Multi-GPU Brain:
A multi-node implementation for an extended Hodgkin-Huxley simulator

M.A. van der Vlag

CE-MS-2019-01

Delft University of Technology

Abstract

Current brain simulators do no scale linearly to realistic problem sizes (e.g. >100,000 neurons), which makes them impractical for researchers. The goal of the thesis is to explore the use of true multi-GPU acceleration on computationally challenging brain models and to assess the scalability of such models given sufficient access to multi-node acceleration platforms. The brain model used is a state-of-the-art, extended Hodgkin-Huxley, biophysically-meaningful, three-compartmental model of the inferior-olivary nucleus. Not only the simulation of the cells, but also the setup of the network is taken into account when designing and benchmarking the multi-GPU version. For network sizes varying from 65K to 4M cells, 10, 100 and 1000 synapses per neuron are simulated. These simulations are executed on 8, 16, 32 and 48 GPUs. A Gaussian-distributed network of 4 million cells with a density of 1,000 synapses per neuron executed on 48 GPUs, is setup and simulated in 465.69 seconds, of which the cell-computation phase takes 4.57 seconds, obtaining a speedup of 50 times the execution time on a single GPU. A uniform-distributed network of same size and density is setup and simulated in 889.89 seconds of which the cell-computation phase takes 10.09 seconds, obtaining a speedup of 8 times the execution on a single GPU. For the implemented design, the inter-GPU communication becomes the major bottleneck, as latency increases when the sent packet sizes increase. This communication overhead does not dominate the overall execution while scaling network sizes is tractable. This scalable design gives a good prospect for neuroscientists, proving that large network-size simulations are possible, using a multi-GPU setup.

TU Delft Delft University of Technology

Quantum & Computer Engineering

Multi-GPU Brain simulator

Cluster implementation for an extended Hodgkin-Huxley simulator

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Michiel Alexander van der Vlag
born in Leeuwarden, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Multi-GPU Brain simulator

by Michiel Alexander van der Vlag

Abstract

Current brain simulators do no scale linearly to realistic problem sizes (e.g. >100,000 neurons), which makes them impractical for researchers. The goal of the thesis is to explore the use of true multi-GPU acceleration on computationally challenging brain models and to assess the scalability of such models given sufficient access to multi-node acceleration platforms. The brain model used is a state-of-the-art, extended Hodgkin-Huxley, biophysically-meaningful, three-compartmental model of the inferior-olivary nucleus. Not only the simulation of the cells, but also the setup of the network is taken into account when designing and benchmarking the multi-GPU version. For network sizes varying from 65K to 4M cells, 10, 100 and 1000 synapses per neuron are simulated. These simulations are executed on 8, 16, 32 and 48 GPUs. A Gaussian-distributed network of 4 million cells with a density of 1,000 synapses per neuron executed on 48 GPUs, is setup and simulated in 465.69 seconds, of which the cell-computation phase takes 4.57 seconds, obtaining a speedup of 50 times the execution time on a single GPU. A uniform-distributed network of same size and density is setup and simulated in 889.89 seconds of which the cell-computation phase takes 10.09 seconds, obtaining a speedup of 8 times the execution on a single GPU. For the implemented design, the inter-GPU communication becomes the major bottleneck, as latency increases when the sent packet sizes increase. This communication overhead does not dominate the overall execution while scaling network sizes is tractable. This scalable design gives a good prospect for neuroscientists, proving that large network-size simulations are possible, using a multi-GPU setup.

Laboratory : Computer Engineering
Codenumber : CE-MS-2019-01

Committee Members :

Advisor: dr. ir. C. Strydis, CE, Erasmus MC Neuroscience

Chairperson: dr. ir. Z. al-Ars, CE, TU Delft

Member: dr. ir. A. van Genderen, CE, TU Delft

Member: ir. G. Smaragdos, CE, Erasmus MC Neuroscience

I would like to thank my supervisors Christos Strydis, Georgios Smaragdos and Zaid Al-Ars for showing me the ropes and their guidance, and the entire Erasmus research group for making me feel at home. Special thanks go out to my wife Irina and my children Noralie and Helena, who have been very supportive and patient during my study, letting me fulfill my dream.

Contents

List of Figures viii

List of Tables ix

1 Introduction **1**
1.1 Problem statement . . . 1
1.2 Overview . . . 2

2 Background **3**
2.1 Spiking Neural Networks . . . 3
2.2 Inferior-olivary nucleus . . . 4
2.3 Multi-node accelerator technology . . . 6
2.3.1 GPU technology . . . 6
2.3.2 GPUdirect . . . 8
2.3.3 Xeon Phi technology . . . 10
2.4 Cluster technology . . . 11
2.4.1 Cartesius . . . 12
2.4.2 SLURM . . . 13
2.4.3 Amazon Web Services . . . 14
2.5 Summary . . . 16

3 Related work: Neural network simulators **19**
3.1 SpiNNaker . . . 19
3.2 Brian . . . 19
3.3 DynaSim . . . 20
3.4 CARLsim . . . 21
3.5 NEST . . . 22
3.6 (Core)Neuron . . . 22
3.7 HRLSim . . . 23
3.8 Neurokernel . . . 25
3.9 NCS6 . . . 25
3.10 STEPS . . . 26
3.11 PSICS . . . 27
3.12 Summarizing . . . 27

4 Design **29**
4.1 Single-GPU version . . . 29
4.2 Design Overview . . . 31
4.3 Regression testing / proof of concept . . . 34
4.4 Connection-generation . . . 35

4.4.1 The cell connection matrix . . . 35
4.4.2 Connections from an external file . . . 37
4.4.3 Connections internally generated . . . 38
4.5 Cell-id-dispersal . . . 44
4.6 Cell-computation . . . 47
4.7 Dendrite-communication . . . 48
4.8 Summary . . . 49

5 Evaluation **51**
5.1 Experimental setup . . . 51
5.2 Connection-generation . . . 52
5.3 Cell-id-dispersal . . . 55
5.4 Cell-computation . . . 56
5.4.1 KNL comparison . . . 59
5.5 Dendrite communication . . . 60
5.6 Total execution time . . . 62
5.7 Energy consumption . . . 65
5.7.1 Energy efficiency . . . 67
5.8 Limitations . . . 68
5.8.1 Memory . . . 68
5.8.2 Roofline model analysis . . . 71

6 Conclusions **75**
6.1 Contributions . . . 76
6.2 Future work . . . 77

List of Figures

2.1 The Olivocerebellar circuitry [1] 4
2.2 Three-compartment dynamics of the IO cell [2] 5
2.3 A neuron pair connected through a synapse [3] 5
2.4 Threads, blocks and grid related to processing structure on the GPU [4] 7
2.5 Hardware model showing SMs and memory resources [5] 7
2.6 Memory transfer paths [6] 9
2.7 Latency specification for GPUdirect on Cartesius [7] 10
2.8 Bandwidth specification for GPUdirect on Cartesius [7] 10
2.9 Slurm components [8] 13
2.10 Burstable AWS architecture using SLURM elastic [9] 16

3.1 Overview characteristics of multi-node brain simulators 28

4.1 Graphical representation of the inferior-olivary network model. (a) 8-neuron network (b) single-neuron model in detail [3] 30
4.2 GPU implementation of the InfOli application [3] 31
4.3 Main flowchart indicating architecture, process and complexity 32
4.4 Workload division of neuron cells across GPUs for 2 and 4 GPUs 33
4.5 Communication interconnections with GPU_worlds 2 and 4 33
4.6 For the uniform-distribution the neighbor-connections are spread more evenly throughout the network, while Gaussian keeps them nearby . . . 36
4.7 Sparsely populated connection matrices for a 4x4 cell network on 1 and 2 GPU('s) 36
4.8 Thread variation for larger and smaller networks using Gaussian generator on OpenMP 41
4.9 Uniform network generation on 6 Xeon cores versus 2 and 16 GPUs . . . 45
4.10 Graphical representation of MPI_Alltoallv function [10] 46

5.1 Execution times for uniform (U) and Gaussian (G) connection- generation phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right) 53
5.2 Execution times for uniform (U) and Gaussian (G) connection- generation phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right) 53
5.3 Execution times for uniform (U) and Gaussian (G) cell-id-dispersal phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right) 56
5.4 Execution times for uniform (U) and Gaussian (G) cell-id-dispersal phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right) 56

5.5 Execution times for uniform (U) and Gaussian (G) cell-computation phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right) 57
5.6 Execution times for uniform (U) and Gaussian (G) cell-computation phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right) 57
5.7 Execution times for uniform (U) and Gaussian (G) dendrite-communication phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right) 61
5.8 Execution times for uniform (U) and Gaussian (G) dendrite-communication phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right) 62
5.9 Total execution times for uniform-distribution for all sizes and densities. Groups with the same colors with different shades represent the GPU_worlds (grey = 1, blue = 8, red = 16, green = 32 and brown = 48) 63
5.10 Total execution times for Gaussian-distribution for all sizes and densities. Groups with the same colors with different shades represent the GPU_worlds (grey = 1, blue = 8, red = 16, green = 32 and brown = 48) 63
5.11 Energy consumption for uniform (U) and Gaussian (G) for all phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right) . 66
5.12 Energy consumption for uniform (U) and Gaussian (G) for all phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right) 66
5.13 Speedup in relation to the energy used for uniform distributions 67
5.14 Speedup in relation to the energy used for Gaussian distributions 69
5.15 Large network behavior for GPU_world 32 and density of 1000 synapses/neuron 71
5.16 Roofline model for K40M, KNL, V100 for the compute-kernel 72
5.17 Roofline model for K40M, KNL, V100 for the generator-kernels 73

List of Tables

2.1 GPU partition overview for Cartesius 14
2.2 AWS P3 compute instance overview 15
2.3 Comparison of important architecture specifications 17

5.1 Parameter space for InfOli benchmark 52
5.2 Speedup for uniform connection-generation - 10 synapses / neuron . . . 54
5.3 Speedup for uniform connection-generation - 100 synapses / neuron . . . 54
5.4 Speedup for uniform connection-generation - 1k synapses / neuron . . . 54
5.5 Speedup for Gaussian connection-generation - 10 synapses / neuron . . . 54
5.6 Speedup for Gaussian connection-generation - 100 synapses / neuron . . 54
5.7 Speedup for Gaussian connection-generation - 1k synapses / neuron . . . 55
5.8 Speedup for uniform cell-computation - 10 synapses / neuron 58
5.9 Speedup for uniform cell-computation - 100 synapses / neuron 58
5.10 Speedup for uniform cell-computation - 1k synapses / neuron 59
5.11 Speedup for Gaussian cell-computation - 10 synapses / neuron 59
5.12 Speedup for Gaussian cell-computation - 100 synapses / neuron 59
5.13 Speedup for Gaussian cell-computation - 1k synapses / neuron 59
5.14 Speedup for total execution time - 10 synapses/neuron - uniform-distribution 64
5.15 Speedup for total execution time - 100 synapses/neuron - uniform-distribution 64
5.16 Speedup for total execution time - 1K synapses/neuron - uniform-distribution 64
5.17 Speedup for total execution time - 10 synapses/neuron - Gaussian-distribution 65
5.18 Speedup for total execution time - 100 synapses/neuron - Gaussian-distribution 65
5.19 Speedup for total execution time - 1K synapses/neuron - Gaussian-distribution 65
5.20 Theoretically possible network sizes in relation to density for GPU_world 16-64 70
5.21 Results increased block sizes K40M 74

1 Introduction

The brain perhaps is *the* central mystery of human life. Biological questions about the brain are often related to medical investigations and are practical by nature. Neurobiologists could also ask questions such as: how can diseases such as Alzheimer or epilepsy be cured? Or, why does the brain respond to this stimulus in this particular way? While directly accessing various parts of the brain would help better to unravel such diseases, current methods are either potentially unethical or have limited physical accessibility, especially in deep brain structures. Many of *in-vivo* brain related experiments on living human test subjects are considered unethical and have been long forbidden. Also, results could easily be contaminated by environmental factors.

An alternative for in-vivo brain experiments is mathematical brain modeling on a computer, called *in-silico* experiments. The brain or part of the brain, consisting of cells the behavior of which can be expressed as mathematical models, could be simulated on a computer. In-silico experiments give insights on for example, single-cell behavior or network dynamics of the whole brain. These experiments could deliver breakthroughs, not only towards curing brain diseases, but also in drug development and the replacement of animal experiments. On a broader scale, these experiments could one day also answer more fundamental questions regarding human life.

At Erasmus Medical Centre, a research group under the name *Erasmus Brain Project* led by dr. Christos Strydis (www.erasmusbrainproject.com) is working on, among other things, *the acceleration of large-scale brain simulations*. This project focuses mainly on cerebellar models in which various high-performance node technologies such as FPGAs, GPUs, dataflow engines or multi-core processors such as the Xeon Phi are explored for delivering largely scalable, high-speed simulation platforms. The end-goal is to provide scientists with a service that integrates the aforementioned technologies and selects the most suitable platform for the task, depending on a variety of workload characteristics [1].

1.1 Problem statement

Neuroscientific researchers often use in silico Spiking Neural Network (SNN) simulators to validate their hypotheses and reproduce certain brain dynamics. The larger and more complex a simulation becomes, the more computationally intensive it will be. Current simulators do no scale linearly to realistic problem sizes (e.g. >100,000 neurons), which makes them impractical for researchers. Also, these simulators take the easy way out, using single or multiple (but isolated) GPUs to perform large-scale simulations. There is much to be gained by performing true multi-GPU simulations now that the technology is finally here.

The goal of the thesis is to explore the use of true multi-GPU acceleration on com-

putationally challenging brain models and to assess the scalability of such models given sufficient access to multi-node acceleration platforms. The performance of multiple truly interconnected GPUs is explored, made only recently possible through enabling NVidia technologies, for simulating large-scale, realistic, compute-intensive neural models. As a benchmark an in-house, extended Hodgkin-Huxley, three-compartmental model of the Inferior-olivary nucleus of the brain is used, which jointly with the cerebellum is responsible for crucial functionality involving motor control, sensorimotor integration, dexterity and more. Not only the simulation of the cells, but also the setup of the network is taken into account when designing a multi-GPU version.

Based on the above problem statement, the mission in this thesis is to tackle the following technical goals:

1. Analyze current SNN model and single-GPU implementation
2. Perform extensive literature survey on existing multi-GPU simulators, platforms and libraries.
3. Select the most optimal candidate and adopt/extend to fit our advanced SNN model's needs.
4. Design, implement and deploy multi-GPU simulator
5. Evaluate performance, scalability and energy-consumption aspects
6. Propose future directions for multi-GPU SNN simulations based on findings.

1.2 Overview

This thesis is structured as follows. Chapter 2 introduces neuroscientific background knowledge necessary to understand the problem. It also discusses multi-node technology necessary for tackling the SNN simulation problem. It especially describes a prior research project of Erasmus which ported the same application to a many-core CPU architecture. This will serve as comparison basis for the evaluation of the design. The chapter concludes with a description of the clusters to which the algorithm will be ported.

Chapter 3 gives an overview of the related work of multi-node SNN simulators, taxonimized according to necessary aspects and optimization technologies used. At the end of this chapter, a table is displayed with an overview of the different brain simulators.

Chapter 4 delineates the main architecture of the multi-GPU implementation. It analyses the currently in the lab existing single-GPU implementation and unveils bottlenecks. It describes the setup of the network and its synapses and the efforts to optimize possible performance bottlenecks. The GPU inter-node communication during setup, the dispersal of the neighboring-neuron information, as well as the communication during simulation are discussed. Lastly, the computation of the neighboring neurons is explained.

Chapter 5 provides an evaluation of the developed simulator to illustrate the speedup obtained from employing a multi-GPU implementation. It depicts the four phases of the simulation, which are encountered in order: the neuron cell-network generation, the cell-identification (cell-id) dispersal, the computation of the cells and the dendrite voltage communication during simulation. These four phases will be illustrated and discussed.

In Chapter 6, conclusions are discussed and suggestions for future work are made.

Background

2

This chapter describes the neuroscientific knowledge necessary for understanding the simulator application and the related work chapter. It focuses on GPU in general and how GPUs could communicate. Two clusters, the Dutch supercomputer called Cartesius and the Elastic cloud computing facility by Amazon Web Services (AWS), are discussed. Because of the similarity to the current research, a research project to port the InfOli application to the Xeon Phi architecture, is introduced in this chapter. This research is discussed outside the chapter for the related work and will serve as a format for the benchmark. This chapter is split up into parts of neuroscientific and engineering related matters.

2.1 Spiking Neural Networks

A *spiking neural network* (SNN) is a type of neuron modelling that mimics biological neural network processes. Next to individual neuronal and synaptic states, an SNN is also dependent on time. When a neuron fires, a signal is generated which travels to other neurons which in turn, in- or decrease their potentials. This in- or decreasing of firing rate across neurons, creating complex patterns when the group becomes larger, is generally considered to correspond to the interchange of information. Neurons can exhibit up to twenty different spiking/firing patterns [11]. As a result they can replicate advanced biological systems. Neurons can be connected to other neurons by a structure called *synapse*. A synapse enables neurons to transmit chemical or electrical signals to other neurons.

There are three main categories for neuron models for SNN: *Leaky Integrate and Fire (LIF), Izhikevich (IZH)* and conductance based models of which *Hodgkin-Huxley (HH)* is a familiar example. The LIF model is widely used. When the membrane potential reaches a certain threshold value, the neuron fires a spike and threshold is reset to another value. It is a simple model because it can only exhibit a few spiking patterns.

The IZH spiking model can display all of the twenty firing patterns. Next to the membrane potential this model also inhibits a membrane recovery variable. When the spike reaches its apex, the membrane and recovery variable are reset according to an equation. It still is a simple model with two state variables and four parameters.

The HH model is a conductance based model. It consists of 4 equations and tens of parameters describing membrane potentials, activation or inactivation of Na and K currents. This latter model is bio-physically meaningful and often demands a lot computational power. The Olivocerebellar model example in [12], 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. The modelling accuracy of this model is at the cell conductance level as is displayed by the HH models. In fig. 2.1 the Olivocerebellar circuitry

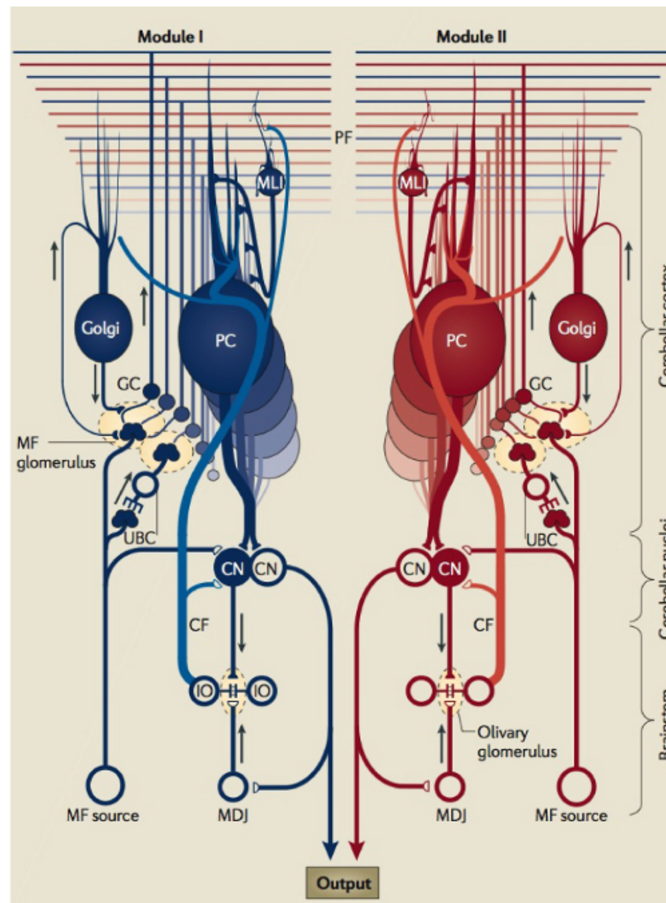

Figure 2.1: The Olivocerebellar circuitry [1]

is shown. Its brain structure is highly repetitive and basically consists of the granule-cell layer (GCL), Purkinje-cell layer (PC), deep-cerebellar-nuclei (DCN), and inferior-olive (IO) nuclei [1].

2.2 Inferior-olivary nucleus

The inferior-olivary (IO) nucleus is one of the most dense regions of the olivocerebellar system and the neuron of this region are densely interconnected through *gap junctions* (GJ). Gap junctions are electrical synapses that allow for rapid communication in the form of electrical current between two or more neurons. They are considered to be complex connections associated with important aspects of cell behavior [3].

Neurons can be modeled with multiple compartments. Neurons for the IO in [3] for instance, are modeled with three compartments: the dendrite, soma and axon according to [13]. In fig. 2.2 a three-compartment representation of a cell from the IO with GJ's is shown. These compartments have electro-chemical characteristics which influences the overall state of the neuron.

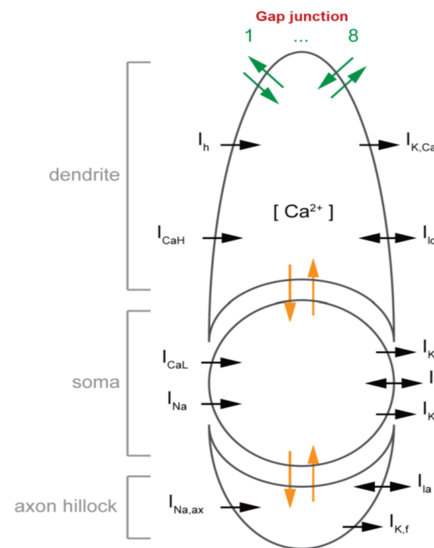

Figure 2.2: Three-compartment dynamics of the IO cell [2]

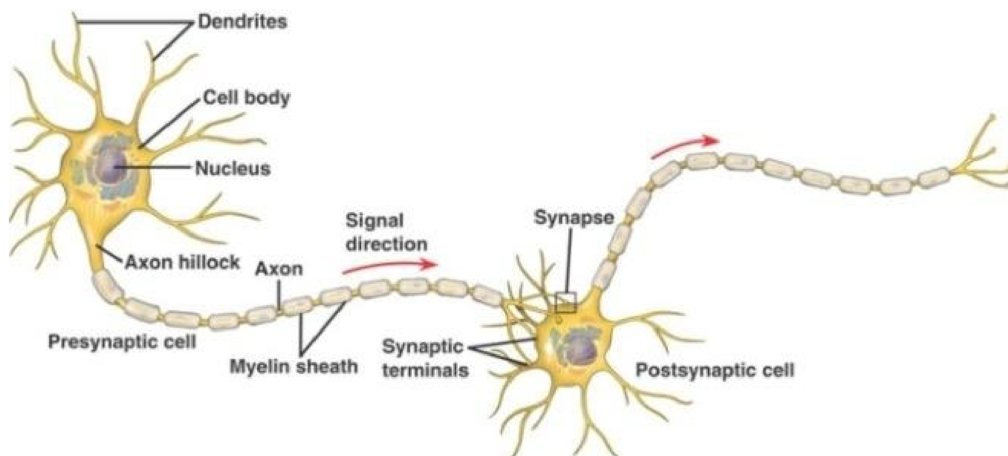

Figure 2.3: A neuron pair connected through a synapse [3]

Figure 2.3 illustrates a neuron pair with different compartments of the cell and its synapses. The part around the nucleus is called the soma. Connected to the soma are the branch shaped dendrites and the axon. The dendrites receive the signals from other cells, usually through the axon which transmits the electrical stimulation. The axon of the presynaptic cell is connected with the dendrites of postsynaptic cell, through the axon tips. When the presynaptic cell fires, it can electrically activate the postsynaptic cell via the axon through the dendrites, over the synapses or gap junctions.

2.3 Multi-node accelerator technology

To analyze related simulators and to understand how to port a simulator to a High Performance Computing (HPC) cluster, an understanding of the clusters, libraries, scheduling managers, launchers and hardware is required. The following sections will go in to details about the aforementioned subjects, before going on to the related work survey on SNNs and to discuss the selected clusters.

The IO nucleus model has a high computational complexity due to the fact that a large number of floating-point operations need to be performed [14]. This makes it an excellent target for high performance computing fabrics such as the GPU. An HPC cluster is an ensemble of computing nodes often consisting of different architectures, such as the prominent GPU technology. The GPU can be used to offload the CPU's compute-intensive parts of an application. The GPU's massively parallel SIMD processing power can boost performance significantly.

2.3.1 GPU technology

A GPU consists of hundreds to thousands compute cores, while a CPU often consists of 4 to 8 larger cores. Nvidia's Tesla GPUs are designed to perform as computational accelerators optimized for scientific computation. As of November 2018, according to the TOP500 list of high performance computing, the Tesla GPUs power the worlds two fastest supercomputers: Summit and Sierra, having a peak performance of 187,659 and 125,435 TFLOPS per second, respectively.

CUDA is an application programming interface (API) developed by NVidia which enables the usage of GPUs for general-purpose processing. It gives access to the GPUs instruction set and parallel computational elements [5]. To provide data parallelism, a CUDA application is partitioned in blocks of threads which execute concurrently. The collective network of blocks is called a grid. On device level, these blocks are allocated on streaming multiprocessors (SM), where they are divided into units called warps. In fig. 2.4 the relation between threads, blocks and the grid and their compute units is visualized.

When blocks are done executing, new blocks are launched on the SMs. SMs consist of a number of: CUDA cores for arithmetic operations, special function units for floating-point functions and warp schedulers. The GPU architecture determines the size of the warp. When an SM is given warps to execute, they are distributed among the schedulers, which at every instruction deploy moment, issue one or two independent instructions for one of its assigned warps. When warps wait for the previous instruction to finish, the scheduler will select another warp to execute. Each thread in a warp is assigned to a compute core and executes the same kernel. Warps from different blocks can be executed concurrently.

SMs have access to the GPUs global memory and all have a local memory, this holds also for the individual cores. Threads have their own local memory and registers. All blocks can access, a texture-, global- and constant memory, which are all are cached. SMs have a level-1 cache for global memory and share a level-2 cache. Figure 2.5 shows the SMs and their relation to memory, in which the device memory is a collective for global,

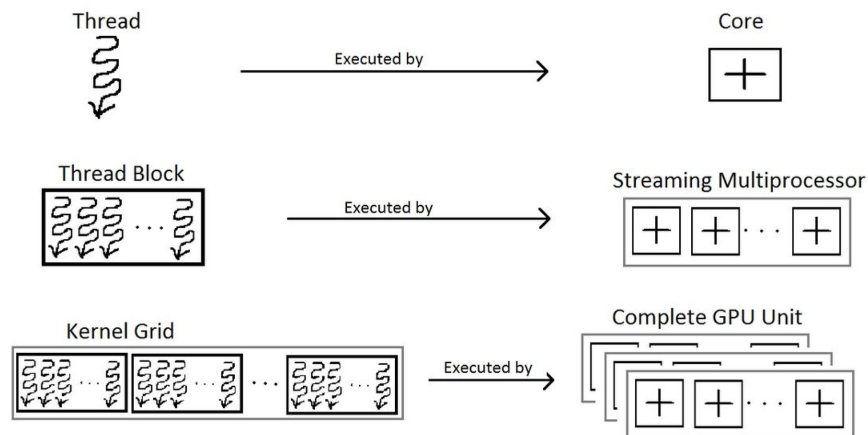

Figure 2.4: Threads, blocks and grid related to processing structure on the GPU [4]

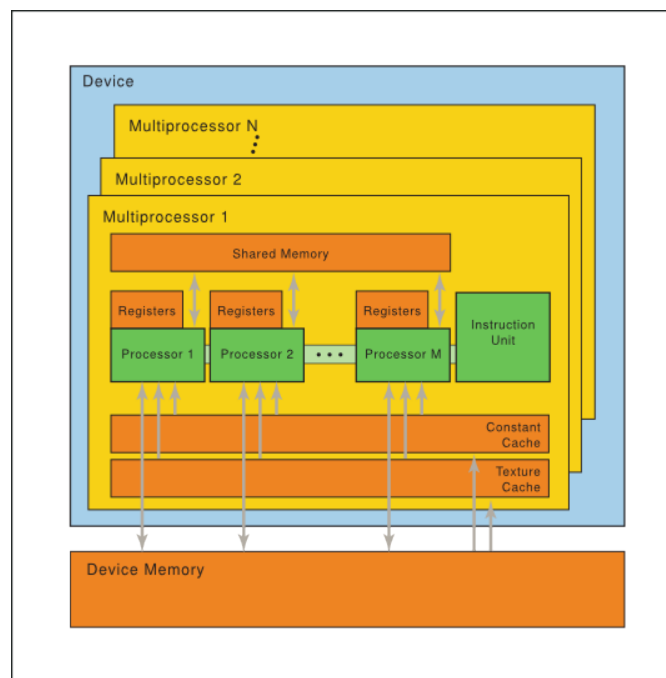

Figure 2.5: Hardware model showing SMs and memory resources [5]

constant and texture memory.

Kernels are written using the CUDA instruction set architecture called PTX, which has an assembly-language style syntax with instruction operation codes and operands, however it is more effective to use the higher-level programming language C. In both cases, kernels must be compiled using the `nvcc` compiler.

The purpose of the GPU design is to hide memory latency, which occurs whenever

data is accessed from main memory. The GPU hides latency by instantiating multiple threads per core and switching between threads. While one thread is waiting on data from main memory, another tread is started and does its part until it also is stalled by memory access. There should be enough threads to hide memory latency but too many threads may lead to a degradation in performance due to the reduction of resources per thread.

An indication of an effective ratio of threads versus latency is the metric *occupancy.* Occupancy is defined as the ratio of active warps to the maximum number of warps supported on a SM and is determined by the amount of shared memory and registers used by each thread block. The registers are a shared resource that are allocated among the thread blocks executing on a SM. Maximizing occupancy can cover latency during global memory loads. The CUDA-compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously [5].

2.3.2 GPUdirect

In a cluster, the connection of nodes among themselves is called *inter-node* connection. The connection of computational units within the nodes is called *intra-node* connection. The NVLink hardware from NVidia, currently has the highest bandwidth for intra-node communication. It has a memory bandwidth of 160 GB/s for the P100 GPU and up to 300 GB/s for the V100 GPU. If NVLink is not available, usually PCI-Express is used, limiting the connection to a bandwidth of 32 GB/s.

GPUdirect is a software technology which provides high-bandwidth and low-latency communications directly between multiple GPUs. It is an umbrella term to refer to specific technologies and covers intra-node and inter-node communication. The main two technologies under this umbrella are Peer-to-Peer (P2P) and Remote DMA (RDMA). P2P transfer technology was introduced in CUDA 4.0 and allows for fast transfers intra-node but does nothing for memory transfers inter-node. It allows for buffers to be copied directly between the memories of GPUs. It employs Unified Virtual Addressing (UVA), which was introduced in CUDA 3.1. With UVA, the host memory and memory of all GPUs in a system are combined into one large virtual address space [15]. A prerequisite for P2P is that 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 (TCC) option. RDMA, introduced in CUDA 5.0, allows to initiate memory transfers between GPUs in a cluster over PCIE or Infiniband. RDMA allows the GPU to send data directly through Infiniband to a remote system without any interactions with the CPU and is available for the Tesla and Quadro accelerator cards, starting from the Kepler architecture onward. In fig. 2.6 the memory-transfer paths are shown with and without GPUdirect.

MPI, is the standard API for communicating data via messages between distributed processes, that is commonly used in HPC to build applications that can scale to multi-node computer clusters [15]. When programming on a cluster, three libraries implementing MPI are commonly used: OpenMPI, MVAPICH and Intel MPI. OpenMPI is a library project combining many technologies from other projects (FT-MPI, LA-MPI, LAM/MPI,

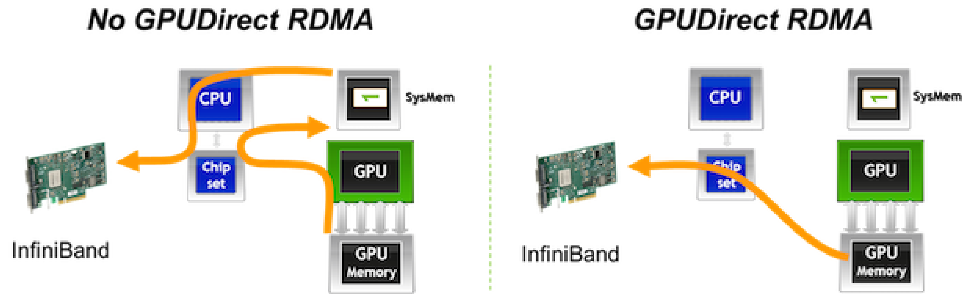

Figure 2.6: Memory transfer paths [6]

and PACX-MPI) and is used by many super computers. MVAPICH is an implementation of MPI following the MPICH standard. MPICH is a freely available and portable implementation of MPI. The Intel MPI is a multifabric message-passing library that also implements the open-source MPICH specification, used to accelerate applications based on cluster technology from Intel.

CUDA-aware MPI (CAM) libraries are the result of the effort of combining the two parallel programming approaches MPI and CUDA. Basically it enables users to accelerate applications with multiple GPUs or to enable single node multi-GPU applications to scale across multiple nodes. OpenMPI and MVAPICH2 are CAM libraries which implement GPUdirect. The version MVAPICH2-GDR has support for Infiniband and GPUs.

These libraries enable the direct passing of the GPU buffers pointers to MPI, instead of needing to stage GPU buffers through host memory using cudaMemcpy, resulting in a direct write from one device to another (RDMA). If RDMA is used when the architecture doesn't support it, the calls for the Message Passing Interface (MPI) will still return successfully, however they will be performed through the standard memory copy paths [16].

GPUdirect can improve performance by reducing latency for small messages, as can be observed in fig. 2.7. It outperforms the Intel MPI and MVAPICH even when message sizes go up. It reports the lowest latency of these techniques for smaller message sizes. Even for larger message sizes, OpenMPI outperforms its competitors, as can seen in fig. 2.8. These results have been obtained on the Cartesius cluster by Cartesius.

Neurons in an SNN, in particular in the IO, are often randomly interconnected by some sort of distribution process. If a cell network is distributed across a GPU cluster, cells will most likely need information from cells hosted on other GPUs. The message size of the data to be send, does not have a fixed size and can even vary greatly among the GPUs. Data movement across GPUs intra- or inter-nodes, could form a serious bottleneck for larger SNNs. Simulators which have GPUdirect implemented preferably through OpenMPI, thus could offer an efficient solution.

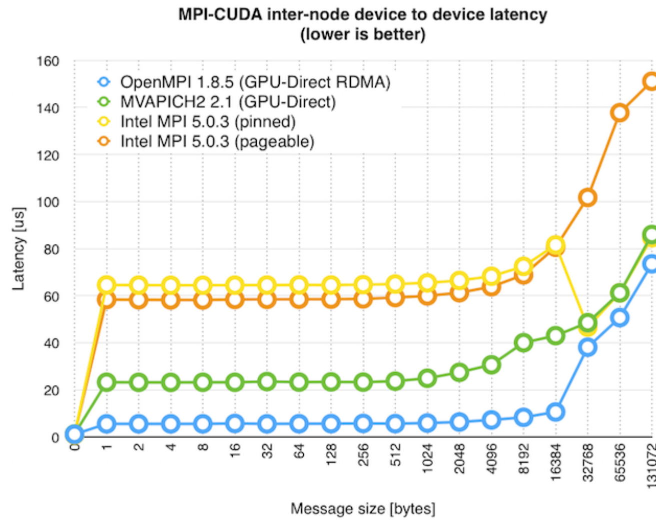

Figure 2.7: Latency specification for GPUdirect on Cartesius [7]

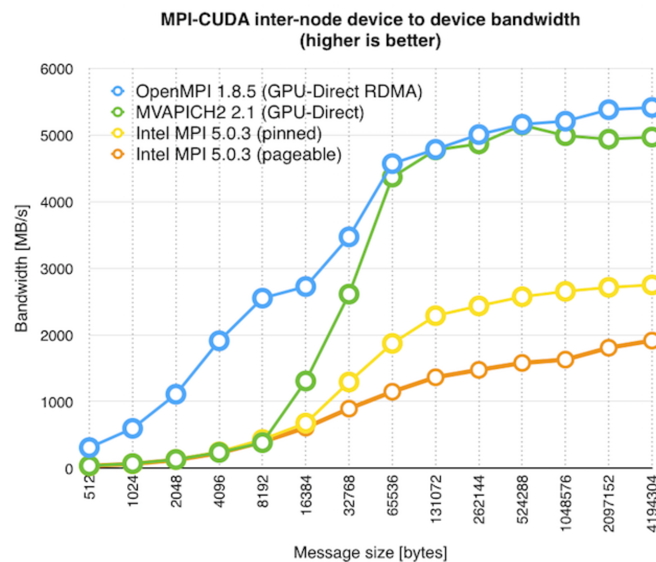

Figure 2.8: Bandwidth specification for GPUdirect on Cartesius [7]

2.3.3 Xeon Phi technology

In [14] a multi-node version of an extended Hodgkin-Huxley simulator has been benchmarked on 1 to 8 of Intels Xeon PHI chips with Knight Landing (KNL) architecture. Because this paper comes close to the implementation developed in this thesis this will serve as a format for the benchmarks in the results section and as a comparison. Since this close relation this subject is treated in the background section but could also be con-

sidered as related work discussed in chapter 3. In this and the next section a comparison between the KNL and GPU is made.

The power of the the KNL comes from the 64 cores able to dispatch 4 instruction sets simultaneously. When employing n KNL processors, a degree of $nx256$ of thread-level parallelism is reached. Each core utilizes two 512-bit wide vectorization processing units (VPU's) which enable AVX512 instructions for parallelized data processing. These cores have a private level 1 and shared level 2 caches and have different configurations how to distribute memory address space across the chip. The cores are connected through the level 2 caches in a mesh fashion using the Omni-Path interconnect for high bandwidth and low latency scaling between nodes, meaning that the processors are not connected all to all.

A feature aiming at reducing memory-access latency, is the 16 GB multi-channel DRAM (MCDRAM). It is a high-bandwidth memory (400 GB/s), spatially located next to the processing cores which has a significantly higher bandwidth than the 384 GB DDR4 RAM (90 GB/s), also available. This memory is configured serving as a last level cache, because larger networks cannot be allocated on the 16 GB entirely.

A 4:64 MPI:OMP ratio is maintained, which means that 4 MPI ranks spawn 64 OpenMP threads. Each neuron is assigned to a single thread and each thread handles an equal number of neurons to balance computational workload. With this configuration it is possible that MPI communication between ranks - when dendrite voltages need to be calculated - can happen on the same KNL node.

Good simulation results are reported for the Xeon Phi implementation. For instance, the computation times for a 2B cell Gaussian distributed network with a density of 1000 neurons per cell, lies between 40 and 50 seconds, executed on 8 KNL nodes. A 1B cell uniformly distributed network with a density of 1000 neurons, which executes best on 2 KNL nodes, runs within 400 milliseconds. Erratic behavior is obtained when 8 nodes are used for execution of large uniformly distributed networks, stating the fact that adding more hardware will not benefit these types of network for this architecture. The reason given for the fact that a 2B version of the aforementioned network cannot be run, is an increase in communication between the cores and memory access of the cores, since the caches are not able to store all the required data, forcing processor cores to search in non-local caches. In this article no times for the setup of the network are portrayed. The mentioned computation times only depict a part of the total simulation, whereas the setup time of a network of cells often is the bottleneck in the simulation. In this thesis all phases of the simulation will be benchmarked and discussed.

2.4 Cluster technology

A number of clusters have been considered to host the multi-GPU version of the InfOli application. The Texas Advanced Computing Center (TACC) has a cluster called *Fabric* which has Power8 nodes equipped with K40M GPUs, however the nodes are not equipped with Infiniband network adapters necessary for GPUdirect. The eXtreme Science and Engineering Discovery Environment (XSEDE) hosts a number of heterogeneous clusters among which is *Comet*, belonging to the San Diego Supercomputer Center (SDSC). The Comet cluster has 72 nodes with in total 432 modern GPUs interconnected with Infini-

band adapters. This is a very suitable cluster, however only accessible for US citizens or through the NeuroScience Gateway (NSG). The NSG, a portal for computational neuroscience, hosts a number of brain applications some of which are discussed in the next chapter. The NSG does not allow the execution of self-developed brain applications, making it unsuitable for the current setting. TU Delft hosts a cluster called the Distributed ASCI Supercomputer 4 (DAS-4) which is part of a six-cluster wide-area distributed system spread out through The Netherlands. However, this cluster contains 8 nodes each with a single consumer grade GPU not suited for GPUdirect and thus not suitable for the task at hand. The Amazon AWS cluster and the Dutch supercomputer Cartesius were found to be the most suitable High Performance Computing clusters for hosting and benchmarking the InfOli application.

2.4.1 Cartesius

Cartesius is a Bullx system extended with one Bull sequana cell. It is a clustered SMP (Symmetric Multiprocessing) system built by Atos/Bull. SMP is a multiprocessor computer consisting of one or more identical processors connected to a single shared memory, having full access to all in- and outputs and controlled by a single operating system. The Cartesius cluster consists of many heterogeneous nodes. The GPU accelerator island consists of 66 Bullx B515 GPGPU accelerated nodes. Each one of these nodes consist of: 2 x 8-core 2.5 GHz Intel Xeon E5-2450 v2 (Ivy Bridge) CPUs/node, 2 x NVidia Tesla K40M GPGPUs/node and 96 GB of memory per node. These Xeon processors and their 8 cores could be used to feed the GPUs their data.

The Tesla K40M is a server-grade PCI-express accelerator board comprised of a single GK110B GPU with NVidia Kepler architecture. It has 2880 processor cores which run at a base clock of 745 MHz. It offers a total of 12GB of GDDR5 comprised of 24 pieces of 256M x 16 GDDR5 SDRAM on-board memory running at 3.0 GHz and with a bus width of 384-bit, totaling the memory bandwidth to 288 GB/sec. The RDMA feature in GPUdirect allows third party devices such as Network Interface Cards (NIC) to directly access memory on multiple GPUs within the system, significantly decreasing the latency for MPI send and receive functionality. It also reduces memory bandwidth and frees the GPUs DMA engines for other CUDA tasks.

The K40M also supports GPUdirect P2P, as described in section 2.3. However when the topology of the nodes are checked, it was found that the GPUs are not connected to the same root PCI complex, a prerequisite for the P2P functionality. Intra-node communication between nodes using the improved RDMA technology on Cartesius is not possible. Thus, per node of the cluster 1 GPU will be utilized such that RDMA can be made use of, limiting the scaling of the algorithm to a maximum of 66 GPUs, of which 58 are currently in use.

The GPGPU nodes have two Mellanox ConnectX-3 Infiniband adapters: one per GPGPU, providing 4 x FDR (Fourteen Data Rate) resulting in 56 Gbit/s inter-node bandwidth. These Mellanox NICs support RDMA, enabling the GPUs to directly access the memory on multiple GPUs within the system. Each node runs under one operating system, bullx Linux, which is a Linux distribution compatible with Red Hat Enterprise Linux and shows a single memory image to the user programs.

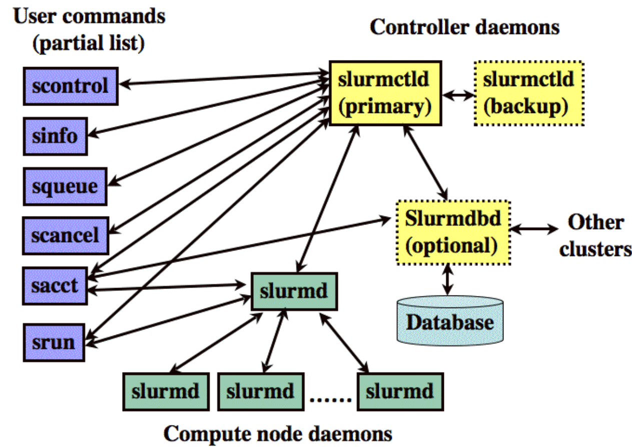

Figure 2.9: Slurm components [8]

```
1 srun --nodes 16 --ntasks 16  --partition gpu_short --time=1:00:00 ./
    your_application
2
3 sacct --format consumedenergyraw,elapsed,jobid,MaxRSS,AveRSS
```

Listing 2.1: Typical srun and sacct command

2.4.2 SLURM

Cartesius uses Simple Linux Utility for Resource Management (SLURM) for cluster management [8]. SLURM is used on 60% of the TOP500 supercomputers. It's an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. It has three key functions. First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform their jobs. Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work.

SLURM consists of a `slurmd` daemon running of the compute nodes and a central `slurmctld` daemon running on a management node. The SLURM components are shown in fig. 2.9.

The entities managed by SLURM include nodes, the compute resources, partitions, which group nodes into logical sets and jobs, which are allocations of resources assigned to a user for a specified amount of time. Partitions can be seen as job queues to which constraints such as job size limit or job time limit can be assigned. When a job is submitted, SLURM will run this job once the resources become available and constraints have been met.

Name	Max wall-time (d-hh:mm:ss)	Max #nodes	Description
GPU	5-00:00:00	48	for production runs
GPU_short	01:00:00	64	for test and debug runs

Table 2.1: GPU partition overview for Cartesius

Frequently used commands are:

- `srun`: submit jobs
- `sbatch`: submit a job script for later execution
- `squeue`: reports the state of jobs, can be filtered on user
- `scancel`: cancels pending or running jobs
- `sinfo`: displays the state of partition and nodes
- `sacct`: display information about finished tasks

The `srun` command is used to submit jobs for execution. It is basically a wrapper around the launcher `mpirun`. Mpirun attempts to determine what kind of machine it is running on and start the required MPI jobs on that machine. It controls several aspects of program execution in OpenMPI and uses the Open Run-Time Environment (ORTE) to launch jobs. The `srun` command has a number of options to specify resource requirements, such as minimum and maximum node count (`-nodes`), the number of tasks per node (`-ntasks`), the partition requested and/or the maximum time constraint (`-time`). The following command is a typical `srun` execution; it will run `your_application` on 16 GPUs launching 1 task per node for a maximum wall-clock time of 1 hour. To collect energy and/or memory information for finished jobs the command `sacct` is executed, also shown in listing 2.1.

On Cartesius a number of partitions are available to launch on, of which 2 are for GPUs. Their specifications are displayed in table 2.1. System usage is measured in System Billing Units (SBU) which is based on wall-clock time. For GPU nodes, 1 SBU is equal to 1 core for 20 minutes. A "core hour" on a GPU node is three times as expensive as a "core hour" on a fat or thin compute node. The `budget-overview` command gives an overview of the remaining budget related to finished or running tasks.

2.4.3 Amazon Web Services

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, re-sizable compute capacity in the cloud. The elastic feature allows to obtain new server instances in minutes, enabling the option to quickly scale capacity, both up and down as the computing requirements change. The EC2 is different than the Cartesius cluster. Cartesius has all its compilers, launchers and schedulers pre-installed and modules are loaded with the simple `load module` command. Amazon lets the user configure the nodes himself.

The partitions on Cartesius are called instances on EC2 and it has a lot more instances than Cartesius. When launching an instance, an Amazon Machine Instances (AMI) has to be selected. These AMI's are pre-configured operating systems with certain software already on board but often applications, libraries, modules or configurations

Model	GPUs	vCPU	Mem (GiB)	GPU Mem (GiB)	GPU P2P	Networking
p3.2xlarge	1	8	61	16	-	Up 10 Gbit
p3.8xlarge	4	32	244	64	NVLink	10 Gbit
p3.16xlarge	8	64	488	128	NVLink	25 Gbit
p3dn.24xlarge	8	96	768	256	NVLink	100 Gbit

Table 2.2: AWS P3 compute instance overview

have to be installed manually. Next to this, EC2 works with so called IAM user profiles. These profiles enable system administrators to administer profiles with only the required authorization level a user needs.

The GPU compute instances are comprised of up to a maximum of 8 NVidia Volta V100 GPUs. The V100 has 5120 cores running at 1230 MHz, 16 GB of HBM2 memory at 900 GB/s. The V100 enable the instance to have up to 1 Pflops of mixed (16 and 32-bit)-, 125 Tflops of single-, and 62 Tflops of double-precision floating point performance. NVLink interconnect allows GPU-to-GPU communication at high speed and low latency intra-node. An overview of the instance option for a p3 accelerated compute instances are given in table 2.2.

EC2 offers 3 different NICs for inter-node communication. The fastest is the Elastic Network Adapter (ENA), a networking chip based on ARM chips, which assures 25 GB/s of dedicated network bandwidth. Next to this, there is also a service called `Elastic Fabric Adapter` enabling 100 GB/s networking and having a latency of 15us. Each instance of type p3.16xlarge would allocate 488 GB of DRAM and 64 virtual CPU's based on custom Intel Xeon E5 Broadwell processor.

To enable the elasticity, a burstable and event-driven HPC Cluster on AWS using SLURM has been deployed, according to [9]. SLURM has an elastic plugin, which allows the coordination of a launch of a set of compute nodes with the appropriate GPU network topology. The job resource can be launched on demand instead of fitting a job in an existing compute topology, like Cartesius. In fig. 2.10 the architecture of this burstable cluster is shown.

This burstable cluster needs to be set up through a cloud formation script, which contains information about certain settings such as: which AMI to use as base and which version to use, etc. Cloud formation provides a common language to describe and provision all the infrastructure resources required in your cloud environment. This script will enable a single instance, R4 in the architecture overview, which will enable user to burst compute instance necessary for the application to run. Input files can be delivered into an S3 bucket, which is an AWS resource for storing and managing files. An S3 bucket has a globally unique name shared by all AWS accounts. AWS provides an S3 API to work with this storage technique.

Unfortunately, for a user to acquire the right authorization to proceed in setting up a cloud formation, a lot of different authorization levels have to be entered. For instance, when creating the burstable cloud, a user needs to create or delete certain roles associated with the process. A user can have the rights to delete a profile, however she also needs to have the rights to show the user profile. After many weeks of deliberation with the support desk, a cloud formation was obtained, but that delayed the implementation

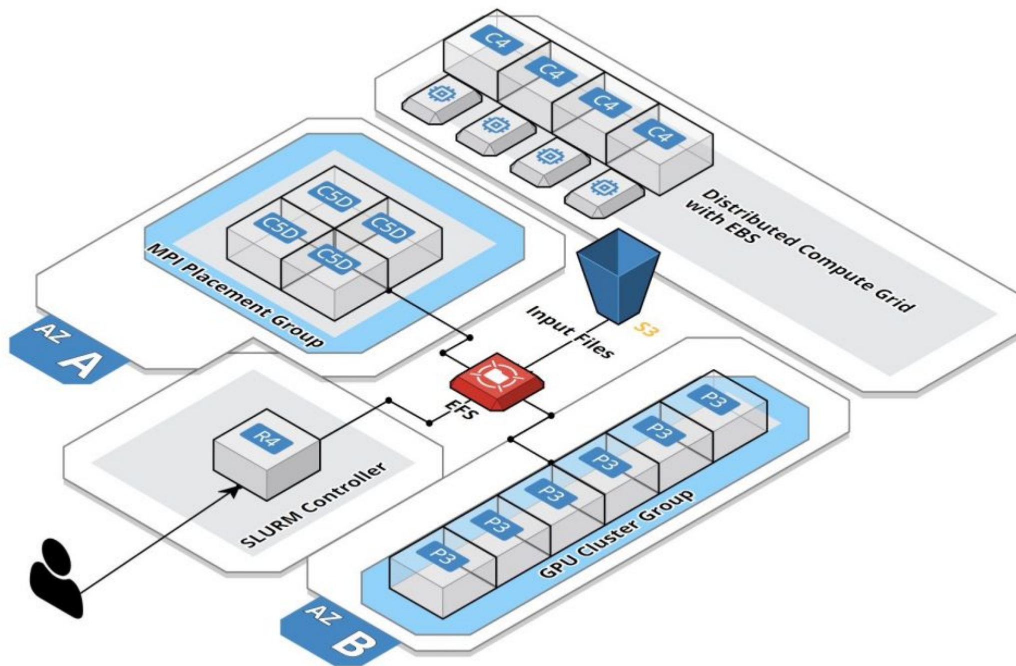

Figure 2.10: Burstable AWS architecture using SLURM elastic [9]

significantly.

2.5 Summary

This chapter has delineated the necessary neuroscientific knowledge and HPC technology. Cluster hardware and software has been analyzed. Cartesius is ready-made and AWS needs more effort to configure the compute facility. Acquiring authorizations for the AWS cluster delayed the development for the InfOli application. The Cartesius cluster proved to be the most suited facility to host and benchmark the InfOli application.

The GPUs to work with and the software for scheduling tasks are known. Limiting factors of the GPUs are the bus- and bandwidth of the memory used. Libraries and interfaces of the individual nodes have been studied. The MPI-CUDA inter-node communication has good performance but will always serve the application best if communication packet size is kept at a minimum. A multi-GPU version should there be efficient in keeping the data communicated over the network as few as possible.

In Table 2.3 the aforementioned specifications with regards to computing units on the nodes of the clusters are summarized. Ranked in terms of performance with the fastest first they could be ordered as V100, KNL and K40M.

Architecture	KNL	K40M	V100
Cores	64	2880	5120
Clock (MHz)	1400	745	1230
Memory On Chip (GB)	16	12	16
Memory Type	MCDRAM	GDDR5 SDRAM	HBM2
Bandwidth (GB/s)	400	288	900
Memory bus width (bit)		384	4096
FP32 (Tflops)	6	4.29	15.7
FP64 (Tflops)	3	1.43	7.8
Multiprocessors	NA	15	64
Resident threads per multiprocessor	NA	2048	2048
Technology (nm)	14	28	12

Table 2.3: Comparison of important architecture specifications

3 Related work: Neural network simulators

This chapter discusses the related simulators which have similar functionality as the IO simulator considered in this thesis. It taxonomizes the simulators, as explained in chapter 2, on support for: LIF, IZH, and HH models, gap-junctions, multi-compartments and neuromorphic or general purpose multi-node technology, in particular for the GPU.

To model neurons for simulations, tools or languages have been developed to unify this process. NeuroML, CellML and Systems Biology Markup Language (SBML) are declarative modeling tools that use XML based descriptions to provide a common data format and the exchange of descriptions of neuron cells and networks. PyNN is a Python based procedural simulator-independent language for building neuron networks. When one of these tools is used as a front-end to the simulator, the LIF, IZH and HH models, gap junctions and multi-compartment capability are supported.

3.1 SpiNNaker

The SpiNNaker project has its architecture inspired by a human brain [17]. It is a massive parallel computing platform which targets neuroscience, robotics and computer science as main area of research. It supports building neural networks in the PyNN description language.

Its neuromorphic hardware consists of over a million of ARM A9 cores and 7T of RAM distributed amongst its network, giving them a bandwidth of 5 billion packets per second. The largest of its form is capable of simulating billions neurons. The machine consumes at most 90 kW of electrical power.

During the design of the architecture of SpiNNaker, three principle axioms of parallel machine computing have been discarded: memory coherence, synchronicity and communication determinism, without loosing the ability to perform meaningful computations. SpiNNaker uses inherently unreliable simple spikes (packets) to communicate, no larger than 72 bits. Therefore, fault recovery mechanisms have been built in at many levels of abstraction. Mimicking the brain also entails breaking up complex tasks, like vision, into many simple computations which can be parallelized.

SpiNNaker source code is accessible through GitHub and their half-million core machine, is accessible through the human brain project portal as an online server.

3.2 Brian

Brian is written in Python and is available on almost all platforms. Brian has an extensive modeling environment and has the option to be driven by (stochastic) differential equations.

Custom models are entered as equations and thus has support for LIF, IZH and HH neuron models. Also PyNN can serve as an input to Brian. Brian has support for conditional, probabilistic and multi synapses and also gap junctions can be modeled. Brian supports multi compartment models. It is possible to create neuron models with a spatially extended morphology, using the SpatialNeuron class. This works the same way as creating a neuron, but now the elements are compartments instead of neurons. These morphologies can be created combining geometrical objects. The length and diameter of each object can be specified, representing individual compartments. Tree structures can then be created by attaching morphology objects together.

Brian has single GPU support in the form of Brian2CUDA (B2C), which generates C++/CUDA code to run on NVidia GPUs. Another back-end that can be used to execute Brian generated models on GPUs is simulator/code generator GeNN [18]. GeNN consists of a C++ source library that generates CUDA kernels and runtime code according to a user-specified network model. GeNN and Brian2CUDA are single GPU orientated. Brian can serve as a front-end for the SpiNNaker project described in section 3.1. This project however has its latest commit dating four years ago and seems to have been abandoned.

3.3 DynaSim

DynaSim is an open-source Matlab/GNU octave toolbox for rapid prototyping of neural models and batch simulation management. It is designed to speedup and simplify the process of generating, sharing and exploring network models of neurons with one or more compartments [19]. GNU Octave is a high-level language, primarily intended for numerical computations [20].

Neuron models can be customized by using modules written in C, C++ or Fortran. Examples of LIF, IZH and HH models are found in their documentation. Besides these models DynaSim supports multi-compartment neurons, however it recommends using NEURON when complex morphologies come into play. In the listed synapse support, a connection mechanism for gap junctions can be found, identified as "Ohmic gap junction".

Next to the fact that a SNNs created in DynaSim could be run on a NEURON, Brian or NEST, it also has native support executing simulations parallel using multi-core processors and high performance clusters. It enables distributed simulation by setting parfor_flag and cluster_flag to 1, for multi-core and cluster support.

The `parfor_flag` originates in the Matlab Parallel Computing Toolbox (PCT). This toolbox uses `parfor` as a statement indicating to execute the marked for-loop iterations in parallel on workers in a parallel pool. The PCT enables the solving of computationally and data-intensive problems on, for instance, CUDA-enabled NVidia GPUs and clusters without using CUDA or MPI programming. However, the parfor instruction for GNU/Octave toolbox reduces to a simple for loop [21]. The status of a cluster implementation henceforth, remains unclear.

The Sun Grid Engine (SGE) is responsible for accepting, scheduling and managing the distributed execution of parallel jobs. The SGE command `qsub` is invoked to queue the job created and has multi-GPU supported batch options [22].

Benchmarking results for DynaSim simulations runs are only reported as a comparison between Brian and DynaSim, without information on which back-end was used. No source

code or documentation about a multi-GPU implementation are reported. A multi-GPU implementation for DynaSim, thus remains only possible in theory.

3.4 CARLsim

The Cognitive Anteater Robotics Laboratory simulator (CARLsim), is a GPU-accelerated library for simulating large-scale SNNs [23]. Parameter details at synapse, neuron or network level can be specified in a PyNN-like programming interface in C/C++. Currently the 4 and 9 parameter IZH with either current-based or conductance-based synapses and LIF models are supported. The HH is not supported.

Neurons in a group can be extended to include multiple compartments. For instance, a group of neurons can be extended to include a dendritic current coming down from a "mother" up to multiple "daughter" compartments. CARLsim has supports for chemical but not for electrical synapses also known as GJ [24].

CARLsim 4.0.0 beta allows for concurrent simulation on up to 8 GPUs in a single simulation. The function CARLsim:createGroup is used to create a group of neurons, which has parameters to specify the preferred GPU partition (0-7). The created groups can then be connected to create, for instance, an all-to-all connected model. MPI and inter-node communication are not supported. Only consumer grade GPUs are supported and therefore P2P technology is not applicable. The message passing between GPUs is staged through the host and device memory.

CARLsim employs a reduced Address-Event Representation (AER) communication protocol for efficient encoding of neuronal communication [25]. AER stores spike events by representing it as an address-time pair. This protocol limits memory bandwidth and reduces memory usage. Duplicate stores may occur when many neurons have the same time step.

To maximize the degree of parallelization, CARLsim employs two approaches to assign SNN computations to threads. These approaches are called N-parallelism (NP) and S-parallelism (SP). NP makes the distinction between organizing computations according to neuronal activity and SP according to synaptical activity. The best results are obtained when a hybrid form between both approaches is used. This reduces load balancing and warp divergence, and increases the overall memory bandwidth. Warp divergence occurs when threads in a warp don't do the same thing at the same time; this can lead to parallel inefficiency. Sparse representation techniques are implemented for spiking events and neuronal firing to decrease memory and bandwidth usage.

Because CARLsim is still a beta version no benchmarking results for the multi-GPU implementation are available. Results from previous versions of CARLsim report that newer hardware and the implemented S-parallelism for the GPU platform, led to 60 times faster execution in comparison to their CPU implementation for a specific network configuration.

3.5 NEST

The Neural Simulation Technology Initiative (NEST) is a simulator for SNN models that focuses on the dynamics, size and structure of neural systems rather than on the exact morphology of individual neurons [26]. NEST is ideal for simulating SNNs of any size for example for models of information processing or models of learning and plasticity.

NEST provides over 50 neuron models. These models include LIF with current or conductance based synapses, IZH or HH models. NEST also support the creation of multi-compartment neurons.

A number of connection strategies can be applied: one-to-one, all-to-all, convergent and divergent and Pairwise-Bernouille. The latter strategy enables the random generation of connections through the Bernouille probability function. If these connection strategies are not sufficient, topology connections which provide more sophisticated functions, can be administered. This topology provides NEST with an interface for creating complex networks with a spatial structure.

NEST provides over 10 synapse models. Next to chemical synapses, NEST also supports the electrical bidirectional gap junctions for the HH model. The creation of gap junctions is restricted to one-to-one or all-to-all connections with equal source and target populations because for each created connection a second connection with the exact same parameters in the opposite direction is required. Random connections cannot be employed for the creation of gap junctions. They have to created on the Python level implementing the random module of the Python Standard library. The drawback here is that on this level it serializes the connection procedure in terms of computation time and and memory in distributed simulations.

NEST is capable of running simulations on multi-core/-processor machines and computer clusters using two ways of parallelization: thread-parallel simulation and distributed simulations. OpenMP and MPI are applied to achieve implementation of these forms of hybrid parallelism. Threads with processes related to the neurons are distributed round-robin onto the MPI processes. A global spike exchange mechanism updates the neurons on the different nodes except when nodes originate and terminate at the same device. In this case the spike exchange is handled by the virtual process to which the neuron in assigned. Before spikes are exchanged they are copied from the spike register to a communication buffer. Once all buffers are set up the buffers are exchanged between MPI processes. The NEST architecture allegedly scales to the largest petascale computer available today, but a multi GPU implementation does not exist.

A NEST affiliated research group called Jülich Supercomputing Centre (JSC), has done interesting analysis of parts of parallel simulation technology [27]. Their research includes different malloc-ing functions, which alleviate the time taken to construct a neuronal network and a memory model that allows the prediction of memory usage related to the number of MPI processes, which help developers find scalability bottlenecks.

3.6 (Core)Neuron

NEURON provides a powerful and flexible environment for implementing biologically realistic models of electrical and chemical signaling in neurons and networks of neurons

[28]. It lets researchers focus on the high-level neuroscience questions without having to go through lower level mathematical questions.

Neuron modeling comes in the form of the high level language NMODL, which allows the expression of models in terms of kinetic schemes or sets of simultaneous differential and algebraic equations. Kinetic schemes are a network of states of which the connections between them, represent the transitions of a dynamical process The constructed models can be published in ModelDB, a database for neuron models. ModelDB contains over a thousand of neural models and many variants of LIF, IZH and HH models can be downloaded for simulation.

The interpreted C-like language 'hoc' has been extended to include function specific properties to the domain of modeling neurons in terms of familiar idioms instead of differential equations in C. It is used to define the morphology and membrane properties of neurons and control the simulation.

NEURONs multi-compartment equivalent is called a section, which is referred to as an unbranched cable. Sections can be connected to form branched trees of which biophysical and anatomical properties can vary along its length. Gap junctions are supported and are called point processes.

CoreNEURON is a simplified engine for the NEURON simulator optimized for both memory usage and computational speed. 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. It only supports fixed time-steps and depends on NEURON to build the network model.

CoreNEURON is used by distinguished institutes as the Jülich Supercomputing Centre and the École polytechnique fédérale de Lausanne (EPFL) to do the HPC computation for the Blue Brain Project (BBP). The BBP is a Swiss initiative, with the aim of digitally reconstructing a brain by reverse engineering mammalian brain circuitry, using a Blue Gene supercomputer. Blue Gene is a project by IBM, building a supercomputer reaching Pflops of computations with low power consumption. The BBP falls under the same umbrella as The Human Brain project (HBP), which has a wider scope, namely revolutionizing the Alzheimer research, replacing the need of certain animal experiments and simulation driven drug discovery [29].

Support for GPUs comes in the form of using the Open Accelerators (OpenACC) programming model which uses an compiler known as Portland Group Inc (PGI) accelerator compiler. OpenACC a programming standard for parallel computing, developed amongst other by NVidia, provides compiler directives that specifies loops and regions of code to be implemented on a high performance computing device. The PGI compiler acts on the directives to accelerate applications by mapping it onto accelerator hardware like GPUs and can target NVidia's Tesla GPU accelerators. PGI compilers also support OpenMPI with GPUdirect, which is configured to use Infiniband hardware if available in the system. However, CoreNEURON does not use for GPUdirect explicitly.

3.7 HRLSim

HRLSim is a high performance spiking neural network simulator for GPU clusters. This GPU simulator is driven by the need for support of the neuromorphic hardware for the

Systems of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE) project[30]. SyNAPSE is a DARPA program, its goals are to build a cognitive computer with similar form, function and architecture to the mammalian brain. SyNAPSE is a joint venture funded by Hughes Research Laboratories (HRL), Hewlett-Packard and IBM Research. HRLSim is not an open source project and not active because no publications are found after 2015. In 2010, MIT listed its chip to learn like the brain by altering synapses in their top ten of breakthrough technologies.

HRLSim supports LIF and a simple Izhikevich model and allows for excitatory and inhibitory synapse modeling, but makes no explicit statements about GJ modeling. A network model is developed in an environment with C++/PyNN like interface.

To achieve optimal performance across the nodes, communication and computation are parallelized and overlapped as much as possible. For communication sending and receiving are two distinct parallel processes, next to computation. The communication threads are allowed to finish in maximum time while communication latency is masked with computational overhead.

To optimize simulation in terms of execution time HRLSim presents three main aspects of its form of communication. Dummy neurons provide a mechanism for compressing the amount of transferable data and remove the need for extra look-up tables to deal with external efferent connection [30]. A novel dynamic spike packing is used to reduce the cost of message passing by placing an upper bound on the amount of data sent regardless of the spike rate of the network. It switches dynamically between AER to a bit representation of the outgoing ensemble of neurons. MPI is used to perform inter-node communication. Spikes are packaged together according to their destination nodes and for Infiniband-based communication the non-blocking P2P methods MPI_Isend and MPI_Irecv perform best.

The following CUDA features were used to optimize the implementation:

- Using network statistics for selecting the grid/block size
- Memory access based on firing rate, led to a dynamical postsynaptic update function
- Kernels that don't depend on large memory copies are assigned to separate streams to execute in parallel
- Reduced precision integer approximation
- Aligning of synapse related data-structures to prevent overlapping memory access
- Distinct kernel for spike message packaging to off load communication threads

For results they report that a 110K neuron and 11M synapse network for 100 timesteps is simulated in 99 seconds on a Tesla C1060 card. To test scaling across nodes, a 100K neuron SNN was simulated on 2, 4, 8, 16, 32, and 64 GPU cards. It was concluded that HRLSim scales well when network sizes increase. They conclude that computational bottlenecks are the integration/synaptic (GJ) updates and the message packaging.

3.8 Neurokernel

In contrast to general-purpose brain simulators, Neurokernel's goal is to develop an open software platform written in Python for the real-time emulation of the entire brain of the fruit fly operating on a multi-GPU back-end. [31]. They use photo-receptors to model the retina and lamina of the fly. The brain of the fruit fly is only 135.000 neurons in size but can display complex behavior.

The connectome of the fruit flies can be decomposed into 40 unique modular subdivisions called local processing units (LPU). These LPUs correspond to anatomical regions of the fly brain associated with specific functional subsystems such as sensation and locomotion and are regarded as functional building blocks of Neurokernel. Next to these LPUs Neurokernel provides the interface for the communication of neuron state data between LPUs. This interface guarantees independence between LPU's such that they can be improved and replaced without modifying other parts of the brain. An Python based API is provided to abstract the LPU implementation and interconnection from the underlying GPU code. The connectivity between LPUs is stored in a sparse multidimensional array.

Neurokernel supports the LIF and Hodgkin-Huxley point neuron models. It supports alpha function synapses and a conductance-based synaptic model.

Their architecture consists of three planes: a high-level application plane which holds the LPUs and interconnect; a low-level compute plane consisting of multiple GPUs for the execution of the neural circuitry; and a control plane consisting of multiple CPUs for the mapping of LPUs to methods and resource management of the GPUs. It uses the package PyCUDA to support Python on NVidia's hardware.

The brain emulations can be run on multiple GPUs in a single computer or a computer cluster. Neurokernel takes advantage of GPUdirect P2P technology by using the OpenMPI library. This can only be employed when source and destination memory locations of an MPI data transfer are both in GPU memory. Because of this and no trace of the RDMA technology being implemented, one must assume that inter-node communication takes place with staged copies through host memory.

For their module communication performance tests they used 4 NVidia Tesla K20XM GPUs running all on the same node. Their tests include scaling the number of ports of the LPU (50-25000) and scaling the number of LPUs (4 to 19). These tests were both performed with and without P2P technology, yielding much better performance with the technology enabled. An increase in the number of ports did not increase the models execution time for the numbers of ports similar to the numbers of neurons in actual LPUs. The speedup from scaling the number of LPUs revealed also that the P2P technology could power multiple LPU models sufficiently. However, the resource allocation and management features to successfully scale a fruit fly brain over multi-GPUs are currently not yet available.

3.9 NCS6

Neural Cortical Simulator (NCS) is an extensible real-time neural simulator for heterogeneous clusters of both CPU's and CUDA-capable GPUs [32]. Advantages of NCS6 are

its computation power, biological capabilities at multiple levels of abstraction and its minimum programming demand. Limitations are the lack of biophysical parameters.

NCS6 has built in support for the LIF, IZH and HH models. The LIF model can be composed of multiple compartments with different synaptic connections, which are distributed on the same device to minimize data transfer. Both excitatory and inhibitory synapses are supported. Due to the possible waveform firing of neurons over multiple time-steps, the synaptic update process contains a list with the total synaptic current for a single neuron.

Elements of the simulation are distributed across the GPUs according to the devices computational power and their dependencies. Neurons are ordered by their number of computational load, where the number of synapses are key. The lowest computational load gets assigned the heaviest computational neuron and so on. When distributed, the elements are partitioned by their sub-types managed by a plugin. The current from stimuli and synapses is computed and used to update the state of every neuron every time step.

The simulation is partitioned into several stages that can be executed in parallel as long as the data is ready. Within a stage the updating of different elements, like for instance a HH and LIF model, can be executed in parallel as well. Due to the unique ID assignment to every element, updates affect separate regions of memory.

NCS simulates up to 1 million neurons with 100 million synapses in real-time: 1 second of simulation of IZH neurons result in 1 second wall time. For their tests they used eight machines having two NVidia GPUs. The GPUs used are GTX 680s, 480s, 460s, and Tesla C2050s with a 1 ms time-step. The main reason for performance loss in very large models is due to memory constraints of the GPU and not due to network limitations.

No traces of P2P or RDMA optimization techniques are found. The project started in 1997 and no publications or software mutations are found after 2013. The project has not been abandoned, the work has focused on the user interface but that is not ready to be released.

3.10 STEPS

STochastic Engine for Pathway Simulation (STEPS) is a stochastic spatial reaction-diffusion simulator implemented in C++ with a Python user interface [33]. STEPS core simulation algorithm is an implementation of Gillespie's SSA, extended to deal with diffusion of molecules over the elements of a 3D tetrahedral mesh. It was mainly developed for simulating detailed models of neuronal signaling pathways in dendrites and around synapses.

STEPS has a well-validated support for the modeling tool and library SMBL [34] which holds various neuronal models using LIF, IZH or HH neuronal models in multi compartmental configuration, or synapse configurations. However, STEPS is meant to simulate stochastic diffusion models of small sections of neurons in contrary to deterministic whole cell models like Neuron.

Articles about their steps undertaken to implement parallelization [33] and [35] speak of a MPI based versus a GPU based implementation. The MPI-based implementation

could serve a foundation for other parallel protocols such as GPU. STEPS currently does not have a GPU nor a GPU cluster implementation.

3.11 PSICS

The Parallel Stochastic Ion Channel Simulator (PSICS) computes the behavior of neurons taking into account the stochastic nature of ion channel gating and the detailed positions of the channels themselves, written mainly in Java and Fortran [36]. It is intended to be complementary to existing tools and has its focus on stochastic behavior and detailed geometry; a class of problems that cannot be handled by other simulators.

PSICS claims to support compartmentalization preserving volume, point positions and surface area. HH style models can be implemented by automatic conversion to kinetic schemes. Models for PSICS are expressed as simple XML file format similar to NeuroML, which also supports electrical synapses.

PSCICS separates the core computations from the rest of the tasks (modeling). The channel updating process is the main cost of the core computation. It implements *embarrassing* parallelism: multiple independent realizations to compute the statistical properties which are run on different processors and combined when finished.

It makes use of the SGE to manage the cluster implementation. Little is known about the hardware to implement PCICS on, no traces of a GPU-cluster nor a multi-GPU implementation is found.

3.12 Summarizing

This chapter delineates a variety of SNNs and their technologies, with the purpose of finding a simulator that could possibly show good scaling potential. In table 3.1 an overview of the surveyed simulators and their characteristics are shown. Most of the simulators support the necessary neuron models, GJ and multi-compartment modeling. They all have in common that they have support for a multi-node implementation, either in the form of a distributed cluster, a single or a multi-GPU implementation.

Regarding SNNs optimization technologies many of the simulators have some form of optimization implemented, of which HRLsim mentions the most formidable. For instance, the parallelization of communication and computation, memory access based on firing rate and the usage of separate streams for kernels that do not depend on large memory access are good hints for optimization, since they could potentially increase performance. Furthermore, they have done some work on investigating scalability of the SNN on a multi-GPU network. When the computation of larger networks form a bottleneck for a simulator, the NP and SP from CARLsim and the ordering of neurons according to their computational load from NCS6 are interesting technologies, because they could reduce the execution time.

There are five simulators which have a multi-GPU implementation: CARLsim, CoreNEURON, HRLSim, Neurokernel and NCS6. Only two simulators implement P2P; but HRLSim is not an open source project and Neurokernel is not a general purpose simulator. CARLsim does not support the Tesla GPUs which most clusters are com-

Simulator	Neuron/Synapse					Front end				Back-ends									Platform Support						Status		
	LIF	IZH	HH	Gap Junc	MultiComp	C/C++	PyNN	NeuroML	SMBL	1GPU	MultiGPU						Distributed	Neuromorphic	MPI	OpenMPI	OpenACC	CUDA	PyCUDA	Matlab PCT	Open Source	Active	General Purpose
											Plane	Tesla	1node	>1node	P2P	RDMA											
SpiNNaker	x	x	x	x	x		x											x							x	x	x
Brian2CUDA	x	x	x	x	x	x	x	x		x	x											x			x	x	x
DynaSim	x	x	x	x	x	x											x							x	x	x	x
CARLsim 4	x	x			x	x	x			x	x		x									x			x	x	x
NEST	x	x	x	x	x		x	x									x		x						x	x	x
CoreNeuron	x	x	x	x	x		x	x		x		x	x	x					x		x				x	x	x
HRLSim	x	x				x	x			x		x	x	x	x			x	x								x
NeuroKernel	x		x	x	?			x		x		x	x	x	x				x	x			x		x	x	
NCS6	x	x	x	x	x					x	x	x	x	x											x		x
Steps	x	x	x	x	x		x		x								x		x						x		
Psics			x		x			x									x								x		

Figure 3.1: Overview characteristics of multi-node brain simulators

prised of and NCS6 was very promising shows no activity for the past years. None of the multi-GPU simulators make use of RDMA.

The answer to the question if there exists a sufficient simulator, is yes and this simulator is CoreNEURON. Next to the fact that it has support for all the neuron models, it is a general purpose simulator and GPUdirect is on their future agenda. Also it currently is a very active project with a well supported front-end.

A simulator that fully supports GPUdirect does not exist. If a larger more densely connected network is the target of the application, the interchange of data could form a bottleneck which none of the multi-GPU simulator frameworks could solve optimally. Therefore it is most likely that the simulators which run on GPUs do not scale linearly with increasing network sizes, making them impractical for researchers. The best way to proceed is not to adopt a current simulator, but to continue with the single-GPU simulator currently used at Erasmus and design a multi-GPU version, which reaps the full benefits of the fast GPUdirect interconnect technology supported by the Cartesius cluster, on which this design will be benchmarked.

Design

4

This section describes the design of the multi-GPU brain simulator. An analysis is made of the existing single-GPU application and the environment to which the simulator is to be ported. From this analysis the main phases of the design are derived and the implementation is described in detail. The main phases are the setup of the network (connection-generation), the dispersal of the required cells (cell-id-dispersal), the computation of the cells (cell-computation) and the communication and the processing of the neighboring dendrite voltages when the network has been setup (dendrite-communication).

In this chapter the following terms will be used. The network of GPUs, will be referred to as `GPU_world`. The size of the network of GPUs, will be referred to as `world_size`. A `local` GPU is the GPU in question which can have relations with `foreign` GPUs. The size of a local network, that is the size of a network of cells on a single GPU, will be referred to as `world_part`, which is defined as `network_size/world_size`, in which the network_size represents the total number of cells which always have squared relation, dimension x is always equal to y.

4.1 Single-GPU version

The original single-GPU application has been designed by [37]. It uses a lighter version of a conductance-based model of the Hodgkin-Huxley neuron; meaning less complicated inter-neuron communication mechanisms.

In the application, three compartments of the cell, namely the axon, soma and the dendrite are modeled according to ordinary differential equations (ODE), which simulate intra-cell chemical exchange of the aforementioned substances.

In fig. 4.1a. a representation of the IO network is shown. This network represent an all-to-all connected network for 6 neurons. Figure 4.1b. zooms in on the cell and shows the 3 compartments and the gap junctions.

Every simulation step is a slice in time of the brain simulation, in which new values of the compartments are calculated according to the previous calculated voltages and/or an external spike input, changing the state of the cells at any moment in simulation time. The previous calculated value, serves as input to the next simulation step iteration; this is the temporal relation of the cells.

Next to the fact that these cell have a temporal relation, the also have a spatial relation. The spatial relation entails a inter-cell connection - a connection between the dendrite compartments of the cell to be exact - called gap junction as described in section 2.1. To calculate the dendrite voltages, the target cell needs to gather all the dendrite voltages of its neighboring cells. This gathering process happens in every simulation step. The simulation lasts a fixed number of simulation steps which represent a constant duration, making the simulator time-driven.

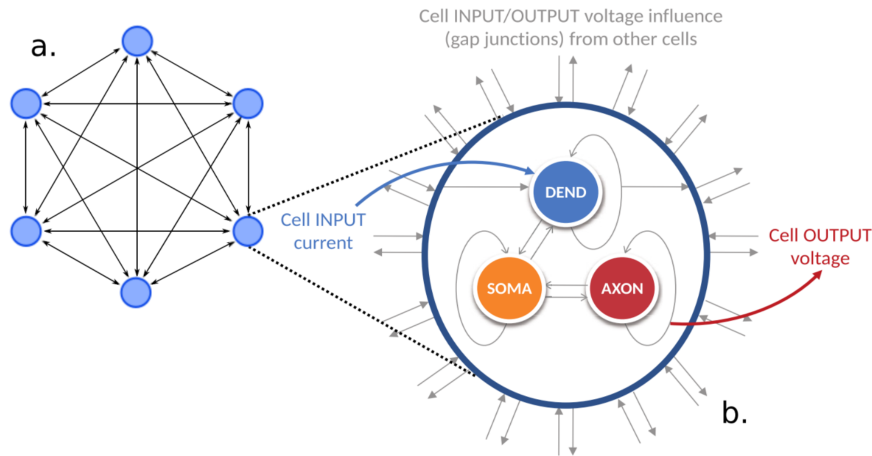

Figure 4.1: Graphical representation of the inferior-olivary network model. (a) 8-neuron network (b) single-neuron model in detail [3]

In the single-GPU version of the simulator all the neighboring voltages reside on the GPU itself. On a multi GPU version, the neighbors could very well reside on other GPUs in the network as well as on its own. The chance that a cell resides on a another GPU will increase because the network will be split evenly over the allocated GPUs. In the single version, this gathering of neighboring cells is the bottleneck of the implementation [37]. Every cell needs to loop though its neighbors, gathering the neighboring dendrite voltages to calculate its own.

When the current implementation is profiled the other two compartments of the simulator have negligible execution times in comparison to the dendrite computations. The current single GPU implementation collects 8 neighbor connections which spatially surrounds the target cell and it simulates 1M neurons within 400 seconds.

The current implementation stores the dendrite voltages in the GPUs texture memory. Texture memory is cached on chip and optimized for 2D spatial locality. Applications where memory access patterns exhibit a great deal of 2D spatial locality can benefit from using this memory. A thread is likely to read from an address nearby an address already read from and is thus more likely to be in cache memory. This is were the current implementation benefits as well. The 8 neighboring cells have a good likelihood to already be in the cache. Thus in a network where the neuron population consists of groups which reside together texture memory could be beneficial.

The shared memory cannot be used as it is not suitable for any type of application data, the data is either broadcast to all the threads in the grid or is too large to be allocated on the shared memory [37].

The best achieved performance for double precision simulations for Tesla platforms with Kepler architecture uses a block size of 32 threads and L1 cache enabled [37]. Because the GPU used on Cartesius, the K40M, also has the Kepler architecture, a block size of 32 threads is maintained with L1 cache enabled.

In fig. 4.2 an overview is given for the single-GPU implementation for the IO applica-

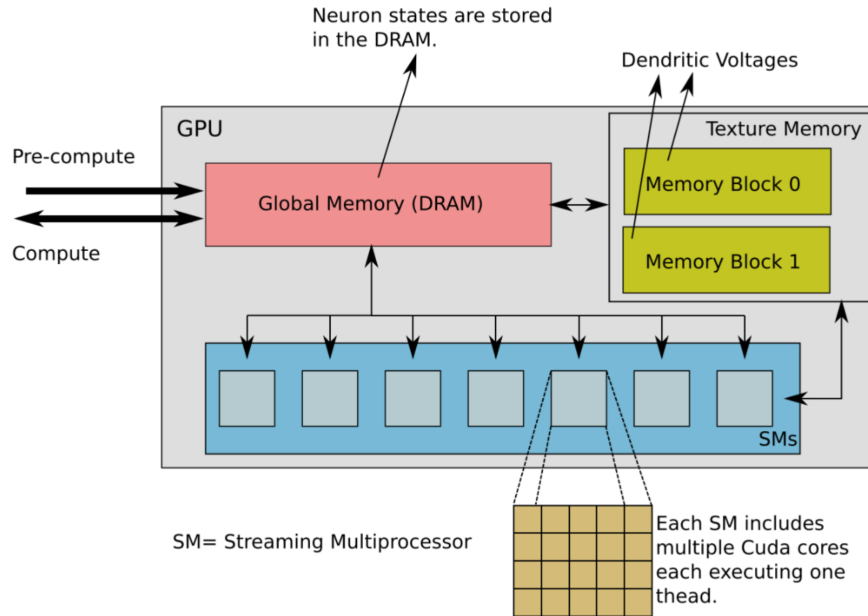

Figure 4.2: GPU implementation of the InfOli application [3]

tion. The distinction in using the global memory and texture memory becomes apparent. The pre-compute part which includes the network setup is for the single version not executed on the GPU. Also the SMs and their relation to memory is shown.

4.2 Design Overview

From the analysis of the single-GPU version two phases become apparent. The first phase is the compartment computation; this is will be known as the *cell-computation* phase. The computation of the soma and axon take place only locally on every GPU and do not have to be adapted for a multi-GPU version.

The second phase, which will have a huge impact on the multi-GPU design, lays hidden in the dendrite- computation part. When the dendrites are calculated the voltages need to be gathered. For the single-GPU version these data are gathered from the texture memory, for the multi-GPU version these data are gathered from all the GPUs in network. This is called the *dendrite-communication* phase.

For the single GPU-version every cell had 8 neighbor cells connected through gap-junctions, the multi-GPU version will have the option to connect each cell to 10, 100 or 1000 neighboring cells distributed by a random process; which will be called the *network-setup* phase. This setup phase consist of two sub-phases, namely the *connection-generation* and the *cell-id- dispersal* process.

Figure 4.3 shows the four phases of the simulation of the algorithm, the architecture on which it can be run and a general high-level pseudo code based overview of the time complexity. These phases coincide with the sections in this chapter discussing their

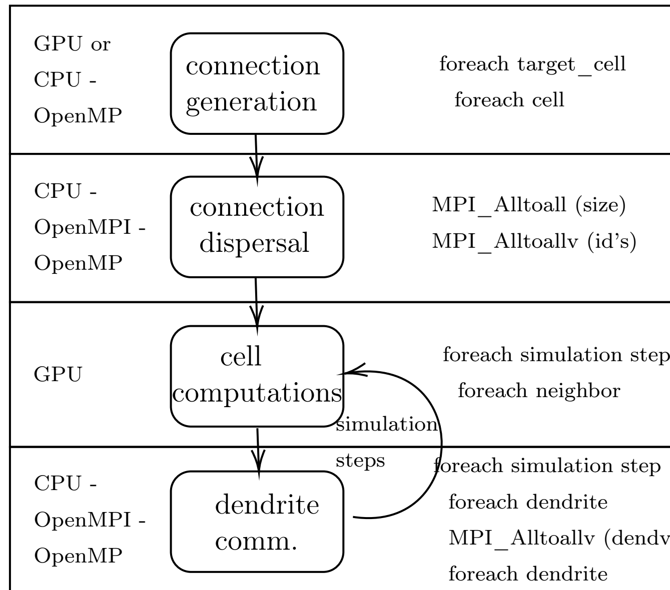

Figure 4.3: Main flowchart indicating architecture, process and complexity

implementation.

The technologies used to support the implementation phases are as follows. The OpenMPI library used to interconnect the GPUs over RDMA, is implemented via MPI collective functions to facilitate inter-node dendrite voltage communication. The OpenMP library is employed to speedup the setup of the network by binding the workload to different threads on the x86 processors to feed the GPUs. In comparison, also the GPUs are used to speedup the cell network generation for which the cuRAND library for random number generation is integrated for the distribution processes. The result is a hybrid version using both GPUs and Xeon cores for parallelization, supporting both the network generation and inter-node communication, and OpenMPI for GPUdirect inter-node communication.

When multiple GPUs are issued, each GPU takes up a portion of the total network size. Meaning that the generation and computation of the network cells are proportionally scaled across the GPUs. The cells to be calculated are split up along the column dimension of the matrix. This means that the dimensions of the CUDA computation kernel, also need to scale along the x-dimension. The column dimension will be divided by the number of GPUs. If the network size stays constant, adding more GPUs will divide the workload across the GPUs, decreasing the workload. In fig. 4.4 a network division for a network of 16 cells is visualized for 2 and 4 GPUs. The total number of cells in a network are divided between multiple GPUs. In the GPU-world of 2 GPUs 8 cells and in the GPU-world of 4 GPUs 4 cells per GPU are calculated.

Enlarging the GPU_world scales down the number of target_cell-loop iterations and thus the workload per GPU. Not only the workload per GPU scales down, also the memory requirements for storing neighbors decreases. The more GPUs the more memory becomes available and less cells per GPU need to be stored, assuming the network size

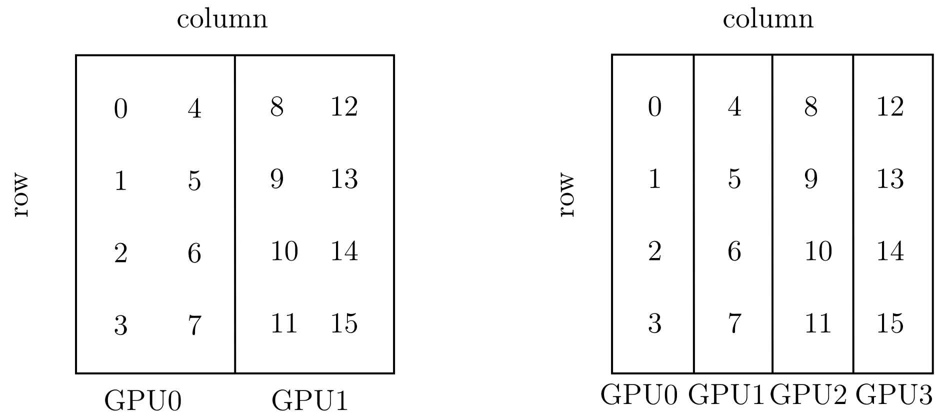

Figure 4.4: Workload division of neuron cells across GPUs for 2 and 4 GPUs

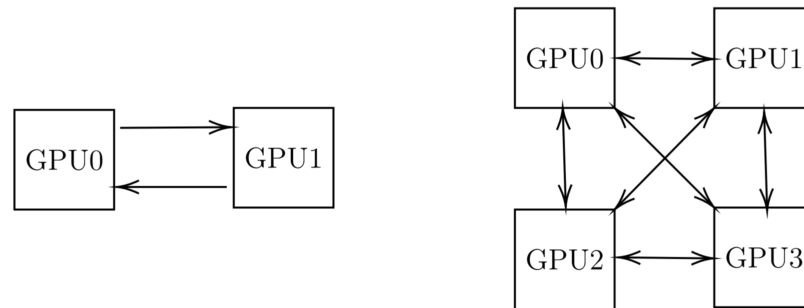

Figure 4.5: Communication interconnections with GPU_worlds 2 and 4

stays constant. The only exception is the texture memory resources. Texture memory will be kept as large as the network is, such that dendrite voltage can be stored and fetched according to their global cell addresses. The dendrite voltages of every cell will be stored here and updated after every run of the cell-computation kernel, such that each kernel can use the global cell id to fetch the neighboring voltage required.

The maximum number of neighbors a cell can have in an all-to-all connected scenario, is
`(network size - 1 ) * network size`. The -1 is due to the fact that cells cannot be connected to themselves. The maximum amount of neighbors a single target_cell can have on a local GPU is world_part-1. (7 for world_size 2, in fig. 4.4) The maximum neighbors a cell can have on a foreign GPU is the world_part (8 for world_size 2 in fig. 4.4).

An increasing GPU_world also means that the communication between nodes increases. Figure 4.5 shows the increase in communication between a GPU_world of world_size 2 and 4. The total number of communication connections - incoming and outgoing - related to the number of GPUs (n) is $n^2 - n$. In a network of 32 nodes, for instance in the case when all GPUs have neighboring relations to all other GPUs, a total of 992 communication lines could be established over which dendrite voltages are communicated.

```
1 MPI_Allgather(
2     cellVDendPtrLocal,
3     *IO_NETWORK_SIZE_H/world_size,
4     MPI_DOUBLE,
5     cellVDendPtrGlobal,
6     *IO_NETWORK_SIZE_H/world_size,
7     MPI_DOUBLE,
8     MPI_COMM_WORLD);
```

Listing 4.1: Basic communication implementation for proof of concept

4.3 Regression testing / proof of concept

Before the implementation of the main design can start, an environment has been set up, to ensure that no functionality is lost along the way. This environment consists of a tool-chain and an overly simple communication protocol implemented in the application. The tool-chain has the goal to test if the cells in the application calculated behave the same way in a multi-GPU environment, as they do on a single GPU. It should be guaranteed that when the GPU_world increases, the cell output remains constant.

The tool-chain consists of a file generator; which produces a unique output file for every GPU, a file concatenator; which pastes the output into a single file and a file comparator; which checks the concatenated file against a base line. The control of this tool-chain is scripted in a Bash-file. It compiles and runs the application in different GPU_worlds, collects the data from the GPUs and compares the output to a base file.

To validate the functionality of the program, a version has been created which communicates all the dendrite voltages of the cell, to all the GPUs by using an `MPI_Allgather`. This is a very inefficient way of communicating these voltages because a lot of unnecessary network traffic is generated. A local GPU will only need the voltages of those foreign GPUs which hosts its neighbors. However, it does the job for smaller network sizes. For larger network sizes this implementation will fall victim to increasing latencies. The `MPI_Allgather` implementation is shown in listing 4.1.

In the `cellVDendPtrLocal` array all the dendrite voltages have been collected by the kernel which does the cell computation. This array is then communicated to all the GPUs and placed in the `cellVDendPtrGlobal` array, which contains all the dendrite voltages of all the GPUs and is equal in content for all the GPUs. The array is bounded to texture memory, which is updated by this function call, after every simulation step.

With only this MPI function and the line of code gathering the dendrite voltages, a basic realization of a multi-GPU version was established and a proof of concept was made. This enables the fine tuning of the compute kernel before a more complex but efficient implementation, along with network cell generation can commence. After every major functionality affecting change in the application, this tool-chain can be run to validate correct behavior.

4.4 Connection-generation

The process of generating or reading a connection matrix is known for long execution times. This section describes the attempts to speedup this process by developing an implementation for an OpenMP and a GPU architecture.

4.4.1 The cell connection matrix

When a simulation starts a network of cells needs to be set up. This entails generation of the connections each cell has with some (or all) of the other cells in the network. The result of this process shall be referred to as the *connection-matrix* of the simulation. Next to the connections of each cell, it also stores a conductance value of the cell. This conductance value represents the weight of the cell, used to calculate the dendrite voltage. When calculating the dendrite voltage of each cell, the dendrite voltages of the neighboring cells need to be gathered. These connections could reside on the same GPU as the cell in question, or they could also reside on any other GPU in the network. To generate a 2 dimensional matrix, usually two for-loops are used, one to iterate over the columns and the other over the rows. Throughout this design these two loops are the main drivers of the generation process.

The connection-generators distribute synapses between the cells according to a uniform and Gaussian probability. These distributions have been found to be naturally occurring and represent different patterns of connectivity in the brain [14]. The difference is that the uniform distribution creates spread-out connections and Gaussian distribution keeps the connections closer to the cell to which the neighbors belong. The differences between Gaussian- and uniform cell-distribution are illustrated in fig. 4.6, borrowed from [14].

Every cell can be connected to any other cell in the network, thus for every cell the entire network of cells needs to be considered. The cells for which neighbors are considered are referred to as *target_cells*. The cells which are considered as possible neighbor candidates for the target_cells are referred to as *neighbor_cells*. To generate the neighbor connection matrix, the number of loop iterations grows quartic in relation to the dimension of the network. For instance, for a network with dimensions 2048x2048, roughly $2048^4 = 17.6$ trillion loop iterations need to take place.

Figure 4.7 shows two connection matrices for a 4x4 cell network, which display 7 neighbor connections, indicated with a conductance value. The x's indicate that cells shouldn't be connected to themselves. When the network is split up the target_cell iteration starts at 0 for every world_part. To ensure that cells are not connected to themselves on a global level, a global_cell id is maintained, which has the relation `global_cell = target_cell + GPU * world_part`.

Storing connections Whenever a connection to another cell is generated, the coordinates - in the form of a global cell id, the conductance value and a counter - to keep track of the total connections - are stored in memory according to listing 4.2. The structure is used for either external or internal connection generation.

The variable `nCount` holds the total number of neighbors per column. The variable

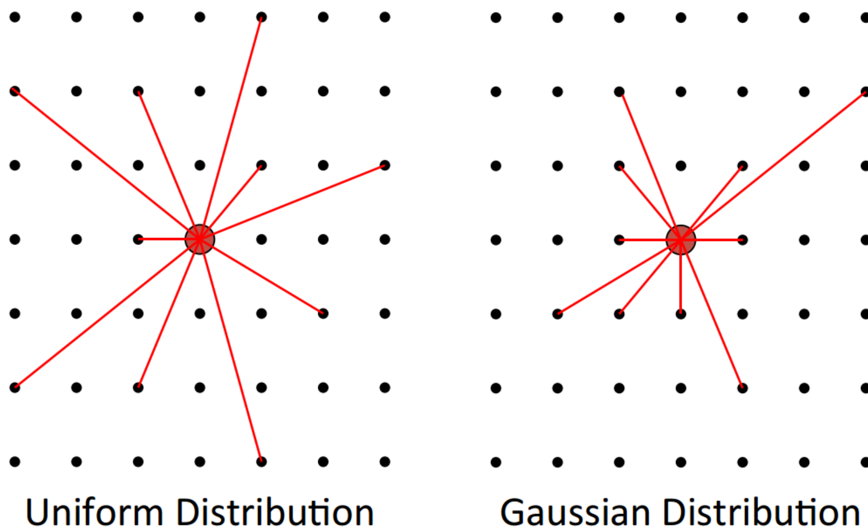

Figure 4.6: For the uniform-distribution the neighbor-connections are spread more evenly throughout the network, while Gaussian keeps them nearby

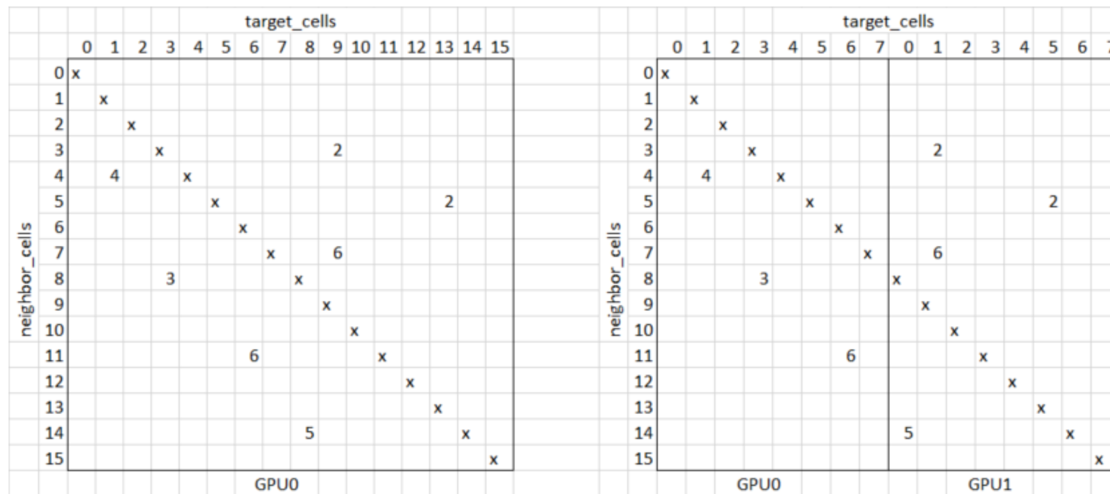

Figure 4.7: Sparsely populated connection matrices for a 4x4 cell network on 1 and 2 GPU('s)

`neighId` stores per new found neighbor the row number of this neighbor. The index of the array, which can be seen representing the column number (`target_cell(t_c)`), together with the row number (`neighbor_cell(n_c)`) form the coordinates of the connection matrix. The variable `conduc` is used to store the conductance of each cell, which can be seen as the weight of the cell. An advantage of this representation is that when to retrieve the neighbor dendrite voltages the total number of neighbors is available and not the entire network needs to be iterated.

```
1 for(int t_c=0; t_c<world_part; t_c++)
2     for(int n_c=0; n_c<world_part; n_c++){
3         nConns[t_c].neigId[nConns[t_c].nCount] = n_c;
4         nConns[t_c].conduc[nConns[t_c].nCount] = CONDUCTANCE;
5         nConns[t_c].nCount++;
6     }
```

Listing 4.2: Structure with 3 members for storing cell connections

Two implementations for neighbor connections regarding memory usage have been dveloped; a memory and computation optimized version. The memory optimized version uses the `realloc()` function to only allocate memory when a neighbor connection has been found ensuring just the right amount of memory to be stored. The computation optimized version pre-allocates space taking into reference the probability entered for the density of the network. This density plus a safe guard portion to ensure that there is always a little too much rather than too little space allocated for neighbor storage.

A hybrid version between both implementation could be considered as well, if memory becomes the limiting bound of the architecture and the safeguard option does not work. A certain amount of space is always pre-allocated and when there is need for more space, another chunk of memory is added. Not completely taking away the reallocation, but reducing these function calls.

However, a memory optimized version is not of the highest necessity since every GPU nowadays is equipped with 12 or 16 Gb of memory. When using multiple GPUs the total memory scales up, while the local network size scales down per GPU added. When taking a large network of 2 billion cells into account and a network density of 1000, all the cells would have 1000 integers for the connection plus 1 for conductance and 1 for the counter, to store. This would amount to 7.5 GB memory used of total storage space, which would fit easily on a single GPU. A version where an over allocated strategy is employed is easily maintained.

To setup the network two strategies have been pursued. The first strategy is to externally read in the connection from a file and the second strategy is to internally generate the matrix. The second strategy is a lot faster and less 'external' memory is required but the first is desirable since it makes usage of the simulator more generic and accessible. More generic since every external generated matrix - when the specifications are met - can serve as a connection index.

4.4.2 Connections from an external file

A huge problem with importing the connection matrix from an the external file is that the file can get very large for larger network sizes, which in turn lead to long importing times. Techniques to import the matrix as well as attempts to optimize execution time and storage space are discussed in this section. A strategy has been devised and implemented, but due to the fact that a single matrix file for larger network can become too large to maintain on a cluster, this option is explored but not pursued.

```
1 int FPointion = (GPU * world_part * elemsize);
2 fseek(pInFile, FPointion, SEEK_SET);
```

Listing 4.3: File pointer distribution for external acquiring of neighbor connections

Every GPU needs a part of the connection matrix. When this file is read every GPU gets allocated a starting point in the file based on the rank it has in the network, as shown in listing 4.3.

The variable `GPU` represents the rank of the GPU in the network. The variable `elemsize` represents the size of the element in the file. This latter variable preludes a first optimization implemented. The matrix generator already present for the single GPU version - to read in a matrix from a file, it first has to be generated - produces double precision elements for the conductance value. Since the initial value was found to be represented by a single digit, the conductance in the matrix was altered to be represented by an integer data-type. This reduced the size of the connection matrix by a half. Also the connection matrix contained spaces between the elements. These spaces are obsolete since it is possible to read in the file per element specified. This reduced the matrix size again by a half.

Another explored optimization, but not implemented is bit fielding of the elements. A single integer data-type, which has 4 bytes to represent its value, is split up into, for instance, 4 parts. A integer could represent 4 conductance values, reducing the matrix 4 fold. Each value would have the range of 256 decimal values. This could be reduced even further to 8 parts, giving a range of 16 decimal values per bit field. A single integer would represent 8 conductance values, reducing the matrix 8 fold in size. This optimization could also be implemented when the CUDA kernels do their cell computations.

A network connection matrix of 2B cells would need $4x10^{12}$ entries representing neighbor connections and conductance values. In the present representation of the matrix, including spaces, the size is 60 Tbytes of data. Implementing all of the aforementioned optimization's would reduce the matrix file to roughly 1 Tbyte. Each GPU would then have to process 1T/n*GPU size of the matrix.

4.4.3 Connections internally generated

Having both the internal and external functionality to create a cell connection matrix is a plus, since they are both useful under different user conditions, however to align the research with the Xeon Phi tier, the internalizing of the cell generation is pursued. This means that the cell generation is done in the same execution run in which the simulation is started. The connection matrices are internally generated according to a uniform and a 3D-Gaussian distribution which have been adopted from [14]. To find the fastest way to generate the connections, a strategy to parallelize these generator through OpenMP and CUDA are pursued.

In listings 4.4 and 4.5 pseudo-code for both distribution algorithms is shown. When comparing the two, the Gaussian generator displays a lot more computational complexity than the uniform generator. The uniform generator is based on comparing a fixed probability against the outcome of the standard GNU random generator, while the latter

```
1 for (int t_c=0; t_c<world_part; t_c++)
2   #pragma omp parallel for schedule(static)
3   for (int n_c=0;n_c<*IO_NETWORK_SIZE_H;n_c++) {
4       rndm = ((float) rand()) / ((float) RAND_MAX);
5       if (rndm <= probability) {
6           // store neighbor and filter cell to for other GPUs
7       }
```

Listing 4.4: Uniform cell distribution

```
1 for (int t_c=0; t_c<world_part; t_c++) {
2   total_amount_connections=generate_gaussian_number();
3   while (connections_found<total_amount_connections) {
4   #pragma omp parallel for schedule(static)
5   for (int n_c=0;n_c<*IO_NETWORK_SIZE_H;n_c++) {
6     s_x=n_c/base_3D_squared;
7     s_y=(n_c%base_3D_squared)/base_3D;
8     s_z=(n_c%base_3D_squared)%base_3D;
9     d_3D=(sqrt(pow(my_x-s_x,2)+pow(my_y-s_y,2)+pow(my_z-s_z,2)))/scale;
10    gaussian_prob = const1 * exp(pow(d_3D-mean,2)*const2);
11    rndm = ((float) rand()) / ((float) RAND_MAX);
12    if (rndm <= gaussian_probability) {
13        // store neighbor and filter cell to for other GPUs
14        #pragma omp atomic
15        conn_gen_buffer[n_c]++;
16        #pragma omp atomic
17        connections_found++;
18        }
19    }
20 }
```

Listing 4.5: Gaussian cell distribution

needs a lot more computations to ensure a close distance to origin. The uniform generator consists of two for loops iterating over the part of the network that is to be processed by a single GPU - world_part - and the entire network. The first loop is the target_cell-loop and the second is the neighor_cell-loop. Every cell in the network gets a probability to become one of the target_cells neighbors and is compared to a fixed probability. The same holds for the Gaussian generator only the random values are now compared to a Gaussian probability. Also a third (while) loop has been added to ensure that enough connections have been found, aggravating the network generation even more.

It will not be a surprise that the setup time for the latter generator will be significantly larger than the first. In reverse, the computation time of a Gaussian network will be significantly smaller than the first, speculatively for two reasons. The first reason is that spatial cache locality - for texture memory - plays a role. The locality will ensure less cache misses. The second reason is that for a Gaussian network the cells will more likely reside on the same GPU as the target cell. It is to be expected that less network traffic is generated for a Gaussian network.

The density setting for the network has been exposed to the user and values for 10, 100 and 1000 synapses per neuron will be bench-marked. This density is related to the probability variable shown in the listings 4.4 and 4.5. For the uniform distribution the probability is equal to a portion of the network size: `probability = Density / Network size`. The larger the network size, the smaller the probability for a potential connection will be, thus ensuring a constant number of connections per cell, when the network size varies. This relation works well for the uniform distribution.

The Gaussian generator comes with more tuning options. According to [14, 38], to make sure the synapses are mostly centered around the neuron a value of 1 for the standard deviation and a 0 for the mean should be used. The Gaussian_probability is not so much related to the network size, as is the case for the uniform distribution, but to the distances cells have to each other. The probability, as defined earlier, plays a role in determining how many iterations the while loop needs to undergo before assuring the right amount of connections are found. However, this does not always assure the minimum required neighbor connections.

The scale option provides a possibility to control the number of neighbors each cell will generate. Namely, the larger the scale the higher the `Gaussian_probability` and the easier it gets to find neighbors. Also a larger density is earlier obtained than a smaller one. This is because of the third loop which continues calling the kernel, when the total amount of connections is not sufficient. A lower probability entails a longer search period for possible neighbor cells. A higher probability means more neighbors in shorter period.

Per network size it was tested to find the optimum setting for the scale option. Setting it too low would mean an increase in setup time, setting it too large means to many neighbors per cell. The optimum setting found was 5, making network setup time very tolerable and producing the right amount of neighbors in relation to network size.

Execution of these generators for large networks (1M-2M cells) sequentially made the generation of the network being executed very time consuming and nearly impossible. The possibility to speed this up came in the form of OpenMP or porting the cell generation onto the GPUs. To see which one is most effective, proof has to be established empirically. The GPU nodes of the Cartesius cluster have Xeon processors with 8 cores available to 'feed' the GPUs. With these 8 cores the generation of the network could very well be reduced to reasonable execution times.

4.4.3.1 OpenMP connection-generation

OpenMP is a library that supports multi-threading for Intel processors, making it possible to bind a thread to a core with a relative simple interface. To set the number of threads, the environment variable
$OMP_NUM_THREADS has to be set in correlation to the `-task per node` flag which indicate the number of tasks that will be instantiated on a node. The number of cores is equal to the number of threads if no more than 8 threads per node are issued. The GPU_world can increase and decrease independently of the number of cores issued. In listings 4.4 and 4.5 the `#pragmas` to parallelize the generators are shown. The most inner loop is a good target because this is relative large. Scheduling nested loops is not

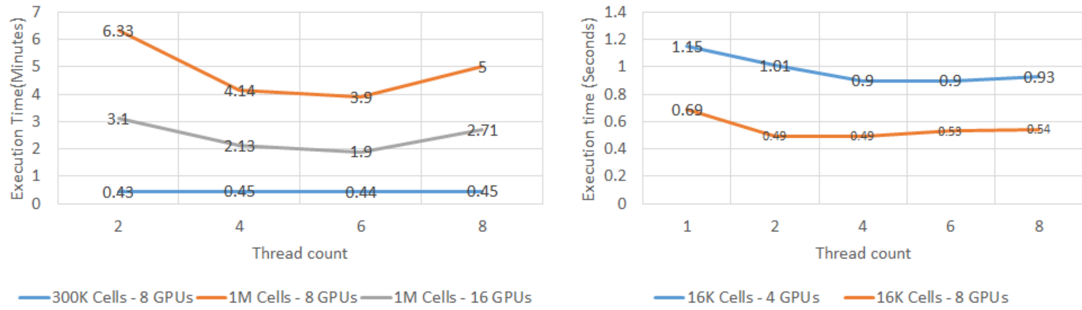

Figure 4.8: Thread variation for larger and smaller networks using Gaussian generator on OpenMP

impossible within OpenMP, but often proof detrimental.

Since the treads are most of the time (depending on the probability) roughly equal in execution time, a static policy will have good result, as this has little overhead. A static policy entails that all the iterations of the loop are divided evenly among the threads. In contrast to static, a dynamic policy could also be adopted. Dynamic policies allow scheduling to take place during execution time. When tasks to be scheduled differ much in length, the overhead of this policy could be overcome by the gain of shorter execution times by scheduling tasks online. The network generation will benefit most likely from a static schedule. The question of what would be the ideal thread number, could also best be answered empirically.

In fig. 4.8 in minutes and in seconds the execution times are displayed for varying network sizes, for a thread count of (1), 2, 4, 6 and 8. A Gaussian generator is used because this has the greatest computational complexity. For the smaller network sizes no significant speedup has been observed after using more than 4 cores. The relative differences are minimal and have no significant effect on absolute execution times. For larger networks the ideal thread number appears to be 6 and not 8. An explanation for these phenomena could be that for 8 threads the cache is misaligned. Cache content is in a way ping-ponged between threads; resulting in more cache misses than for instance 6 threads. Using more threads and thus cores is not beneficial for larger network sizes.

4.4.3.2 CUDA connection-generation

The other option to decrease setup times of larger networks is to port the generators and part of the cell-id-dispersal phase, to the GPUs. It is possible to issue multiple kernels for different grid sizes in a single application. A big advantage is that the generated network would already be present on the GPUs and doesn't need to be copied in a distinct step from the generation module/kernel to the calculating kernel. A downside is that a different library for randomness needs to be implemented and it is unsure how this would effect performance. To realize this, a single GPU version which simulates the multi-GPU implementation was first developed and later integrated with the calculating kernel.

The `#pragma omp parallel for` and `#pragma omp atomic` from section 4.4.3.1 al-

```
1 for (int tcell=0; tcell<world_part; tcell++)
2   srand((unsigned int) time(0) + GPU);
3       for (int i = 0; i < *IO_NETWORK_SIZE_H; ++i)
4           rndm[i] = ((float) rand()) / ((float) RAND_MAX);
```

Listing 4.6: Standard randomness for generator-kernels

```
1 curandStateXORWOW_t localState;
2 curand_init((unsigned long long)clock() + k + GPU, 0, 0, &localState);
3 float cuRndm = curand_uniform(&localState);
```

Listing 4.7: cuRAND randomness for generator-kernels

ready indicate which part is the candidate for parallelization on the GPU and which part of the code needs to be an atomic operation. Basically, the last loop of both generators - shown in listings 4.4 and 4.5 - is a good candidate to be parallelized on the GPU. This loop has the size of network, the kernel dimensions will be matched accordingly.

Randomness Random number generation is a vital part of both network generators. Unfortunately, the library function `rand()` cannot be issued inside a kernel. Two solutions to create randomness inside the kernel are tested.

The first solution is to fill an array on the CPU side with as many random numbers as necessary - in this case the size of the cell network, because every cell needs a distinct random value. This array is copied to the device. Listing 4.6 shows the random generation for the kernel. Observe the built in factor to add the GPU rank to seed the random function differently for every GPU and the fact that it is inside the first loop. To create randomness for every target cell, the random numbers have to be regenerated for each target_cell, placing a heavy burden on the generators. As was observed, the execution time of neighbor generation grows exponentially when the network size grows.

For the second solution, the cuRAND library is used. This library provides facilities that focus on the simple and efficient generation of high-quality pseudo-random and quasi-random numbers. The advantage of quasi-random over pseudo-random is, that the results are more evenly distributed, the latter is bothered by clustering. Since scaling of the algorithm is under investigation - number crunching is more important than distribution, a pseudo-random generator is selected.

The library consists of two parts: a library on the host and on the device side. The user can either generate random numbers and store them in global memory to be used in kernels later or generate sequences of random numbers and use them directly. The first option can be issued from host or device and stored on the host or device. The header file `curand.h` needs to included. The second option enables direct consummation of the random numbers without writing and reading to and from memory. The device header file `curand_kernel.h` needs to included.

Both options were explored but the second option fits the application best. Listing 4.7 shows the implementation of the device random generator.

The `curand_init(seed, subsequence, offset, state)` is the function that seeds

```
int neigh_cell = j*(*IO_NETWORK_DIM2_H)+k;
if cuRndm <= gaussian_probability) {
  ... store neighbor information ...

    atomicAdd(&conn_gen_buffer[neigh_cell], 1);
    atomicAdd(connections_found, 1);

    // foreign cell determination
    GPUdomain=(neigh_cell/world_part);
    if (GPUdomain!=GPU) {
        // these cells will be requested from other GPUs
        if (duplicateRegister[neigh_cell]==0) {
        xcell[atomicAdd(xcell_size_local, 1)] = neigh_cell;
        atomicAdd(&xcellno_perGPUlocal[GPUdomain], 1);
        duplicateRegister[neigh_cell]++;
    }
}
```

Listing 4.8: Storing and filtering the foreign cells

the random function. This initializes a XORWOW state in `state` with the given `seed`, `subsequence` and `offset`. The seeds are arbitrary bits used to seed the function. This function gets seeded by the UNIX time at the moment of initialization; guaranteed to be unique. The subsequence is the starting point of the sequence and the offset is an absolute offset into the sequence. Both values are recommended to keep at `0` for faster implementation.

The XORWOW generator 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. In the XORWOW variation the period is increased by adding a simple additive counter module 2^{32} to produce a period of $2^{160} - 2^{32}$. The function `curand_uniform(&localState)` returns a uniformly distributed float between 0.0f (excluded) and 1.0f (included).

Foreign cell selection Listing 4.8 shows the last part of the implementation of the generators inside the kernels, which for both generators is basically the same. The main difference is the comparison of the random value generated with `cuRAND` with a fixed probability - in case of the uniform, or with a Gaussian probability, in case of the Gaussian distribution. All cells are considered in parallel whether or not they are a candidate to be a neighbor for the target_cell. All to be considered cells are given a thread on the GPU and thus the thread id is the neighbor_cell id to be stored.

Whenever a neighbor_cell does not belong to the local GPU, it is stored in the `xcell` array. The `duplicateRegister` on line 16 has been added to filter the foreign cells such that they appear only once in the collecting `xcell` array. The dendrite voltage of a cell only needs to be collected and send once per simulation step. A neighbor_cell residing on the same GPU, is very likely a neighbor to another cell, however this value needs to be send over only once after it has been recalculated. For larger densities the uniqueness of the connection is limited by the total number of cells minus the cell that resides on

```
for (int colNo=0; colNo<*IO_NETWORK_SIZE_H/world_size; colNo++) {
    global_cell_id=colNo+GPU*world_part;
    uniGenn_kernel <<< gridDim, blockDim, 0, streamUNI>>>(...);
    cudaDeviceSynchronize();
}
```

Listing 4.9: Execution of the uniform kernel

```
for (int target_cell=0; target_cell<world_part; target_cell++) {
        global_cell_id=target_cell+GPU*world_part;
    while (*connections_found<total_amount_connections) {
        gausGenn_kernel <<< gridDim, blockDim, 0, streamGAUS>>>(...);
        cudaDeviceSynchronize();
}
```

Listing 4.10: Execution of the Gaussian kernel

the local GPU.

Also the size of each portion of foreign cells destined for each GPU - minus its own - is determined such that later on these cells, and only these cell, are communicated to other GPUs. This can be see in the listing on line 14. Each GPU most likely has a different set of neighbor connections required from the other GPUs, therefore an effective communication should take into account the different size of the total required cells.

Kernel execution evaluation The kernel is executed for every target cell as can be observed from Listings 4.9 and 4.10. The calling of the kernels are enclosed in the first loop which iterates over the target_cells. This loop scales when the GPU_world network varies. The difference between the kernel calls for both generators, is that the Gaussian kernel is enclosed in a second loop. The `*connections_found` is the loop iterator variable that needs to be accessible on both the GPU and CPU side, and is updated when the kernel finishes. This extra loops increases the complexity of the Gaussian connection generation.

The results are shown for generation of the network for 8 Xeon cores versus 2 and 16 GPUs, in fig. 4.9. Even, when run on 2 GPUs the speedup is roughly 10 times. Increasing the GPU_world increases the speedup even more. Generating the network on 16 GPUs gives a speedup of almost 77x. The GPUs make it possible to create a network within seconds instead of many minutes for the Xeon cores.

4.5 Cell-id-dispersal

After the CUDA network generation the `xcell` array contains the global cell count of all the neighbors foreign to the local GPU. This information needs to be transmitted to all the foreign GPUs such that they know which dendrite voltage from which cell id to gather and send back to the local GPU after each simulation step. This section describes the first part, the dispersal of the required global cell id's from local to the foreign GPUs.

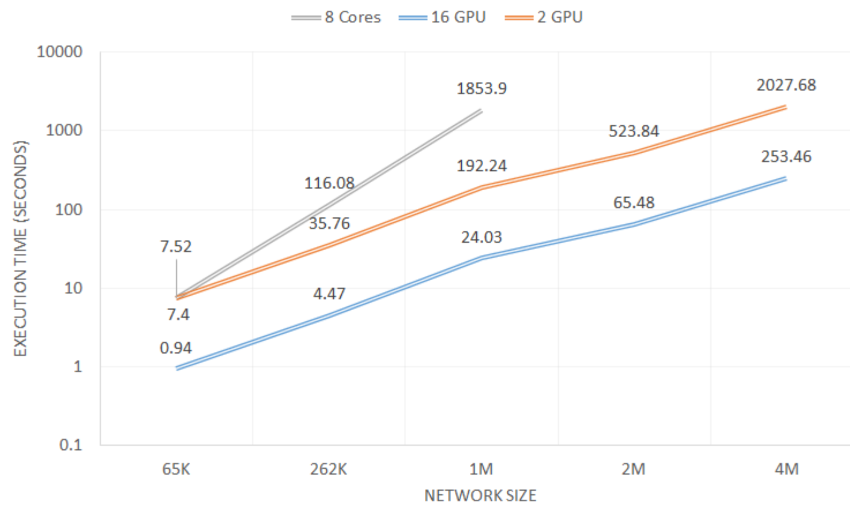

Figure 4.9: Uniform network generation on 6 Xeon cores versus 2 and 16 GPUs

```
1 for (int i = 0; i < world_size; ++i)
2     src_length[i]=1;
3
4 MPI_Alltoall(
5     xcellno_perGPUlocal,
6     1,
7     MPI_INT,
8     cellno_perGPUglob,
9     1,
10    MPI_INT,
11    MPI_COMM_WORLD
12 );
13
14 int xcell_size_global=0;
15 for (int j = 0; j < world_size; ++j)
16         xcell_size_global+=cellno_perGPUglob[j];
```

Listing 4.11: Foreign cell count communication

To communicate the foreign cell-id's, first the total count of foreign cells, has to be communicated. The collective function `MPI_Alltoall()` is used for this part of the dispersal, shown in listing 4.11. The Alltoall function takes the first part of the `xcellno_perGPUlocal` from all the GPUs, in this case only one element, and sends this to the first GPU, and then a second part of all the array's and sends to the second GPU, and so on. In fig. 4.10 a graphical representation of the function is displayed.

This figure visualizes the Alltoallv function, used in the next part, in which the 'v' stands for 'varying'. The difference with a regular Alltoall is that for the second function, the size of the data to be send can vary. An Alltoallv with a fixed send count behaves the same as the Alltoall function.

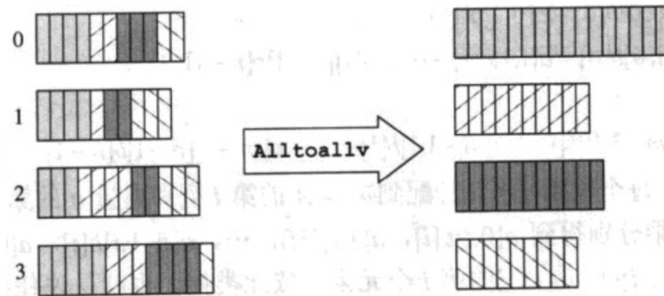

Figure 4.10: Graphical representation of MPI_Alltoallv function [10]

```
1 sdispls[0] = 0;
2 for (int i = 1; i < world_size; ++i)
3         sdispls[i] = sdispls[i-1]+xcellno_perGPUlocal[i-1];
4
5 rdispls[0] = 0;
6 for (int i = 1; i < world_size; ++i)
7         rdispls[i] = rdispls[i-1]+cellno_perGPUglob[i-1];
8
9 MPI_Alltoallv(
10        xcell,
11        xcellno_perGPUlocal,
12        sdispls,
13        MPI_INT,
14        cellstoRetrieve,
15        cellno_perGPUglob,
16        rdispls,
17        MPI_INT,
18        MPI_COMM_WORLD);
```

Listing 4.12: Foreign global cell id communication

From the figure it becomes apparent that this function basically performs a transpose on the data across the GPU network. After the execution of the Alltoallv every GPU now knows the count of cells that need to be send to every GPU but itself. This count is the total of the cells that will be received per GPU, represented in `cellno_perGPUglob`. The GPUs yet only know the count of the to be received global cell-id's, which they have to send back.

Because the GPUs know the cell count, memory can be allocated on the GPUs for the to be received global cell id's. To send the global cell-id's the collective function MPI_Alltoallv() is utilized, shown in listing 4.12. The number of neighbors vary per GPU, thus not every GPU needs the same amount of cells-id's to be send, therefore with this function the exact amount of cells are send, keeping communication to the exact minimum. For this function to operate, the displacement of the send and receive array's needs to be determined. The displacement is the location from where this function grasps and puts its data in the send and receive buffers respectively. Both the local and global sizes are required to set up the send and receive displacement.

```
1  int cell = j*(*IO_NETWORK_DIM2_D/world_size)+k;
2
3  for (int i=0; i<neighConns[cell].neighCount; i++)
4  {
5      neighbor=neighConns[cell].neighId[i];
6      V = prevdend_IC[cell] -
7      fetch_double(textMemPointer, (neighbor % *NETWORK_DIM1), ((int) (neighbor
        / *NETWORK_DIM2)));
8
9      f = 0.8 * exp(-1*pow(V, 2)/100) + 0.2;    // SCHWEIGHOFER 2004 VERSION
10     condu=((double)(dev_neighConns[cell].neighCondu[i])/100000);
11     I_c = I_c + (condu * f * V);
12 }
13 return I_c;
```

Listing 4.13: Target_cell dendrite voltage computation

After the execution of this function, every GPU knows which dendrite voltage to gather and to which GPU it has to be send. These global cell-id's are stored in `cellstoRetrieve`. The length and displacement are the indicators for returning the gathered dendrite voltages to the requiring GPUs. After this second step the network has been setup and the computation can commence.

4.6 Cell-computation

In every simulation step the three cell compartments, the axon, soma and dendrite voltages, of the neuron are computed. Whenever a cell is a connected to another cell, the dendrite voltage of this neighboring cell is required for the computation of the target_cell. The computation of the dendrite voltages is the biggest bottleneck in performance for the single-GPU version. It is a bottleneck because it includes the gathering of the dendrite voltages from the neighbors of the target_cell.

The simulation runs for several simulation steps. For 100 ms real brain time, the simulation needs to execute 2000 simulation steps. The computation of the dendrite voltages, as shown in listing 4.13, takes place every simulation step for each neighbor_cell of the target_cell. The target_cell has a relation to the threads of the CUDA kernel, which is used to fetch the correct neighbor information.

When calculating the new dendrite voltage, the neighboring dendrite voltages are subtracted from the dendrite voltage of the target_cell. The neighbor_cell is a global id, which is transformed into coordinates to obtain the dendrite voltages from texture memory. All the GPUs have an equally sized texture memory according to the size of the cell network, this is the only part of the network which does not scale when the GPU_world is increased. The population of this memory is different for every GPU, since the required neighboring cells vary from GPU to GPU. The difference of this voltage is then multiplied with constants and the conductance value, to form the new dendrite voltage of the target_cell.

Next to the computation of the new voltages for the dendrite compartment, also the

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < xcell_size_global; ++i)
        dendVtoSend[i]=cellVDendPtr[cellstoRetrieve[i]];

MPI_Alltoallv(
        dendVtoSend,
        cellno_perGPUglob,
        rdispls,
        MPI_DOUBLE,
        dendVtoStore,
        xcellno_perGPUlocal,
        sdispls,
        MPI_DOUBLE,
        MPI_COMM_WORLD);

#pragma omp parallel for schedule(static)
for (int i = 0; i < xcell_size_local; ++i)
        cellVDendPtr[xcell[i]]=dendVtoStore[i];
```

Listing 4.14: Dendrite voltage communication during simulation

next voltages for the soma and the axon compartments are computed in each simulation step. The kernel calls three sub-functions, each relating to one of the compartments, computing the next voltages using the equations in [13]. None of the soma or axon computations need voltages from other cells. These computations only need their own previous voltages and a external input and thus do not require any data from foreign GPUs. Therefore, these calculation do not have an implication for the multi-GPU design.

4.7 Dendrite-communication

The newly calculated dendrite voltages are stored after each computation step in an the array `CellVDendptr` which is bound to texture memory. Only after the CUDA kernel is finished, the update process of the texture memory can be conducted.

After every time step the new dendrite voltages are communicated to the requiring GPUs and the texture memory is updated with the new value, such that when the new computation starts, the new neighboring voltages can be fetched from texture memory. Thus after each simulation step the dendrite voltages need to be gathered, scattered across the network and updated in the texture memory. In listing 4.14 this part of the code in shown.

The first for-loop gathers the calculated dendrite voltages and packs them in `dendVtoSend[]`. The global cell-id's serve as the index to retrieve the dendrite voltages. When retrieved the cells can be send back to the needing GPUs. This is a reciprocal operation of the previous `MPI_Alltoallv` function, described in section 4.5. The results are returned to the same location as where they came from.

The second for-loop unpacks the received results into the `cellVDendPtr` which is bound to the texture memory, using once again the global cell id as the array index and thus updating the memory with the required values for computation in the next

simulation step. Both loops are implemented with multi-threading implemented with OpenMP to speedup the packing and unpacking of the dendrite voltages.

4.8 Summary

This section has discussed the design for the multi-GPU version for the four main phases of the application. The application setup for the benchmarking will contain the connection generators implemented on the GPUs using the cuRAND random generator, for the cell-connection phase. The cell-dispersal phase does not contain any parallelizing enhancements and communicates the cell-id's through OpenMPI across the GPU_world. The cell-computation phase is executed on the GPUs on which every thread resembles a neuron cell with 3 compartments. The dendrite-communication phase occurs after every computation step and uses OpenMPI to transfer the data across the GPU_world. The packing and unpacking of the voltages are parallelized with the aid of OpenMP.

5 Evaluation

In this chapter the benchmarking is reported of the proposed design from chapter 4. This evaluation is done to analyze the behavior of the four phases of the design: connection-generation, cell-id-dispersal, cell-computation and dendrite-communication. For the four phases the absolute execution time is measured to establish which phase forms the bottleneck for the design and to give insight in how long the simulations run. When the execution times are known the speedup relative to the execution time of a single GPU_world is calculated. The speedup gives an indication in the performance improvements when parallel computing resources are added.

Next, the energy usage of the design is determined. The energy used is one of the metrics on which a cluster bases its billing, thus this analysis also indirectly takes the cost in terms of financial budget of the simulation into account. Because the speedup is known an analysis of the efficiency of the design, in terms of energy versus speedup, can be established. If a great speedup comes at the price of consuming a lot more energy, and thus has a greater impact on the financial budget, it might be more efficient to execute with fewer GPUs at the cost of a higher execution time. Since the consumed energy is a more generic metric than for instance the billing units of the cluster, these measurements will provide a more general comparison afterwards, when for instance other clusters are used for the execution of the design.

To chart the behavior of the design beyond regular execution times and financial cost, an extrapolation is made based on the gathered data to predict which factors limit this design and to show how the execution times of the phases behave for larger network sizes. Also, for the phases where the GPUs are used, the connection-generation and the cell-computation phases, a roofline model is constructed to analyze the behavior of the GPUs in terms of memory bandwidth and peak performance. This gives insight in hardware limitations and could offer potential optimizations by analyzing if the design is memory-bound or compute-bound.

5.1 Experimental setup

There are three dimensions which are varied running the benchmark, these are the network size, the density and the GPU_world. This benchmark is executed for uniform and Gaussian distributed networks. These are natural occurring distributions which also gives insight in the impact of different network topology on the design.

The network size varies from 65K to 2M neurons and always has a squared relation, meaning that the width is always equal to the height of the network. For each network size, densities of 10, 100 and 1000 synapses per neuron are simulated. The settings for these dimensions match and surpass connectivity encountered in the IO nucleus and aim at revealing the simulator performance trends [14].

The GPU_world is increased to 8, 16, 32 and 48 GPUs. Every increase is a doubling of the amount of GPUs, except for the last. Because some nodes are under maintenance, 48 and not 64 GPUs are used. By increasing the GPU_world, the workload per GPU will decrease and the expectation is that the execution time will decrease as well. In table 5.1 an overview of the parameter space in displayed.

Name	Range
Network size	65K-2M neurons
Density	10-1000 synapses/neuron
Synaptic pattern	Uniform (U) and Gaussian (G)
World_size	8-48 GPU nodes

Table 5.1: Parameter space for InfOli benchmark

The simulator is executed for 2000 simulation steps, which resemblances 100 ms of brain time. The C-function `gettimeofday()` is utilized to get the execution times of the phases that do not use the GPUs: cell-id-dispersal and dendrite-communication. For the other two phases the functions `cudaEventRecord` for recording the begin and end of the kernels, and `cudaEventElapsedTime` to compute the elapsed time between these two events, are utilized to get the execution times. Input and output have been minimized to measure only the bare execution times.

The gathered data are visualized in 3-dimensional column graphs with a logarithmic time scale with a base of 10 on the vertical axis versus the network sizes and densities on the horizontal axes. To directly compare the uniform with the Gaussian distribution, both are displayed in each graph for a particular GPU_world. The graphs show the execution times for all the network sizes and all the densities for both uniform and Gaussian distributions. The execution times for the four main phases are reported separately. The speedup is discussed, for the total execution times, the connection-generation and cell-computation phase. The other two phases are not invoked when executed on a single GPU and thus no speedup can be calculated. Per simulation the average number of generated neighbors were checked for each density setting. In all cases the average number met the required density setting.

5.2 Connection-generation

In figs. 5.1 and 5.2 the setup times for the connection-generation are displayed. As expected the Gaussian generator takes more time to setup a cell network than the uniform version. This is simply because the Gaussian generator is more complex than the uniform. The fastest time to generate the largest network, is roughly 66 seconds for uniform and 152 seconds for Gaussian distributions in a GPU_world of 48. Doubling the amount of GPUs reduces the overall network setup by half. The speedup relative to a GPU_world of 1 and the execution times are displayed in tables 5.2 to 5.4 for the uniform-generator and tables 5.5 to 5.7 for the Gaussian-generator. It shows that overall the speedup for the Gaussian is a little higher than for the uniform-generator. The Gaussian-generator benefits better from an increasing GPU_world. This is due to the fact that the occupation

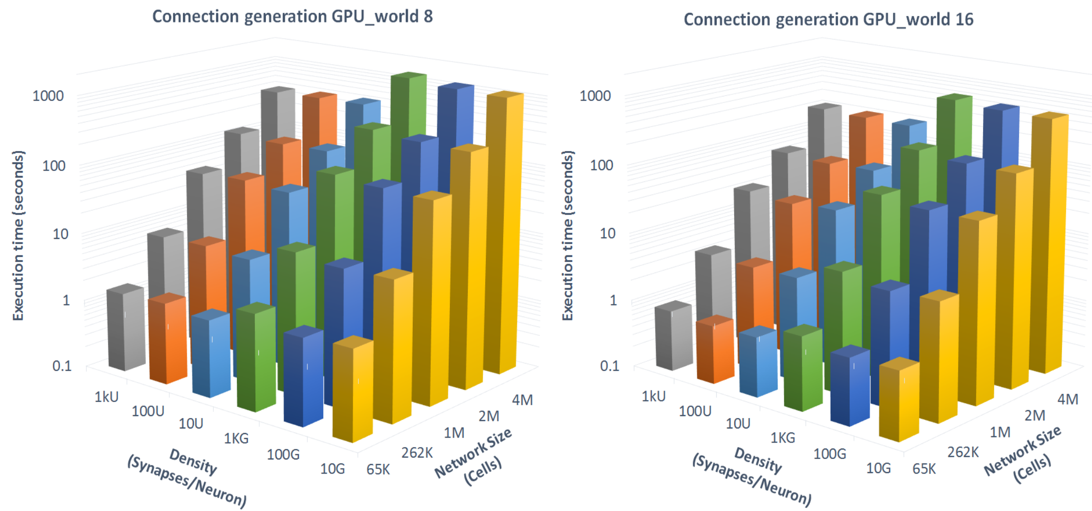

Figure 5.1: Execution times for uniform (U) and Gaussian (G) connection- generation phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right)

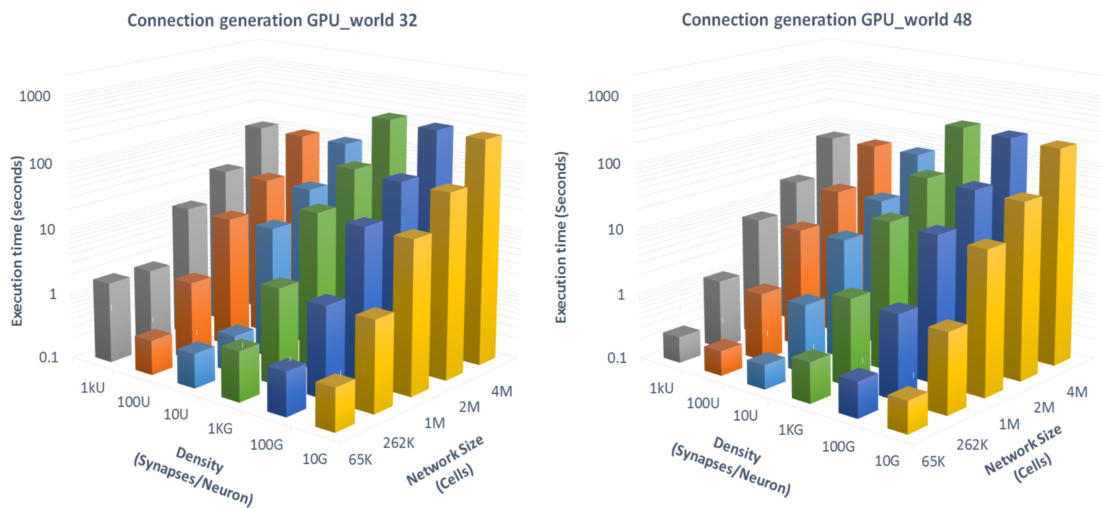

Figure 5.2: Execution times for uniform (U) and Gaussian (G) connection- generation phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right)

of the computing units is higher, the GPU's utilization is more efficient for the Gaussian-generator.

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	9.34	1.33	7.00	0.75	12.48	0.33	28.02	0.23	40.19
262K	44.92	6.28	7.15	3.39	13.24	0.36	124.14	1.10	40.90
1M	290.79	39.82	7.30	21.79	13.34	10.01	29.05	6.75	43.08
2M	791.11	113.54	6.97	58.11	13.61	26.99	29.31	17.85	44.31
4M	4554.97	412.04	11.05	196.89	23.13	97.33	46.80	65.34	69.71

Table 5.2: Speedup for uniform connection-generation - 10 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	9.16	1.55	5.90	0.72	12.79	1.67	5.48	0.24	38.53
262K	44.04	6.94	6.34	3.31	13.29	1.57	28.11	1.07	41.13
1M	286.58	44.13	6.47	19.60	14.57	9.84	29.02	6.63	43.10
2M	776.93	109.11	7.12	54.96	14.14	27.13	28.64	17.89	43.43
4M	4473.34	402.48	11.11	200.8	22.28	97.37	45.94	65.62	68.17

Table 5.3: Speedup for uniform connection-generation - 100 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	9.82	1.45	6.78	0.79	12.44	0.34	29.27	0.25	39.30
262K	45.85	6.63	6.92	3.54	12.95	1.63	28.07	1.13	40.57
1M	299.26	40.78	7.34	22.08	13.55	10.2	29.24	6.79	44.07
2M	814.14	118.76	6.86	59.93	13.58	27.7	29.41	18.30	44.49
4M	4687.58	386.28	12.14	210.4	22.28	101	46.59	66.84	70.13

Table 5.4: Speedup for uniform connection-generation - 1k synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	12.36	1.94	6.39	0.96	12.88	0.44	28.17	0.30	41.08
262K	71.88	10.21	7.04	5.07	14.16	2.34	30.76	1.61	44.53
1M	677.75	80.91	8.38	41.93	16.16	20.48	33.09	15.01	45.17
2M	1999.98	260.74	7.67	130.35	15.34	64.41	31.05	47.74	41.89
4M	8431.42	1086.42	7.76	542.85	15.53	267.56	31.51	197.81	42.62

Table 5.5: Speedup for Gaussian connection-generation - 10 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	18.06	1.76	10.24	0.92	19.59	0.46	39.24	0.34	52.38
262K	96.89	9.87	9.82	4.75	20.41	2.44	39.71	1.88	51.50
1M	703.50	86.65	8.12	42.19	16.68	21.70	32.42	16.75	42.01
2M	2075.96	270.58	7.67	133.27	15.58	67.44	30.78	51.19	40.55
4M	8751.71	1128.00	7.76	554.41	15.79	279.75	31.28	211.74	41.33

Table 5.6: Speedup for Gaussian connection-generation - 100 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	24.58	2.49	9.87	1.19	20.65	0.57	43.12	0.41	60.02
262K	171.59	11.58	14.82	6.07	28.27	2.90	59.17	2.08	82.57
1M	1900.48	98.57	19.28	50.40	37.71	24.72	76.88	18.19	104.50
2M	5608.14	308.02	18.21	155.01	36.18	75.79	74.00	55.58	100.89
4M	23642.49	1284.04	18.41	613.50	38.54	306.36	77.17	229.48	103.03

Table 5.7: Speedup for Gaussian connection-generation - 1k synapses / neuron

Enlarging the network sizes within a GPU_world, leads to growth in the execution times, which is expected. Looking at the execution times across densities, for the same network sizes the differences in execution time become smaller as the GPU_world grows. For smaller GPU_worlds the execution times are more substantial than for larger GPU_worlds. For larger GPU_worlds the workload is smaller than for smaller GPU_worlds, thus the difference in execution time must also be smaller.

5.3 Cell-id-dispersal

In figs. 5.3 and 5.4, the execution time is shown for the phase where the required neighbor-cell-id's are dispersed throughout the GPU_world. This phase does not directly benefit from increasing the GPU_world, because it is not executed on the GPUs. The small overall speedup, for both generators is due to the fact that a smaller amount of cells need to be communicated to an increasing GPU_world. Data sizes decrease while the count of packets per communication instance increase.

It stands out that the cell-id-dispersal for a Gaussian-distribution takes longer time than for a uniform-distribution. As was mentioned in section 2.3, GPUdirect is best when the packet sizes are below a certain threshold. When packet sizes to be send are above 16 Kbytes the latency starts to grow. In this part of the architecture the required neighboring cell-id's are communicated from the local to the foreign GPUs. In case of the uniform-distribution, relative small packets containing cell-id's, are send to many foreign GPUs. In case of the Gaussian-distribution relative large packets are send to fewer foreign GPUs. For the Gaussian-distribution it is more likely that the average packet size will reach the threshold sooner than the uniform-distribution. On the other hand, with a uniform-distribution the average packet size will reach the threshold much later, not causing as much latency overhead as for the Gaussian- distribution.

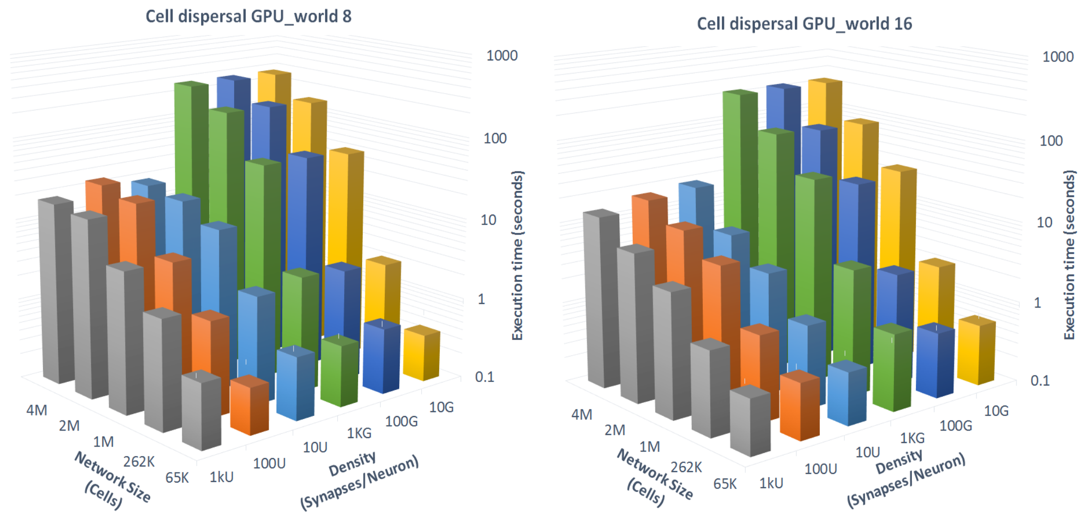

Figure 5.3: Execution times for uniform (U) and Gaussian (G) cell-id-dispersal phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right)

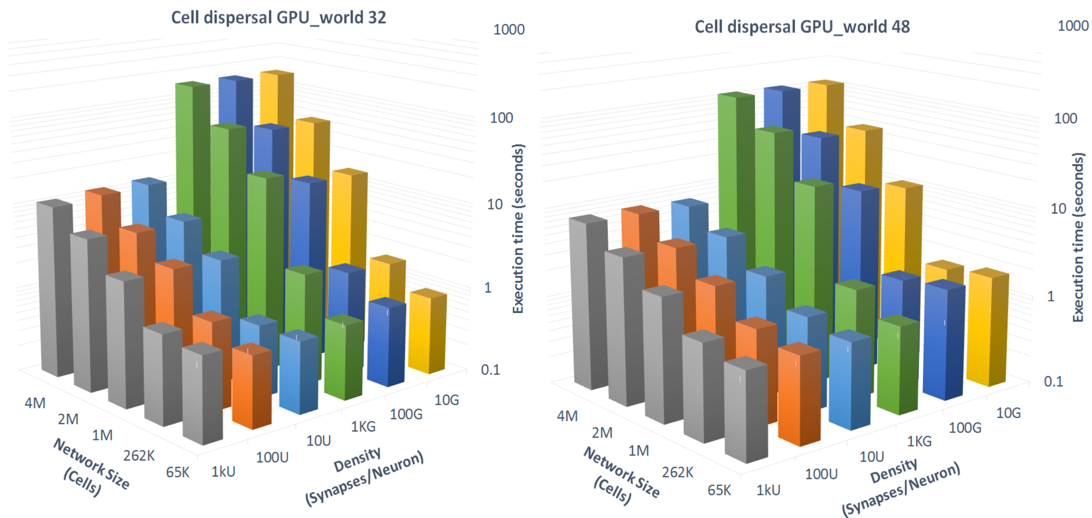

Figure 5.4: Execution times for uniform (U) and Gaussian (G) cell-id-dispersal phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right)

5.4 Cell-computation

The execution time for the cell-computation phase, shown in figs. 5.5 and 5.6, reports solely the execution time of the computation kernel during simulation.

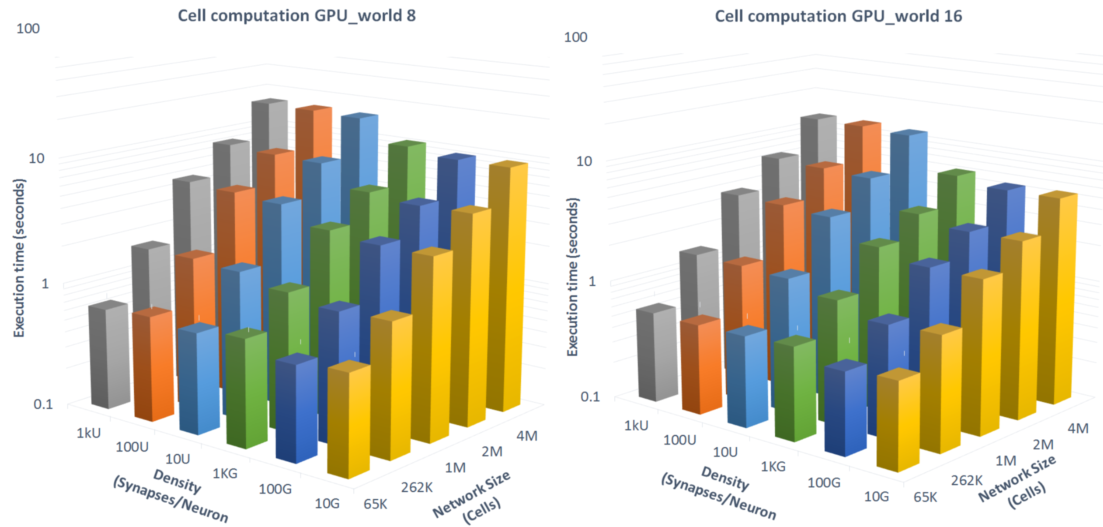

Figure 5.5: Execution times for uniform (U) and Gaussian (G) cell-computation phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right)

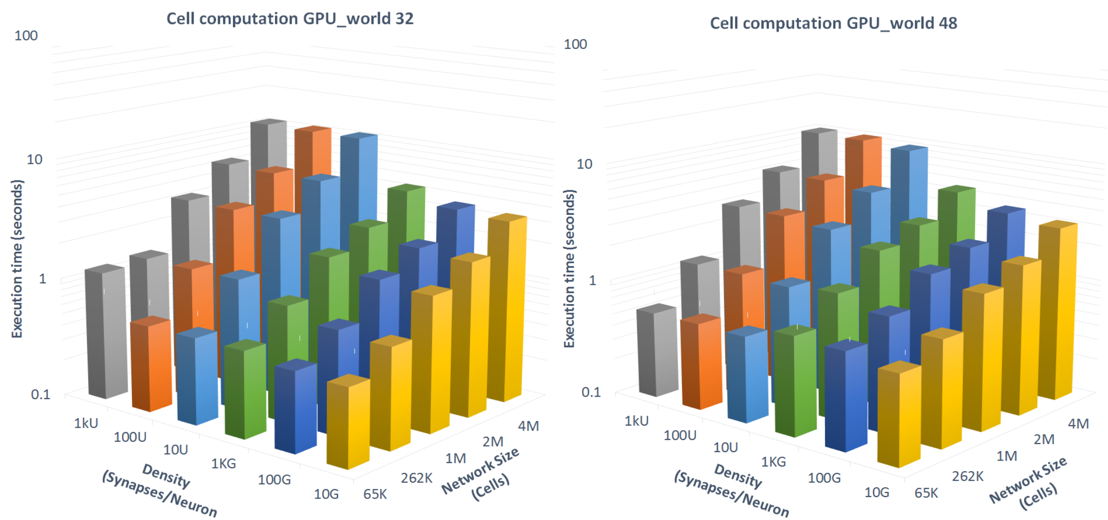

Figure 5.6: Execution times for uniform (U) and Gaussian (G) cell-computation phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right)

In general, it holds that the larger the GPU_world becomes the smaller the world_part, the fewer computations per GPU need to be preformed and the smaller the loop for collecting the dendrite voltage per GPU loop will become. Next to an increase in speed of simulation, increasing the GPU_world has the other benefit that the total

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.26	0.64	1.97	0.57	2.21	0.52	2.43	0.55	2.32
262K	4.12	1.54	2.68	1.32	3.12	0.53	7.79	1.10	3.73
1M	15.43	4.32	3.57	3.44	4.49	2.89	5.33	2.73	5.65
2M	28.83	7.73	3.73	6.00	4.81	5.02	5.74	4.68	6.16
4M	65.33	15.37	4.25	11.72	5.57	9.79	6.67	9.15	7.14

Table 5.8: Speedup for uniform cell-computation - 10 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.38	0.70	1.96	0.57	2.43	0.53	2.60	0.56	2.47
262K	4.47	1.64	2.73	1.39	3.22	1.25	3.59	1.16	3.84
1M	16.57	4.62	3.59	3.66	4.52	3.19	5.19	3.01	5.51
2M	30.96	7.93	3.90	6.29	4.92	5.44	5.69	5.14	6.03
4M	70.14	15.82	4.43	12.33	5.69	10.53	6.66	9.90	7.08

Table 5.9: Speedup for uniform cell-computation - 100 synapses / neuron

memory is increased. The limitations of the memory and implications for the network sizes are studied in section 5.8.

For a uniform-distributed network the computation speedup diminishes as the GPU_world grows, as can be observed in tables 5.8 to 5.10. For a Gaussian-distributed network a greater speedup is gained than for the uniform, as can be observed in tables 5.11 to 5.13. The reason for this difference between distributions and the diminishing returns for the uniform-distribution, was foreboded in the previous chapter. The spread of the cells per GPU in a Gaussian-distributed network is relatively better balanced, all GPUs have a similar workload, than in a uniform one. In case of the uniform-distribution it is very plausible that, for instance, one GPU needs to process a large and another a small workload. This on average could take a longer time to complete than in the case where both GPUs do an equally sized job, where resources are more efficiently utilized.

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.39	0.66	2.12	0.58	2.41	1.19	1.16	0.55	2.52
262K	5.16	1.64	3.15	1.43	3.61	1.26	4.10	1.16	4.43
1M	19.68	4.85	4.06	3.78	5.20	3.25	6.05	3.07	6.41
2M	36.76	8.36	4.40	6.57	5.59	5.57	6.60	5.24	7.02
4M	83.29	16.13	5.16	12.50	6.66	10.67	7.80	10.09	8.26

Table 5.10: Speedup for uniform cell-computation - 1k synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.23	0.62	1.99	0.52	2.39	0.44	2.80	0.57	2.18
262K	3.97	1.15	3.45	0.88	4.51	0.68	5.82	0.79	5.02
1M	15.09	2.81	5.37	1.87	8.05	1.33	11.33	1.42	10.63
2M	28.38	4.85	5.85	3.00	9.46	1.95	14.53	1.91	14.88
4M	63.12	9.19	6.87	5.43	11.62	3.36	18.76	3.13	20.20

Table 5.11: Speedup for Gaussian cell-computation - 10 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.44	0.57	2.54	0.48	3.01	0.46	3.12	0.67	2.16
262K	4.12	1.11	3.70	0.85	4.86	0.74	5.58	0.97	4.27
1M	15.17	2.86	5.31	1.91	7.94	1.46	10.39	1.67	9.06
2M	28.52	4.76	5.99	2.99	9.54	2.10	13.57	2.22	12.86
4M	63.45	9.19	6.90	5.40	11.76	3.57	17.80	3.53	17.96

Table 5.12: Speedup for Gaussian cell-computation - 100 synapses / neuron

	GPU_world								
	1	8		16		32		48	
Network Size	Run (s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	1.29	0.72	1.80	0.59	2.18	0.53	2.46	0.70	1.85
262K	4.05	1.28	3.15	1.08	3.75	0.93	4.34	1.21	3.35
1M	15.27	3.16	4.83	2.32	6.59	1.83	8.33	2.18	7.01
2M	28.71	5.22	5.50	3.52	8.15	2.60	11.05	2.87	9.99
4M	63.86	10.27	6.22	6.12	10.44	4.32	14.79	4.57	13.96

Table 5.13: Speedup for Gaussian cell-computation - 1k synapses / neuron

5.4.1 KNL comparison

The KNL version of the application [14] is evaluated using only cell- computations, thus this is the only phase to which the multi-GPU design can only be compared to with regards to performance. This section compares the results of the KNL setup to the GPU setup in terms of network scalability, execution times and speedup.

The KNL evaluation reports that when increasing the KNL_world to more than 8 nodes no speedup gains are observed, particularly for heavier workloads. Even more

so, simulating uniform-distributed networks of 1M cells results in erratic behavior. The reason given is that performance and scalability are hindered due to data messages being exchanged between cores, especially those belonging to different KNL nodes. The mesh interconnect as described in section 2.3.3 which connect some, but not all, cores to other cores, creating expensive extra memory reads and writes to get information across nodes, is most probably the underlying architectural design to blame.

Firstly, the multi-GPU design does not show to have difficulties scaling larger network sizes as for both distributions up to 4M cells, all densities can be simulated. The GPUdirect technology implemented through OpenMPI, which connects the GPUs with the lowest latency possible, shows good scalability across worlds and differing network sizes. For the GPU-setup also the communication part eventually becomes the bottleneck of the design. However, larger network sizes do not lead to erratic behavior as is the case for the KNL setup. The dendrite voltages keep being delivered to other nodes and it is just the execution time that increases with network size.

Secondly, when comparing the execution times of the KNL-setup to that of the multi-GPU all the simulation times for all distributions, sizes and densities stay below 11 seconds, even for a GPU_world of 8. For larger densities and network sizes, the KNL-setup reports executions times of 100-130 seconds for uniform and 50-110 seconds for Gaussian-distributed network simulations.

Thirdly, the speedup reported for Gaussian-distributed network simulations, is lower than the speedup observed for the GPU setup. In table 5.13 the speedup for the cell computation are given for the largest density. In relation to the execution on a single KNL node for Gaussian-distributed network simulations, the speedup reported for KNL is 2.1, 4 and 7.2 when scaling from 1 to 2, 4 and 8 KNL nodes respectively. For the GPU setup the speedup of 7.2 is obtained for a GPU_world of 16 and network size of 1M. The speedup obtained by the KNL-setup can be matched by the GPU-setup.

The KNL-setup has been compared to the GPU-setup and overall it delineates a picture how both setups behave. This is a comparison where the results serve as an absolute reference point, however the GPU is not a KNL. The GPU for instance has a lot of smaller cores while the KNL has fewer but larger cores. The multi-GPU setup shows that network sizes above the 2M for the uniform distribution can still be executed. The cell-computation phase scales good when the GPU_world increases. Enlarging the GPU_world does not show any erratic behavior. The GPU setup either matches or outperforms the KNL setup and it must be concluded that at least for larger network sizes the multi-GPU is very suited to scale the InfOli brain application. A more thorough comparison should be made where, for instance, also the consumed energy is considered to establish the efficiency of the KNL setup.

5.5 Dendrite communication

The dendrite-communication phase, which performs the update of the dendrite voltages after each simulation step, has become the bottleneck of the simulation. In figs. 5.7 and 5.8 can be observed that the largest execution times are reported in comparison to other phases.

Next to the communication of the dendrite voltages, also the execution time for

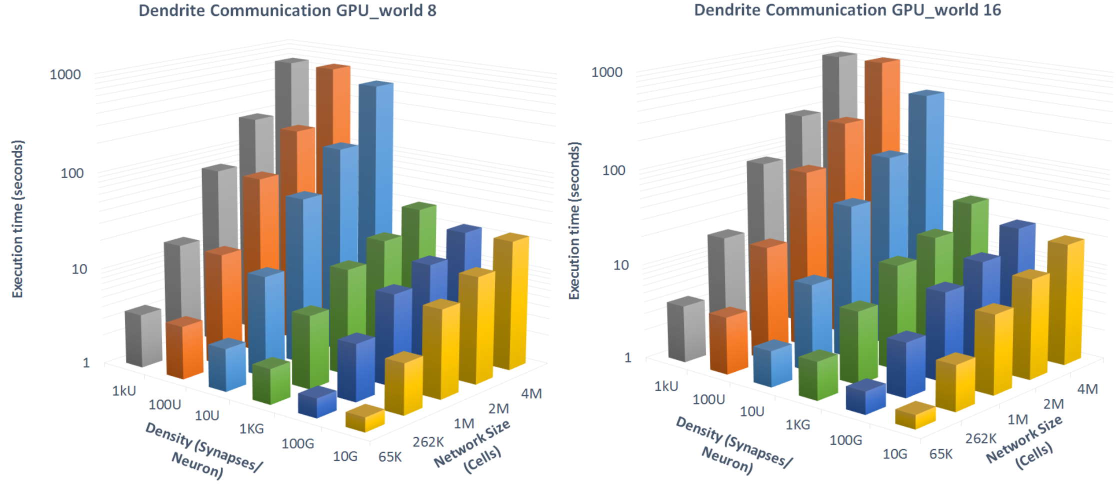

Figure 5.7: Execution times for uniform (U) and Gaussian (G) dendrite-communication phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right)

gathering and unpacking of the dendrite voltages are included, which are both parallelized using OpenMP as described in section 4.7. As the networks grow larger, less speedup is obtained by increasing the GPU_world. This process also does not benefit directly from a larger GPU_world, since it is executed on the host cores instead of the GPUs.

The results observed here are reciprocal to the results from the cell-dispersal process. The observed execution times for uniform are larger than for the Gaussian, which was the opposite when looking at the cell-dispersal process. The larger the network the larger the difference between Gaussian- and uniform-distributed networks. In this phase the foreign GPUs have to send back the gathered dendrite voltages according to the cell-id's received in the cell-id-dispersal phase. The packets sent are larger than in the cell-id-dispersal phase, because the data-type used, is `MPI_double` instead of `MPI_int`. For a uniform-distributed network many foreign GPUs send back the gathered voltages to local GPUs. Thus, a uniform network generates more traffic for larger packet sizes. For a Gaussian network fewer foreign GPUs transfer data to local GPUs.

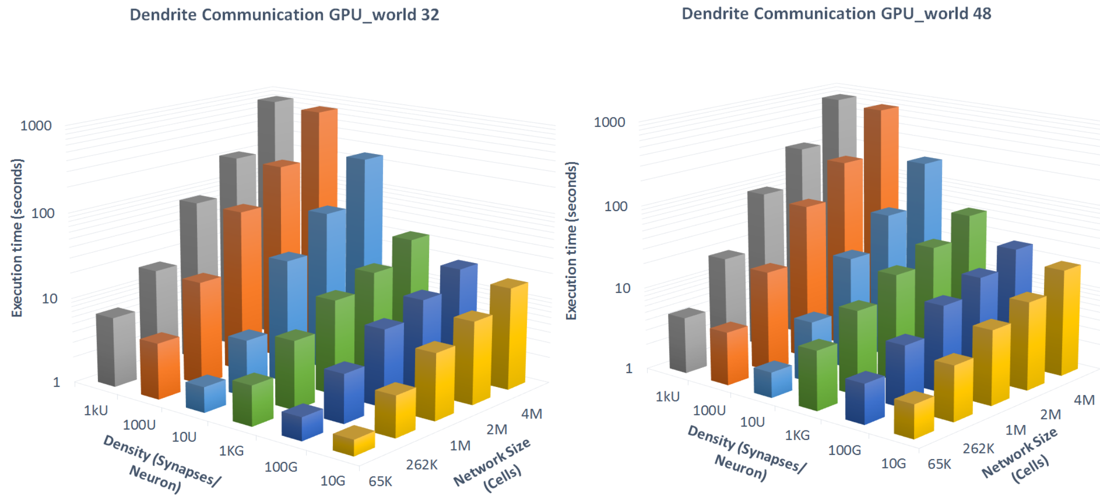

Figure 5.8: Execution times for uniform (U) and Gaussian (G) dendrite-communication phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right)

5.6 Total execution time

The execution phases mentioned in the previous paragraph are summed and visualized in figs. 5.9 and 5.10 and the speedup in relation to execution on a single GPU is displayed in tables in this section.

To clearly show the scalability of the design, again three dimensional graphs are used, that show the uniform and Gaussian-distributed simulations. The different shades of a certain color belong to a distinct GPU_world, the darker the color the larger the density within the GPU_world.

The communication is the dominant factor in the total execution times; either influenced by the cell dispersal phase for Gaussian or dendrite communication for uniform. In fig. 5.9 it can be seen that for the larger densities of the uniform distribution the execution time diminishes slowly for an increasing GPU_world. For the smaller densities a decrease in execution time is visible. In fig. 5.10 it can be seen that for an increasing GPU_world, the total execution time decreases from roughly 1,100 to 200 seconds for all densities. The execution times for a Gaussian-distributed network diminishes faster than the for the uniform-distributed network.

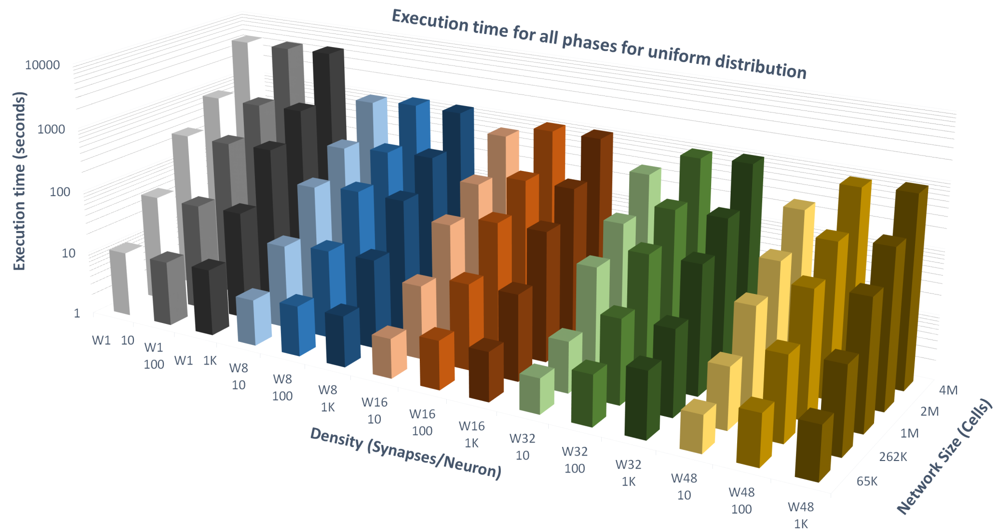

Figure 5.9: Total execution times for uniform-distribution for all sizes and densities. Groups with the same colors with different shades represent the GPU_worlds (grey = 1, blue = 8, red = 16, green = 32 and brown = 48)

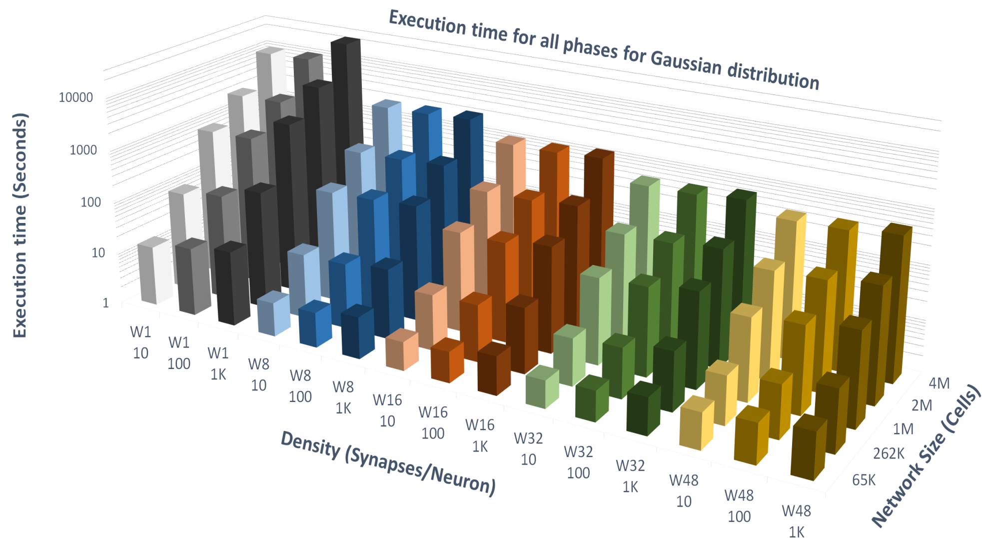

Figure 5.10: Total execution times for Gaussian-distribution for all sizes and densities. Groups with the same colors with different shades represent the GPU_worlds (grey = 1, blue = 8, red = 16, green = 32 and brown = 48)

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	10.60	5.33	1.99	4.15	2.56	3.52	3.01	3.71	2.86
262K	49.04	20.99	2.34	14.10	3.48	6.19	7.92	8.82	5.56
1M	306.23	106.96	2.86	69.67	4.40	43.23	7.08	34.41	8.90
2M	819.95	275.59	2.98	175.66	4.67	111.08	7.38	83.69	9.80
4M	4620.30	962.60	4.80	614.01	7.52	367.82	12.56	270.04	17.11

Table 5.14: Speedup for total execution time - 10 synapses/neuron - uniform-distribution

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	10.53	6.16	1.71	5.76	1.83	10.42	1.01	6.09	1.73
262K	48.51	24.68	1.97	21.69	2.24	20.09	2.41	19.78	2.45
1M	302.15	124.20	2.43	104.57	2.89	96.28	3.14	87.98	3.43
2M	807.89	315.20	2.56	272.90	2.96	249.04	3.24	228.39	3.54
4M	4543.48	1118.95	4.06	976.70	4.65	874.94	5.19	799.59	5.68

Table 5.15: Speedup for total execution time - 100 synapses/neuron - uniform-distribution

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	11.21	6.31	1.78	5.82	1.93	6.06	1.85	6.51	1.72
262K	51.01	25.02	2.04	21.86	2.33	20.87	2.44	21.56	2.37
1M	318.94	121.36	2.63	106.18	3.00	97.38	3.28	97.66	3.27
2M	850.91	344.27	2.47	276.43	3.08	255.95	3.32	268.27	3.17
4M	4770.87	1102.56	4.33	984.69	4.85	967.73	4.93	889.69	5.36

Table 5.16: Speedup for total execution time - 1K synapses/neuron - uniform-distribution

The overall speedup is calculated in comparison to the single-GPU execution. In tables 5.14 to 5.16 the speedup for the uniform- and in tables 5.17 to 5.19 the speedup for the Gaussian-distributed simulation is shown.

For the uniform-distributed simulation, the largest speedup observed is 17.11 for the simulation with 10 synapses per neuron, a GPU_world of 48 and a network size of 4M cells. For this distribution the speedup decreases when network density increases. Larger densities, mean increasing neighbors and thus increasing communication overhead. The speedup gained from increasing the GPU_world is attenuated by this overhead. This distribution benefits less than, for instance, a Gaussian-distribution from an increasing GPU_world. Still, significant speedup is obtained when the GPU_world increases.

The Gaussian-distribution, as opposed to the uniform, does show an increasing speedup when density increases. For this distribution the neighbors are more local, which means less communication overhead and more to gain from the increasing calculating capacities. The largest speedup is 50.91 which is registered for the simulation of 1k synapses per neuron, a GPU_world of 48 and a network size of 4M cells. The results displayed for GPU_world 48, are partly extrapolated based on the almost linear relation in speedup

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	13.60	4.34	3.13	3.44	3.96	3.23	4.21	4.56	2.98
262K	75.85	16.79	4.52	11.11	6.83	7.55	10.05	7.69	9.86
1M	692.84	132.46	5.23	76.79	9.02	41.35	16.76	33.40	20.74
2M	2028.36	422.49	4.80	226.56	8.95	121.92	16.64	98.41	20.61
4M	8494.54	1786.23	4.76	944.71	8.99	475.35	17.87	358.78	23.68

Table 5.17: Speedup for total execution time - 10 synapses/neuron - Gaussian-distribution

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	19.49	4.55	4.28	3.75	5.19	3.65	5.35	5.31	3.67
262K	101.01	17.22	5.87	11.64	8.68	8.46	11.94	9.22	10.96
1M	718.67	143.69	5.00	75.62	9.50	44.84	16.03	39.87	18.02
2M	2104.48	442.94	4.75	233.43	9.02	128.45	16.38	106.02	19.85
4M	8815.16	1873.20	4.71	964.29	9.14	491.89	17.92	376.37	23.42

Table 5.18: Speedup for total execution time - 100 synapses/neuron - Gaussian-distribution

in the previous results. This extrapolation holds only for the Gaussian-distribution and had to be done due to unavailability of 48 GPU nodes.

	GPU_world								
	1	8		16		32		48	
Network Size	Run(s)	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup	Run (s)	Speedup
65K	25.87	6.08	4.26	5.18	5.00	4.78	5.41	6.63	3.90
262K	175.64	21.20	8.29	16.54	10.62	12.14	14.47	13.37	13.14
1M	1915.74	160.13	11.96	98.53	19.44	58.90	32.52	52.82	36.27
2M	5636.84	490.12	11.50	269.06	20.95	155.21	36.32	134.29	41.97
4M	23706.35	2057.25	11.52	1075.62	22.04	577.87	41.02	465.69	50.91

Table 5.19: Speedup for total execution time - 1K synapses/neuron - Gaussian-distribution

5.7 Energy consumption

In figs. 5.11 and 5.12 the energy consumption is reported in kilo Joules for every node count. The larger the network, the larger the execution time and thus more energy is consumed. The energy is collected according to the commands mentioned in section 2.4.2. SLURM is not configured to distinguish between GPU or Xeon processor on the node, so these figures represent the total energy consumed by the whole node.

Smaller densities consume relative less energy than larger densities. The simulation of Gaussian-distributed networks consume more energy than the uniform-distributed networks. Since the Gaussian process has longer setup time and setup is done on the GPUs, using the GPUs are more expensive in energy, than using the Xeon cores. The Xeon cores are used for the communication and some related tasks, which does take the longest over-

all time, but apparently does not take up as much energy.

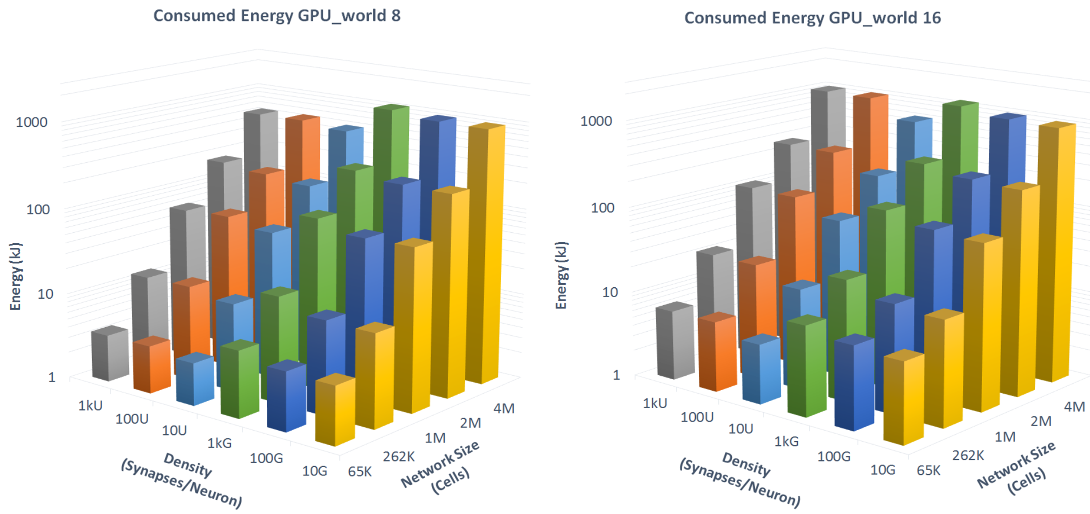

Figure 5.11: Energy consumption for uniform (U) and Gaussian (G) for all phases for all network sizes and densities, for GPU_worlds 8 (left) and 16 (right)

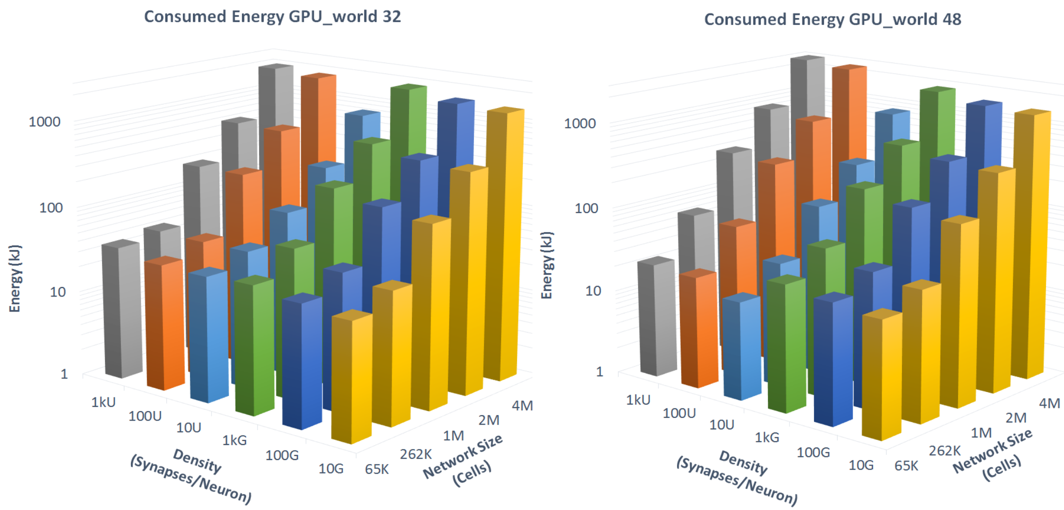

Figure 5.12: Energy consumption for uniform (U) and Gaussian (G) for all phases for all network sizes and densities, for GPU_worlds 32 (left) and 48 (right)

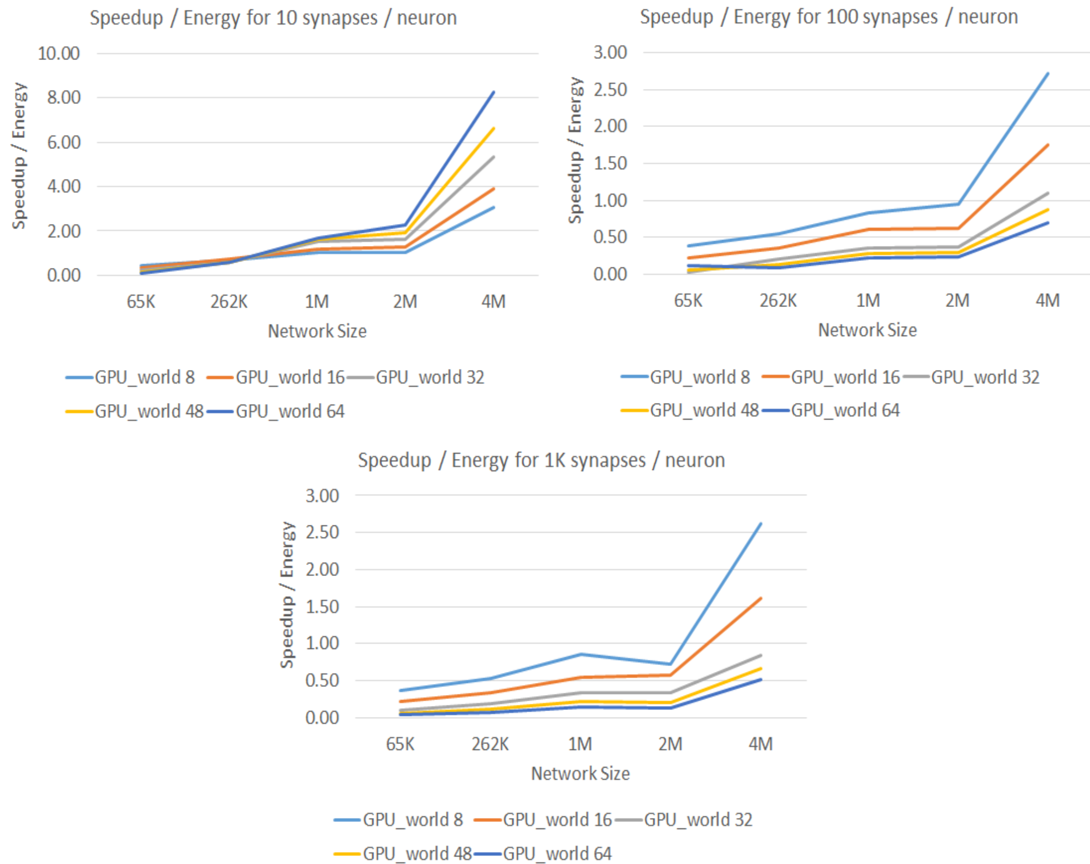

Figure 5.13: Speedup in relation to the energy used for uniform distributions

5.7.1 Energy efficiency

Now that the energy per GPU_world per network size per density is known, a cost-benefit analysis can be made. The energy consumed is also a rate for the financial cost, as clusters base their billing on this metric. A larger speedup is often obtained by increasing the GPU_world, but if the energy consumed rises higher than a rate of a lower GPU_world, it might be worth waiting longer for the simulation to finish. Also, the results for GPU_world 64 have been extrapolated and added to the graph to observe how the design would behave if the GPU_world were to increase further.

In fig. 5.13 the speedup gained across GPU_worlds has been put into relation to the relative energy increase compared to values obtained from GPU_world 1. GPU_world 1 serves as a base line for the speedup and the percentage for energy cost. A higher value is better, because this means that the speedup, which is the old execution time divided by the new, is relatively bigger than the percentage of energy increase, which is the old energy divided by the new.

From the graph it can be observed that for smaller densities, it pays off to increase the GPU_world. The energy used is relatively low in relation to the speedup gained.

For an increasing network density 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. For a uniform-distribution holds that when the density increases, the number of neighbors on foreign GPUs increase, the communication overhead increases, which attenuates the speedup gained from the increasing GPU_world. If there is time to kill, a smaller GPU_world for larger densities is a more energy efficient solution.

When the network size becomes larger then 2M cells, the GPUs become relatively more energy efficient. The GPU_world of 8 is most efficient for an input of 4M cells at larger densities. The larger contributor to the speedup gained from an increasing GPU_world for a uniform- distributed simulation is the matrix generation process. Thus, this increase in energy efficiency is caused by a more efficient execution by the GPUs. An explanation is that this has to do with the fact that the computing units of the GPUs become fully occupied. This is confirmed in [37] where utilization reaches 100% when network size reaches 262,144 cells, which is roughly the same for a network size for a single GPU when 2M cells are simulated for a GPU_world of 8. The GPUs use their resources to the fullest and a higher speedup per energy, is obtained. This means also that for a network size of 8M, the GPU_world of 16 would become most energy efficient. This is supported by the graphs, for it shows a rise in efficiency for all GPU_worlds when network size increases.

For Gaussian-distributed simulations the speedup versus energy relation is shown in fig. 5.14. A different trend than for the uniform-distribution is observed. From the graphs it becomes apparent that for smaller network sizes, smaller GPU_worlds are more energy efficient with regards to speedup, this holds for all densities. When network sizes increase, the GPU_world of 32 becomes the most energy efficient, followed by 48, 64, 16 and then 8. The speedup obtained in a GPU_world of 32 is the highest when compared to the energy used.

The magnitude of speedup/energy is much larger in comparison to the uniform-distribution, especially for 1K synapses/neuron. The Gaussian-distributed simulations can gain a higher overall speedup than the uniform, due to the fact that the dendrite-communication phase takes less time and the speedup of the connection-generation phase is more significant.

5.8 Limitations

5.8.1 Memory

From the previous sections it can be concluded that the design scales well with increasing network sizes and densities. When the data sets are analyzed, it can be observed that all measured metrics follow a predictable growth pattern. The limiting factor which determines to which magnitude cell networks can be simulated, has become the available memory. In this section an estimation is given on the limitations of the design, when extrapolating the obtained data, such to predict its behavior when dimensions are maximized. The two major factors which determine the memory usage, are the size of the network and the density. The growth pattern of the measured results can roughly be mimicked. The next value for the metric is obtained by multiplying the relation between

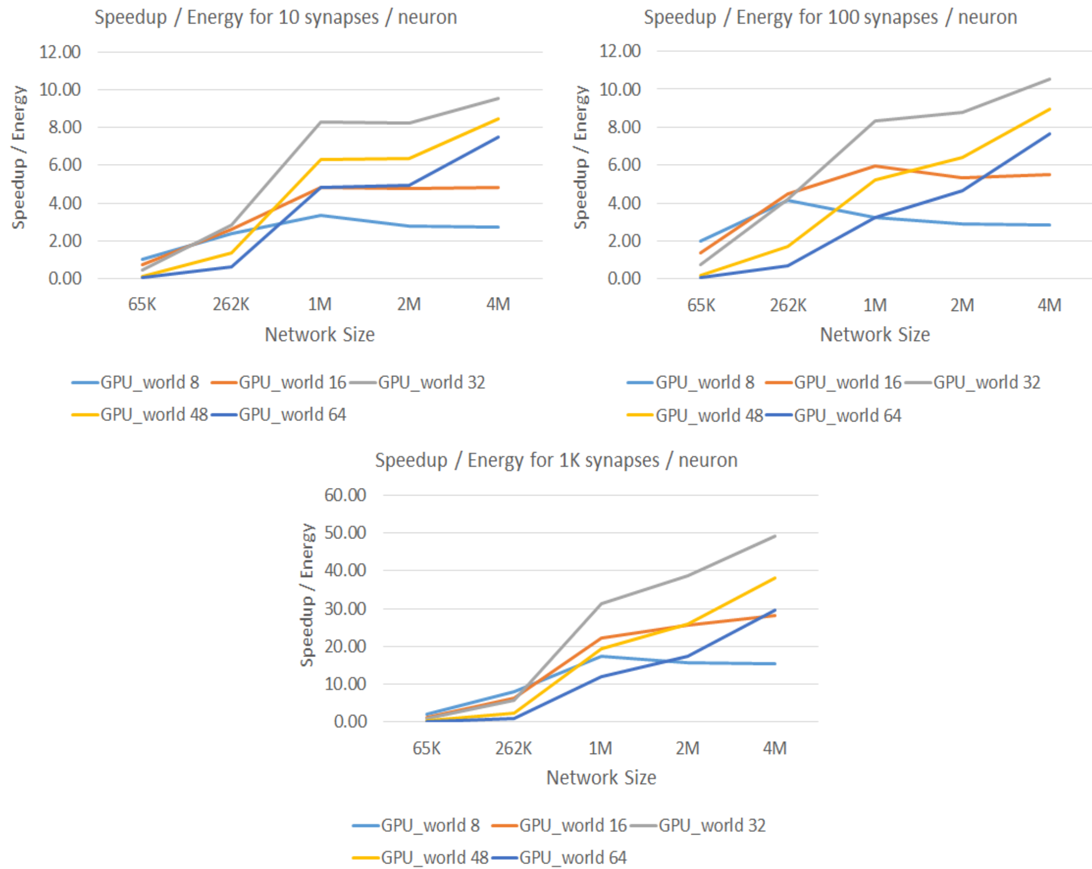

Figure 5.14: Speedup in relation to the energy used for Gaussian distributions

the previous and the current value, multiplied by the current. The results of the extrapolation roughly equalled the observations obtained from real experiments and thus can serve for as an indication to lay bare the limitations of the design. Extrapolating along these dimensions, the look-up table 5.20 can be obtained, showing the limitations of the cell network size in relation to the density of what that GPU_world could simulate.

It shows the theoretical limitations of the design, when scaled over 16, 32, 48 and 64 GPUs. It should be possible to simulate a network of 4096M cells at a density of 10 synapse per neuron when 64 GPU are used. Also observable is that the maximum number of synapses/neuron attainable per network size is a density of 100k for a network of 262k cell when using either a GPU_world of 32, 48 or 64.

As an example all the metrics for varying network sizes for a GPU_world of 32 and a density of 1000, are partly empirical obtained and partly extrapolated. From 10M cells the data is extrapolated until memory usage reaches 100%. The results of this extrapolation can be observed in fig. 5.15. In a GPU_world of 32 a network of 16M cells with a density of 1000 can be simulated before memory runs out (see memory). This then would take roughly 2.5 hours for the simulation to complete. The dendrite-

NS\Density	10	100	1K	10K	100K
65K					W16
262K					W32/W48/W64
1M				W16/W32	
2M				W48	
4M				W64	
8M					
16M			W16		
32M			W32		
64M			W48/W64		
128M					
256M		W16			
512M	W16	W32			
1024M	W32	W48			
2048M	W48	W64			
4096M	W64				

Table 5.20: Theoretically possible network sizes in relation to density for GPU_world 16-64

communication would still by far be the largest contributor to the execution time with 1.68 hours.

Adding more GPUs would not provide greater performance benefits, as was seen in the previous sections, however it would supply more memory. Thus even larger cell-networks become feasible when the GPU_world increases. If all the 64 GPU nodes on Cartesius were to be utilized (which is practically not possible because 6 nodes are under revision and scheduling that many node will be hard between the other jobs from other users), in theory a twice as large network could run in roughly the same time. Thus, simulating 32M cells with a density of 1000 synapses per neuron could be possible within reasonable time.

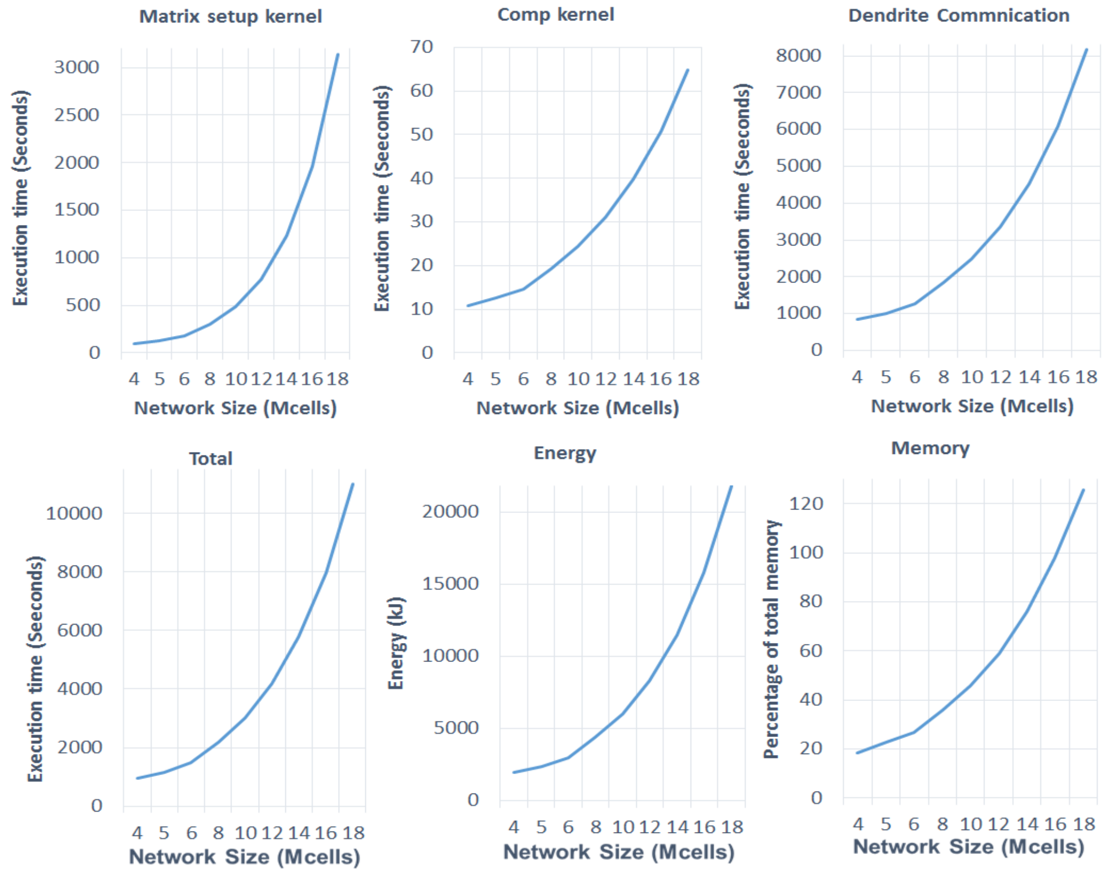

Figure 5.15: Large network behavior for GPU_world 32 and density of 1000 synapses/neuron

5.8.2 Roofline model analysis

A roofline model is used to provide performance indications for applications, executed on, for example, architectures such as GPUs, by visualizing hardware limitations. Various theoretical performance ceilings could be defined based on the properties of the application [39]. The model shows the operational intensity which is the ratio of work to the memory traffic, in relation to the peak performance and the peak bandwidth. This model can be used to study the design and potentially help in improving its performance.

The operational-intensity (OI) is defined as the ratio of floating point operations (flops) to memory operations. The number of double precision flops have been derived with the NVidia NVPROF profiling application, the profiler has a FLOP counter metric output (`flop_count_dp` for double precision and `flop_count_sp` for single precision). The memory operations have been derived by summing the write and read accesses from and to the DRAM of the GPU for which the profiler also has counters(`dram_read_transactions` and `dram_write_transactions`). The DRAM, of which the K40M has the type GDDR5, is the psychical device memory residing on the GPU

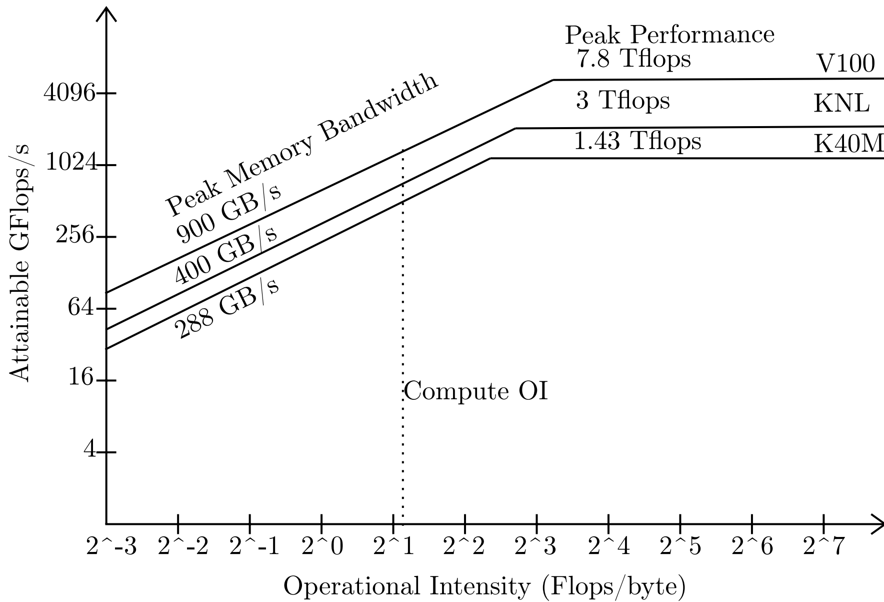

Figure 5.16: Roofline model for K40M, KNL, V100 for the compute-kernel

which is accessed on level 2 cache misses. For the compute-kernel, the flops and DRAM accesses scale proportionally when the network size and the GPU_world varies.

The OI is computed as:

$$OI = \frac{\text{Flops}}{(\text{DRAM_reads}+\text{DRAM_write})*(\text{size of transaction} = 32\text{ Bytes})}$$

In fig. 5.16 three different rooflines can be observed for the K40M, KNL and V100 compute-units. To make a comparison between architectures also the KNL has been added. The figure shows that OI for the compute-kernel is 1.63 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. The attainable performance is 211 GB/s. For the KNL and the attainable performance is 245 and for the V100 this is 661 GB/s. When compared to the ridge point of the roofline, which for the K40M GPU is at 1.43T / 288GB/s = 4.96, the OIc is at 33% of its peak performance.

Next to a roofline model of the compute kernel, a model of the uniform and Gaussian generator has been created to verify its behaviour. These kernels only handle single precision floating points. The flop_counts do not scale down when the GPU_world is increased, as opposed to the compute-kernel. This is because the dimension of these kernels is always equal to the network size. The dimension of the kernels are not scaled in relation to the GPU_world. Only the frequency of kernel execution, scales when GPU_world increases, but this has no effect on the OI.

Also, the number of flops do not scale with the same ratio as the DRAM memory movement when network sizes vary. This can explained from the fact that the probability

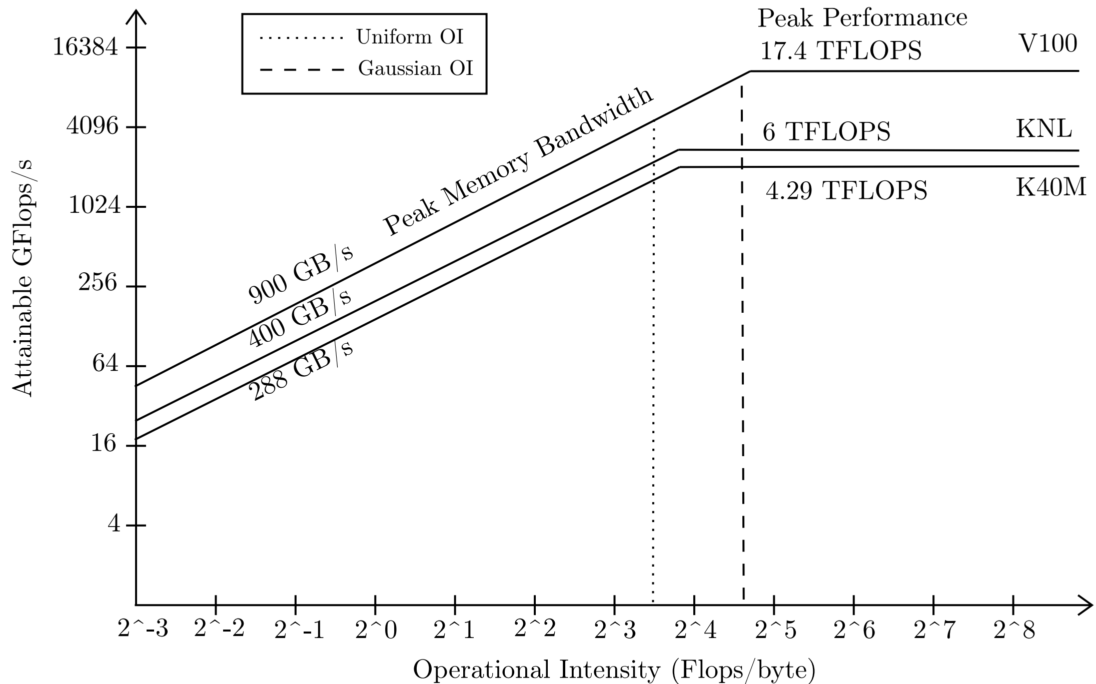

Figure 5.17: Roofline model for K40M, KNL, V100 for the generator-kernels

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 insure a steady density. The main contributor to floating point operations of the generator-kernels which determines the OI for the generator kernels is the cuRAND random-generator, which is executed for every cell. As network size increases the probability for neighbors decreases and less DRAM memory needs to be written, resulting in an increases in OI.

For a network-size of 256k cells the uniform OI is 10.11 and rises due to the fact that the flops count increases more rapidly then the DRAM memory movement. For network size of 512k, 1M, 2M the uniform OI is 32, 109 and 374. The same holds for the Gaussian generator, the gaussian OI for 256k cells is 28, then 37 for 512k cells and increases even more steep then the uniform. The results are shown in fig. 5.17. It should be no surprise that the Gaussian generator-kernel has a higher and faster increasing OI then the uniform. This is because for this kernel more floating point operations take place on roughly the same DRAM memory movement.

The results are displayed in fig. 5.17, from which can be concluded that the generators are compute-bound, meaning that the performance is limited by the speed of the GPUs.

Only the compute-kernel proves to be memory-bound, for increasing network sizes the other kernels becomes computation-bound very rapidly. For memory-bound applications, improving the occupancy of the GPU could help better hide memory latency and therefore improve performance. For compute-bound kernels this could have a counterproductive effect, introducing more overhead and possible degradation of performance. By increasing the block dimension a higher occupancy could be achieved. A few random tests have been

Block Size	Ratio	Occupancy	OIcompute
32	4x8	0.18	1.63
64	8x8	0.34	1.57
96	8x12	0.36	1.58
128	8x16	0.36	1.51
192	12x16	0.36	1.49
256	16x16	0.35	1.46

Table 5.21: Results increased block sizes K40M

done to check if this would have a positive effect, of which the results are displayed in table 5.21. A network size of 2M cells with a density of 100 synapses/neuron has been used. The overall occupancy of the kernel does improve, but it does not have a positive effect on the compute OI which decreases with every increase in block size.

6 Conclusions

This chapter concludes the research for the multi-GPU simulator. Chapter 2 has discussed the InfOli model which is used as a benchmark and modern GPU interconnect, used to facilitate the multi-GPU implementation. The extended literature survey of multi-node neural net simulators and their technology has been discussed in chapter 3. The proposed design which lays bare the main phases of SNN simulation on a cluster, is discussed in chapter 4. The main phases are setup of the network, which consists of connection-generation and cell-id-dispersal across the GPU_world, and the simulation phase, which consists of the computation of the soma, axon and dendrite voltages and communication of the latter voltages across the GPU_world. The impact of these phases is reported in chapter 5.

Because the design shows good scalability across nodes for varying network sizes, the performance of the design is limited by the available GPUs in the cluster. The application has proven to execute on as many GPUs as were available and no erratic behavior has been noticed. All processes related to running on the GPUs show speedup when network size is increased. When studying the total execution times, it is observed that for larger densities, for the uniform distribution a minimal speedup is gained. The dendrite communication process becomes the major bottleneck, as latency increases when the sent packet sizes increase. This communication overhead does not take the overhand in the overall execution while scaling network sizes is tractable. This scalable design gives a good prospect for neuroscientists, proving that large network size simulations is possible, using a multi-GPU setup.

Uniform distributions are more scattered across nodes indicating larger communication overhead, while the Gaussian is more local and is less bothered by communication overhead. Both distributions benchmark different features when it comes to test the application for a cluster. The Gaussian laid bare the fact that setup time is the largest contributor to total execution time. For uniform distributions the dendrite communication time is the largest contributor to total execution time.

The different phases of the multi-GPU simulator were programmed as modules. In principle these modules are generic modules, in the sense that they could be used to indicate how to implement a multi-node version for other similar architectures as well. It was depicted that the atomic functions for instance, indicate for GPU and OpenMP how to implement these parts of the application. The libraries used are platform specific but the modules functionality is generic across platforms. The modular programming approach simplifies the coding of the GPUs as well. Modules can be tested separately, using controlling variables to switch them on or off which help to identify problems such as segfaults more easily.

The estimated limitations can be helpful for future related work. If simulating larger networks is a goal, then table 5.20 is a visual aid showing what sizes are possible. For

future work related to the inferior olivary experiment, complexity of the neural cells, could also be increased. Instead of simulating 3 compartments, more compartments can be simulated at the cost of smaller cell size networks. Increasing the complexity of the cells, means increasing computational demands which could be supported by the GPUs.

Furthermore, this thesis does not limit itself to brain simulations only, it also indicates the sheer force the GPUs have in combination with a sound interconnect technology like GPUdirect. These technologies can also help other project currently being executed at Erasmus. For instance, the complex whisker tracking algorithm, which tracks individual whiskers of mice or rats, could be aided by such computational power. In fact, many projects where mathematical models are to be calculated, will gain from the GPU power benchmarked in this thesis and could use this document as a motivation that scaling computational power is very much a possibility.

The GPUs are relatively easy to program. The CUDA API is well supported and solutions to many problems are reported on fora. The functionality of the MPI collective functions is challenging at times. For instance, the difference between `MPI_Allgather(v)`, `MPI_Alltoall(v)` or `MPI_Sendrecv` all have similar but distinct functionality, only required for specific tasks.

When designing the multi-GPU application, a demanding part was to keep track where data resides. Inherently, data is local only to a single GPU, when data need to be interchanged or needs to be accessed by all GPUs the complexity of the implementation can increase significantly. Especially, the implementation of the cell-dispersal phase proved a complex undertaking. A solution for the problem that all GPUs need to communicate only parts of collected data to distinct GPUs, took more than a few iterations to solve. The designer has to always bare in mind that the application is executed on all GPUs, but is written for a single GPU. This next to the fact that a GPU is already a parallel device, on which the threads communicate implicitly through memory. A GPU-cluster is a large massive parallel unit of parallel units.

6.1 Contributions

This thesis has contributed:

1. an analysis of the current SNN model and single-GPU implementation
2. an extensive literature survey of multi-node neural net simulators, their architectures and optimization techniques
3. designed, implemented and deployed a multi-GPU simulator on the Cartesius cluster and made preparation to implement the simulator on the AWS cluster
4. an evaluation of the performance, scalability and energy-consumption of the implemented multi-GPU design
5. describing the bottlenecks according to the evaluation results and outlining the theoretical limitations of the design
6. proposing future direction for multi-GPU SNN simulation based on the findings

6.2 Future work

Potential next steps for this project could be:

- Increasing the number of compartments per cell such that more complex cell behavior could be simulated. The GPU network can then be used to aid the acceleration of parameter space exploration.
- The Euler solver used to calculate the differential equations is of the most basic sort, a first order Runga-Kutta method. Experimenting with higher order solvers, which produce more accurate simulation results, come at the price of decreasing performance, which can be overcome in with a multi-GPU setup.
- Implementing the application on the AWS cluster, is a step forward for performance but also for accessibility. The state-of-the-art hardware, could produce even better results when simulating even larger networks and the cloud interface could make it accessible to a greater public. The basis is the burstable cloud formation using SLURM, which dynamically enables the elasticity to enlarge the number of compute nodes necessary, which has been discussed.
- Comparing the multi-GPU implementation against the most promising candidate found from the survey - CoreNEURON, could be interesting. Because CoreNEURON is not making use of GPUdirect to scale across nodes, this simulator does not provide the most efficient implementation. However when contacted, developers stated such an optimization *to be on the agenda*, thus in the future this may prove lucrative. It would bring to light the similarities and differences between implementations. Since CoreNEURON is used by distinguished institutes as JSC and by EPFL, a supplementary research could serve as an indication for future research.
- The development of data retrieval for the cell connections from an external file can be made more efficient with the already proposed solution of bit- fielding mentioned in section 4.4.2. Another idea to speedup this process, is to copy the matrix directly to GPU memory and use a kernel to retrieve the cells. This is an highly experimental idea, no API's or predefined ways to do so exist. Getting a proof of concept can be challenging, but the results could be beneficial.

Bibliography

[1] "BrainFrame: Brain Simulation Library on HPC Platforms and Brain Experimentation Support." [Online]. Available: http://www.erasmusbrainproject.com/index.php/en/themes/brainframe

[2] H. Nguyen, Z. Al-Ars, G. Smaragdos, and C. Strydis, "Accelerating complex brain-model simulations on GPU platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, 2015, pp. 974–979. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=7092530&searchWithin=simulation&ranges=2013_2015_p_Publication_Year&pageNumber=2&queryText=gpu

[3] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. De Zeeuw, and C. Strydis, "BrainFrame: A node-level heterogeneous accelerator platform for neuron simulations," *Journal of Neural Engineering*, vol. 14, no. 6, p. 66008, 2017. [Online]. Available: http://stacks.iop.org/1741-2552/14/i=6/a=066008

[4] D. Kirk, "Thread Optimization Slides adapted from the course at UIUC Single-Program Multiple-Data (SPMD)." [Online]. Available: http://users.umiacs.umd.edu/~ramani/cmsc828e_gpusci/lecture9.pdf

[5] NVIDIA, "{CUDA C} Programming Guide," no. October, 2014. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[6] ——, "GPUdirect," https://developer.nvidia.com/gpudirect. [Online]. Available: https://developer.nvidia.com/gpudirect

[7] "Cartesius The Dutch supercomputer - Multinode compute cluster," https://userinfo.surfsara.nl/systems/cartesius. [Online]. Available: https://userinfo.surfsara.nl/systems/cartesius

[8] "SLURM - Cluster Management System," https://slurm.schedmd.com/. [Online]. Available: https://slurm.schedmd.com/

[9] G. Murase, "Deploying a Burstable and Event-driven HPC Cluster on AWS Using SLURM," 2018. [Online]. Available: https://aws.amazon.com/blogs/compute/deploying-a-burstable-and-event-driven-hpc-cluster-on-aws-using-slurm-part-1/

[10] Stackoverflow, "MPI collective functions," https://stackoverflow.com/questions/15049190/difference-between-mpi-allgather-and-mpi-alltoall-functions. [Online]. Available: https://stackoverflow.com/questions/15049190/difference-between-mpi-allgather-and-mpi-alltoall-functions

[11] E. M. Izhikevich, "Izhikevich2004-Which Model to Use for Cortical Spiking Neurons," *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[12] C. De Zeeuw, F. Hoebeek, L. Bosman, M. Schonewille, L. Witter, and S. Koekkoek, "Spatiotemporal patterns in the cerebellum. Nat Rev Neurosci," *Nature reviews. Neuroscience*, vol. 12, pp. 327–344, 2011.

[13] J. R. de Gruijl, P. Bazzigaluppi, M. T. de Jeu, and C. I. de Zeeuw, "Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting," *PLoS Computational Biology*, vol. 8, no. 12, pp. 1–10, 2012. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1002814

[14] G. Chatzikonstantis, H. Sidiropoulos, C. Strydis, M. Negrello, G. Smaragdos, C. I. De Zeeuw, and D. J. Soudris, "Multinode implementation of an extended Hodgkin-Huxley simulator," *Neurocomputing*, 2018. [Online]. Available: https://doi.org/10.1016/j.neucom.2018.10.062

[15] J. N. C. Kraus, "An Introduction to CUDA-Aware MPI," https://devblogs.nvidia.com/introduction-cuda-aware-mpi/, 2013. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[16] "Comparison Between NVIDIA GeForce and Tesla GPUs," https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/. [Online]. Available: https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/

[17] "Research Groups: APT - Advanced Processor Technologies (School of Computer Science - The University of Manchester)," http://apt.cs.manchester.ac.uk/projects/SpiNNaker/. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/SpiNNaker/

[18] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: A code generation framework for accelerated brain simulations," *Scientific Reports*, vol. 6, p. 18854, 1 2016. [Online]. Available: http://dx.doi.org/10.1038/srep18854http://10.0.4.14/srep18854https://www.nature.com/articles/srep18854#supplementary-information

[19] J. S. Sherfey, A. E. Soplata, S. Ardid, E. A. Roberts, D. A. Stanley, B. R. Pittman-Polletta, and N. J. Kopell, "DynaSim: A MATLAB Toolbox for Neural Modeling and Simulation," 2018. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2018.00010/full

[20] J. W. Eaton, "GNU Octave - Scientific Programming Language," https://www.gnu.org/software/octave. [Online]. Available: https://www.gnu.org/software/octave/

[21] J. S. Sherfey, A. E. Soplata, S. Ardid, E. A. Roberts, D. A. Stanley, B. R. Pittman-Polletta, and N. J. Kopell, "Welcome to DynaSim! - github," https://github.com/DynaSim/DynaSim/wiki. [Online]. Available: https://github.com/DynaSim/DynaSim/wiki

[22] "Running Parallel Batch Jobs," https://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/#gpu. [Online]. Available: https://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/#gpu

[23] J. Krichmar, "CARLsim: a GPU-accelerated SNN Simulator," http://www.socsci.uci.edu/~jkrichma/CARLsim. [Online]. Available: http://www.socsci.uci.edu/~jkrichma/CARLsim/index.html

[24] "CARLsim 4.0.0 - userguide," http://uci-carl.github.io/CARLsim4/index.html.

[25] K. D. Carlson, M. Beyeler, N. Dutt, and J. L. Krichmar, "GPGPU accelerated simulation and parameter tuning for neuromorphic applications," *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 570–577, 2014.

[26] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, 2007. [Online]. Available: http://www.nest-simulator.org/

[27] T. Ippen, J. M. Eppler, H. E. Plesser, and M. Diesmann, "Constructing Neuronal Network Models in Massively Parallel Environments," https://www.frontiersin.org/articles/10.3389/fninf.2017.00030/full, 2017. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2017.00030/full

[28] M. L. Hines and N. T. Carnevale, "The NEURON Simulation Environment, Neural Computation," pp. 1–26, 1997. [Online]. Available: http://citeseer.uark.edu:8080/citeseerx/showciting;jsessionid=CA934E44E6AFD9849B676C4BEB820241?cid=84470

[29] S. Theil, "Why the Human Brain Project Went WrongâĂŤand How to Fix It," 2015. [Online]. Available: https://www.scientificamerican.com/article/why-the-human-brain-project-went-wrong-and-how-to-fix-it/

[30] K. Minkovich, C. M. Thibeault, M. J. O'Brien, A. Nogin, Y. Cho, and N. Srinivasa, "HRLSim: A high performance spiking neural network simulator for GPGPU clusters," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 2, pp. 316–331, 2014. [Online]. Available: ttps://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6579754&tag=1

[31] L. E. Givon and A. A. Lazar, "Neurokernel: An open source platform for emulating the fruit fly brain," *PLoS ONE*, vol. 11, no. 1, pp. 1–25, 2016. [Online]. Available: http://neurokernel.github.io/index.html

[32] R. V. Hoang, D. Tanna, L. C. Jayet Bray, S. M. Dascalu, and F. C. Harris, "A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling," *Frontiers in Neuroinformatics*, vol. 7, no. October, p. 19, 2013. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2013.00019/abstract

[33] W. Chen and E. De Schutter, "Parallel STEPS: Large Scale Stochastic Spatial Reaction-Diffusion Simulation with High Performance Computers," *Frontiers in*

Neuroinformatics, vol. 11, no. February, pp. 1–15, 2017. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fninf.2017.00013/full

[34] I. Hepburn, W. Chen, and E. De Schutter, "Accurate reaction-diffusion operator splitting on tetrahedral meshes for parallel stochastic molecular simulations," *Journal of Chemical Physics*, vol. 145, no. 5, 2016. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fncom.2013.00129/full

[35] I. Hepbur, R. Cannon, and E. De Schutter, "Efficient calculation of the quasi-static electrical potential on a tetrahedral mesh and its implementation in STEPS," *Frontiers in Computational Neuroscience*, vol. 7, no. October, pp. 1–11, 2013. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fncom.2013.00129/abstract

[36] R. C. Cannon, C. O'Donnell, and M. F. Nolan, "Stochastic ion channel gating in dendritic neurons: Morphology dependence and probabilistic synaptic activation of dendritic spikes," *PLoS Computational Biology*, vol. 6, no. 8, 2010. [Online]. Available: http://journals.plos.org/ploscompbiol/article/file?id=10.1371/journal.pcbi.1000886&type=printablehttp://www.psics.org/

[37] H. Du Nguyen, "Gpu-Based Simulation of Brain Neuron Models," 2013. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3A59d5a00d-6bb7-490d-9887-32e9530eb33e

[38] "Gaussian function," https://en.wikipedia.org/wiki/Gaussian_function. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_function

[39] "Measuring Roofline Quantities on NVIDIA GPUs," http://performanceportability.org/. [Online]. Available: http://performanceportability.org/perfport/measurements/gpu/