MPI_Alltoall` transfer, shown in Algorithm 4. The first loop gathers the newly calculated dendrite voltages, which are stored according to their global-cell-IDs spread out through the texture memory, and packs them consecutively in an array. When packed, the array can be sent back to the requesting GPUs. Every GPU only sends the necessary data to the requesting GPUs, keeping the sent packet sizes as small as possible. This is a reciprocal operation of the previous `MPI_Alltoallv` function, as used in the connectivity-network dispersal phase; the results are returned to their location of origin. In this phase, the GPU-Direct technology is invoked for the dendrite communication to reduce latency and increase bandwidth. A second loop unpacks the received results, using once again the global-cells-IDs as index. The texture memory is then updated with the required dendrite voltages necessary for the neuron computations in the next simulation step. Both loops are implemented with OpenMP multi-threading to speedup the packing and unpacking of the dendrite voltages. This concludes the process of simulation.

### 5 EVALUATION

The multi-node GPU design is benchmarked for its strong and weak scalability as well as energy efficiency for both uniform and Gaussian connectivity networks of the ION model. Network sizes of 65K, 262K, 1M, and 2M neurons, densities of 10 and 1,000 synapses/neuron (s/n) and GPU-world sizes of 1, 8, 16, 24, and 32 GPU nodes are evaluated, collectively putting up a large parameter space. This space matches and surpasses populations and connectivities encountered in the actual human ION—the largest among mammals—and aims at revealing performance trends of the design and general conclusions for HH models at large [1].

Simulations are run for 2,000 steps, which equals 100 ms of simulated brain time. The C function `gettimeofday` is utilized to get the execution times of the phases that do not use the
GPUs (connectivity-network dispersal and dendrite-communication) but also for the connectivity-network generation. For the computation phase, the function `cudaEventElapsedTime` is utilized to get the GPU execution times. For robustness reasons, simulation results were averaged over three runs and verified by a second researcher.

5.1 Performance Evaluation

Execution time is used as the performance metric for our evaluation. For both uniform- and Gaussian-connected networks, performance is calculated by comparing results (i) for 1M cells when scaling GPU-world size from 1 to 32 GPUs and densities from 10 to 1k (weak-scaling experiments) and (ii) for a high network density of 1k when scaling network sizes from 65k to 4M cells and for GPU-world sizes from 1 to 32 GPUs (strong-scaling experiments). The network size of 4M cells could not be simulated for 1k s/n due to insufficient memory on the GPU. The results mentioned for this size have been extrapolated based on the ratio of the number of neighbors to execution time and have been added for qualitative comparison. Both network connectivity types are compared to highlight the different impacts on performance. The performance of the four phases of the simulation are discussed first, resulting in the overall performance in which the effects of these phases are reflected.

5.1.1 Connectivity-network Generation. Measurements from the connectivity-network generation phase, shown in Figure 5(a) and (b), reveal that generating a Gaussian network, will take more time than a uniform one. This is to be expected, because the Gaussian generator requires more operations and it often needs more iterations to ensure a certain amount of neighbors than the uniform one, as discussed in Section 4.1. The difference becomes more pronounced with increasing network sizes, shown in Figure 5(c) and (d). The fastest execution measured for generating the largest network of 2M cells is roughly 22 s for uniform and 43 s for Gaussian networks in a GPU population of 32.

Figure 5(c) and (d) also reveal that, when the network size increases, performance increases rapidly. The largest reported speedup for uniform-distributed simulations is $\times 22$ and for Gaussian-distributed ones $\times 17$. Adding more GPUs does not only add more cache, memory, and computing cores, but it also reduces GPU-workload sizes. Smaller workloads will fit better in cache, reducing memory transfers.

5.1.2 Connectivity-network Dispersal. The connectivity-network dispersal phase does not directly benefit from a larger GPU-world, because it is not executed on the GPUs. The small overall decrease in execution time observed in the connectivity-network dispersal phase for both generators, shown in Figure 6(a) and (b), is due to the fact that a decreasing amount of cells needs to be communicated across an increasing GPU-world size. Data sizes decrease while the numbers of packets per communication instance increase.

From the results, it can also be derived that the Gaussian network dispersal phase takes significantly longer than the uniform, especially for larger network sizes (mind the scale). Figure 6(c) and (d) show that, for smaller network sizes (65k and 262k), the dispersal time increases when scaling from 8 to 32 GPUs. However, for larger network sizes, there is a decrease in dispersal time when scaling from 8 to 32 GPUs. It indeed agrees with our intuition that fewer GPUs are more efficient with smaller network sizes. In this part of the architecture, the required neighboring cells are communicated from the local to the remote GPUs. In the case of a uniform network, relatively small packets containing cell values, are sent to many remote GPUs. In the case of a Gaussian network, relatively large packets are sent to fewer remote GPUs. For a Gaussian network, it is more likely that some packets are larger than for a uniform network, causing much more transmission overhead.
5.1.3 Cell Computation. In general, the larger the GPU-world size becomes, the smaller the network segment to map per GPU and the fewer computations per GPU needed to be preformed, when the network size remains constant. Simulation results reveal that, for both connectivity-network types, the execution time drops as the GPU-world size grows, shown in Figure 7(a) and (b). The results also reveal that smaller densities benefit less from an increasing GPU-world size, as reported speedups are lower than for larger densities. A uniform network with small densities hardly benefits from increasing the GPU-world size, as dendrite communication times increase and little computation speedup is gained. For a Gaussian network a greater speedup is gained compared to a uniform one, as shown in Figure 7(c) and (d). The reason for this difference between connectivity networks and the diminishing returns for a uniform network, was foreboded in the previous paragraph: The impact of the dendrite-communication phase is larger for uniform than for Gaussian-distributed networks, as is shown next.

5.1.4 Dendrite Communication. Next to the communication of the dendrite voltages, the execution times for gathering and unpacking of the dendrite voltages are also included, which are both parallelized using OpenMP. This phase does not benefit directly from a larger GPU-world size as well, since it is executed on the host CPU cores instead of on the GPUs. For a uniform network, the communication times reported are 3 to 4 times larger than for a Gaussian network,
Fig. 6. Obtained connectivity-network dispersal times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained connectivity-network dispersal times when scaling neuron network size with a density of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graph represent the relation of the transfer time to the 8 GPU-world execution.

as shown in Figure 8(a) and (b). Smaller densities benefit from an increasing GPU-world size as communication times diminish, but for larger densities the communication times increase. As networks grow larger, the communication times increase for both connectivity types, as is shown in Figure 8(c) and (d).

The results observed are opposite to the results from the connectivity-network dispersal phase (see Section 5.1.2). The execution times for the uniform network are larger than for the Gaussian network, which was opposite when looking at the connectivity-network dispersal phase. The packets sent in this phase are larger than in the connectivity-network dispersal phase, because the data sent back are floating-point dendrite voltages instead of cells represented by an integer. In a uniform-distributed network, many remote GPUs send back the gathered voltages to local GPUs. Thus, a uniform network generates more traffic with larger packets. For a Gaussian network, fewer remote GPUs transfer data to local GPUs.

5.1.5 Overall Performance. The total execution time is the sum of all four phases, mentioned above. The results are shown in the following figures.

From the strong-scaling experiments—Figure 9(a) and (b)—we observe that the initial speedup (GPU-world of 8) for a uniform network, is ×4.6 for the smallest, in comparison to ×7.8 for the largest density. The uniform network has a higher initial speedup for the largest density than the Gaussian network and vice versa for the smallest density. For the Gaussian network, the initial speedup is ×6.9 for the smallest and ×7.2 for the largest density.
Fig. 7. Obtained simulation execution times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained simulation execution times when scaling neuron network size with a density of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graph are the speedups of the results in relation to the single GPU-world execution. All graphs also show the execution time for the dendrite communication.

From the weak-scaling experiments, when scaling the network size, the smallest observed overall speedup is \( \times 8 \) and the largest is \( \times 24 \) for uniform networks in relation to a GPU-world of 1, shown in Figure 9(c). For the Gaussian-distributed simulations of Figure 9(d), the smallest observed overall speedup is \( \times 9 \) and the largest is \( \times 28 \), when scaling the network size. For both network distributions, the speedup increases when the network size increases, but a Gaussian network gains a higher overall speedup when increasing the network size. The reason for this is that the dispersal and communication overheads in a uniform network are overall larger than those in a Gaussian one. This is due to the fact that the cell neighbors in the latter are statistically more proximal (in terms of physical GPU locations), which means less communication overhead and more gain from enlisting more GPUs.

5.2 Comparison with a Manycore Architecture

To assess the benefits of the current multi-GPU implementation, we compare it with a multi-node implementation of the ION model on Intel’s manycore Xeon Phi platform (Knights’ Landing (KNL) version) [1]. This version runs the ION model in a cluster on a maximum of eight KNL-equipped nodes. For the KNL, no GPU Direct equivalent is available, and MPI-based communication over Infiniband is used. In this work, the manycore, multi-node implementation was evaluated based on results of the computation phase only; no results for the connectivity-network setup were
Fig. 8. Obtained dendrite communication times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained dendrite communication times when scaling neuron network size with a density of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graph represent the relation of the transfer time to the eight GPU-world execution.

reported, and thus this is the only phase to which the multi-GPU design can be meaningfully compared.

The multi-KNL evaluation reveals that simulating uniform-distributed networks larger than 1M cells result in erratic behavior. The reason given is that performance and scalability are hampered due to data messages being exchanged between cores, especially those belonging to different KNL nodes. The KNL mesh interconnect links some but not all cores among them, creating expensive extra memory reads and writes to get information across nodes and is most probably the underlying architectural element to blame.

In contrast, the multi-GPU design does not appear to have difficulties scaling to larger network sizes, because for both connectivity types every density (10 and 1K) up to 2M cells is simulated. The GPUDirect technology implemented through OpenMPI, which connects the GPUs with the lowest latency possible, shows excellent—near linear—scalability across GPU-world sizes and different network sizes. Larger network sizes do not lead to erratic behavior.

Second, when comparing the execution times, the following results are obtained. For larger densities and network sizes, the multi-KNL design reports execution times of 100–130 s for uniform and of 50–110 s for Gaussian-distributed network simulations. The multi-GPU design reports these execution times when 24–32 GPUs are utilized: For uniform simulations, the execution time is 168 s for 24 and 120 s for 32 GPUs, while for Gaussian simulations the execution time is 134 s for 24 and
Fig. 9. Obtained total execution times when scaling GPU-world size for 1M neurons for densities of 10 and 1k synapses/neuron (s/n). For (a) uniform and (b) Gaussian connectivity networks. (c) Obtained total execution times when scaling neuron network size with a density of 1k synapses/neuron (s/n) for 1 and 32 GPU(s) for uniform and (d) Gaussian connectivity networks. The numbers in the graphs are the speedups of the results in relation to the single GPU-world execution.

98 s for 32 GPUs. Thus, the multi-GPU can match the multi-KNL design on this front, albeit at the cost of more accelerator nodes.

Third, the multi-KNL speedup reported in Gaussian simulations is lower than the respective multi-GPU speedup. For Gaussian simulations, the multi-KNL speedup reported (vs. a single KNL) is \( \times 2.1 \), \( \times 4 \), and \( \times 7.2 \) when scaling from 1 to 2, 4, and 8 KNL nodes, respectively. For the multi-GPU design, a speedup of \( \times 8.3 \) is obtained for a GPU-world size of 8 and a network size of 1M. Thus, the speedup obtained by the multi-KNL design can be matched by the multi-GPU one.

These results serve as an quantitative comparison point; however, GPUs have some significant qualitative differences to KNLs as well. The GPU design shows that network sizes above 2M for a uniform network can still be executed. The cell-computation phase scales more gracefully when the GPU-world size increases. Increasing the GPU-world size does not show any erratic behavior. Judging by the above and by the fact that our GPU design matches or outperforms the KNL design, it must be concluded that, at least for larger network sizes, the multi-GPU design is a good fit for scaling up applications such as the Inferior Olive model and, in general, large-scale eHH representations.

5.3 Energy Evaluation

We next assess the energy efficiency of the multi-GPU design. Here, we calculate energy efficiency as speedup over energy consumption of different neural-network sizes and densities. Note that the
Fig. 10. Multi-GPU speedup normalized by the energy used for uniform for (a) 10 s/n and (b) 1k s/n and for Gaussian (c) 10 s/n and (d) 1k s/n connectivity networks. Note that GPU-world sizes of 48 and 64 have been extrapolated.

The total number of neighbors differs per GPU; this holds for both connectivity types. Thus, especially in the case of Gaussian connectivity, some GPUs will have to process more neighbors than others. These GPUs will consume more energy as well. The registered energy consumption is the total energy consumed on the node. This includes the GPUs, CPUs, Infiniband adapters, and so on. This total energy does not take the cooling of the nodes into account. The energy has not been registered for each GPU individually. The single-GPU system serves as the performance baseline for calculating the speedup figures of the different multi-GPU configurations. It should be noted that the results for GPU-world sizes of 48 and 64 have been extrapolated and added to the graph to observe how the design would behave if the GPU-world were to increase further. This extrapolation was done by projecting the observed linear relation between GPU-worlds of 16 and 32, onto the GPU-worlds of 48 and 64.

For uniform networks, shown in Figure 10(a) and (c), it can be observed that for smaller connectivity densities, it pays off to increase the GPU-world size. The energy used is relatively low compared to the speedup gained. For an increasing network size, however, the energy cost increases and it is more efficient when a GPU-world of 8 is used. This, again, has to do with the communication overhead. When the connectivity density increases in a network with a uniform distribution, the number of neighbors on remote GPUs increases, and, thus, the communication overhead increases, which attenuates the speedup gained from increasing the GPU-world size.

When the network size becomes larger than 2M cells, all GPU configurations become relatively more energy efficient. For uniform connectivities, sparser networks favour larger GPU-world sizes,
and denser networks favour lower GPU-world sizes. This is to be expected due to an increase in communication overheads.

For Gaussian-distributed simulations, shown in Figure 10(b) and (d), energy-efficiency trends are different. For smaller network sizes, smaller GPU-world sizes are more energy efficient with regards to speedup for all densities. When network sizes increase, the GPU-world size of 32 becomes the most energy efficient across the board, followed by 48, 64, 16, and, last, 8. The speedup scored for a GPU-world of 32 is the highest when compared to the energy used.

A crucial observation is that the energy-efficiency magnitudes are much larger here compared to the uniform case, especially for larger densities (1k s/n). Gaussian-distributed simulations can achieve a higher overall speedup than the uniform ones due to the fact that the dendrite-communication phase takes proportionally less time and the speedup of the connectivity-network generation phase is more significant.

5.4 Roofline-model Analysis

We will conclude our evaluation by performing a typical roofline-model analysis, which is a visual performance model well suited for manycore and multi-node architectures. Operational intensity (OI)—or, simply, intensity—is defined as the ratio of floating-point operations (FLOP) to memory operations \[26\] for a given compute kernel; the ION model, in our case. The number of double-precision FLOP has been derived from the NVIDIA NVPROF profiling application; the profiler has a FLOP-counter metric output (\texttt{flop\_count\_dp} for double precision and \texttt{flop\_count\_sp} for single precision). The number of memory operations has been derived by summing the write and read accesses from and to the DRAM of the GPU for which the profiler also has counters (\texttt{dram\_read\_transactions} and \texttt{dram\_write\_transactions}). The DRAM, which is a GDDR5 in the K40M card, is the physical device memory residing on the GPU which is accessed on Level-2 cache misses. The OI is computed as follows:

\[
OI = \frac{\text{FLOP}}{(\text{DRAM\_reads} + \text{DRAM\_write}) \times (\text{sizeof\_transaction} = 32\text{Bytes})}
\]

Based on this formula, in Figure 11 rooflines have been plotted for the K40M and V100 GPUs regarding the simulation kernel. The V100 roofline is also shown, because the multi-GPU design can benefit greatly from the recently released and faster V100 GPUs. For comparing between GPU and manycore architectures, the KNL roofline has also been added. Last, to make clear the computational impact of the simulation kernel of the ION model (i.e., communication is excluded), it has been separated into a compute kernel and a neighbor kernel. The compute kernel solely represents the cell-computation kernel, doing computations on the three compartments of the neuron, but without the intensive fetching and computation of the dendrite voltages from neighboring cells. The neighbor kernel represents solely the fetching and computation of the dendrite voltages. This distinction shows the difference in the attainable performance between the two parts of the overall simulation kernel. In both plots, bandwidth ceilings, in-core ceilings, and locality walls are also shown.

Figure 11(a) shows that the OI for the simulation (neighbor & compute) kernel is 1.36 and the kernel is, therefore, memory bound for all architectures, meaning that this part of the design is mostly busy with reading and writing to memory. When compared to the ridge point of the roofline, which for the K40m GPU is at 1.43T/288GB/s = 4.96, the OI is at 33% of its peak value. An architecture is considered memory bound, when the OI is below the ridge point. The attainable performance for the K40m is 211 GFLOP/s, for the KNL it is 294 GFLOP/s, and for the V100 it is 661 GFLOP/s.
The neighbor kernel scores a higher performance (918 GFLOP/s) than the compute kernel (279 GFLOP/s), because it performs less floating-point operations and DRAM write-operations while the DRAM read-operations roughly stay equal, compared to both kernels.

Next to a roofline model of the simulation kernel, a model of the uniform and Gaussian generators has been created to verify their behavior, the results of which are shown in Figure 11(b). These generator kernels only handle single-precision floating-point operations. The flop_count's are invariant of the GPU-world, as opposed to the compute kernel. The dimensions of the kernels do not grow with GPU-world value. Only the frequency of kernel execution scales when GPU-world increases, but this has no effect on the OI.

Also, the number of FLOPs does not scale at the same rate as the DRAM data-movement rate when network sizes vary. This can be explained by the fact that the probability that a neighbor is found reduces when network size increases. When network size increases, the kernel is executed more often but less neighbors need to be found to ensure a steady density. The main contributor to generator-kernels' FLOPs, which determines the OI, is the cuRAND random-number generator, which is executed for every cell. As network size increases, the probability for finding neighbors decreases and less DRAM memory needs to be written, resulting in an increase in OI.

For a network-size of 256k cells, the uniform-generator OI is 10.11 FLOP/byte and rises due to the fact that the FLOP count increases more rapidly than the DRAM data-movement rate. For network sizes of 512k, 1M, and 2M, the uniform OI is equal to 32, 109, and 374, respectively. The same holds for the Gaussian generator; the Gaussian OI for 256k cells is equal to 28 and then for 512k cells to 37 and increases even more steeply than the uniform OI. It should be no surprise that the Gaussian-generator kernel has a higher and faster-rising OI than the uniform one. This is because, for this kernel, more FLOPs take place for roughly the same amount of DRAM data movements. Besides, both generators are compute bound, meaning that the performance is limited by the compute power of the GPUs.
6 DISCUSSION

From the evaluation of the multi-GPU implementation it is observed that, for larger densities, a uniform network displays a lower overall speedup when compared to its Gaussian counterpart. The dendrite-communication phase becomes the performance bottleneck of the simulation process as traffic among GPU nodes increases in volume. However, this communication overhead does not dominate the overall execution while scaling network sizes is tractable. The ION-model workload has proven to execute on as many GPUs as available and no erratic behavior (as in the case of the multi-KNL design) has been noticed. The multi-KNL design shows erratic behavior for large uniformly distributed, and even more for Gaussian-distributed networks, due to the performance penalties incurred for keeping the many CPU caches of the KNL architecture coherent.

The energy analysis shows that, for uniform networks, the smallest tested GPU-world size of 8 is more beneficial for larger network sizes. For Gaussian networks, a GPU-world size of 32 GPUs is more beneficial in relation to speedup for both measured densities. The observed behaviour is related to the utilization of the GPUs and the inter-node communication overhead. As network sizes go up, different GPU-world sizes become more energy efficient due to the fact that the GPU resources are better utilized.

From the roofline analysis, it follows that the compute kernel is memory bound due to the gap-junction calculations. The uniform generator kernel is memory bound and the Gaussian kernel is compute bound. The compute kernel is memory bound due to the fact that, for every cell which is a GPU thread, every other connected cell needs to be taken into account. Compared to the computations executed by this kernel, the memory fetches of the dendrite voltages dominate this phase.

Every brain simulator which has a communication profile like the ION model could benefit from the technology described in this work. Code generators/simulators such as GeNN and ANNarchy, which let users introduce their own neuron and synapse models, could easily adopt the modular designed connectivity-network generators and dispersal algorithms to setup and simulate their produced CUDA code. Another simulator which could benefit, is CoreNeuron. It supports GJ coupling with the fixed-time-step method, meaning that after each time step an update phase needs to also take place for providing the neurons with the latest values. The low-latency implementation presented in this work could fit very well in this scenario.

Moreover, projects where mathematical models need to be calculated will benefit from multi-GPU setups with low intra- or interconnect latencies. Examples of such projects include the numerical finite-element method used to solve problems in the area of heat transfer and the derivation of electromagnetic fields or fluid dynamics. A multi-GPU design could target the process of first computing and then recombining the finite elements back into a global system. Also, multi-agent systems where nodes have intrinsic dynamics and some coupling functions, such as modelling social systems or epidemiology, are also fields which could benefit severely from a true multi-GPU design such as the one presented here.

7 CONCLUSIONS

In this work, the setup and simulation of a computationally challenging SNN from the field of extended Hodgkin-Huxley models implemented on a multi-node GPU setup has been presented. Next to the actual simulation process, the network-setup phase—often neglected but equally significant—has also been studied and sped up. Four main phases have been identified and implemented. With the aid of OpenMPI and CUDA, these phases have been parallelized, while GPUDirect RDMA has been used to instantiate low-latency and high-bandwidth communication among the GPU nodes. When using GPUDirect, the design displays excellent scalability between number of GPU nodes increasing network sizes. The performance of the design indeed is upper-bounded
by the number of available GPUs in the cluster. More specifically, all phases running on GPUs show speedups when the number of GPUs is increased. This scalable design gives a good prospect for neuroscientists, proving that large-network-size simulations are attainable using a multi-GPU design. The large-scale network capabilities displayed by these simulations are not only relevant for achieving ION simulations at full scale but also for the whole Olivocerebelar system that is comprised of tens of millions of neurons. This article does not limit itself to brain simulations only; it also highlights the sheer power that GPUs have in combination with a sound interconnect technology like GPUDirect.

ACKNOWLEDGMENTS

This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

REFERENCES

[1] G. Chatzikonstantis, H. Sidiropoulos, C. Strydis, M. Negrello, G. Smaragdos, C. I. De Zeeuw, and D. J. Soudris. 2018. Multinode implementation of an extended Hodgkin-Huxley simulator. Neurocomputing 329 (2018), 370–383. DOI: https://doi.org/10.1016/j.neucom.2018.10.062

[2] Ting-shuo Chou, Hirak J Kashyap, Jinwei Xing, Stanislav Listopad, Emily L Rounds, Michael Beyeler, Nikil Dutt, and Jeffrey L Krichmar. 2018. CARLsim 4 : An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN'18). 1158–1165.

[3] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. 5, 1 (1998), 46–55.

[4] Andrew P. Davison. 2008. PyNN: A common interface for neuronal network simulators. Front. Neuroinform. 2 (Jan. 2008), 1–10. DOI: https://doi.org/10.3389/neuro.11.011.2008

[5] Jornt R. de Gruijl, Paolo Bazzigaluppi, Marcel T.G. de Jeu, and Chris I. de Zeeuw. 2012. Climbing fiber burst size and olivary sub-threshold oscillations in a network setting. PLoS Comput. Biol. 8, 12 (2012), 1–10. DOI: https://doi.org/10.1371/journal.pcbi.1002814

[6] Chris De Zeeuw, Freek Hoebeek, Laurens Bosman, Martijn Schonewille, Laurens Witter, and Sebastiaan Koekkoek. 2011. Spatiotemporal firing patterns in the cerebellum. Nat. Rev. Neurosci. 12 (2011), 327–344. DOI: 10.1038/nrn3011

[7] Hoang Du Nguyen. 2013. Gpu-based Simulation of Brain Neuron Models. Master’s thesis. Technical University of Delft.

[8] Wulfram Gerstner and Werner Kistler. 2002. Spiking Neuron Models: An Introduction. Cambridge University Press, New York, NY.

[9] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural simulation tool). Scholarpedia 2, 4 (2007), 1430.

[10] Dan Goodman and Romain Brette. 2008. Brian: A simulator for spiking neural networks in Python. Front. Neuroinform. 2 (2008), 5.

[11] Michael Hines, Sameer Kumar, and Felix Schürmann. 2011. Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. Front. Comput. Neurosci. 5 (2011), 49. DOI: 10.3389/fncom.2011.00049

[12] M. L. Hines and N. T. Carnevale. 1997. The NEURON simulation environment, neural computation. Pubmed 9, 6 (1997), 1–26.

[13] Roger V. Hoang, Deyvani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris. 2013. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. Front. Neuroinform. 7 (Oct. 2013), 19. DOI: https://doi.org/10.3389/fninf.2013.00019

[14] Eugene M. Izhikevich. 2004. Izhikevich2004-which model to use for cortical spiking neurons. IEEE Trans. Neural Netw. 15, 5 (2004), 1063–1070. DOI: https://doi.org/10.1109/TNN.2004.832719

[15] James C. Knight and Thomas Nowotny. 2018. GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. Front. Neurosci. 12 (2018), 941. DOI: https://doi.org/10.3389/fnins.2018.00941

[16] Jiri Kraus. 2013. An Introduction to CUDA-Aware MPI. Retrieved from https://devblogs.nvidia.com/introduction-cuda-aware-mpi/.

[17] Pramod Kumbhar, Michael Hines, Jeremy Fouriaux, Aleksandr Ovcharenko, James King, Fabien Delalandre, and Felix Schürmann. 2019. CoreNEURON: An optimized compute engine for the NEURON simulator. Front. Neuroinform. 13 (2019), 63. DOI: https://doi.org/10.3389/fninf.2019.00063

ACM Transactions on Architecture and Code Optimization, Vol. 16, No. 4, Article 53. Publication date: December 2019.
Exploring Complex Brain-Simulation Workloads on Multi-GPU Deployments

53:25

[18] H. Lasn, B. Winblad, and N. Bogdanovic. 2001. The number of neurons in the inferior olivary nucleus in Alzheimer’s disease and normal aging: A stereological study using the optical fractionator. J. Alzheim. Dis. 3, 2 (2001), 159–168. DOI: https://doi.org/10.3233/JAD-2001-3201

[19] Wolfgang Maass. 1996. Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons. In Proceedings of the 9th International Conference on Neural Information Processing Systems (NIPS’96). MIT Press, Cambridge, MA, 211–217. http://dl.acm.org/citation.cfm?id=2998981.2999011

[20] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. Neur. Netw. 10, 9 (1997), 1659–1671. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7

[21] Micheal V. Mascagni. 1989. Numerical methods for neuronal modeling. In Methods in Neuronal Modeling, Christof Koch and Idan Segev (Eds.). MIT Press, Cambridge, MA, 439–484. http://dl.acm.org/citation.cfm?id=94605.94628

[22] Rene Miedema. 2019. flexHH: A Flexible Hardware Library for Hodgkin-Huxley-based Neural Simulations. Master’s thesis. Technical University of Delft.

[23] Kirill Minkovich, Corey M. Thibeault, Michael John O’Brien, Alexkey Nogin, Youngkwan Cho, and Narayan Srinivasa. 2014. HRLSim: A high performance spiking neural network simulator for GPGPU clusters. IEEE Trans. Neur. Netw. Learn. Syst. 25, 2 (2014), 316–331. DOI: https://doi.org/10.1109/TNNLS.2013.2276056

[24] Mario Negrello, Pascal Warnaar, Vincenzo Romano, Cullen B. Owens, Sander Lindeman, Elisabetta Lavarone, Jochen K. Spanke, Laurens W. J. Bosman, and Chris I. De Zeeuw. 2019. Quasiperiodic rhythms of the inferior olive. PLOS Comput. Biol. 15, 5 (05 2019), 1–41. DOI: https://doi.org/10.1371/journal.pcbi.1006475

[25] NVIDIA. 2010. GPUDirect. Retrieved from https://developer.nvidia.com/gpudirect.

[26] Performance Portability. 2018. Measuring Roofline Quantities on NVIDIA GPUs. Retrieved from http://performanceportability.org/perfport/measurements/gpu/.

[27] S. Plimpton. 1995. LAMMPS—Fast parallel algorithms for short-range molecular dynamics. J. Comp. Phys. 117, 1 (1995), 1–19.

[28] Romelia Salomon-Ferrer, David A. Case, and Ross C. Walker. 2013. An overview of the Amber biomolecular simulation package. Comput. Molec. Sci. 3, 2 (2013), 198–210. DOI: https://doi.org/10.1002/wcms.1121 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1121

[29] Georgios Smaragdos, Georgios Chatzikonstantis, Rahul Kukreja, Harry Sidiropoulos, Dimitrios Rodopoulos, Ioannis Soudris, Zaid Al-Ars, Christofoerchos Kachris, Dimitrios Soudris, Chris I. De Zeeuw, and Christos Strydis. 2017. Brain-Frame: A node-level heterogeneous accelerator platform for neuron simulations. J. Neur. Eng. 14, 6 (2017), 66008. DOI: https://doi.org/10.1088/1741-2552/aa7fc5

[30] Georgios Smaragdos, Craig Davies, Christos Strydis, Ioannis Soudris, Catalin Ciobanu, Oskar Mencer, and Chris De Zeeuw. 2014. Real-time olivary neuron simulations on dataflow computing machines. In Supercomputing. Lecture Notes in Computer Science, Vol. 8488. DOI: https://doi.org/10.1007/978-3-319-07518-1_34

[31] Marcel Stimberg, Dan F. M. Goodman, Victor Benichoux, and Romain Brette. 2013. Brian 2 - the second coming: Spiking neural network simulation in Python with code generation. In BMC Neuroscience. BioMed Central, 1471–2220.

[32] Surfsara. 2016. Cartesius: the Dutch supercomputer. Retrieved from https://userinfo.surfsara.nl/systems/cartesius.

[33] Mellanox Technologies. 2013. Accelerating High Performance Computing with GPUDirect RDMA. Retrieved October 14, 2019 from http://on-demand.gputechconf.com/gtc/2013/webinar/gtc-express-gpudirect-rdma.pdf.

[34] M. M. ten Brinke, H. J. Boele, and C. I. De Zeeuw. 2019. Conditioned climbing fiber responses in cerebellar cortex and nuclei. Neurosci. Lett. 688 (2019), 26–36. DOI: https://doi.org/10.1016/j.neulet.2018.04.035 The Cerebellum in Health and Disease.

[35] Julien Vitay, Helge U. Dinkelbach, and Fred H. Hamker. 2015. ANNarchy: A code generation approach to neural simulations on parallel hardware. Front. Neuroinform. 9 (Jul. 2015), 1–20. DOI: https://doi.org/10.3389/fninf.2015.00019

[36] Nora Vrieler, Sebastian Loyola, Yasmin Yarden-Rabinowitz, Jesse Hoogendorp, Nikolay Medvedev, Tycho M. Hoogland, Chris I. De Zeeuw, Erik De Schutter, Josef Yarom, Mario Negrello, Ben Torben-Nielsen, and Marylka Yoe Uusisaari. 2019. Variability and directionality of inferior olive neuron dendrites revealed by detailed 3D characterization of an extensive morphological library. Brain Struct. Funct. 224, 4 (01 May 2019), 1677–1695. DOI: https://doi.org/10.1007/s00429-019-01859-z

[37] Esin Yavuz, James Turner, and Thomas Nowotny. 2016. GeNN: A code generation framework for accelerated brain simulations. Sci. Rep. 6, 1 (2016), 18854. DOI: https://doi.org/10.1038/srep18854

Received May 2019; revised October 2019; accepted November 2019