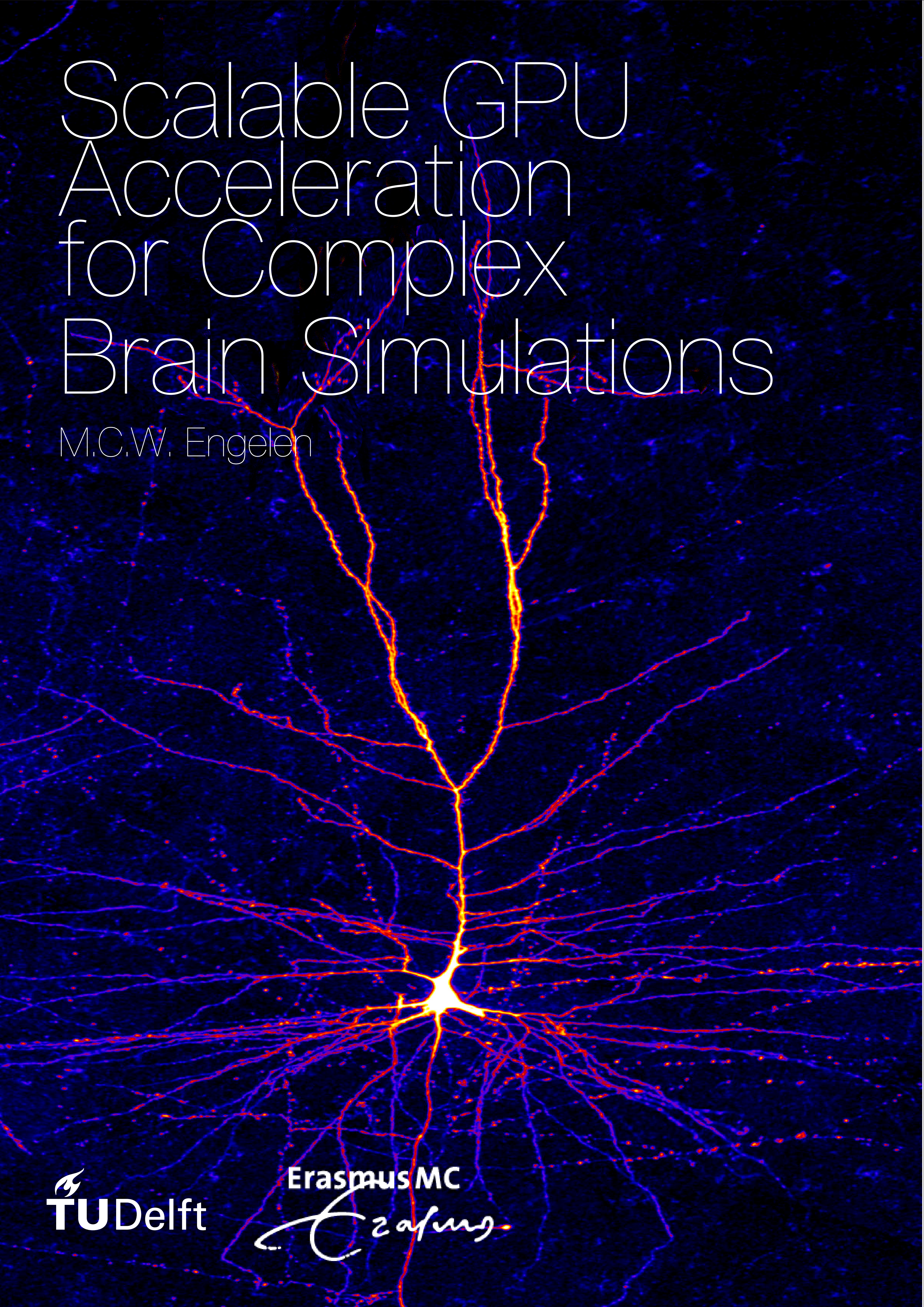


Scalable GPU Acceleration for Complex Brain Simulations

M.C.W. Engelen

Scalable GPU Acceleration for Complex Brain Simulations

by

M.C.W. Engelen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on the 24 of February 2021.

Student number:	4470532		
Laboratory:	Computer Engineering		
Codenumber:	Q&CE-CE-MS-2021-01		
Project duration:	April, 2020 – February, 2021		
Thesis committee:	**Chairperson**	Dr. Ir. Zaid Al-Ars,	TU Delft , CE, QCE
	Member	Dr. Matthias Möller,	TU Delft , Num. Analysis, AM
	Advisor	Dr. Ir. Christos Stydis,	Erasmus MC, Neurocomputing
	Member	Dr. Mario Negrello	Erasmus MC, Neurocomputing

Abstract

Complex mathematical models are used in computational neuroscience to stimulate brain activity to understand the biological processes involved. The simulation of such models is computationally costly, and thus high-performance computing systems are selected as a potential solution to increase performance. This thesis aims to implement a new versatile, multi-GPU eHH simulator (mgpuHH), explore its performance and make general observations on performance scalability over different modeling and cluster-configuration properties. This work offers a multi-node multi-GPU solution that offers excellent scalability performance due to how the simulator is constructed, with the use of OpenMPI and CUDA. The simulator is configured with JSON configuration files, containing the neural descriptions and simulator-specific settings. Consequently, enabling a user-friendly environment, for the neuroscientists, without the need of recompiling or understanding the source code. The gap junction calculations are identified as the critical function bottlenecking performance of the simulator. Therefore, an algorithm tailored to utilize GPU performance is implemented to decrease wallclock time for these specific calculations. For inter-node communication, OpenMPI can be configured in two ways. Eiter share all possible compartments potentials with every node in the network or only share the compartments potentials to nodes that need them. These methods rely internally on MPI Allgather and Alltoallv respectively. When available, GPUDirect, NVlink, and RDMA are supported. The implementation hides communication overhead, when possible, by concurrently executable compute kernels. A neuron model from the Inferior Olivary Nucleus is selected for benchmarking. Reported results go up to 32 Nodes with a total of 64 GPU cards. The design shows linear weak and strong scaling within the experimental setups for intra-node and inter-node scalability. With this simulator, networks over 10 million cells become available to model on large-scale GPU clusters, setting a new standard for eHH simulations. Comparisons against related work on CPU and FPGAs have been conducted, a 100x speedup is achieved versus a single cpu threaded solution. Furthermore, a 2x speedup is achieved over an FPGA solution (flexHH) and 10 fold over a multithreaded CPU (GenEHH, with 128 threads) solution, both reported speedups are for a fully connected network with 7000 IO cells.

Acknowledgements

I want to thank my thesis advisor, Christos Strydis, and his entire lab, the Neuro Computing Lab at the Erasmus MC, for the advice and guidance during this project. The whole project was conducted in a challenging time due to a global pandemic at the time. However, Cristos and his lab managed to keep my morals high and created a workable environment suiting the restrictions that were in place. Likewise, a thank you to Zaid Al-Ars, my supervisor from the TUDelft, for helping me select this thesis subject and advice on my work during the project. I also want to acknowledge multiple institutes supporting the project with hardware resources. The support of NVIDIA Solutions Lab with the donation of the Tesla V100 GPU development cluster, the Greek Research and Technology Network for making Aris available for this project, and the Swiss National Supercomputing Centre to access their development cluster AULT.

To finish, I want to thank my parents and friends, who supported me through my studies.

Package managers are overrated (just kidding), it is true for pretty printing in ASCII.

Max Engelen
Delft
Januari 17, 2021

Contents

List of Figures

1 Introduction

1.1. Thesis scope

Since the 1970s, there has been a lot of specialized graphics circuits. At the start, Graphics Processing Units (GPUs) were nothing more than integrated frame buffers. Since then, GPUs have evolved into programmable and highly parallel devices [25]. The ability to execute many instructions concurrently is used heavily in all sorts of computational problems. Ranging from gaming to scientific compute loads.

An example of those scientific compute loads is present within the field of computational neuroscience. Computational neuroscience focuses on building models that can explain and/or predict experimental neuroscientific data. The appliance of GPUs will generate the ability to simulate more complex computational models within a reasonable time, enabling the possibility to gain insights into such models' functionality. Eventually, create a better understanding of the functioning of the brain.

Questions in computational neuroscience can span a wide range of traditional analysis levels, such as the brain's development, structure, and cognitive functions. Research in this field utilizes mathematical models, theoretical analysis, and computer simulation to describe and verify biologically plausible neurons and nervous systems. For example, biological accurate neuron models are mathematical descriptions of spiking neurons that can represent both the behavior of single neurons and the dynamics of neural networks. Computational neuroscience is, therefore, often referred to as theoretical neuroscience. The neuron model depends on the goals that the neuroscientists want to achieve. If these goals depend on neuronal behavior that depends on measurable physiological parameters, Hodgkin-Huxley (HH) type models are the best fit [23].

However, Izhikevich stated in 2004 that only tens of coupled spiking neurons could be simulated in real-time. This computational problem is solvable with the ever-improving computational platforms. Therefore, models can extend to a lot more than tens of neurons, real-time, in the future. Hodgkin-Huxley modeling has evolved with many modern extensions (extended HH, eHH), intercellular connections, multiple compartments per cell, and additional user-defined custom ion gates. A widely used model to benchmark simulators is the Inferior Olive (IO)-model [14]. The IO-model utilizes a lot of the abilities the extended HH models can capture. The Erasmus MC's Neurocomputing department uses this model to benchmark their Brainframe platform [36], working on a heterogeneous accelerator platform for neuron simulators. (The work done for this thesis will supplement this platform.)

To support the full range of models that one can create using the extended HH formulation, an effort is made, by Rene Miedema, to build a flexible hardware library (flexHH) [26]. Generalizing the computational task helps to create scalable solutions without knowledge of network dynamics. This library is focused on FPGA solutions (Data flow Engines (DFE)) but contains a particular approach that can potentially benefit a broader computational platform range. Based on the flexHH method, an in-depth analysis of scalable GPU platforms is performed.

Simulation of the full brain, with high detail, is still far out of scope for any neuroscientists, simply because the compute power available does not come close to handling such models. These simulations will stay out of scope for quite some time after writing this thesis. However, an attempt to show the current potential of computing systems known at the time of writing could create exciting insights for the future road to walk. To display the potential, distributing the task over multiple computing systems would be beneficial for computing throughput.

This thesis aims to implement a new versatile, multi-GPU eHH simulator named mgpuHH and explore its performance, and make general observations on performance scalability over different modeling and cluster-configuration properties. Each neuron has a set of ordinary differential equations (ODE) that needs to be solved. The typical approach is to use a numeric solver (e.g., forward Euler). Implying that for every timestep, the intermediate state of all cells is calculated in parallel, suiting GPUs very well.

Exploring the boundaries of what is possible, in terms of neuron count and average connection number amang the neurons in a system, on specific hardware configurations, and bottlenecking the experiments' scalability will be the main focus point. However, an exploration in best practices when designing GPU and/or scalable solution for this type of computing load will be covered when developing a platform that suits the goal to simulate of the biggest IO-model ever seen in the field of neuroscience, with a reasonable biological time over simulation time ratio.

1.2. Challenges and Research Questions

Scalability will increase complexity for the simulator but boils down to the problem of communication and memory management of the model under simulation. The neurons in a network share a predefined set of interconnections, making it challenging to maintain high performance while scaling the model across multiple units (compute nodes and/or GPUs). A distributed solution also suffers from a problem known as load balacing. Where it is important each unit has the same amount of computational work and perfromancem. When this is not the case units are in idle when having dependencies to units that still need to finish their task.

Using a solver technique to solve the ODEs for every cell means there is a need to share specific data from one neuron to all connected neurons for every solver's timesteps. Furthermore, one or multiple kernels are developed to apply the solving technique for each timestep. The platform is designed in CUDA (Compute Unified Device Architecture), NVIDIAs parallel computing platform, and application programming interface (API), giving the most programming freedom on GPUs, but limiting development to NVIDIA products only. A new C++ Runtime API and Kernel language know as HIP could change this, making it possible to develop for AMD and NVIDIA products using the same codebase. Tools exist to automatically covert CUDA to HIP, making it possible to change to codebase later on relatively quickly when there is a need to support AMD GPUs.

In this thesis, we will face multiple research questions, which will be centralized around the following problem statement: **How can we efficiently simulate neural models, growing numbers of densely connected neurons keeping the solution as high performing as possible, utilizing multi-node GPU compute platforms?** This problem statement gives rise to a set of research questions that are answered in this thesis.

- How does growing neural networks influence the simulation on multi-node GPU compute platforms?
- What can be learned from other platforms and migrate over to a GPU platform?
- How does a multi-GPU platform and/or multi compute node configuration scale with respect to performance?
- What is bottlenecking scalability, and how (if possible) could these bottlenecks be resolved?
- How can we optimize the compute functions to reduce memory footprint and reduce wallclock performance utilizing the GPU specific architecture?

1.3. Overview

This thesis is structured as follows:

Chapter 2: Gives the reader all the background information they need to have about the topics covered in this thesis. For example, high-performance computing, CUDA, and the basics of neuro-computing are covered.

Chapter 3: With the background information in hand, it should be possible to follow the related work's breakdown provided in this chapter. The related work mainly focuses on the predecessors that lead to this research. It also briefly covers other projects in the same field to give an overview of exciting projects that could potentially provide ideas for this work.

Chapter 4: Will cover the high-level design of this work. It will give insight into data flow and critical paths in the simulator. It will also highly why design chooses were made and what possible other options could be.

Chapter 5: When implementing the design, there are still different option that can be selected. These will be covered in this chapter. Mostly, there is no predefined optimal way to do it, and therefore experimentation will be used to deduce the best way to implement the design.

Chapter 6: Will cover evaluation of the design and focusses on answering the research questions, by utilizing multiple compute platforms to verify findings. Results will be presented and compared with each other.

Chapter 7: Will conclude the work. It also will conduct a reflection on the research questions and answers found in Chapter 6. Furthermore, the open research questions and future work are formulated.

2 Background

This chapter provides a deeper dive into the modeling and the hardware solutions available to create a performance-driven simulator. It is essential to notice that this is a computer engineering thesis, and any neuroscience covered is only there to support the given explanation.

2.1. Neuron Simulations

The brain consists of roughly 8.6×10^{10} (eighty-six billion) neurons on average [20]. The average synaptic connection count between the neurons is 7000 [15].

A sparsely saved graph to capture all these connections would theoretically be 2500 terabytes (4 bytes per edge). Unfortunately, this is still unthinkable in standard compute systems and will not be there soon. For example, take Fugaku (number one of the top500 at the moment of writing [3]) and perfectly utilize all system memory. It would come short, more than 1800 terabytes. Summit (number two of the top500 at the moment of writing) could theoretically hold the graph in system memory. Nevertheless, that is just the synaptic connection graph (and only if the program could perfectly distribute the connection graph across all nodes).

The way neuroscientists simulate neural networks is dominated by modeling them as Spiking neural networks (SNNs). SNNs can more closely mimic natural neural networks. The concept of time is integrated into the network, together with the neuronal and synaptic state. When a neuron fires (spikes), the spike travels along the synaptic connections and influences connected neurons' potential. In the field of neurocomputing, it is established that the firing patterns of neurons are limited to a particular set. Multiple spiking models are ranging from very basic to quite complex. The differences are discussed in the Izhikevich paper [23]. The most challenging neuron models, compute wise are considered to be neurons of the Hodgkin-Huxley type.

2.1.1. Hodgkin-Huxley Models

Conductance-based models in the Hodgkin-Huxley-type subset are giving biophysically meaningful and measurable results. An overview of the basic Hodgkin-Hexley model is presented in Figure 2.1. However, they come at the price of high computational complexity [21]. This work focuses on these models because of the computational complexity, which poses interesting compute challenges. The basic Hodgkin-Huxley equations can be extended in various ways. One common case is adding intercellular connections (gap junctions), custom user-defined gates, and multiple compartments per cell. Then, the resulting modified HH models are generally known as the extended Hodgkin-Huxley (eHH) formulation. The Inferior Olive model (IO-model) [14] captures these extensions and is often used as a base network to benchmark/test simulators within the Neuroscience Department of the Erasmus MC. The IO-model is a part of the brain that coordinates signals from the spinal cord to the cerebellum, regulating motor coordination and learning [33].

2.1.2. Numerical Solvers

Numerical approximations can be utilized to solve Ordinary Differential Equations (ODEs). Many ODEs cannot be solved through symbolic computation, or in other words, analysis. A numeric approximation

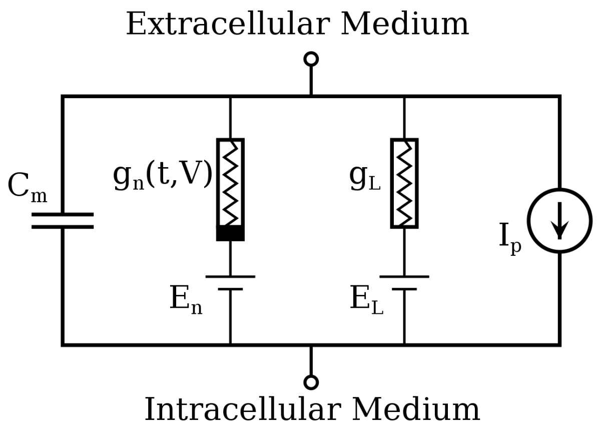


Figure 2.1: The components, representing the cell membrane's biophysical characteristics, of the Hodgkin–Huxley-type models. The capacitance (C_M) represents the lipid bilayer. The ion channels are modeled by linear (g_L) and nonlinear (g_n) conductances. The voltage sources (E_n and E_l) represent the electrochemical gradients driving the flow of ions. The current source (I_p) represents the ion pumps and exchangers.

is, most of the time, sufficient. For neural simulations, a numerical approximation is adequate. The ODEs in question have an unknown derivative with respect to time. The problem can be tackled with implicit or explicit methods. Implicit methods find a solution by solving an equation involving both the current state of the system and the later one, where explicit methods only take the current state into account. The space in time of interest is divided into a certain amount of steps the solver will take. The smaller these time steps are, the better the resolution will be. When the timesteps are too big, the networks will show instability. Both the instability and resolution are also depending on which solving techniques are used.

2.2. Compute Hardware

Since processor clock frequencies have not significantly increased since 2006 [5], modern CPUs are developed to do as much as possible in each processor cycle. By using techniques like instruction-level pipelining and out-of-order execution, modern CPUs maximize the effective number of instructions per cycle. However, this has some significant drawbacks. Processor-level optimization requires many extra transistors, power, and engineering, but for increasingly little performance gains. Therefore different approaches to the computing world gained traction.

In the early 2000s, the first commercial multicore processors are developed. By simply duplicating a slightly less complex processor core, suddenly performance could, theoretically, be doubled. As transistor counts continue to go up, it has been predicted that core counts will also increase exponentially. [12] However, as Amdahl's laws teach us, adding more processor cores will stop making sense as long there is a sequential part in the execution path.

The ambitions to get additional processing power drove the industry in making faster and/or architectural different processors. However, it also fueled people to build various architectures and configurations with off the shelf components. With the development of computer interconnections, the compute clustering principle was born. The Clustering approach will usually but not generally connect existing compute units through fast interconnection to create a system that has more power shared than a single compute unit.

High-performance computing (HPC) uses large scale compute resources for computational problems that are not suited for standard computers. Compute clusters fit perfectly under the definition of large scale compute resources. A wide variety of Parallel techniques, such as distributing and parallelization libraries, are used to utilize these large scales compute resources. These tools are useful for all science that rely upon computing throughput.

2.2.1. GPU

The GPU, or Graphical Processing Unit, represents an extension of the idea of multicore processors. GPUs often consist of many cores that, individually, are not all that powerful. They can perform some simple instructions but are not optimized for a maximum performance like CPUs often are. Instead, the GPUs strength comes from the fact that it has so many parallel cores. That way, parallelizable programs' workload can be divided over all cores, severely decreasing the required runtime.

Utilizing the GPU in software is not straightforward for the developer. In a ”normal” multicore system with just a few cores, the program is already being run on multiple processors, requiring synchronization of instructions, data access, and hardware access. When programming for a GPU, one can have hundreds of cores, only further worsening the problem. In addition to this, GPUs usually have much smaller memory caches and altogether different memory structures, which the developer also has to take into account [7].

Luckily, this is a problem that many people had since different applications try to use GPUs to improve performance. Furthermore, as is often the case for such common issues, solutions exist in the form of software tools. One of the most advanced GPU programming frameworks is called CUDA. This framework allows the user to (relatively) quickly develop software that can execute on a GPU. CUDA, however, is vendor specific and is made and maintained by Nvidea. Limiting this library to Nvidea products only.

GPUs offer the opportunity to parallelize calculations massively. Suited calculations can, therefore, massively benefit from GPU support. The perfect example is pixel rendering. The reason GPUs were invented in the first place. However, this is for sure not the only field of calculations suitable. The Scientific field quickly saw potential in using GPUs to increase performance on a wide variety of calculations and is therefore getting more and more traction in a wide range of application fields.

2.2.2. Multi-Node Systems

Nowadays, the scalability of computing systems is getting quite common. The well-known top500 computers exploding multi-node systems with local interconnection going up to 100 Gb/s with InfiniBand EDR, amazingly powerful systems can be configured and exploited for all sorts of applications.

A single node of such systems can again hold multiple processors, and expansion cards such as GPUs, FPGA, and custom logic cards can be added to target specific bottlenecks of applications. Management software such as Slurm [4] are well developed, makes using these systems very user friendly.

2.2.3. Cloud Based Compute Systems

Cloud computing makes the hardware, software, and data available on request via a network, often the internet. Services as Google Cloud, Microsoft Azure, and Amazon Web Services provide a wide variety of services one can utilize. Setting up compute units with a particular hardware configuration with a few mouse clicks unlock tremendous potential. There is no need to buy and maintain high-end machines anymore if there is the possibility of just renting the desired services. Of course, this does not suit all projects equally well. Nevertheless, for the sake of testing, for example, self-made application on a wide variety of machine configurations, this would do the job for just a few bucks.

2.3. Platform Scope

For this work, a specific set of tools and possible platforms are within the project's scope. A typical computing system that lies in the project's scope is graphically shown in Figure 2.2. All components will be briefly highlighted to understand their potential and why the project can benefit from utilizing them.

2.3.1. CUDA

CUDA gets often described as an API or even a stand-alone language, which is not the case. It is best described by a computing platform/programming model that uses a GPU for general-purpose computing. The developer still programs in one of the supported languages (C, C++, Fortran, ..) and incorporates extensions of these languages in the form of a few keywords. Using these keywords will direct the compiler to map parts of the application to the GPU, unlocking the parallelism a GPU offers.

CUDA has the capability of launching Kernels within a particular dimension, called a grid. This grid consists of blocks build of threads. Within a kernel call, everything is symmetrical. This results in two parameters passed alongside a kernel call, the grid dimension, and the block dimension. Dimensions can be one, two, or three dimensional. Every thread then executes the same functions passed to the

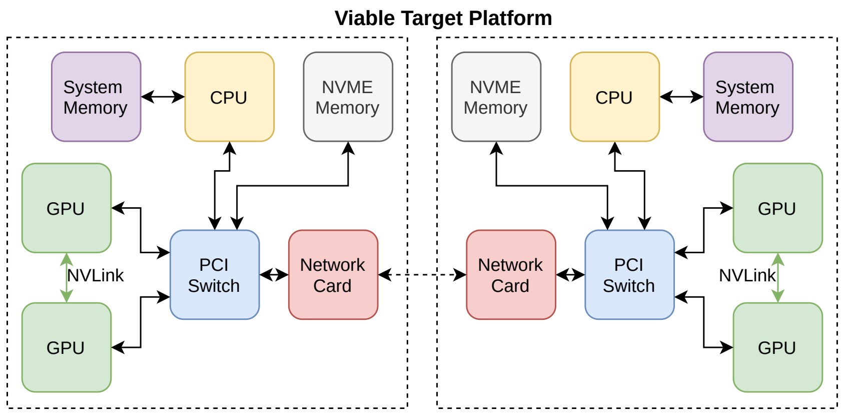


Figure 2.2: Overview of a viable target platform. This platform consists of 2 nodes, 2 GPUs each. Where each node has his own NVME memory, the GPUs are equipped with the NVlink bus. The dotted line illustrates inter-node connection. The solid lines represent intra-node connections between components.

kernel. Within these functions, the thread's location with respect to the grid is known, giving all sorts of freedom with the way parallelization is applied.

An Important Part of the performance is memory latency. Therefore, for NVIDIA architecture from Fermi and up, the architecture consists of four hierarchical levels. Device memory, l2 cache, shared memory, and registers. Device memory is where all data allocations goto and has the most significant latency. L2 cache is an ordinary cache for this device's memory. The architecture has multiple streaming multi processors (SMs). Each of these streaming multi processors is responsible for executing a block of threads described in the launched kernel. This SM has then shared memory, which is accessible by all threads within the same block. An SM consists of multiple cores. A core can handle a single thread. A simplified overview is given in Figure 2.3. Note that an SM will consist of more cores, but only two are drawn for simplicity.

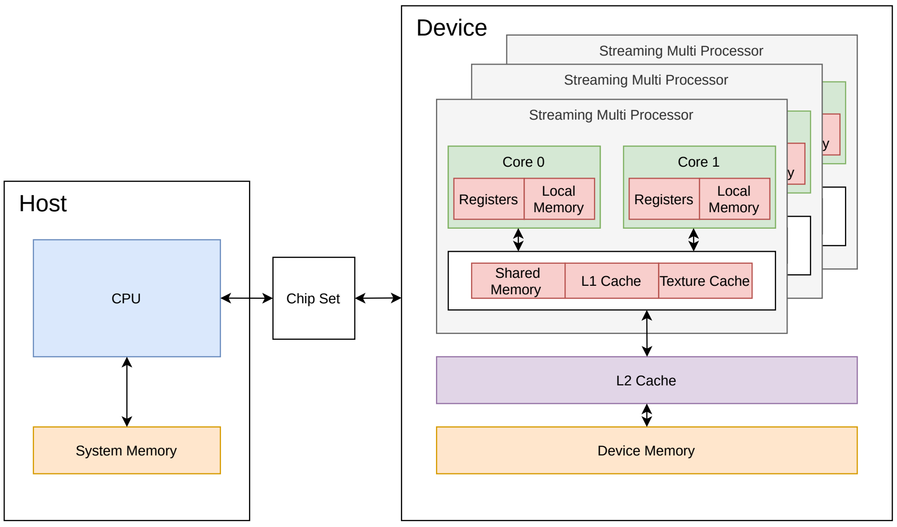


Figure 2.3: An logicall overview to illustrate the design NVIDIA GPUs are using.

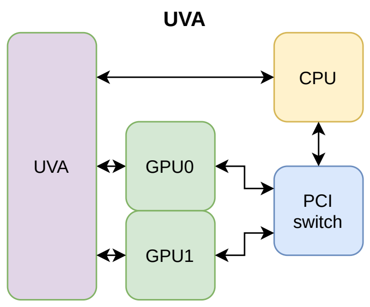
Nvidia GPUs are using Single Instruction Multiple Thread (SIMT) execution. Execution happens in warps, of 32 parallel threads, that each having its own registers. However, they are still able to load and store from divergent addresses. The threads within a warp can follow divergent control flow paths, but performance takes a hit because all threads within the warp that are not on the same flow paths are inactive during divergent execution. CUDA provides warp-level primitives such as $\_\_shfl\_down\_sync()$ to perform a tree-reduction within the warp.

The multiprocessor occupancy is the ratio between active warps and the maximum amount of warps supported. Each SM has a certain amount of registers available. These registers are a shared resource among the thread block executed on the SM. Together with the amount of shared memory, the amount of active warps is derived, and the occupancy can be calculated. The higher the occupancy, the more thread blocks that can be active in the machine simultaneously.

Host to device data transfers Memory copies between system memory and Device memory are passing through the CPU every time. CPU data allocations are by default pageable. GPUs, however, cannot access pageable host memory directly. Therefore, the CUDA driver must first allocate page-locked (pinned) memory and then start transferring data to the device memory. ($cudaMallocHost()$ is used for pinned memory allocations)

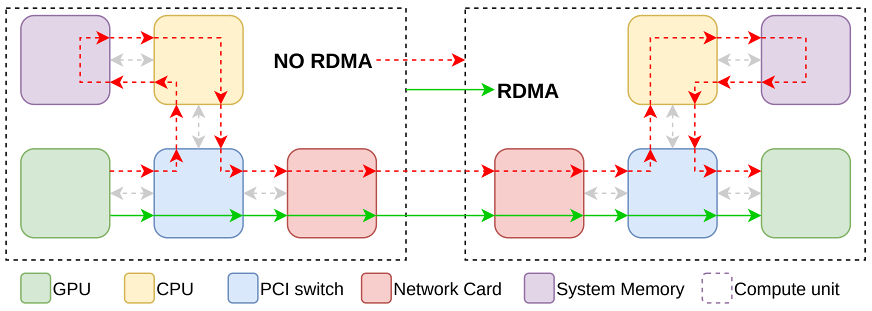
NVlink High-speed GPU interconnects connecting multiple GPUs is what NVLink offers. Therefore, it is a significantly faster connection than multi-GPU systems that can utilize traditional PCIe connections. If available, CUDA drivers will automatically select this faster interconnect. For example, p2p data transfers will use this connection.

Unified Virtual Addressing (UVA) CUDA 4.0 (compute capability two and up) introduced the concept of UVA. With UVA, the host memory and all device memory of GPUs residing in a single compute node are combined into one large virtual address space. Making memory accesses available without knowing where the data is physically located. ($cudaMallocManged()$ is used for UVA allocations is available)

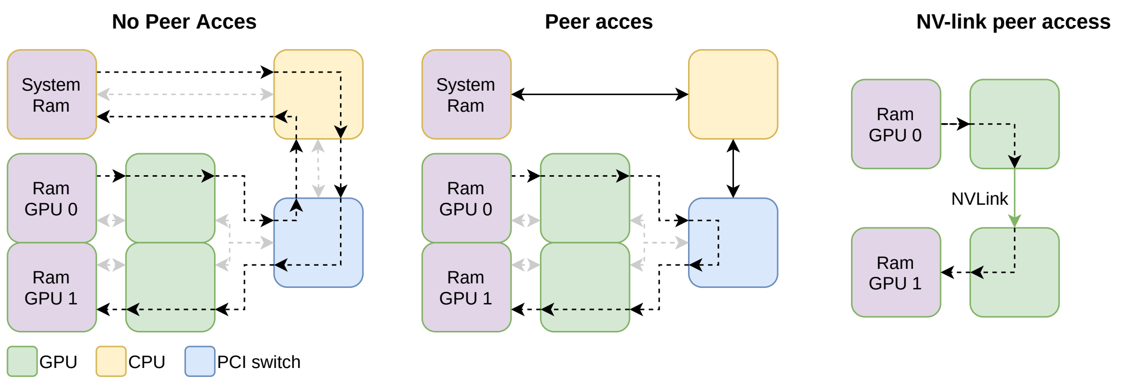


GPUDirect ”NVIDIA GPUDirect®” is a family of technologies, part of Magnum IO, that enhances data movement and access for NVIDIA data center GPUs. Using GPUDirect, network adapters and storage drives can directly read and write to/from GPU memory, eliminating unnecessary memory copies, decreasing CPU overheads, and reducing latency, resulting in significant performance improvements. These technologies - including GPUDirect Storage, GPUDirect Remote Direct Memory Access (RDMA), GPUDirect Peer to Peer (P2P), and GPUDirect Video - are presented through a comprehensive set of APIs.” [2]

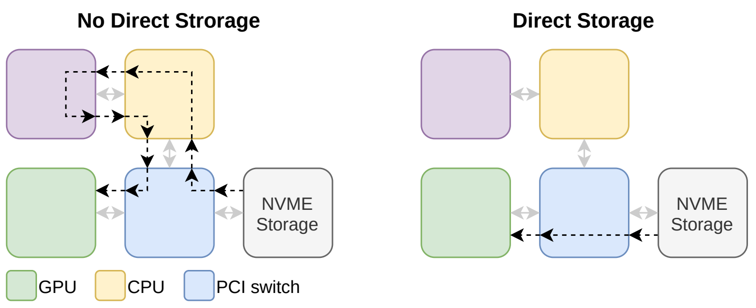
- **RDMA** - Is the ability for third party PCIe cards to directly access the GPU memory address space. Making direct communication between GPUs across a cluster possible. Mpi will benefit significantly from this technology. (Requirement that Both (GPU and network) PCIe cards are on the same PCIe root complex)



- **P2P** - Multi-GPU systems have to pass memory copies between GPU usually through system memory. This is not needed anymore with P2P support. It enables systems to directly transfer data through the PCIe bus to a different GPU residing in the same PCIe root complex.



- **GPUDirect Storage** - The standard route data travels from an NVMe drive to GPU device memory is through a bounce buffer in system memory. This one-way data path is simplified by having a direct path from e.x. NVMe drives to GPU device memory. It only provides this path and not reversed. PCIe Peer-to-Peer is a prerequisite for GPUdirect Storage to work.



- **GPUDirect Video** - Ability to directly pass buffers to video IO. Not in the scope of this work.

Coalesced memory access The combination of multiple memory accesses into a single transaction refers to coalesced memory access or memory coalescing. For example, in a single transaction, any successive memory of 128 bytes can be accessed by a warp. However, uncoalesced load can result in memory access becoming serialized: misaligned memory access, memory is not sequential, and memory access are sparse. Serialization of memory accesses will affect memory performance. Therefore this is a vital feature to keep in mind while implementing CUDA Kernels.

2.3.2. OpenMP

OpenMP (Open Multiprocessing) enables easy multi-threading. One multiprocessing approach is creating multiple threads and splitting the program into parallel chunks executed by a specific thread. OpenMP takes a different approach. It tries to distribute tasks or parts of the task that can be executed in parallel, at runtime over a thread team. OpenMP works out, through compiler directives, which parts could be executed in parallel. OpenMP then determines at runtime which threads should perform which part of which task. OpenMP is prone to Non-Uniform Memory Access (NUMA) effects because multiple processors can access the same system memory. Therefore, applications need to be designed with these effects in mind.

2.3.3. OpenMPI

When systems get bigger and grow, for example, to multi-node systems, some communication and synchronization between distributed processes are needed. The open MPI project is a message passing interface that brings a solution to these problems. OpenMPI is widely used across HPC platforms and, therefore, a perfect fit. Because each openMPI process has its pool of resources, it can simplify design over openMP, but potentially performance will take a hit. Therefore a hybrid system with openMP inter-node and OpenMPI intra-node is a popular design choice. OpenMPI works on with a modular architecture, which allows for a wide variety of supported hardware and software configurations. OpenMPI is built upon the De facto standard MPI, and an active community backs an open-source project. [16]

CUDA-aware openMPI When openMPI is built with extensions to support CUDA, the libraries are called CUDA-aware openMPI (CAM). The significant difference is that instead of passing host buffers to mpi, it is possible to pass GPU buffers directly. Depending on platform support, CUDA-aware openMPI behaves differently. GPUdirect plays a significant role here. Depending on RDMA support or P2P support, MPI will not have to stage the buffers through the CPU. However, if GPUDirect is not available, MPI will still be able to transfer the request through the standard memory copy paths successfully.

3
Related work

Various HPC projects are focussing on neural network simulations. Scalability is often a problem within these projects and, therefore, either not present or not researched extensively. Most common projects focus on x86_64 CPU architectures with some, extending there focussed to GPU support. Various projects focus on only supporting a certain subset of added features to their implementations and exploring the boundaries of what is possible computational wise on certain hardware platforms. The other approach makes the implementation extremely parameterizable to support a diverse as possible range of neuron descriptions such as NEURON, NEST, BRIAN, which are well established neural simulators. Pushing the boundaries of computing possibilities and learning valuable lessons for the future of neurocomputing will contribute to this work. To model neurons for simulations, specific tools are designed to unify the description. NeuroML [17] and PyNN [13] are examples of such descriptions. The idea is that simulator platforms can be build taking this description as input, Making it fairly easy for a neuroscientist to pick a simulator that suits their wishes and deploy the neural descriptions without the need to (re)write code and/or model description.

The chapter will start with historically recounting the Hardcoded Inferior-Olive work conducted by the Neurocomputing department lab (NCL) at the Erasmus MC. Following the NCL developments, the next section will focus on stepping away from hardcoded models and scaling to multi-node systems. The last section of the chapter will highlight different standalone neural simulators other research groups are developing.

3.1. The Inferior-Olive Race

Platforms pushing compute possibilities of neural-network simulations tend to select the extended Holdkin-Huxley (eHH) type of neural descriptions. These are very compute-heavy descriptions of models and, therefore, perfect for exploring the possibilities that lay in accelerations via different compute. A model that is often utilized for research into this spectrum of the field is the inferior Olive model (IOmodel) from de Gruijl [14]. The de Gruijl model uses all features of the eHH type.

The model is build-out of three compartments the dendrite, soma, and axon hillock. Each compartment has a potential affected by the channel currents ($I_{Channels}$). These consist of: the leak current (I_{leak}), the interaction current induced between the compartments ($I_{interact}$), the interaction currents trough the synaptic connections(I_{gap}) and the externally applied current (I_{app}). These currents act on on the membrane capacity (C_m) which create a potential (Equation (3.1)). (The formulations of these currents are described in the additional material of the paper from de Gruijl [14], and repeated in the Appendix A)

$$\frac{dV}{dT} = \frac{I_{App} - I_{Channels} - I_{Gap} - I_{Interact} - I_{Leak}}{C_m} \tag{3.1}$$

A single channel can consist of multiple gates. These gates influence the current generated by a channel and are dependent on the current compartment potential. All these equations describe an Ordinary Differential Equations (ODEs) system, making simulation only possible with a numerical approach as expected. In Christoph et al. [8] a comparison is made between different solvers and exploring their timestep requirements. Using exponential time differencing does seem to make sense for HH, like partial and ordinary differential equations. However, it adds complexity with respect to the standard explicit time-stepping schemes, which may or may not be worth it in specific cases. To compare standard time-stepping schemes (e.g., FWD-Euler and Runge-Kutta methods (rk2 and rk3), Miedema et al. [26] implemented FWD-Euler, rk2, and rk3 resulting in the fastest execution for the fwd-Euler method for the edge cases of stability. It is important to note that the timestep states' quality is different with different methods, so it is always weighing performance against result resolution.

The reason that time steps need to be small comes from membrane potential overshooting. The rapid rising of the membrane potential often only lasts for a few 10s of microseconds. Some kind of adaptive stepping method could be beneficial for a single cell. However, as Christoph et al. [8] states, conventional timestep control mechanisms would force the whole network to scale down the time step whenever a single neuron fires disabling all the benefits when the network is not in perfect synchronization. All implementation considered in the rest of this section only supports the standard time-stepping schemes.

In Smaragdos et al. [34], the IO model went under the loop. Their Xeon Phi implementation of a hardcoded IO model could simulate up to 24 neurons matching simulation time with biological time, with a timestep of 0,05ms. Furthermore, simulation results up to 7,680-cell networks are presented with fully connected gap junction networks. These accomplishments can be seen as two types of experiments. Type 1: With fine-tuned model parameter supporting networks from 10 till 100s in real-time. An example of such an experiment is given in Yamazaki et al. [39] where they were able to simulate a Cerebellar model containing 102400 granule cells, 1024 Golgi cells, 16 Purkinje cells, 16 basket cells, one inferior olive, and one neuron in the cerebellar nucleus in real-time. Simulating a more extensive network or even a more biologically accurate model could potentially benefit this field substantially. Experiments of Type 2 are large-scale networks which Smaragdos et al. [34] classifies as everything larger as 1000 neurons. They also consider two fundamentally different implementations for the IO model, a Data Flow Engine and a Xeon Phi implementation. Types 1 experiments seem suited for a data flow engine, where type 2 experiments are better suited for the Xeon Phi implementation. Graphical processing units (GPUs) are not considered.

Nguyen et al. [28] considered Graphical processing units as an accelerator, with a maximum speedup of 160 times over a CPU implementation. However, the catch is that the implementation supports only eight neighboring connections via gap junctions and nothing more. Making the simulation inherently more parallelizable than, for example, a fully connected network. The implementation is a hardcoded IO model, as previously mentioned. The work is reporting results for network sizes of over 1 million neurons. With 120,000 simulation steps, this network takes over 40,000 seconds of simulation time. Where a comparable FPGA based implementation, at the time, could only simulate up to 14,400 cells but can simulate 96 neurons in real-time while having a fully connected gap junction network [35].

A point can be made that FPGAs require a certain skill set to work with that most neuroscientists do not possess. Therefore Miedema et al. [26] suggested a flexible hardware library (flexHH) that does not need new firmware to run a different network configuration. FlexHH is reporting to support 166 neurons of the IO model in real-time while heaving a fully connected gap junction network. It still supports simulations of up to 24,576 compartments (8192 Neurons). The point that Smaragdos et al. [34] made that FPGA are better suited for Type1 experiments and CPU better suited for Type2 experiments seems to hold with the remark that GPU seems more promising for Type2 experiments.

Brainframe [36] is a project that tries to bring all these possible solutions together under one roof, giving users of Brainframe control and freedom to choose simulator type and configuration. Brainframe aims to extend and complete the previous work by Smaragdos et al. [34]. Brainframe only shows results for a hardcoded IO-model. However, these are excellent benchmarking results for more general approaches to compare against. The work is presented as a single compute node application, but it is intended to be extended to Multi-node computing platforms.

Supporting multi-node systems opens up this whole new area of High-Performance Computing. In Chatzikonstantis et al. [10], a multi-node CPU implementation is presented for the hardcoded IO-model. The work presented only saw a Small speedup between utilizing 1 and 2 nodes. Additionally, scaling to more nodes showed no further improvements. They did, however, create a new standard for CPU simulation of the IO-model. This work simulated a neural network simulation of 2 million neurons, with 1000 connections per neuron.

Vlag et al. [38] took another approach by utilizing the best of both worlds with there brainGPU simulator. A multi-node multi-GPU setup, reporting promising scalability trends. However, when reviewing the source code, a significant bug was found in the generation of the gap junction networks, potentially favoring the published performance results. The bug appears in the generation of the Synaptic connection networks, where one kernel per cell is launched to generate the connection list. However, each thread within this kernel starts with setting the connection count back to zero. One can never be sure that all threads are running concurrently, and therefore generated connections can get lost. Furthermore, the memory allocated for the connection lists does not have any overflow protection. And because of the stochastic nature, it can never be ensured to stay within the allocated memory bounds. However, this will not affect the trends shown in the paper but potentially shifts the tipping point. Therefore, brainGPU will not be used for comparison at this stage.

3.2. Stepping Away from Hardcoded Models

While the IOmodel is pushing compute to the limits and setting new performance standards. It is not useful for neuroscientist that want to have the ability to tweak the experiments without having knowledge about HPC programming or even programming in general. Therefore Miedema et al. [26], work that was already mentioned in the IO race, designed a way to generalize networks of the eHH type. This hardware flexible design is called flexHH. The general idea without the Data Flow Engine (DFE) implementation can also be beneficial for experiments of large networks, Type2, where the DFE was focussing on smaller experiments classified as Type1.

flexHH

A way to generalize the extended HH equations would be preferable. It will allow for a single logic process to simulate a range of different neural configurations. In Miedema et al. [26], a flexible hardware library is proposed, which makes this possible. The IO-model is used as a basis, and extra features such as custom ion gates are supported within this general description. This is established by creating a set of functions that makes it possible to describe all gates using the same formula set. This adds overhead to hardcoded models but gives back the flexibility to use the same hardware/code base for variations of models. As shown in Miedema et al. [26], remarkable results can be achieved on an FPGA platform. This thesis will use the basic idea of the flexible library and port it to a GPU accelerated codebase.

The generalized mathematical equations used in flexHH are shown in Equations (3.1) – (3.10). Where Equation (3.10) might change if custom gates are not supported. All variables written in lower case are constant values that are model configuration specific. This collection of equations need to be solved for each neuron in the system and timestep synchronous because of the intercell dependency of the gap junction calculation (Equation (3.3)). Therefore the problem is twofold. Firstly update the compartments, secondly share the synaptic potentials across the network.

$$I_{interact,i} = g_{int} \sum_{j=0}^{n_{comps,i}-1} \frac{V_i - V_j}{p_{i,j}} \tag{3.2}$$

$$I_{gap,i} = \sum_{j=0}^{n_{conected_cells}-1} (w_{i,j}(c_0 exp(c_1 \times V_{i,j}^2)V_{i,j}) \tag{3.3}$$

$$I_{leak,i} = g_{leak,i} \times (V_i - v_{leak,i}) \tag{3.4}$$

$$I_{app}(step) = \begin{cases} A & \text{if } step_{start} \leq step < step_{end} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

$$\begin{aligned} I_{channels} &= \sum_{j=0}^{n_{channels}-1} I_{channel,j} \\ &= \sum_{j=0}^{n_{channels}-1} g_{c,j} \times (V - v_{c,j}) \times YProd_j s \end{aligned} \tag{3.6}$$

$$YProd = \prod_{i=0}^{N_{gates,i}-1} Y_i^{p_i} \tag{3.7}$$

$$\frac{dY_i}{dt} = (1 - y_i) \times \alpha - Y_i \times \beta_i \tag{3.8}$$

$$\frac{dY_i}{dt} = \frac{inf_i - Y_i}{tau_i} \tag{3.9}$$

$$fCustom(V, x[8], f_t) = \begin{cases} \frac{x_5(x_1-V)}{x_0 exp((x_1-V)x_2)+x3} + x_8 & \text{if } f_t = 0 \\ \frac{x_8}{x_0 exp(x_2(x_1-V))+x_3+x_4 exp(x_5(x_6-V))+x_7} & \text{if } f_t = 1 \\ \frac{x_0((x_1-V)x_2)+x_3}{x_4 exp((x_6-V)x_5)+x_7} + x_8 & \text{if } f_t = 2 \\ min(x_0 V, x_1) & \text{if } f_t = 3 \end{cases} \tag{3.10}$$

Equation (3.7) does not show support for instantaneous gates in this description, which means that the gate does not rely on a DFE system that needs to be solved but directly dependent on the compartment voltage. This is, however, supported in the implementation that flexHH uses to validate it is functionality. Instantaneous gates are necessary if the gate descriptions are not stable because small changes push the state to infinity and minus infinity quickly. Instantaneous gates can be solved by using $fCustom$ to calculated the activation function. The results can be plugged directly into Equation (3.7) as that Y_i value of that gate.

The flexHH paper did optimize the equations to some extent to be better suited for FPGA usage. Calcium concentrations are modeled as a gate and, therefore, can only depend on previously described channels. However, calcium concentration is compartment-specific and will depend on some subset of the $I_{Channel}$ currents. Equation (3.6) makes the channel current calculation sequential. The same sequential requirement hold for the Calcium updates. Updating the gate DFE system does not have any such sequential requirements. Therefore it could make sense to categorize calcium concentration updates as part of a compartment. Making solving the gate DFE systems completely parallelizable. s Equation (3.2) supports different compartment ordering. However, the implementation only supports chained compartments, which is not a major implementation compromise but something to keep in mind when designing the simulator for this thesis.

The implementation supports FWD-Euler, rk2, and rk3 Solving methods. A conclusion on performances is made on these solvers, resulting in FWD-Euler coming out on top for all experiments. These results are not included in the paper but can be found in the MSC thesis work, with the same title, where the paper is based upon [9].

A novel simulator for extended Hodgkin-Huxley neural networks

FlexHH is a powerful generalization that enables high-performance computing while maintaining flexibility. Panagiotou et al. [31] took this philosophy and ported it to an x64 CPU Architecture. The implementation is named GenEHH: "a highly-configurable conductance-based neuronal network simulator." GenEHH can run on multi-core x64 architectures, utilizing openMP for parallelization over multiple threads. A JSON file configuration interface is proposed to make GenEHH usable for General users. The platform also supports concurrent output generalization, reducing the RAM footprint of the application because intermediate states are offloaded to disk memory. The implementation is utilizing an FWD-Euler solver, and results about scaling out to multiple CPU cores are in favor of adding more compute power. With a 2x degrees in runtime for every doubling of CPUs available. The implementation does support fully heterogeneous neural models and is deployable on any x64 system. Combining that with the JSON input configuration makes this platform extremely useful for an end-user. Also, in the IO-model race, its performance better than the work from Nguyen et al. [28] while supporting higher connectivity densities. The DFE implementation of flexHH outperforms GenEHH but does not support the same large scale networks as GenEHH can support. GenEHH focusses more on experiments of Type2.

Scaling to Multi-node

Scaling the idea of flexHH to a multi-node platform is not done yet. Chatzikonstantis et al. [10] scalability results are not promising. However, Vlag et al. [38] does report great results on scaling their hardcoded IO model over multiple nodes. Both implementation, however, suffers from communication overhead of the Gapjunction network. The work by Hahne et al. [19] proposes a unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. The framework is based on waveform relaxation techniques. Results seem very promising but also come with a big note that the communication overhead is replaced by more compute overhead. When the added compute overhead is lower than the communication overhead, it makes sense to use frameworks as described in this paper.

Jordan et al. [24] researched the feasibility of scaling neural simulators out to the post-petascale high-performance computing facilities. The communication scheme selected is utilizing the `MPI_Alltoallv` API function. It is argued that it becomes beneficial for neural systems with very sparse interconnection networks to have a more tailored approach to the communication problem that ”just” sharing everything with every process (`MPI_AllGather`) contributing to the simulation. The work by Vlag et al. [38] also adopted this approach in a similar fashion. However, a fair comparison between the option is not given which limits the insight into where the turning point, in terms of performance, of the different approaches, lies. This work does go into the ”Gaussian” versus ”Uniformly” generated interconnection networks. Where a gaussian generated interconnection network has more locality in its connections with results in less stress on the communication tasks. The work shows that a neural network that is more locally connected performance better in terms of execution time.

3.3. Standalone Neural Simulators

Some standalone neural simulators that are taken interesting approaches are SpiNNaker, HRLSim, and Arbor. There are many more simulators out in the world, with the focus laying on brain simulations, which will not be covered. It is chosen to highlight some exciting approaches rather than highlighting the most successful approaches.

Arbor

Arbor [6] is a high-performance library of multi-compartment, morphologically complex cells, from single-cell models to vast networks, for computer neuroscience simulations. To help neuroscientists efficiently use current and future HPC systems to fulfill their simulation needs. Arbor is written from scratch with several CPU and GPU architectures in mind. Arbor supports several neuron models but focuses on the simulation of multi-compartmental neurons. It supports spiking patterns with an asynchronous MPI-based spike communication scheme. Arbor enables extensive parallelism with one CUDA thread per state Mechanism update. It shows strong scalability results when deployed on multi compute node systems. Arbor is an opensource project that is still heavily under development and expected to develop to a full fletch way of supporting neural simulations.

NEST

The Neural Simulation Technology Initiative (NEST) is a simulator for SNN models that focuses not on the exact morphology of individual neurons but the dynamics, size, and structure of neural networks.NEST offers over 50 models of neurons, including LIF with synapses based on current or conductance, IZH, or HH models. Also included are several multi-compartmental neurons.

NEST uses two types of parallelization, thread-parallel simulation and distributed simulations, to conduct simulations on multi-core machines and computer clusters. To accomplish this, OpenMP and MPI are utilities. Allegedly, the NEST architecture scales to the biggest available petascale machine, but there is no multi-GPU implementation.

The Julich supercomputing center (JSC) has a NEST affiliated research group. Their research showed that the algorithms generating neural model instances and their connections scale well for neural networks on the scale of ten thousand neurons but do not offer the same speedup for millions of neuron networks. The work uncovers that the lack of scaling is due to inadequate memory allocation strategies and demonstrates that thread-optimized memory allocators recover excellent scaling. [22]

Neuron

NEURON provides a versatile and robust environment to implement biologically realistic models of electrical and chemical signaling in neurons and neuron networks. Neuron modeling comes in the form of NMODL. This high-level language allows models to be represented in terms of kinetic schemes or collections of differential and algebraic equations at the same time.

CoreNEURON is a simplified NEURON simulator engine designed to simulate larger network models on supercomputing platforms, optimized for computational speed and memory use. It can target architectures for GPUs and CPUs. Building the network model depends on NEURON.

Via the Open Accelerators (OpenACC) programming model, CoreNEURON supports GPUs. OpenACC is a parallel computing programming standard that offers compiler directives that define code regions to be implemented on a high-performance computing system. OpenACC does not restrict itself to NVIDIA devices but supports a variety of accelerator platform.

Brian

Brian is an open-source simulator for spiking neural networks written in the Python programming language. Brian has an extensive modeling environment and can be driven by (stochastic) differential equations. Custom models are used as equations that endorse a wide variety of neural models. The PyNN language can be used as an input for Brian. Two software packets, Brian2CUDA and Brian2GeNN, are available to accelerate Brian.

Brian2CUDA is an extension of the Brian spiking neural network simulator, implementing a standalone system to generate C++/CUDA code to run simulations on NVIDIA graphics processing units for general purpose (GPGPUs). This package is under development and has not yet been released.

Brian2Genn is a software package that connects Brian and GeNN. GeNN is a C++-based meta-compiler for accelerating spiking neural network simulations using consumer or high-performance grade graphics processing units (GPUs) GeNN [40]. In this way, the users can use Genn GPU acceleration without requiring any technical skills about GPUs, Genn, or C++. [37]

HRLSim: A High Performance Spiking Neural Network Simulator of GPGPU Clusters

HRLSim [27] is a spiking neural network simulator with GPU cluster systems as their targeted compute platforms. The main reason this work is of interest is the optimizations that were used to increase performance. The modeling, for example, is happening with a reduced precision integer approximation. For the message packaging, a different kernel is created to offload communication threads. Furthermore, kernels that do not have large dependencies on memory are assigned to run concurrently and resulting in better compute throughput. Also, the alignment of synapse related data-structures which prevents memory access overlapping is a great feature. They report having a simulator that scales well over multiple GPU cards, reporting results for up to 64 GPUs. The performance bottleneck is the synaptic updates. However, the project is discontinued and therefore not relevant beyond the lessons learned.

CARLsim

Due to the memory and computation needed to iteratively process the broad collection of dynamics and updates of the neural state, large-scale spiking neural network (SNN) simulations are challenging to implement. CARLsim 4 is a user-friendly SNN library written in C++ that can simulate large biologically detailed neural networks to meet these challenges. CARLsim 4 is the next iterations improving the previous version, which can now use multiple GPUs and multiple CPU cores concurrently in a heterogeneous computing cluster. [11]

The work shows results demonstrate the simulation of 8.6 million neurons and 0.48 billion synapses.

When using 4 GPUs, up to a 60x speedup over a single-threaded CPU implementation is observed. In addition, the latest release introduces new features, such as leaky-integrate-and-fire (LIF), nine parameter Izhikevich, multi-compartment neuron models, and incorporation of Runge-Kutta fourth-order. Hodkin Huxley is not supported. CARLsim does not support electrical synapses but does chemical synapses are supported.

CARLsim maximizes the degree of parallelizing with two approaches. Firstly the distinction between organizing computations according to neuronal activity and secondly to synoptical activity. A hybrid of these two reports the best results showing the best load balancing and the least warp divergence.

SpiNNaker

Spiking Neural Network Architecture (SpiNNaker [30]) is a massively parallel, multi-core supercomputer architecture developed at the University of Manchester. It consists of 57,600 processing nodes, each with 18 ARM9 processors (specifically ARM968) and 128 MB of mobile DDR SDRAM, totaling 1,036,800 cores and over.

It supports building neural networks in the PyNN description language. Three principle axioms of parallel computing are discarded: Synchronicity, memory coherence, and communication determinism. It, however, is still able to perform meaningful computations. Communication happens by simple spike mechanisms no larger than packets of 72 bits. The computing platform is accessible through the Human Brain Project [32] and the source code available on Github.

Summarizing

This section has highlighted a subset of available neuron simulators. The focus on selecting these simulators was their similarities to this thesis subject. In Table 3.1 an overview of this subset is presented. What has become very clear is that there is already a good collection of neural simulators available for neuroscientists to use. This thesis work focuses on the eHH subset of neuron models supported by all except HRLSim, CARLsim, and GeNN3. SpiNNaker is the only simulator that does not support GPUs but is included in this review because of its approach to coping with distributed computing. This thesis problem statement: "How can we efficiently simulate neural models, with growing neuron counts keeping the solution as high performing as possible, utilizing multi-node GPU compute platforms" raises the questions if one of these platforms could be suitable to answer the formulated research questions, with or without alternations to the software. However, as the objective already stated, this work will implement a new versatile, multi-GPU eHH simulator, which will be heavily based upon flexHH and GenEHH. These two implementations show massive potential in their respective subset of neuron models they support. With this new multi-GPU simulator, this thesis will show that simulating the entire Inferior Olive complex of a human could come into scope for the neuroscientist. None of the simulators mentions can achieve this simply because they are way more general than a dedicated eHH simulator.

Simulator	Neuron models						Frontend				Characteristics				
	LIF	IZH	HH	Multi Compartmental	Current-based Synapses	Conductance-based Synapses	Python	C \C++	PyNN	NeuroML	Single GPU Support	Multi GPU Support	Distributed	Neuromorphic	Open Source
Arbor	x		x	x	x	x	x	x			x	x	x		x
NEST	x	x	x		x	x	x		x	x			x		x
CoreNeuron	x	x	x	x	x	x			x	x	x	x	x		x
Brian	x	x	x	x	x	x	x	x	x	x	x	x			x
GeNN 3	x	x	x		x		x	x			x				x
CARLsim 4	x	x		x				x	x		x	x			x
SpiNNaker	x	x	x		x				x					x	x
flexHH/GenEHH			x	x		x				*					

Table 3.1: Overview of the characteristics of the review subset of neural simulators that are still active
* Indicates using a interfaced which is NeuroMLcompliant

4 Design

The main focus of this chapter is on the design, of our simulator mgpuHH, choices for which first the requirements of the design are discussed. As the thesis title suggest, the project focuses on GPU-powered platforms to do the calculation, but the same scalability principles hold for a CPU-only implementation. Several design cycles are done, firstly to get acquainted with the CUDA syntax and project and secondly to test certain ideas. Explanations will mainly be on the final design.

4.1. Design Considerations

The ideas presented in flexHH [26] and the CPU revision GenEHH [31] are selected as a starting point for this work. These works were created and maintained by the Neuro Computing Lab (NCL) at the Erasmus MC (EMC). This thesis project is part of the NCL, and collaborators on the mentioned work are still lab members. This work is considered the next iteration, exploring the possibilities of utilizing GPU and multi-node computing systems. The choice to follow the same neuron support as flexHH and GenEHH does not cover all possible neuron modeling features. For example, chemical synapses and geometrically accurate neurons are not in the scope of this simulator. Electrical synapses (gap-junctions) are more challenging compute-wise due to their direct nature, meaning that in each simulation step, all the communication required for gap-junctions needs to be performed. Chemical synapse do not have this requirement, and therefore their implementation needs to be designed differently, which is left out of scope. Geometrical data about neuron models is left out of scope because the focus is large network simulation and researching scalability; therefore, such features on cell level are not considered.

4.1.1. Compute Challenges in Neurocomputing

A few key challenges in the world of computational neuroscience came out of discussions with computational neuroscientists:

1. Dense gap-junction connectivity in a large-scale network
2. Numeric instabilities in small active compartments
3. Large parameter spaces for network composition
4. Very long simulations (hundreds of seconds of biological time)
5. Stochastic input sources and seed management
6. Local Field Potential simulations of the cerebellar network
7. Simulate an entire olivo-cerebellar complex.

This work will tackle mainly problems 1,3,4 and 7. One example of a stochastic input source is included in the design. Seed management is done through a user interface, ensuring experiments can be recreated. However, the design will allow for easy addition of input sources, with the side note that programming know-how is needed. Combining these statements with the research questions presented in the introduction chapter gives rise to the following presented requirements.

4.1.2. Requirements

An essential aspect of every design is meeting the requirements. Because this is a research project, there is a certain amount of freedom to explore the design space as no full-fledged application is expected. However, to be recognized as a usable neural simulator, a set of requirements needs to be met. The essential requirement is that the simulator is accurate, and users can trust the results. An overview of the requirments is given in Table 4.1.

Feature	Requirements	Design Specifics
Interface	Designed for LINUX-based systems	Source code only, user will have to build the executable.
	Configuration-file controllable	JSON-based, building further upon. the work of GenEHH
	CLI (non interactable) interface	
Scalability	Multi-Node support	OpenMPI will be used for message passing between computes nodes.
	Multi-GPU support	Intra-Node multiGPU support without OpenMPI dependencies.
GPU Support	CUDA	Limits to NVIDIA GPUs only.
Output	Metadata output file	Meta data results on performance and memory usages, intended to profile the design. Example in Appendix C.
	File-based network-state output	Output in either raw or ASCII data to give the user insights into their simulations.

Table 4.1: Overview of the design requirements

Interface

The programming skills of neuroscientists might be limited. Therefore, it is required to be 'user-friendly' enabeling neuroscientists to utalize the simulator. The program will achieve this by using configuration files in JSON format. The JSON format is lightweight and designed for storing and transporting data that is widely supported and can be used with Python and Matlab. An example configuration file is included and annotated in Appendix B. The simulator will be a command-line interface (CLI) executable that is designed to run on Linux systems. The focus lies on a scalable solution, and almost all big compute nodes/clusters are Linux powered.

Neural model support In the related work, most of the implementations supports heterogeneous models. This work should support fully heterogeneous networks. Homogeneous networks are having the same constructions and parameter settings for all cells in the network. Heterogeneous networks can have different constructions and parameter settings for all cells in the network. However, most experiments consist of significant clusters of the same cell structures. The neuroscientist wants to control specific simulation parameters with some randomization instead of creating massive heterogeneous networks with only some randomization at the parameter level. This work will support handling those randomizations, making it easier to use the JSON files helping relax memory-size needs. The performance implications introduced because of this will be discussed in the Implementation chapter.

The parameters that are getting randomization support are the "Channel LeakConductivity" and the "Channel InversionPotential." These two are selected because there was a need for these parameters in collaboration with the neuroscientist. The implementation can be easily altered to support a different and/or bigger subset of parameters. However, this will potentially need more memory resources and can slow down the simulations. On the other hand, when all parameters need randomization, one can still explicitly describe every neuron in the system, maintaining full flexibility.

Gap-junction network generation To properly test the simulator, some gap-junction network count needs to be supplied or generated. This work will have two generative algorithms built-in, RandomBinary (A uniform-distribution generation) and a 1D-Gaussian. It will also support reading the interconnection graph from a file. RandomBinary and 1D-Gaussian are selected because of either the lack of locality or strong locality in the generated connections, which either stresses communication and cashing more in the case of no locality or relieves stress in the case of strong locality. The normal mode of operation is considered to be reading from a connection file because neuroscientists like to have precise control over the interconnection network they deploy.

Scalability

Scalability of compute resources is critical to answer the research questions of this thesis. Primarily, two sorts of scalability are taken into consideration. Firstly, intra-node scalability. Secondly, inter-node scalability. Intra node scalability is the possibility of using all GPU resources available to a single node. The amount of GPUs a node can carry depends on the amount of PCI-e lanes the processor has available. In extreme cases, setups with 19 GPUs are out there at the time of this writing, reducing the amount PCI-e connection lanes from x16 to a lower count and therefore reducing the communication bandwith. The focus will be on systems with two up to eight GPUs because those systems are available to use and are sufficient to showcase intra node scalability. Inter-node scalability means utilizing multiple nodes within a compute cluster. Many nodes are clustered together through some networking interface. The simulator will distribute processes and data using these network interfaces between nodes. MPI, the Message Passing Interface, will be the standard here. OpenMPI is chosen to be the implementation of MPI that this work will use. It is an opensource and free-to-use library that interacts through an API, reducing complexity by not managing all network interfaces manually. [18]

GPU Support

This work focuses on utilizing GPU(s) and comparing results against prior CPU and DFE implementations. The created program will only run on platforms with CUDA support in the intended design form. However, some effort to incorporate the CPU design from [31] into the final executable to run some smaller-scale experiments with the same code base is made. That design does not support multi-node implementations, and for this reason incorporating multi-node CPU support stays out of scope.

Output

The most crucial requirement for the neuroscientist is the network's state at specific simulation time steps. Four values can be of importance: The compartment potentials, the calcium concentration within a compartment, the channel currents, and the gate-variable states. The JSON input configuration will decide what needs to be written into an output file. These output files can either be raw binary data or human-readable ASCII data. Printing out values for all cells or a specific cell needs to be supported. The network's state must be possible to be written to file for every timestep in the simulator but also be reduced to every N'th step to spare disk space and potentially increase performance. The simulator will not be interactive through callback functions, or command-line interface (CLI) commands. However, already generated output data can be looked into before the simulator finishes the full simulation run. This choice is made because there are not any added benefits to this thesis and it only increases complexity.

4.2. Analysis

In Figure 4.1, the extensibility which the neural description utalized in the simulator possesses, is presented structurally. It is important to note that the designed simulator cannot control what a neuroscientist would like to simulate. It could be a single cell with millions of compartments consisting of only a single channel and a single gate. Nevertheless, it could also be millions of cells with a single compartment, a single channel, and one-hundred gates for this channel. Whether these configurations make any biological sense is not up to this simulator to determine. The simulator should support every degree of freedom while keeping the solutions scalable and functionally correct.

Supported Network Structurs The network structures this work supports are presented in Figure 4.2. A few concessions are made that were also made in flexHH and GenEHH. Firstly, the externally applied current and the gap-junction current can only act on the first compartment described in a cell description. Secondly, the compartments are ordered in a chain following the order of the cell description. These concessions simplify the design but still creates a design that can answer many research questions and meet requirements.

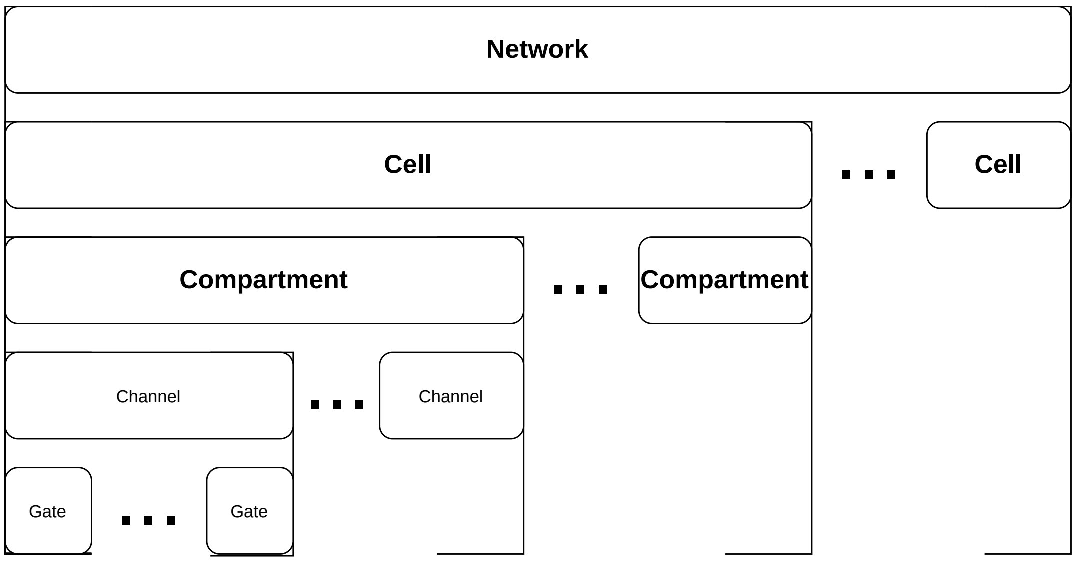


Figure 4.1: An overview of the neural network structure. A network can contain multiple cells, which can contain multiple compartments etc. The dotted lines represent the extensibility of the object.

By including support for gap-junctions to act on the different compartments, converts the gap-junction problem form a cell-dependency to a compartment-dependency problem. The same holds for the applied current and the chain linking of compartments. The chain linking of compartments is a somewhat harder problem to provide a generalized solution for. Support to save the cell structure would need to be built into the cell description. The formulation of flexHH's interaction-current Equation (3.2) does support arbitrary compartment ordering. However, the IO model does not utilize this, and also the flexHH implementation does support arbitrary compartment ordering. GenEHH does only support chain-style compartment ordering, and this work will follow those design choices.

Gap-junction currents are dependent on a connectivity list. This connectivity list is supplied by the neuroscientist as an input file. Alternatively, the parameters for a generation algorithm are provided by the neuroscientist. This work will support three options: An input file, a RandomBinary (uniformly distributed) generation, and a RandomGaussian generation. The weight that each gap-junction connection has can be uniform or connection-specific.

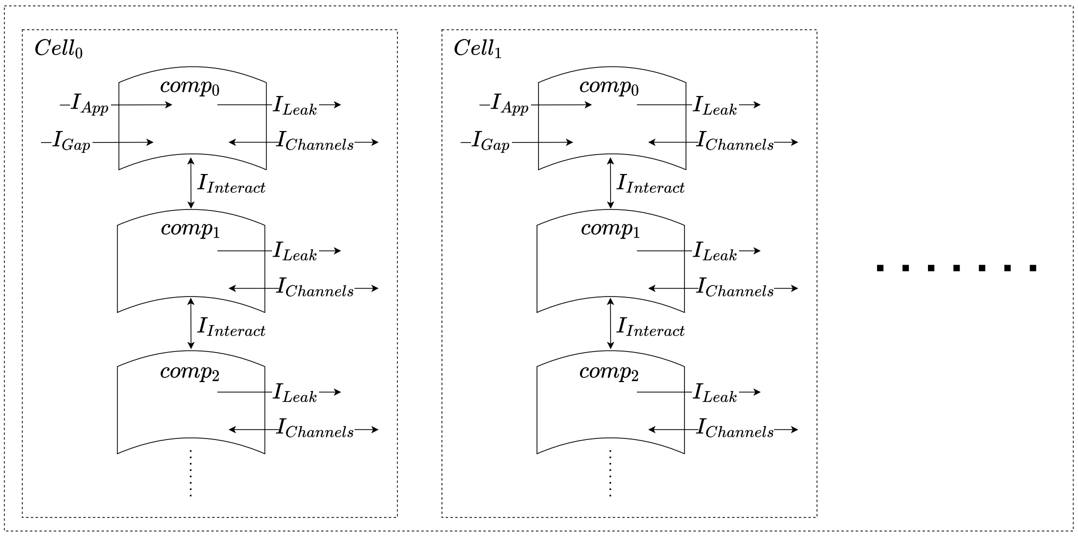


Figure 4.2: An overview of the neural network style that is supported in this work. Two neurons are shown, each consisting of three compartments. I_{app} and I_{gap} can only act on the first compartment in the chain like compartment list each neuron has. The dotted lines represent the growing freedom of the neural network.

4.2.1. Simulation Flow Breakdown

flexHH and GenEHH are designed in a way that one function can handle the complete simulation. This function takes a compartment and updates it. This function is then executed as often as the number of instantiated compartments for each timestep in the simulation. The timesteps required are depending on which numerical solver is used but will stay sequential depending on each iteration, because there is a relation to the previous state of the network for each case. Pseudocode to run a simulation can be simplified to Algorithm 1. This idea that a single function can handle the complete simulation and is within a fat loop that can be executed in parallel indicates the potential of GPU usage.

Algorithm 1 Run Simulations

```
1: for $0 \leq i < N_{steps}$ do
2:     for $0 \leq k < N_{comps}$ do
3:         $UpdateComp()$
4:     end for
5: end for
```

Compartment update A carefully tailored implementation is of the utmost importance to meet requirements and still get high-performing solutions. Analysis of the work that the targeted compute platform needs to perform is crucial here. A dataflow overview can be made when breaking down the implementations of flexHH and GenEHH and altering them to best suit a parallel implementation. See Figure 4.3. It is important to note that it is a numerical solution on a ODE system, which means repetition is needed. The critical path depends on the configuration of the model. An important note is that the gap connection and externally applied currents are only present in the first compartment of each cell, adding additional logic to check if the current compartment is indeed the first one or not. The alteration in design is that the compartment now gets the option to add a dedicated calcium concentration, instead of the need to model it as a gate, which flexHH and GenEHH are doing. This choice makes the gate updates independent of each other and generalizes the support for calcium concentrations.

As can be seen in the overview in Figure 4.3, the rest of the formulas show resemblance to Equations (3.1) – (3.10), where Equations (3.8) – (3.10) are combined in the Y_{update} function. At gate level, instantaneous gate support is added, which is not present in the formulations of flexHH but is implemented in flexHH and GenEHH in the same manner.

Leak Current - The leak current is straightforward and only depends on the current compartment potential and two compartment-specific variables.

Interaction Current - The supported network structures limit the cell structure to have chained-style compartment connections resulting in a maximum of two neighboring compartments. An improvement could be modifying the design to support three structures that can be tailored by the user. This work will not go deeper into that direction, resulting in a reasonably easy way to calculate all interaction currents.

Gap-junction Current - The most challenging of all because it needs the number of compartment potentials as it has connections, resulting in many memory accesses that tend to be very chaotic and uncoallesed.

Externally Applied Current - The externally applied current is time-dependent and therefore takes the current simulation step as input. When it is enabled, the amplitude provided will add to the compartment potential update. The way the amplitude gets calculated does not matter for the design, given that it is a formula that depends on the compartment or cellID, the compartment potential, and the time variables. Constant values or random noise sources can be added. What will be supported to showcase this the versatile functionality are the following types of applied current: DcCurrentStepandHold, DcCurrentRampandHold, DcCurrentPulseModulo, and OrnsteinUhlenbeck noise.

Channel Current - The channel-current calculation is the most complex one. There are still 2 degrees of expandability: The number of channels and gates of a channel can differ across cells making this an

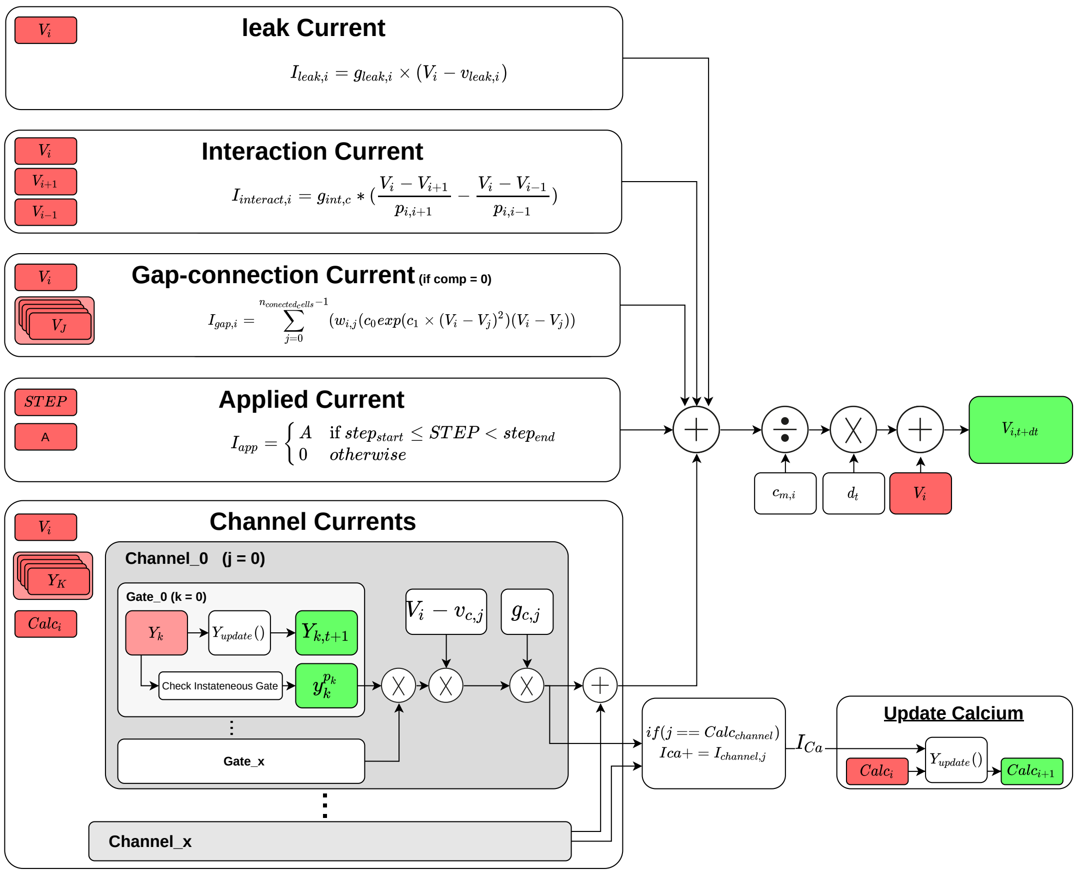

leak Current
$I_{leak,i} = g_{leak,i} \times (V_i - v_{leak,i})$
Interaction Current
$I_{interact,i} = g_{int,c} * (\frac{V_i - V_{i+1}}{p_{i,i+1}} - \frac{V_i - V_{i-1}}{p_{i,i-1}})$
Gap-connection Current (if comp = 0)
$I_{gap,i} = \sum_{j=0}^{n_{conected_cells}-1} (w_{i,j}(c_0 exp(c_1 \times (V_i - V_j)^2)(V_i - V_j))$
Applied Current
$I_{app} = \begin{cases} A & \text{if } step_{start} \leq STEP < step_{end} \\ 0 & otherwise \end{cases}$
Channel Currents
Channel_0 (j = 0)
Gate_0 (k = 0)
Check Instateneous Gate
Gate_x
Channel_x
$if(j == Calc_{channel})$ $Ica+ = I_{channel,j}$
I_{Ca}
Update Calcium
$V_{i,t+dt}$

Figure 4.3: This overview shows everything that needs to be computed for a single solver step of the simulation for one compartment. All upper-case variables are timestep-dependent. The lower-case variables are constants that are defined in the configuration file. The color red represents inputs, while the color green represents the output of a specific step in the simulation. Dotted lines represent the ability to add multiple channels and/or gates to a compartment.

unpredictable summation for the single-channel currents and multiplication for the gate's contribution to a single-channel current. Y_{update}, which is an collection of equations (3.10), (3.8) and (3.9), is not dependent on other factors within the compartment-update phase. One function to handle all possible gate descriptions is designed.

Calcium Concentration - The calcium concentration is unique for each compartment and can be influenced by one or multiple channels in the implementation presented in this work. It is necessary to be able to model the calcium update formulas as a flexHH gate. However, it is compartment-specific and dependeds on one or more channel currents ($I_{channel}$). Therefore, it is made part of the compartment description, resulting in the fact that all gates stay independent of each other but still can be influenced by the compartment's calcium concentration.

4.2.2. Parallelizable Parts

Memory dependencies heavily bound parallelization. Therefore, it is immediately evident that the solver steps, regardless of the type or method, are sequential because of the dependencies on previous calculated values. Updating the neurons, on the other hand, is not sequentially depend on other neurons during a single timestep. Parallelization at cell level or compartment level gives the same chucks of calculations each parallel pipe needs to perform. Compartment-wise parallelization would be preferable because it splits up the calculations in more parallel pipes and will not lose performance when compartments counts differ across the network. Compartments with gap-junctions will have more computational load that compartments without gap junctions. This imbalance could be tackled by grouping all compartments with gap junctions together, eliminating warp divirgence, or by taking the gap-junction calculations out of the compartment update and create a new kernel for this task.

If this parallelism is applied on a GPU platform, we would use a CUDA thread per compartment and launch a kernel to update a compartment. This would limit the freedom of further parallelization of the function. For example, the gap-junction calculations could benefit from having multiple threads available to calculate the gap-junction current to parallelize the accumulation of single connections further. Also, the gate updates can benefit from having a thread per gate available. Because of the diversity of calculation and accumulations, it would not make sense to create kernels for each contributing current in the compartment update. However, it could make sense, performance-wise, to split up the $UpdateComp()$ function into a separate gap calculation function, gate update function, and compartment update kernel. The different options will be implemented, and the implementation chapter will report the results.

Setting up the network pertains mostly allocating and initializing the memory correctly based upon the configuration file. However, the gap-junction network generation is something that can be done with a GPU kernel where, for example, one thread for each cell is launched. This works for the RandomBinary and Gaussian distribution incorperated this codebase. However, when a neuroscientist designs a connection list, she can link a file in her configuration, and the program will read out the connection list from the hard disk. The CPU will do this and then copy it to the on board memory of the selected GPU. GPUdirect memory could potentially improve performance. However, because most targeted systems do not have direct NVMe support, and clusters are working most of the time with network-attached storage, no further design resources are put into this feature.

4.2.3. Scalability

Scalability is of great importance to this implementation. It makes it possible to use anything from a computer at home up to high-performance compute platforms such as supercomputers from the TOP500. As described in the Background chapter, openMP and openMPI are the best-suited libraries to facilitate this requirement.

Because this work focuses heavily on GPU acceleration, it will not benefit a lot from multi-threading. However, in the case of multiple GPU's available, it is required to call API functions per GPU to utilize them. These API function calls are handled on a thread basis and therefore can be parallel. This work uses the same amount of openMP threads as GPUs available to simplify the implementation, there is no additional performance benefit expected from using multiple threads.

Another performance killing requirement is writing results into an output file. The writing is a sequential process but does not require execution to be sequential with the compute kernels. Writing the output of a timestep can therefore be made parallel with a simulation timestep. Pretty-printing floating-points numbers from raw to ASCI is a cumbersome operation. Therefore, the user is given the option to either pretty print or dump the raw data to a file, resulting in significant performance differences.

Expanding to multi-node or, in other words, a multi-process program requires communication between processes. OpenMPI is perfectly suited to handle this. Analysis of the communication needed boils down to communication of the compartment potentials when gap-junction are utalized. Everything else is known in each separate process, either by the process ID or input handling that each process does at startup. Beneficial would be to overlap communication with computing. However, communication is sequential with the compartment updates, simply because the next timestep needs the connected cells' compartment voltages. However, the gate updates do not have a sequential dependency within a time step and can potentially be overlapped with communication.

Memory-wise, it is mandatory to save all compartment potentials needed for the gap junction calculations, of every local cell connection list, in GPU-accessible memory. Spanning easily to a point where all GPUs need to have every synaptic connection potential stored inside accessible memory to perform gap-junction calculations. Accessible memory can be memory directly located on a GPU card. However, it can also be located in the Unified Virtual Addressing (UVA) space of a compute node. Using UVA saves memory on multiple GPU compute nodes but is outperformed by GPU-specific memory allocations as presented in the Implementation chapter. A tradeoff between memory and performance has to be made here. The added amount of data latency of getting the GPUs data versus getting the data in the unified memory space cannot be neglected. It is important to note that, before CUDA compute capability 6.0, it was impossible to have multiple devices (GPUs / CPUs) access the same memory locations in UVA, making UVA not suitable for older cards with the application under design.

The first described compartments in the neural description are the only compartments capable of having gap-junctions connected to them. Therefore, when in this work we talk about sharing the gap-junction potentials, we refer to sharing the first compartment's potentials only. The gap-junction potentials are the only values that need to be shared in the system. To communicate between MPI processes, two implementations are possible. Firstly, communicate everything to everyone. Secondly, communicate the necessary data to specific nodes only. Experimentation will give insight into the performance of different configurations. The more local the connections get, performance is expected to benefit from the second implementation because gap-junctions spanning MPI processes are reduced.

4.2.4. Memory Usage

It is important to understand how memory usage relates to the modeling experiment a neuroscientist wants to explore. All modeling constructs discussed so far have has linear memory scaling, except for the synaptic-connection lists: In the worst-case case scenario, it scales at a cubic rate. Each cell needs to allocate memory to save a connection to all other cells in the network. The synaptic-connection lists can either be saved sparsely or densely. Densely saving will always have the same size, $N_{cells} * N_{cells} * sizeof(bool)[bytes]$. Saving sparse connection lists will have a great benefit memory-wise when the network has a density under $sizeof(bool)/sizeof(uint)$ (normally under 0.25). This is derived from the memory space needed for densely saving the network count and sparsely saving it, which takes up $N_{cells} * N_{cells} * Density * sizeof(bool)[bytes]$ of memory when taking the density as the average connections each node has.

However, the problem lies in randomly generating networks. One can never be sure that that generated list will not overshoot some smaller allocation than the network size. CUDA does not natively support something like std::vector pushback that we know from the C++ language (that works as desired at the time of writing). However, the goal is to simulate massive networks in terms of the number of cells without quickly running out of memory. Therefore, sparse will be the only valid option. Furthermore, some decisions on connection-list growth must be made to ensure memory safety but still ensures a near to perfect generation of the connection list obeying the set of requirements the generation takes as an input. The details are discussed in Section 4.3.3.

4.3. Design Overview

The analysis broken down to a functional-level description for a particular system configuration is graphically shown in Figure 4.4. The network setup consists of initializing GPU and CPU memory and optionally generating the synaptic-connectivity list. Memory management must ensure data correctness because the system works with separate kernels to do gate updates and compartment updates. Therefore, it requires a double-buffering system to ensure that the previous-timestep-calculated compartment potentials are not updated before they are read by an operation that is still dependent on these values. Employing two memory allocations, where one holds the current values and one that holds the updated values, this is ensured. An extra benefit is that we can also use the first buffer, with the current values, to copy concurrently back to system memory if required. Therefore, each state of the network, compartment potentials and calcium concentrations, channel currents, and gate-activation states are double-buffered. Because of the double buffering, the GPU-to-CPU data transfers can be done entirely concurrently with the compute kernels. Also, as discussed in the analysis, the most promising and advanced simulation would be to do the calculations in three separated kernels. This implementation is shown in the design and will be justified later in this chapter. Using three separate kernels makes the gate update completely independent from the gap and compartment kernels, which are still holding sequential relations.

4.3.1. User IO

For the user input of the design there is chosen for a JSON input file. This decision has been made because some of the groundwork was already done, in GenEHH, for this system, and JSON is a widely accepted data-format making it highly usable for any user. Being able to simulate the full eHH class of neurons with the system that a user can come up with is one requirement. This is possible due to the fact that anything in the eHH class of neurons can be described with the flexHH formulations.

For described cells, one can multiply the description they made for a single cell by adding a multiplier entry to the cell description. However, it would not be representative of biological systems to be exact replicas of each other. Therefore, a randomization factor can be added to each channel's conductivity and reversal potential; these randomizations were chosen by expert end-users within the NCL and are

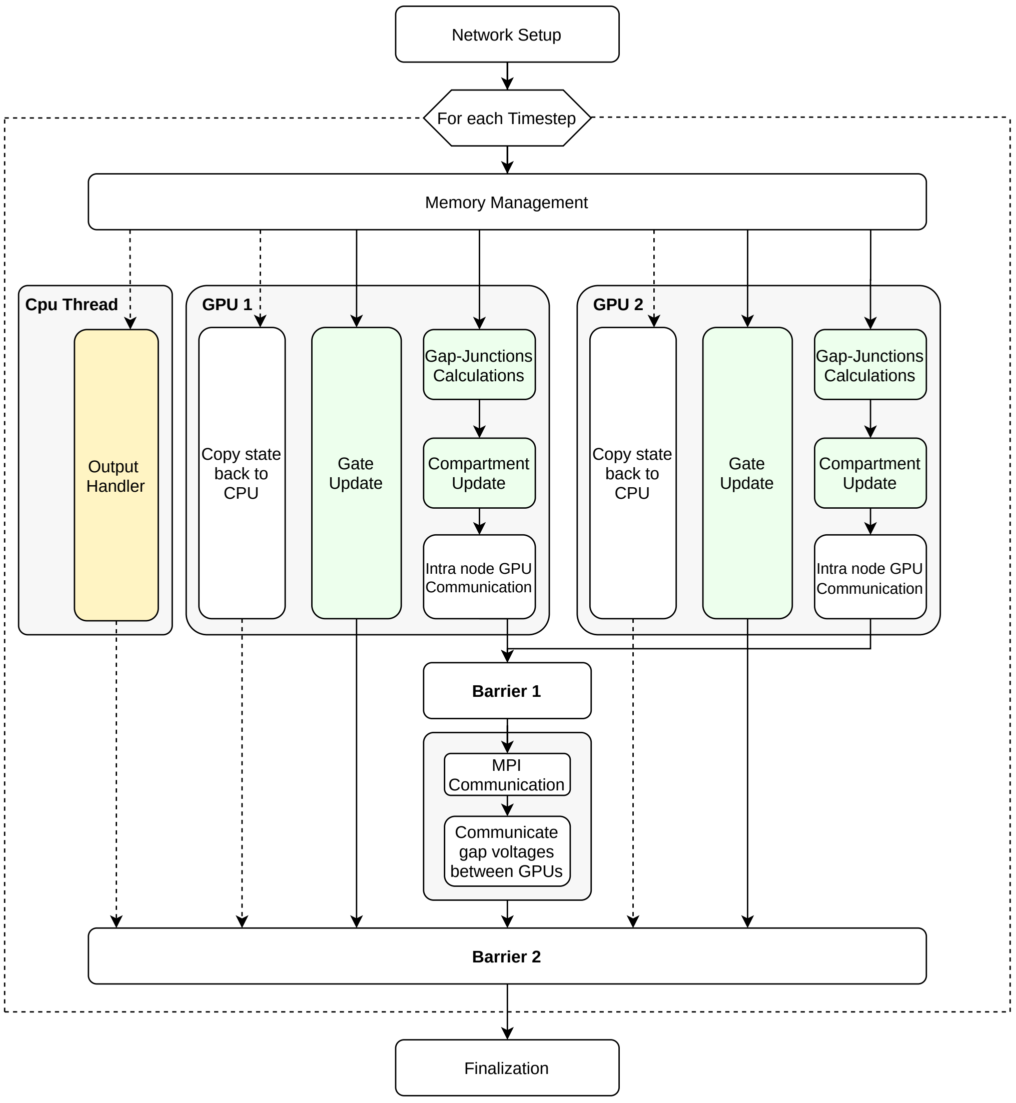


Figure 4.4: A functional overview of the system where green-colored blocks define CUDA kernels, and yellow-colored blocks define CPU tasks. All white blocks are data management and control flow specific. The overview in 4.3 is captured in the Compatment update with the execption for the gap-junction calculations and gate update which both have a dedicated kernel represented by ”Gap-Junction Calculations” and ”Gate Update” in this overview. This functional overview is for a single process with two GPUs connected to one compute node. The MPI communication happens inter-node to a system running a similar process, when multiple compute node are utalized. In the case of a single GPU, the ”Inter-node GPU communication” can be seen as a NULL function. The same holds for the ”MPI communication” in the case of single-process execution.

sufficient for a meaningful simulation run. When the user wishes to describe each cell with different configurations and geometries explicitly, this is also possible by setting the multiply entry to one. A full description of using the input file will be included in the repository, which is managed by the NCL and not opensource. A example of such a configuration file is included in Appendix B.

The output can be adjusted to the user’s requirements. The compartment potentials and calcium concentrations, channel currents, and gate-activation states are available to the user on each timestep in formatted ASCII format written to a text file. However, float-to-ASCII conversions are very costly in terms of execution time. Therefore, one can also choose to opt for raw binary outputs for the selected variable. The user can set a timestep interval between each output generation. Higher performance can be achieved, especially with non-local storage locations; writing the output to a file can delay simulation runs.

4.3.2. Scalability

The scalability requirement is met utilizing OpenMPI intra-node. OpenMP is used to manage GPUs connected to a node. Mapping one-to-one different threads to separate GPUs (via OpenMP) does not offer a performance benefit compared (for instance) to launching all GPU calls from a single thread. However, it allows for elegant programming style and higher maintainability of the code. The implementation autodetects the GPU number each node has, and is able to utalize them all or a selected subset. The program does not guarantee the fastest run time when combining different GPU types or ,having multiple GPU, available versus a single GPU implementation. This was not inside the scope of this thesis and would be great future work on the project. An overview of how the design's scalability is graphically shown in Figure 4.5. Combining this overview with the functional overview in Figure 4.4 should give the reader insight into how the overall simulator design works.

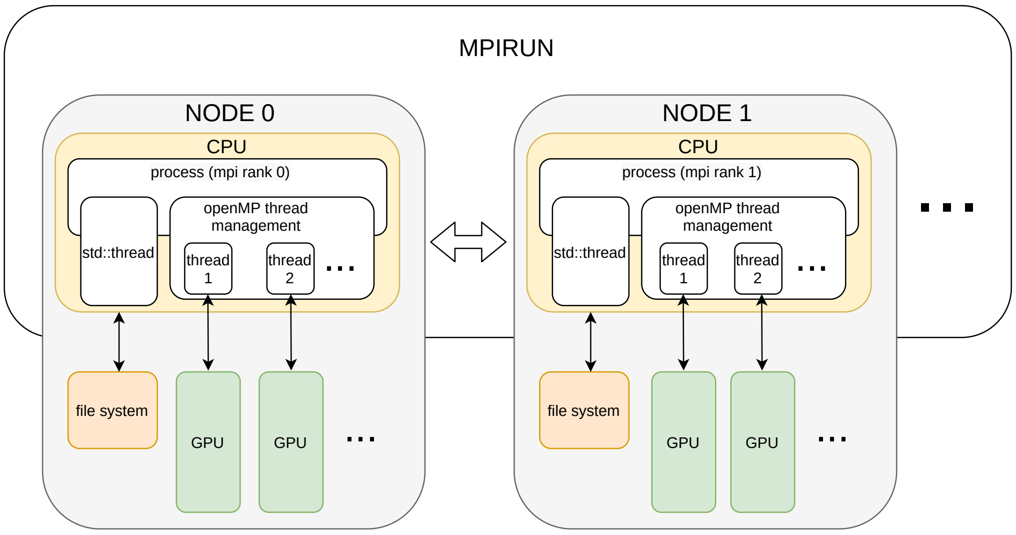


Figure 4.5: Architecture-level diagram of the design. It illustrates the scalability of the design where the dotted lines represent the ability to scale out. MPI manages the processes where each node receives a process to run. Within this process, openMP-thread management takes care of the communication with the available GPU cards in the system. A separate CPU thread takes care of writing the state of the simulator to an output file.

MPI: Allgather versus Alltoallv Both communication schemes will be supported and usable via a flag in the configuration file. `Allgather` will bluntly share all data across all processes where `Alltoallv` has a tailored sharing approach where only the necessary data will be sent over to the process that requires the data. Sharing all data with everyone adds more communication overhead. `Alltoallv` adds more computational complexity because of ordering and collecting the correct data before sending it over. It will be investigated in chapter 6 whether both implementations will have the same scaling properties and whether the synaptic-connection-list type of generation (denser or sparser or more localized) does influence the preferred choice in term of performance.

4.3.3. Kernel Breakdown

Designing CUDA kernels is the most challenging part of the design phase. Programming a CUDA kernel is quite delicate and somewhat complex to estimate its performance. However, the CUDA C++ programming guide gives a starting point [1]. Nvidia delivers a toolchain within CUDA that is very helpful for validating applications under design. The available profilers are heavily used to determine performance and/or validate design choices (Visual Profiler and Nsight Compute). An overview of all CUDA kernels implemented in our mgpuHH simulator is shown in Figure 4.4. Next, we will go over the details of each one of these kernels.

Gate and Compartment Update The gate- and compartment-updates kernels are a straightforward design following the analysis. The gate-update kernel has the same amount of threads as there are gates in the neurons. Each thread is responsible for updating its gate with the $Y_{update}()$ function. The compartment update kernel does the same but then on a compartment level, implementing the overview of Figure 4.3 except for the gate updates and gap-junction connection current calculation. The gap junction connection current will be a direct input provided by the gap-junction calculation kernel. Updating the Y_k value to $Yk, t+1$, which is handled by the gate update kernel.

Gap-junction Calculations For the gap-junction calculation, a different approach is taken. For each cell, a certain adjustable amount of threads (N) will be called. All these N threads will work together to fetch from memory and accumulate to a shared result: The gap-junction current of that specific cell. The accumulations rely on warp-level-primitives, limiting the maximum amount of threads to 32 per cell (one warp). The system could potentially benefit from even more threads by utilizing shared memory for the additions making to possible to use a full CUDA thread block; this was tested briefly and did not offer any improvements. Because the focus is on big network experiments, it is estimated that the GPU(s) will be 100% utilized in any case, and dividing up further would reduce instead of increasing performance because more threads require to be managed in terms of communication and memory. Therefore, it was not further taken into consideration for the design. The optimal N threads will be explored in the implementation chapter. Also, pre-buffering through shared memory will be explored in the implementation chapter. It is not expected to affect smaller network densities. Still, it can potentially benefit more dense gap-junction networks, where a larger number of cells need to access the same memory locations. The overall design is graphically presented in Figure 4.6.

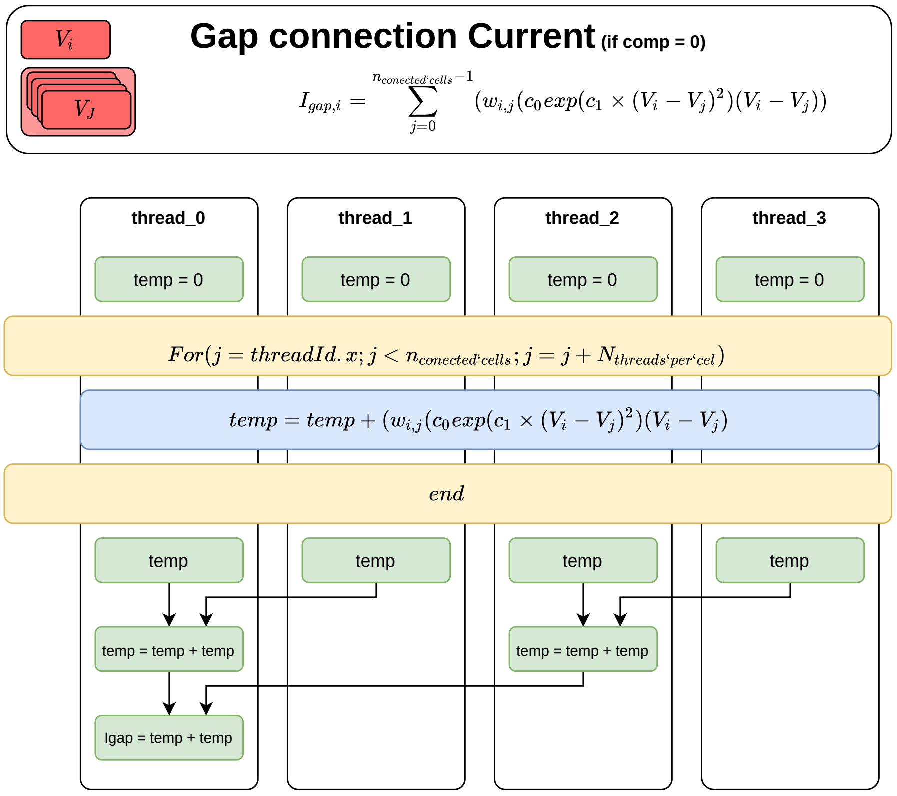

V_i
V_J
Gap connection Current (if comp = 0)
$I_{gap,i} = \sum_{j=0}^{n_{conected`cells}-1} (w_{i,j}(c_0 exp(c_1 \times (V_i - V_j)^2)(V_i - V_j))$
thread_0
thread_1
thread_2
thread_3
temp = 0
$For(j = threadId.x; j < n_{conected`cells}; j = j + N_{threads`per`cel})$
$temp = temp + (w_{i,j}(c_0 exp(c_1 \times (V_i - V_j)^2)(V_i - V_j)$
end
temp
temp = temp + temp
Igap = temp + temp

Figure 4.6: A dataflow-style overview of the gap-junction kernel design. The colour yellow represents a for-loop where it iterates the function represented by the colour blue. Each thread will execute this function and then accumulate the temporary results together resulting in the value for the gap-junction current of a single cell in the network.

Connection-graph generation The generation of the connection graph is important for performance experiments of the network. Most end-users will probably provide their custom-tailored connections-lists that suit their specific needs. Therefore, the generation is optimized for GPU execution but not with high optimization priority. Two schemes are implemented natively: RandomBinar (shown in Algorithm 2) and 1D-Gaussian (shown in Algorithm 3). The Randombinary generation is considered to be the most challenging interconnection network because it is completely random. The 1D-Gaussian generation has many localities in the connection list, which will benefit computation and communication performance. A 2D and 3D Gaussian is also created in the same fashion but will not be further discussed in the report because they do not add value to the research questions and only complicate the understanding of locallity and memory / neuron ID tracking. The networks used are all ranked in a 1D list ranging from 0 to the maximum number in the network. This could be easily changed to a 2D ordering or even 3D ordering but does add complexity, which will make the performance results dependable on more parameters, which is not in the scope of this work. It is estimated that it will not change performance scalability.

Algorithm 2 RandomBinary Generation (Density,Networksize)

```
1: TargetCell = threadID
2: for $0 \leq NeighborCell < Networksize$ do
3:     if $Rand(0,1) \leq Density/Networksize$ then
4:         -> Add NeighborCell to TargetCell
5:     end if
6: end for
```

Algorithm 3 1D-Gaussian generation (mean,variance,Networksize,Target Connection count)

```
1: TargetCell = threadID
2: xd = 1 / (variance * sqrt(2 * PI))
3: yd = -(1 / (2 * pow(variance, 2)))
4: for $0 \leq Dinstance < Networksize$ do
5:     checkcell = TargetCell - Dintance
6:     if checkcell < Networksize then
7:         probability = xd * exp(pow(distance - mean, 2) * yd)
8:         if $Rand(0,1) \leq probability$ then
9:             -> Add NeighborCell to connection graph
10:        end if
11:    end if
12:    checkcell = Targetcell + Dintance
13:    if Targetcell > Dintance then
14:        probability = xd * exp(pow(distance - mean, 2) * yd)
15:        if $Rand(0,1) \leq probability$ then
16:            -> Add NeighborCell to connection graph
17:        end if
18:    end if
19:    if Connection count ≥ Target Connection count then
20:        Break()
21:    end if
22: end for
```

4.4. Added Value to Research

The design of mgpuHH, as captured in this chapter, should contribute to the IO-model race described in the Background chapter and potentially set the new standard for Large-network experiments. The codebase is not hardcoded but utilizes the methodologies from flexHH and GenEHH, making it a universal eHH simulator. The design will hopefully be very didactic for current neuron simulators that are being developed in the field of multi-node and GPU support. This design should give insights into the challenges involved when creating neural simulators with gap-junction style interconnections. It is the first codebase that can utilize multi-node and/or GPU systems that have the flexibility of experimenting with all sorts of conductance-based models through a JSON based configuration where the codebase does not need to be recompiled. A benefit of this codebase's usability and due to close relations with neuroscientists is that this design will be used to run experiments that were earlier out of reach without (and outside) the NCL. The ability to easily deploy mgpuHH on supercomputers brings much potential, and the first human-sized IO model can be simulated with hardware resources that are available for this thesis.

5 Implementation

With the proposed design in Chapter 4, there is still a lot of room for different implementations. These options will need to obey the design but need to best-suite the hardware configuration. This chapter will explore these different paths and validate the design choices.

5.1. Development Resources

For the development of the design, a range of platforms is selected. Important when selecting compute platforms is firstly availability secondly whether the platform suits the requirements for the evaluation. For example, profilers from Nvidia are powerful but disabled on most larger-scale platforms. Platforms that we can control the software installed on them are better-suited for this. An overview of all available hardware to this thesis is presented in Table 6.2.

EMC resources The Neurocomputing Lab of the Erasmus MC has a range of development options available, one server in particular suits our needs. This server is named ”ComputeDev” and carries a V100 GPU and a dual-socket server CPU (AMD EPYC 7551) with 128 threads. Having full control over this machine enables us to use `nvprof`, `nsight compute` and `cudamemcheck`, which makes this machine a good choice for the development of single-GPU code. Also, it is an ideal platform to run CPU versus GPU benchmarks to show the potential GPUs are bringing to the table.

TU Delft development resources The TU Delft has a CUDA development server, which is part of the Q&CE IT infrastructure. The server carries 2 GPUs, a K40 and a RTX2080Ti. Permissions to use profiling tools are enabled and therefore suitable for developing multi-GPU code. They also have a cluster available within the same resource pool comprising four nodes with 2 K80 GPUs. Unfortunately, MPI is not supported, which makes it harder to use its full potential. Optionally, OpenMPI can be built from source code, and the cluster could still be used in the multi-node configuration. This, however, was not worth the time, and the platform is only used for single-node dual GPU development. The somewhat older GPUs, unified memory addressing is available but does not support simultaneously reading from UVA memory from different devices.

TU delft HPC cluster The HPC cluster has the same problem as the TU delft development resources. It does not support MPI out of the box. Because of lacking dedicated interconnection between nodes, it is not considered a multi-node option. However, some nodes within this cluster are packing eight GTX1080ti GPUs, a potential platform for benchmarking the simulator’s multi-GPU scalability.

Google Cloud Google Cloud resources are falling into a different category of computing units. They offer Cloud-computing resources that are adaptable to the user’s wishes at any time. However, this project is limited to a maximum of resources. Specifically, the number of GPUs available is limiting the possibilities. For example, the project can only use one V100 GPU but four Tesla T4 GPUs at the same time. Therefore, this platform is selected for multi-node development. Because it is a Cloud-computing-based solution, the user has full administrator rights over the virtual machine instances they manage making it tedious work to set up, but possible, to tailor the correct environment to run multi-node experiments. Due to hardware resource limitations, configurations of four nodes with one GPU each or two nodes with two GPUs each, or one node with four GPUs are possible. The rest is out of scope for this platform. However, this is sufficient to verify that the codebase is correctly functioning in multi-node environments.

CSCS The Swiss National Supercomputing Centre Offers a wide range of high-end supercomputers that were out of scope for this project. What was, however, in scope was their development cluster named ”Ault”. This development cluster enabled us to finalize implementation experimentation and draw a conclusion about which configuration is working best for this design. Ault consits of four Nodes with four V100 GPUs per node.

5.1.1. Software-Specific Considerations

CUDA support is required, preferably CUDA 5.0 or higher, to support GPUdirect technologies. None of the discussed systems is running anything below CUDA 10.1 out of the box, which means no measures need to be taken for the correct functioning of the implementation. For inter-node scaling, OpenMPI can be built either CUDA-aware or not CUDA-aware, meaning one can pass GPU buffers directly to an openMPI call. Therefore, in any case, it is beneficial to have a CUDA-aware openMPI library installed. However, this is not common in most installations. Consequently, the design will have to support a non CUDA-aware openMPI library as the end-user is not expected to built these libraries from source code. Performance for CUDA aware libraries will be higher for most configurations.

5.1.2. Hardware-Specific Considerations

The implementation should support different hardware configurations to show that even with home compute units simulating complex brain models becomes accessible. However, the main goal lies in scalability. With powerful systems, some different tricks become available to the user. Intra-node, the most significant introduced delay will be the sharing of the gap-junction potentials between GPUs. This most straightforward approach will be to use UVA memory allocations. However, this is not expected to be achieving the best performance. Best performance is expected to come from direct peer2peer memory copies when GPUdirect is available or otherwise use regular CPU staged memory copies when GPUdirect is not available. NVlink, when available, will make all direct peer access faster, and therefore, UVA and peer2peer memory copies will benefit from having this bus available. However, on all described platforms earlier, this bus is not present, so testing this hypothesis will be out of scope.

Intra-node, the networking interfaces, and the way openMPI is built are important. OpenMPI is performance bound by network latency and bandwidth. When GPUdirect is available on a system, CUDA-aware MPI can reach its full potential utilizing tricks as RDMA to increase data-transfers' performance. Performance is hard to estimate upfront, but in HPC computing systems, latency between nodes utilizing QDR InfiniBand can be as low as a few microseconds. Any implementation of MPI can detect and use systems to their full potentials if everything is set up correctly. Any network interconnection can be used from 100 Mbits up to QDR Infiniband, reaching 100 Gbits bandwidth speeds. The latency, which is also an essential factor to consider, is harder to estimate upfront and will depend on the networking topologies of the system in use.

All this hints that the more hardware available, the better it is. But this is most definitely not the case. There will always be a tradeoff between the extra added compute power versus the communication overhead. These tradeoffs will be explored and discussed. This work leaves combining differnt GPU types or computes nodes out of scope it thus considers only homogeneous HPC resources; otherwise, the design space exploration would explode. MgpuHH is built to assume homogeneous resources but will run correctly on heterogeneous configurations. However, it will most not be a performance balanced implemention in the case of heterogeneous configurations.

5.2. Memory Requirements

The memory requirements of the design can be calculated when the network configuration is known. A *tool* named mgpuHH memory-estimation tool (mgpuHH MET) is built to do this with any given JSON network configuration used as input for the simulator. This tools is implementing Equations(5.1) - (5.6) to determine the memory needs per simulation. Where *unique* stands for a unique description. For example, the IO model has three unique compartments, nine unique channels, and twelve unique gates. The global identifier stands for the total number of neurons described in the network. To get the *local* GPU identifier, the global counterparts need to be divided by the number of GPUs present in the system. The numerical multipliers attached in the equations represent respective byte-sizes of the various memory structures used in the source code of the simulator.

$$MemUsage(bytes) = MemUsage_{config} + MemUsage_{States} + MemUsage_{Connectionlists} \tag{5.1}$$

$$\begin{aligned} MemUsage_{config} =& nCompartments_{unique} * 84 + \\ & nChannels_{unique} * 13 + \\ & nGates_{unique} * 124 \end{aligned} \tag{5.2}$$

$$\begin{aligned} MemUsage_{States} =& nCells_{global} * 8 + \\ & nCells_{localGPU} * 20 + \\ & nCompartments_{localGPU} * 32 + \\ & nChannels_{localGPU} * 8 + \\ & nGates_{localGPU} * 16 \end{aligned} \tag{5.3}$$

$$MemUsage_{Connectionlists}$$

$$RandomBinary = 8 * nCells_{localGPU} * 2 * density * nCells_{global} \tag{5.4}$$

$$1D - Gaussian = 8 * nCells_{localGPU} * 2 * density * nCells_{global} \tag{5.5}$$

$$Read = 8 * nCells_{localGPU} * 1 * MaxDensity * nCells_{global} \tag{5.6}$$

5.3. CUDA Kernel Breakdown

As discussed in the parallelization analysis in Section 4.2.2, it could be beneficial to spread a single simulation step out over multiple kernels, adding communication overhead between the kernels in the case of data dependencies. This section will explore the different options when parallelizing the system shown in Figure 4.3 and will highlight how the kernels described in Chapter 4 are implemented.

One kernel fits all When taking a look at the tasks that need to be done to update a single compartment potential presented in Figure 4.3, it is quite clear how to convert this to a GPU kernel. Launching this kernel with grid dimensions that match the total number of compartments in the simulator completes a single simulation step. As seen in Figure 4.3, where all upper-case variables are compartment-specific and updated every step and lower-case variables are, in fact, constants related to the description of the network. All variables need to be looked up in memory, probably leading to a memory bottleneck. Therefore, this kernel could potentially be better-performing than splitting the kernels out because all variables are only pulled into one kernel and stay in cache or registers. However, because only one kernel is launched, concurrent ke execution is not available, resulting in less parallelization. This may not be a problem as long the GPU is fully utilized.

Compartment-potential update The same as ”One kernel fits all” excluding the gap-junction calculations and gate updates.

Activation variable update (gate update) Updating the activation variable is shown to be completely independent of updating the compartment potentials. This kernel can therfore, if hardware allows, be computed concurrently with the other kernels. It can also be parallelized on gate-level.

Compartment and gate update For comparison, a kernel is designed that does both compartment and gate updates. The kernel is similar too the ”One kernel fits all” kernel without the gap-junction calculations. Potentially this is beneficial because all cell constants are loaded into the same kernel instead of two separate kernels. This kernel is parallelized at the compartment level.

Table 5.1: Overview of the kernels for internal experimentation to make implementation decisions

Kernel	GJ-calculations	Compartment-updates	Gate-updates
One kernel fits all	X	X	X
Compartment- and gate-update		X	X
Compartment-update			X
Gate-update		X	

Evaluation To evaluate the different described update kernels, two networks of the IO model are deployed. The GJ-connectivity-density range is chosen to be multiples of 10, starting at 0.01% up to 100% or full memory utilization. All kernel evaluations are done on the ComputeDev server utilizing a V100 GPU, forcing all kernels to be executed sequentially for fair comparison because concurrent kernels can influence each other performance. An overview of the kernels used for internal experimentation for the implementation process is found in Table 5.1. Results are displayed in Figure 5.1. Where it can be noted that The compartment- and gate-update performance does not get influenced by the GJ-connectivity-density. This comes from the fact that the only part in Figure 4.3 that depends on the GJ-density are the GJ-calculations. This dependency is present in the one kernel fits all bars. The GJ-calculations and how these can be further optimized as a stand-alone kernel will be discussed in the next subsection. The Compartment and gate update combined is outperformed by the two separate kernels to conduct these tasks. This is due to the fact a higher parallelization is achieved by splitting up the task in the respective separate kernels. How this scales with growing network sizes is non proportional, e.g., 2x more neurons results in more than 2x the simulation time. As can be concluded when comparing the subfigures, this scalability will be discussed in the Evaluation chapter.

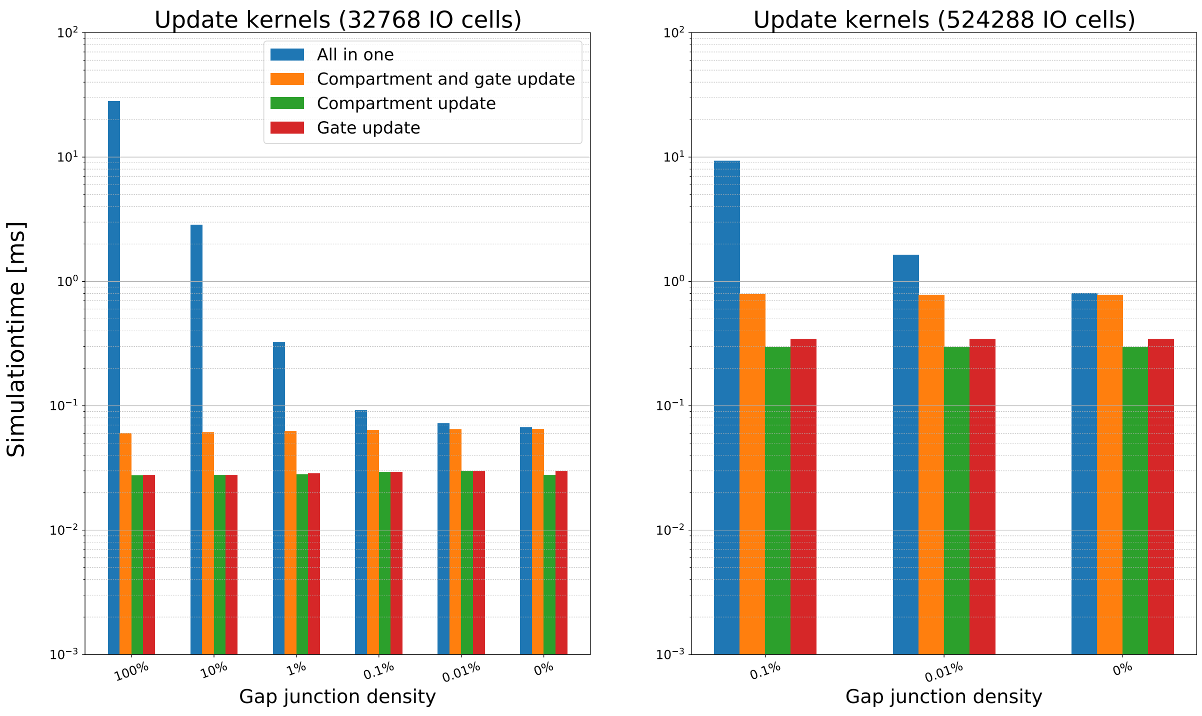


Figure 5.1: A visual representation of different kernel approaches to update the network for a timestep. The simulation ran for 1000 Simulation steps and results presented are an accumulation of these steps. Results are generated utalizing ”ComputeDev” resources.

Gap-Junction Calculations

The gap-junction (GJ) calculation is a challenging problem performance-wise due to the random nature and amount of memory lookups. The proposed design in Figure 4.6 is giving us freedom in the number of threads (2,4,8,16,32) to allocate for each neuron. They are going to work on the same-neuron's incoming gap-junction current. The needed memory can either come directly out of device memory or be staged through a shared memory buffer that can potentially improve memory latency when multiple cells need to access the same memory location.

Evaluation Figure 5.2 and 5.3 presents results for the same parameter space and experimental setup as Figure 5.1. Note that that when the gap-junction density is 0%, the gap-junction kernel is still launched for this experiment, leading to non-zero execution times due to the launching-overhead costs shown, this is not the case for the released version of the simulator. When the group of worker threads becomes larger, this overhead increases; the same behavior is observed for shared-memory against no shared memory. Shared memory is dynamically allocated when the kernel is launched, and therefore adds more overhead. When the work done by the kernel is sufficient, it becomes clear that running without shared memory and 16 threads per neuron seems to be performing best. This is not a clear winner, but in both edge cases (a neural network with high neuron count and a dense gap-junction neural network), this configuration is favored. Shared memory buffering is most likely not impacting growing densities due to L1-cache mechanisms in place that reside in the same physically located memory as shared memory. For sparse networks, more threads seem to have a less pronounced and eventually a detrimental effect on performance. This is due to the fact that the amount of work that need to be done by the GJ-calculation kernel is low and the overhead of launching more thread is therefore a higher performance penalty the the slightly more work a thread needs to perform.

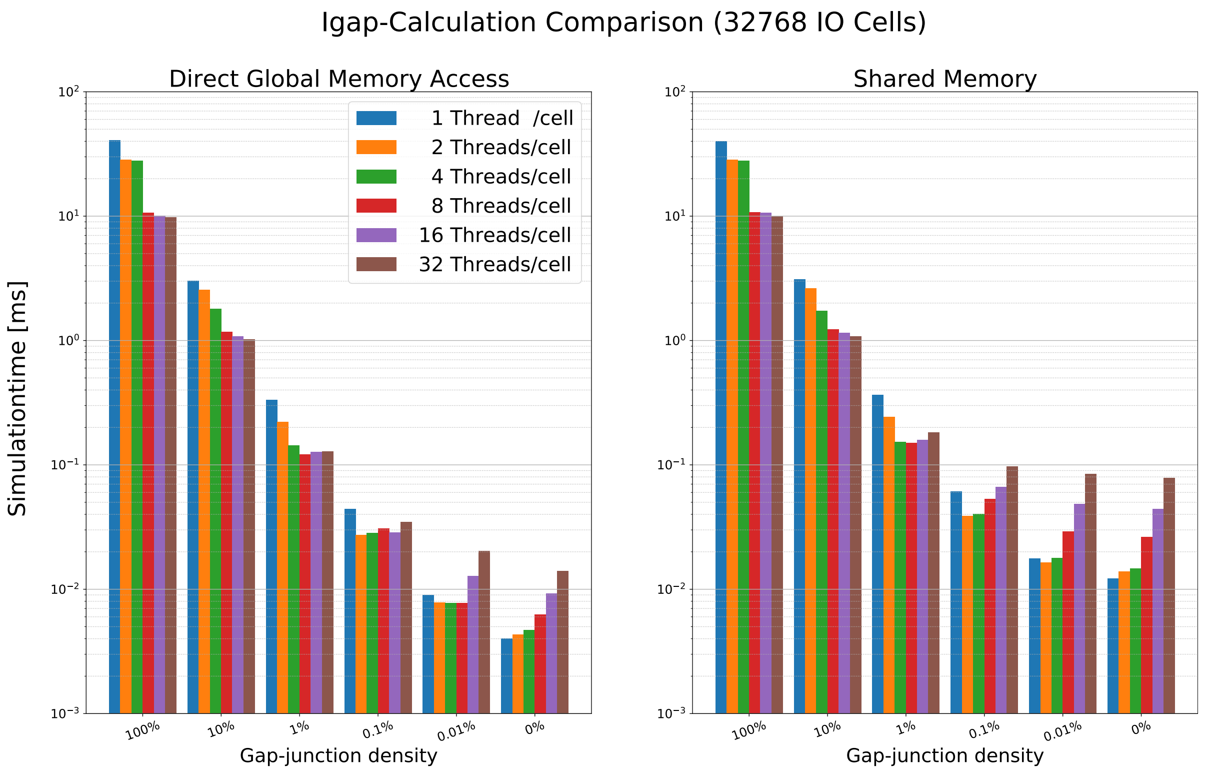


Figure 5.2: Synaptic current calculation for an IO network with 32768 cells, with different densities for a uniformly generated synaptic connection list. Results are generated utilizing "ComputeDev" resources.

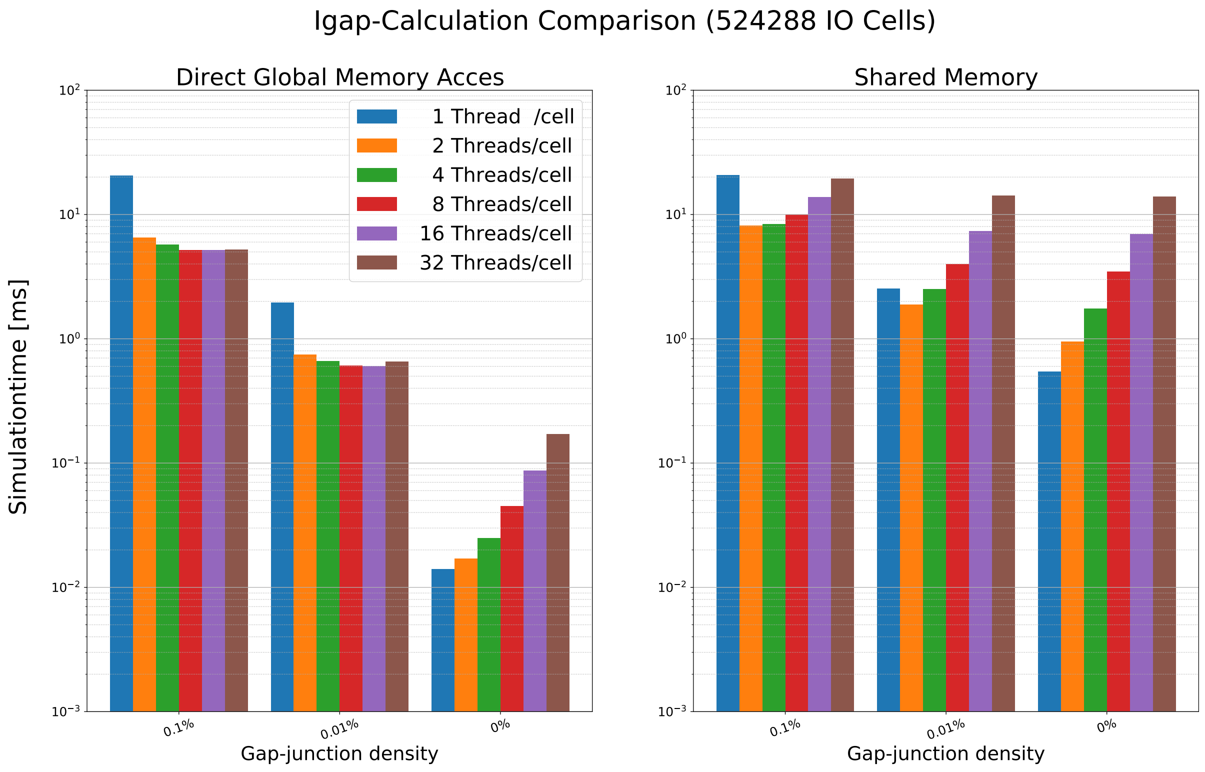


Figure 5.3: Synaptic current calculation for an IO network with 524288 cells with different densities for a uniformly generated synaptic connection list. Results are generated utilizing ”ComputeDev” resources.

5.4. Memory Arrangement to Support Warp Equality

CUDA works great when all threads within an SM are executing the same instructions and taking the same branches in the code. This comes from the fact that it can only launch one instruction to all threads residing in that SM. Therefore, it becomes clear that each block should launch the same compartment or gate, for that matter. The gap-junction calculations are not affected by this because of the already similar tasks for each tread. CUDA also heavily favors coalesced memory accesses, the network should be arranged in this fashion at a memory level.

Performance plots for different data arrangements in memory are shown in Figure 5.4. ”Ordered” is grouping compartments and gates in blocks and ”Not Ordered” is arranging them in description order.”Not Ordered”: $(cell1comp1 - cell1comp2 - cell1comp3 - cell2comp1..$ vs ”Ordered”:$cell1comp1 - cell2comp1....cell1comp3 - cell2comp3..)$.. shows consistently better results which can be explained by the fact that warp equality is implemented. However, it restrict the input configuration to have cell descriptions containing multiplication factors of the blocksize. This can be fixed by padding the internal descriptions. The effort to build this mechanism was not made. Ordering can be turned off or on at compile time.

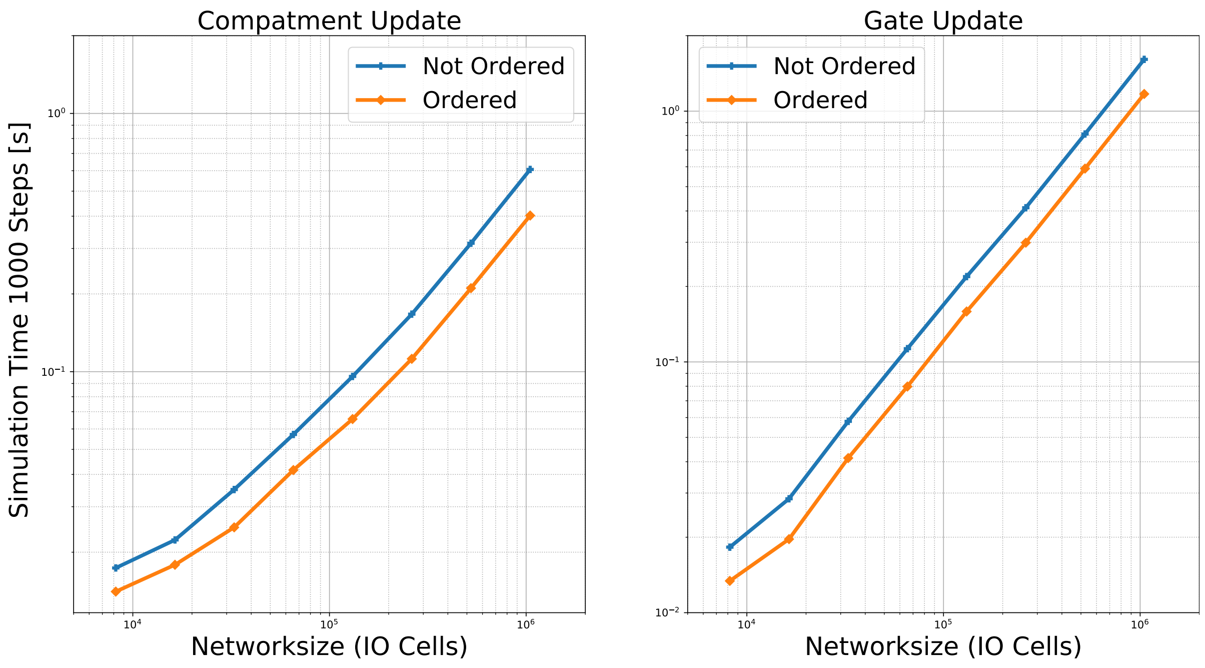


Figure 5.4: Experimental results to support ordered memory configuration to achieve warp equality for the compartment-update and gate-update kernel. Results are generated utilizing ”ComputeDev” resources.

5.5. Output Handler

Formatting the output of the simulator does decrease performance. Dumping raw data to disk versus formatting can increase the writing time with a factor of over one hunderd. Therefore, it is made optional to format the output data but not recommended to do so. Also, not every timestep is always important for a neuroscientist. Consequently, it is possible for the user to select a particular sampling interval, such as every 10th, 100th or any number tof simulation time-steps. Because a separate thread manages the writing to disk, it will only synchronize every N-th step making it concurrently with more than one simulation step. The only drawback is the need for a separate writing buffer to keep data correctness throughout the writing process. During experimentation, generating output has never been in the critical path as long as it is not requested to generate output at each time-step. The design has support to export the full network state at each timestep with 4 discrete groups: Compart Potentials, Calcium Levels, Channel Currents, Activation Variables.

5.6. Scalability

Scalability in either multi-GPU or multi-node usage depends on data movement. Compartment potentials that add to the synaptic connection from neurons located on other GPUs or even compute nodes need to be shared. Within a compute node, the memory space is shared, and data can be shared either through UVA memory or by directly copying it between GPU buffers. Between different compute nodes, this is done by utilizing MPI. However, some hardware and software configurations can influence the performance of these methods, which will be discussed next.

UVA memory There are two suitable implementations for UVA memory 2.3.1. The first one just allocates the needed space without specifying a specific GPU. The second method determines the GPU where the allocation needs to occur and matches it with the generated data. In both cases, the UVA memory space is accessible for all devices in the system, but in the latter case, it is potentially faster to update each entry because it should reside in physical memory on the GPU that is generating the data. This is a hypothesis developed during implementation and not discussed in any NVIDIA manual or literature, the results are presented in Figure 5.5. ”OldUVA” stands for the first described method and UVA for the secondly described method, p2p will be discussed in the next paragraph. The second design outperforms the first design except for the case with 65536 neurons. This is due to the small network and low density, it is expected that the added complexity, keeping track where the data is actually located and requesting the right data, the second method is not worth it in this case. It also becomes very clear that UVA is never worth it compared to the results where UVA is not enabled. These chosen neural networks should benefit from UVA memory because of the low densities involved: only a few lookups to other GPUs are happing, but still, UVA is a poor design choice. Therefore, no more investigation will be conducted in utilizing UVA memory.

Testing for high connection densities will only increase the performance difference between UVA and manual memory management. This is because no extra memory operations need to be conducted for manual memory management. The simulator already shares everything with each GPU inside the same process. However, UVA will need to conduct even more memory lookups and thus will not decrease the two options’ performance gap.

P2P access P2P access 2.3.1 can impact the durations of memory copies which is shown in Figure 5.5. It is clear that with p2p enabled, simulation times are slightly faster. The cases with UVA enabled are not taken into account because the CUDA driver most likely utilizes other systems to manage UVA, the results are included and shows that it indeed does not seem to matter if p2p is enabled or not in the case of utalizing UVA memory. However, this information is not openly available, so no conclusions can be drawn.

Internode communication Internode communication is handled by OpenMPI. OpenMPI can be either built CUDA-aware or not. This is checked and passed to the codebases at compile time. When it is built CUDA-aware, GPU buffers can be directly passed to the OpenMPI API. When this is not the case, there is a need to first copy the buffers into host memory before passing them to the OpenMPI API. However, when there is no GPUdirect support present in the interconnection medium openMPI will still copy the buffers into host memory before passing handeling them.

When OpenMPI is built CUDA-aware, it does not explicitly mean OpenMPI will use GPUdirect with or without RDMA support. A CUDA-aware build library can be utalized on any system: systems without GPUdirect or RDMA, even systems without a GPU in them. OpenMPI is following the so-called ”Law of Least Astonishment.” This means that it will automatically select the option with most performance to use for communication, the CUDA-aware built library adds the option of GPUdirect and RDMA as an option and they will be utalized when present on the target system. Because of this, it is easy to deploy on a broad range of different platforms. The same holds for Infiniband versus Ethernet. When InfiniBand is available, it will be automatically selected as the medium to communicate.

For internode communications, two styles of message passing are supported in OpenMPI, `mpiallgather` and `MPI_alltoallv`. The `MPI_alltoallv` implementation does not support GPUDirect. This stems from the fact that the send buffer needs to be created out of all the selected compartment potentials which to be send to a specific node. A CPU is better-suited for this task, resulting in a

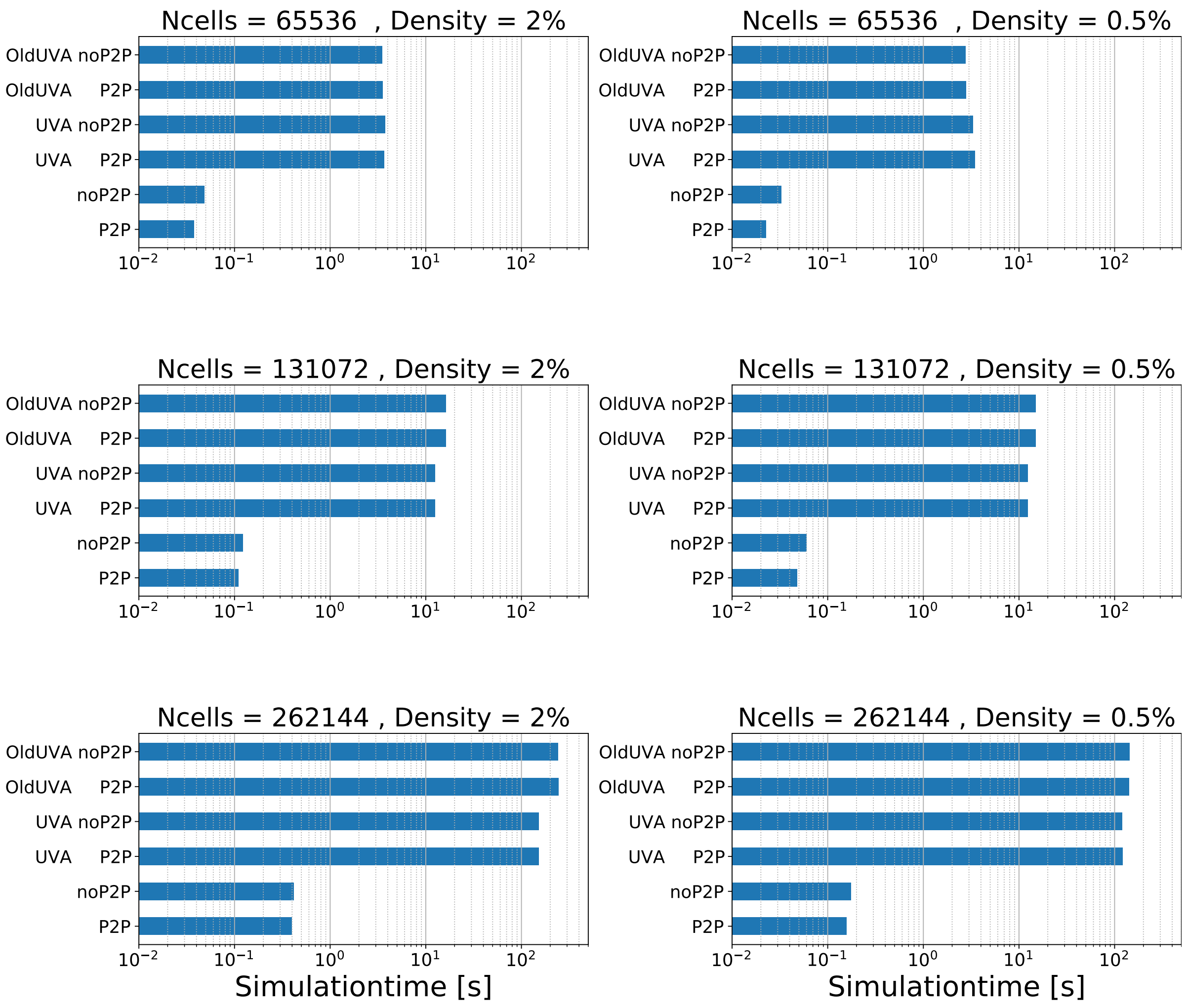


Figure 5.5: Experimental exploration of UVA and p2p access either enabled or disabled. All results are for a single-node with 4 V100 GPUs, between all GPUs p2p access is possible. Results are generated utalizing the "Ault" development cluster at the CSCS

sendbuffer that is created in host memory making GPUDirect obsoleet. This task could also be performed by a GPU kernel. However, letting the CPU perform these tasks allows for concurrency with the Gate-update kernel, balancing the compute resources better.

Synaptic Connection Graph Generation

To implement a synaptic connection graph, system-memory size is one of the biggest bottlenecks when the simulator is scaled to larger network sizes. When a dense representation is selected, the memory footprint will not vary with a requested gap-junction density. However, it will require $sizeof(bool) * netsize^2[bytes]$ in memory allocations which explodes memory usage for large network sizes. Therefore, a sparse representation is selected, which would ideally require $sizeof(bool) * netsize^2 * density[bytes]$ of memory space. This is not beneficial for growing networks, but at least for low-density configurations, it keeps the memory footprint smaller. However, the random patterns at which users are generating networks, places this ideal complexity far from real-world scenarios. In a 100-cell neural network with a density of 10%, cell #1 can have eight connections, and cell #2 can have 15 connections in a randomly generated network. This poses a big problem with memory management and potentially the hazard of overflowing buffers. When for example, each thread determines the connections of a single cell, which would be a logical approach for parallelizing the problem, the simulator needs to allocate the arrays upfront to hold the generated connections. Evenly distributing would only allow for ten connections per cell, which does not comply with the example above.

From file When importing hand-tailored connection matrices, the simulator has no idea of the pattern used to create the matrix. Therefore, it will search for the most extensive connection list a single cell has and therefore allocated that space per cell. This is not an optimal solution in terms of memory footprint, but for the scope of this project, it solves the problem of reliable memory allocation.

RandomBinary Generation RandomBinary is the most challenging connectivity-generation problem because of the variances in connections list per cell, making it prone to memory probles. Therefore the following approach is taken. Instead of allocating $sizeof(bool) * netsize^2 * density[bytes]$, double of this allocation will be taken for the connection lists. Also, instead of 1 cell per thread, the problem will take N cells per tread. This will automatically balance out the network lists sizes of the accumulation of N selected cells. N is selected to be 25, which seems reasonable but can be altered when memory problems occur. All kernels check for overflowing the memory buffer. When this happens, the simulator will throw a warning and stop the generation for that specific cell group. When all generation is done, the total amount of connections will be checked and compared against the targeted density. If these do not differ by a certain factor, the simulator will continue execution. When the two differ with more than this factor, the simulator will raise an error and terminate to maintain reliable performance data. The simulator can then be rerun the configuration with a different seed or excluded the simulation from the experimentation results. The meganism is build in to ensure correct data for this thesis.

Gaussian Generation The Gaussian connection-list generation depends on the variance mean and density of a Gaussian distribution. As described in Algorithm 3. The kernel will never create more connection than requested. It can, however, be prone to making not enough connections. Therefore this will be checked after generation because it could result in faulty experiment results. It will be handled in the same manner as described in the RandomBinary Generation.

Both kernels are written in CUDA so as to execute natively on GPUs. They have been tested to perform correctly and show performance as expected. Network-generation simulation times are reported in Figure 5.6 up to the piont where the simulator runs out of system memory. Experimentation is done on the ComputeDev server, which is equipped with one V100 GPU with 32GB of graphical memory.

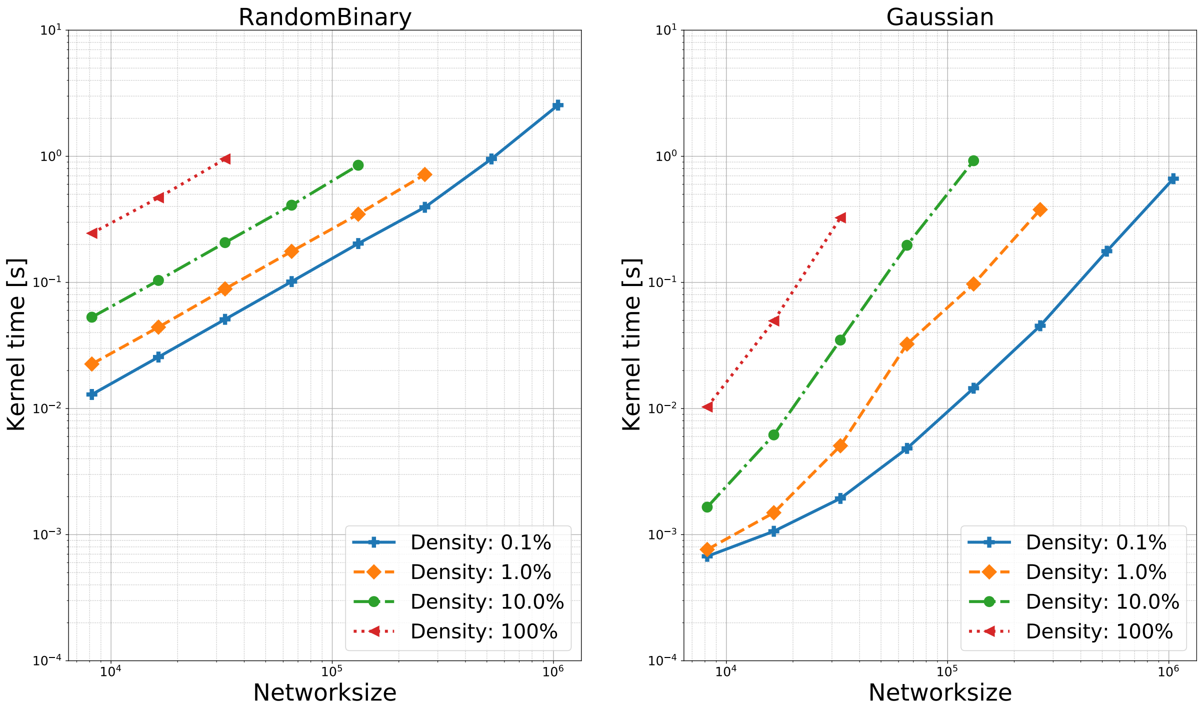


Figure 5.6: Overview of the generation kernel performances. Results are generated utilizing ”ComputeDev” resources.

6
Evaluation

This chapter will evaluate the design implemented in Chapter 5. The evaluation will cover different aspects of the program, including full-scale performance runs to validate the design scalability. Benchmarking the simulator shows that improvement on previously written simulators is made by utilizing GPU(s) and multi-node computing solutions. It also verifies the design choices and sets an example of how neuron simulators can benefit from multi-node systems and GPU utilization. A short section will be devoted to the practical case for resident neuroscientists and the experiments they will conduct with this implementation.

Single-node, single-GPU benchmarking up to multi-node, multi-GPU benchmarking will be presented to show this implementation's versatility and the potential benefits and potential drawbacks when scaling out to higher node counts. Python scripts have been employed control the exploration space's parameter sweep and is utilized to gather all benchmarking information. Every experiment with the same hardware configuration is combined into a single job, reducing job counts and avoiding short simulation runs on hard to access computing platforms. The exploration space will consist of a parameter sweep over the number of IO neurons in the network versus the gap-junctions density. These sweeps will run with different configurations such as `MPI_allgather` (sharing and collecting everything) versus `MPI_alltoallv` (sharing only what is necessary). Also, control over p2p access, having the ability to be disabled to see the effect of p2p memory copies if available. Unfortunately, a system with NVlink is not available to see if NVlink can bring even more performance benefits. Therefore, all communications on all compute platforms used in our benchmarking are utilizing the PCIe bus. UVA memory can be used or disabled. However, it was already shown in Chapter 5 that UVA memory is never beneficial for design of this type. Consequently, it not be investigated any further.

Furthermore, a roofline model will be presented for the different kernels to determine whether the design is memory- or compute-bound, giving insight into possible future improvements.

6.1. Experimental Setup

The parameter space consists of 6 parameters that will determine the exploration space as listed in Table 6.1. However, it is impossible to iterate and mix and match all possible parameter values on the same hardware platform due to hardware limitations. Therefore, various computational platforms are being used, where each one will support a subset of this parameter space and answer a different research question. Available platforms will be listed underneath and referred to when discussing specific experiments.

Performance-Evaluation Resources

Different computing platforms are selected to evaluate performance compared to the implementation chapter, mainly because more compute resources are required for scalability evaluation. However, almost all platforms mentioned do not support profiling tools due to security reasons and therefore give limited insight into the simulator's exact workings. Also, these platforms are heavily used HPC clusters, development purposes are not their main purpose. The resources at NCL will be used for the single-GPU and -CPU benchmarking. The Ault resources (described in chapter 5) are not selected,

Table 6.1: Exploration space for tthe evaluation of the mgpuHH simulator.

Parameter	Range
Problem size	0 - 10M neurons
GJ connection density	0 - 100%
GJ connection pattern	RandomBinary and Gaussian
GPU(s) per Compute Node	1 - 4 GPUs
Compute Node(s)	1 - 32 Nodes
MPI communication pattern	`Allgather` and `Alltoallv`

but results are validated against other platforms to ensure correct results across multiple computing platforms.

NVIDIA Solutions Lab cluster (NSL-A) At the Nvidia solutions lab, a cluster is available which comprises nine nodes with four V100-32GB PCIe cards and four nodes with four V100-16GB PCIe cards. Both types are equipped with FDR Infiniband interconnection (56GB/s), which makes an excellent setup for performance experiments.

Aris [29] The Greek Research and Technology Network or GRNET contains the national HPC infrastructure providing state of the art supercomputing named ARIS based in Greece. Aris infrastructure consists of Thin, Fat Phi, and GPU nodes. For this project, the GPU nodes are of interest, containing dual Tesla K40m cards and Infiniband FDR interconnection. There are 44 GPU nodes available to the user. The K40m GPUs are having a compute capability of 3.5 and Kepler architecture, which limit UVA memory to not be accessible by multiple devices at the same time. This platform will be used for scale-out experiments over up to 32 nodes.

Table 6.2: Overview of all available hardware this project has access too. Combining resources form this Chapter and Chapter 5

	Number of node	CPU-type	GPU-type	Compute Capability	GPUs Per Node	Experiment	Interconneciton
Computedev at NCL	1	AMD EPYC 7551 32-Core Processor (2 sockets,128 threads)	V100	7.0	1	- General development single GPU - CPU vs GPU - DFE vs GPU - Single GPU benchmark	-
Google	4	Skylake, 4 threads	T4	7.5	1-4	- General development multi GPU/Node - CUDA-aware vs no CUDA-aware openmpi	10GB/s
TUDelft Dev cluster	1	Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz	K80	3.7	2	Development Multi GPU	-
CSCS Ault Dev cluster	4	Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz	V100	7.0	4	Dual node full benchmark	IB 100GB/s
Nvidia Dev cluster	8	Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz	V100	7.0	4	Scalability benchmark	IB 56GB/s
ARIS at GRNET	32	Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz	K40m	3.5	2	Scalability benchmark	IB 56GB/s

Memory Evaluation

Section 5.2 mathematically described the relation between the neural-network configuration and memory usage. These formulas calculate the amount of memory used per GPU system when the neural-network is evenly distributed over all available GPUs. System memory resources required are neglectable. Even with multi-node support, it will never grow larger than:

$$(Networksize_{global} + Networksize_{localnode} * (1 + 2 * N_{Compartments} + N_{channels} + N_{Gates})) * size(float)$$

Indicating that it could become a problem when scaling with extensive networks, but inside the range stated in the experimental setup, this will not be a possibility. Furthermore, the only relevant question is how many GPUs are required to fit a particular model configuration, which is calculated through the equations in Section 5.2. As a result, Figure 6.1, to determine the amount of GPUs required for a specific neural-network configuration. This is assuming IO cells, this overview is created using the tool mgpuHH MET which takes in any model configuration and creates memory requirements for that particular configuration.

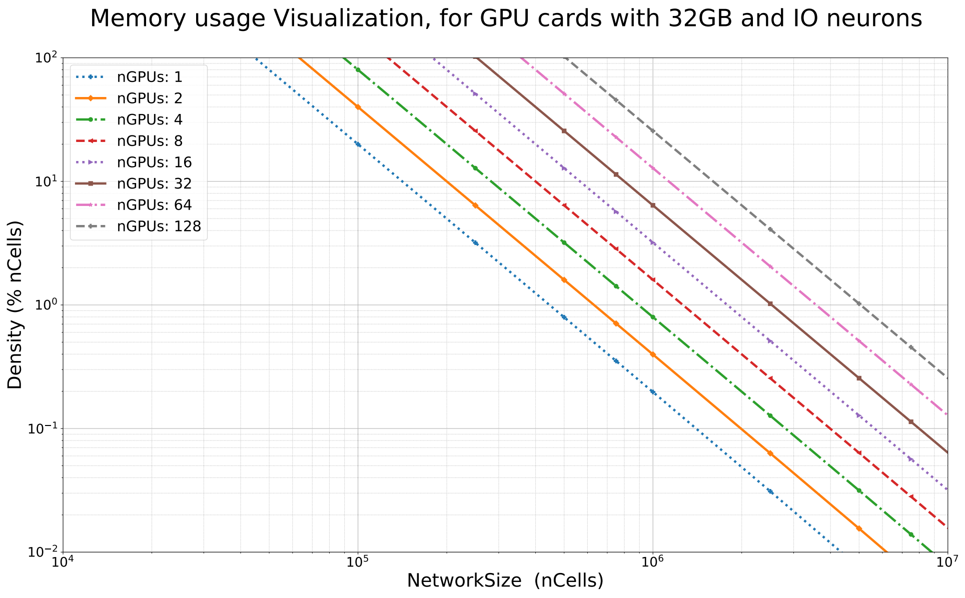


Figure 6.1: Overview of memory usage for different problem sizes of the IO model to determine the required GPUs for 32GB cards. Everything on the left of a line is possible with the amount of GPUs the line represents. On the right of the lines is not feasable due to memory resource limitations.

6.2. Roofline Model

The Roofline model is an intuitive, visual performance plot that is used to provide performance estimates of a given computer kernel or application running on multi-core or accelerator-processor architectures by demonstrating inherent hardware limitations and optimizing potential benefits and priorities. By integrating location, bandwidth, and various parallelization paradigms into a single figure.

The `Nsight Compute Profiler` from CUDA can return the `flop_counts` for each kernel for single and double precision. Combining this with the specifications of a specific type of GPU results in a Roofline model such as presented in Figure 6.2. The problem with creating a roofline model for this application is that with growing network sizes and/or different configurations, the performance per kernel changes. However, as long as the neural model stays the same, for example, the IO-model, the trends and results are very similar for different network sizes and densities. Therefore, only one neural network configuration is presented, representing the roofline model for the simulator with IO-model neural configuration, in Figure 6.2.

A conclusion that can be drawn for this specific case is that the gap-junctions calculation kernel is heavily memory-bound. The kernel is often stalled because it has to wait on memory-accesses. The compartment- and gate-update kernels still show room for improvement. Both are, in principle, memory-bounded. However, it can be observed that the device does not achieve the peak performance within this bound. This is mainly because warps are stalling to wait for dependencies. The most occurring stall is the `Stall Long Scoreboard` indicating that memory access patterns are not optimal for these kernels. The compartment-update kernel does not achieve maximum warp occupancy because of the number of registers required per thread are not available for this specific neural-model. A neural-model with fewer state variable would be possible to fit.

Possible improvements for the compartment- and gap-update kernels are better memory-access patterns by arranging the memory, used for the configuration parameters, differently or better utilizing shared memory. The gap-junction calculations kernel is bottlenecked by memory bandwidth. Efforts can be made to better hide this latency by adding more compute complexity. The gap-junction calculations are also lacking concurrency in the memory accesses, due to their random nature, limiting the maximum achievable performance. Shared memory is not the solution, as the implementation chapter showed in Section 5.3.

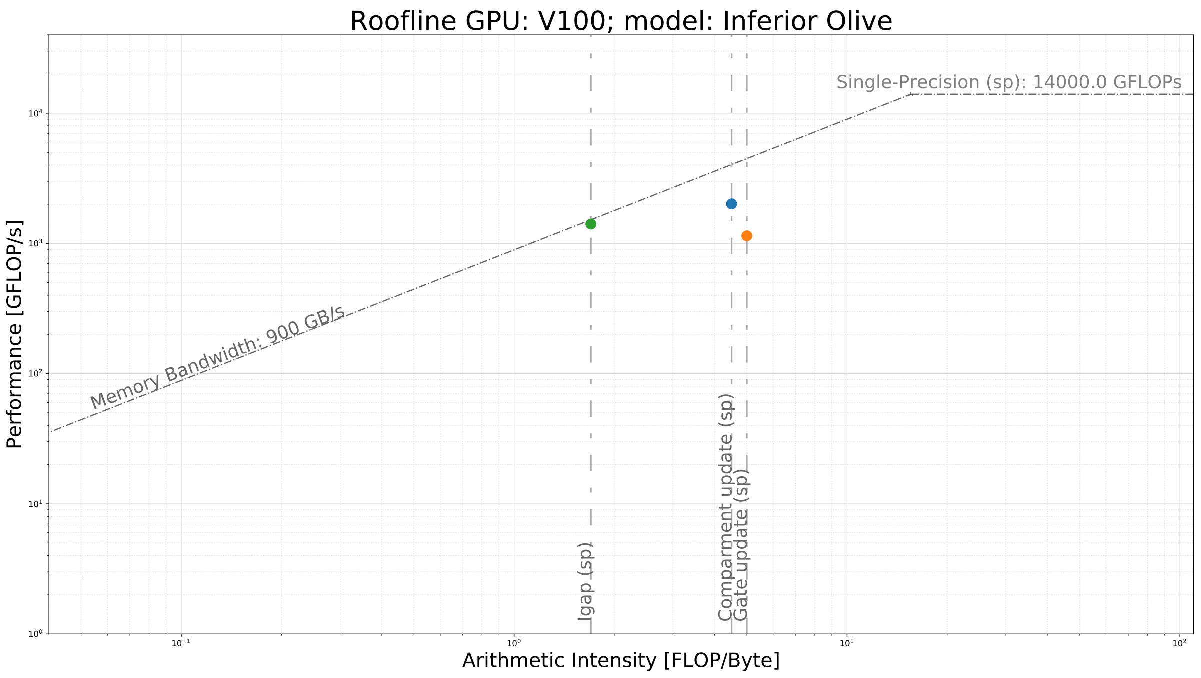


Figure 6.2: Roofline model for the mgpuHH kernels. Single precision (sp) kernels are presented in this figure. The exact configuration for the presented results is: 131072 cells, 1% Density for a RandomBinary generated synaptic-connection network and the IOmodel for each cell in the network. The dots represent the measured performance.

6.3. Functional-Flow Performance

Evaluation of the kernels is already done extensively in the implementation chapter. The focus there was on which implementation is performing better in terms of implementation options. Here, experimentation will show if the design is in balance and how different tasks scale or bottleneck the performance when scaling out the problem over multiple nodes. The NVIDIA Solutions Lab cluster (NSL-A) is selected to be the platform to run experiments. In Table 6.3 the exploration space is given.

Table 6.3: Exploration space, NSL-A functional flow Performance

Parameter	**Range**
Model	IOmodel by DeGruijl [14]
Number of simulation steps	100
Problem size	[26144]
GJ connection density	[0,0.01,0.1,1]
GJ connection pattern	RandomBinary
GPU(s) per compute node	[4]
Compute node(s)	[1,2,4,8]
MPI communication pattern	`Allgather`

Figure 6.3 shows the simulation time, split up in the respective tasks, forming the critical path, which links to the functional-flow diagram in Figure 4.4. When observing the results, it becomes clear that every time we double the number of nodes, halves the gap calculation, compartment update kernel and local memory movements execution times, represented by `stall barrier 1`. The Kernel launch times are becomming shorter when the local network sizes are going down for higher node count, the dimensions of grids are launched decreases and seems to affect the time it takes to launch these kernels. Which can be noted in Figure 6.3. MPI communication times are growing by adding more nodes because of the management overhead it adds for each additional node. However, those overheads are not scaling by the same factor as adding nodes to the system, making it possible to get better performance when scaling up to multiple nodes. The gate updates, represented by `stall barrier 2`, are almost entirely executed concurrently to the other tasks and do not add anything to the critical path, concluding that the design is well balanced for this specific network configuration.

It is important to note that with increasing GJ densities, the GJ calculation kernel also increases, as shown in Figure 5.1. This contributes to the overall simulation time and to `stall barrier 1` in Figure 6.3. These calculations are distributed over the respective nodes in the configuration, and therefore better scaling is achieved for higher densities.

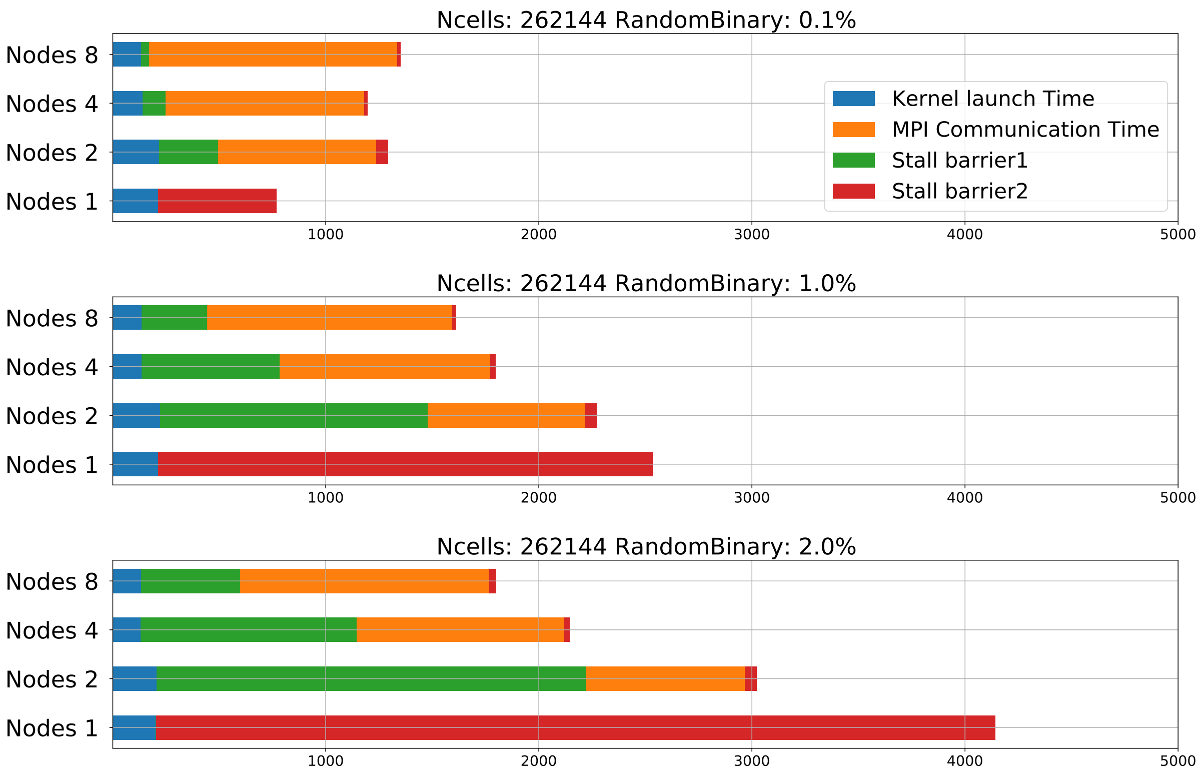


Figure 6.3: Functianal-flow performance which can be linked to Figure 4.4. Nodes x indicated the amount of Nodes and each configuration has 4 GPUs per node configured

6.4. Performance Scalability

The main goal of the project is to show the scalability potential in neural simulators. The design was build in a way to automatically adapt to the hardware configuration it is deployed on. Two compute platforms are selected to run the parameter sweep. The NVIDIA Solutions Lab cluster (NSL-A) and Aris are selected for scalability measurements.

NSL-A

The way this experiment is setup is that a particular parameter sweep is executed for each possible hardware configuration. Not all parameters can be found in the presented results. When the configurations are not possible, results are left out. Two experiments are selected. The first one, too, determines the scalability of multiple GPUs on a single node presented in Figure 6.4. The second one is determining the scalability over multiple nodes with one GPU per node shown in Figure 6.5. For the Synaptic connection pattern, RandomBinary is selected, stressing the scalability because it has no locality across the network. For the MPI communication, it is set to share everything across all nodes, taking away a variable that can influence the results. The exploration space is found in Table 6.4

The results show that scalability performance looks very promising with multiple GPUs on a single node and multiple nodes with a single GPU. What can be determined is that the implementation seems the scale very well for larger growing network sizes. Figure 6.4 Shows that adding more GPUs starts making sense from a certain network size. This point is determined by the utilization of the GPUs in use. The bigger the workload (growing density and growing neural network size), the faster it makes sense to utilize multiple GPUs. This is explained by the fact that when one GPU is not fully utilized, it does not make sense to use more than one GPU. In the form of adding more compute nodes, scalability

Table 6.4: Exploration space, NSL-A scalability experiment

Parameter	Range
Model	IOmodel by DeGruijl [14]
Number of simulation steps	1000
Problem size	[8192,16384,32768,65536,131072,262144,524288,1048576]
GJ connection density	[0,0.01,0.1,1]
GJ connection pattern	RandomBinary
GPU(s) per compute node	[1,2,4]
Compute node(s)	[1,2,4,8]
MPI communication pattern	`MPI_allgather`

shows that the communication overhead can be hidden if a single node has enough computing tasks to hide the overhead. As soon as this point is reached, the implementation is showing the same trend. The location of this point is following the same reasoning as the GPU scalability experiment. More detailed experiments on the communication overhead will be presented later in this chapter. In terms of scalability, the design shows reliable scalability results, up to eight nodes, as long as the GPUs are not underutilized.

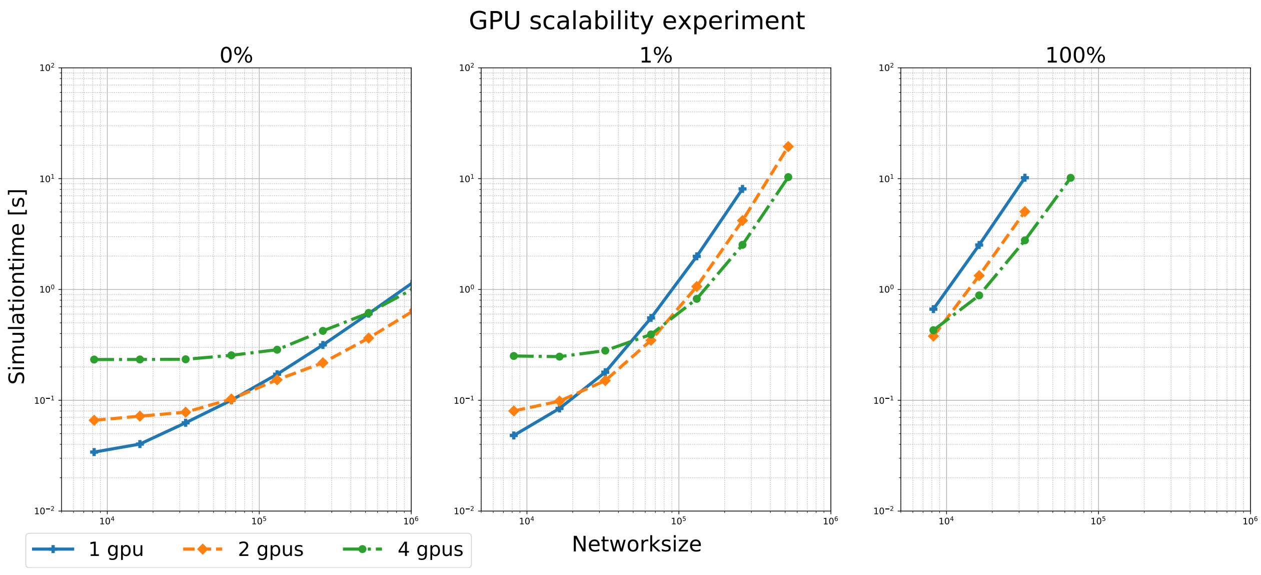


Figure 6.4: Experimental results to determine the impact of adding more gpus to one node. One node is selected and results are reported for 1,2 and 4 Gpus enabled. Compute platform selected is NSL-A

Aris

Results from running benchmarks on Aris are not comparable with NSL-A performance because of different GPU architectures. However, Aris offers more than 32 GPU nodes, making it possible to see if the scalability found in previous results is still showing the same behaviors when expanding the number of nodes. Results are visually presented in Figure 6.6 and the exploration space is found in Table 6.5.

The experiment shows the same behavior as the previous experiment with some unclear imperfections for smaller network sizes, which can be due to nonoptimal GPU utilization. Furthermore, the Codebase performs as expected and communication can still be overlapped with computation, creating very predictable results up to 32 Nodes with a total of 64 GPUs. Adding more GPUs per node would not influence the intra node communication overhead were adding more nodes will affect this overhead. Moreover, it is expected to introduce the same ”balance” point, as observed in the previous experiment when exploring higher node counts.

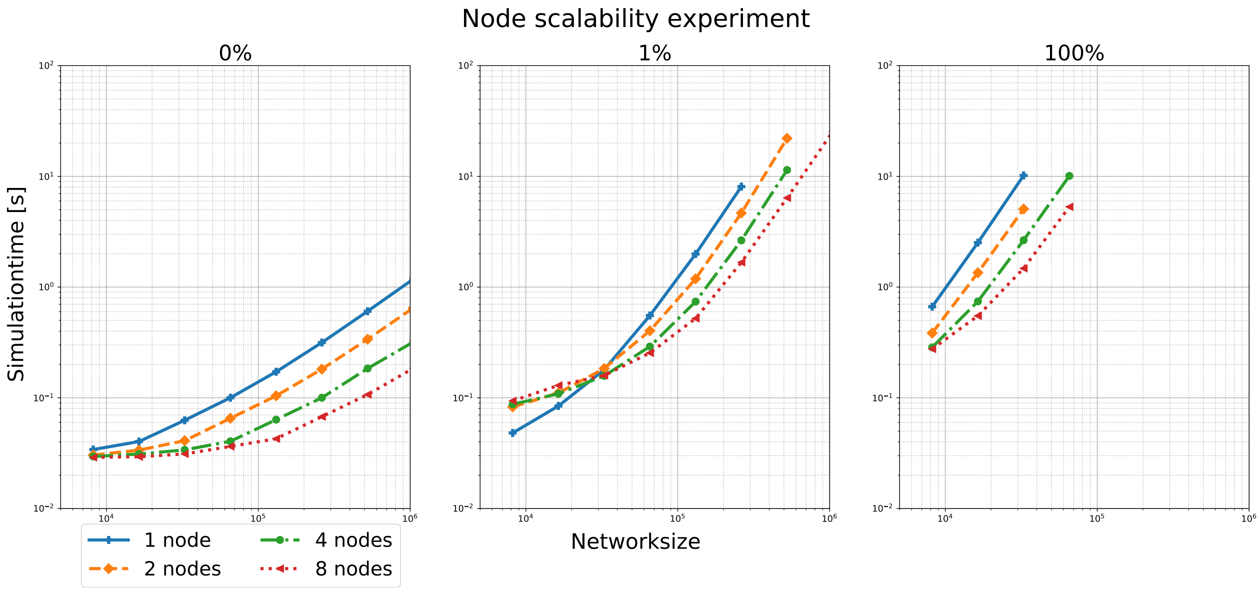


Figure 6.5: Experimental results to determine scalability over a varing size of nodes results are reported for 1,2,4,8 nodes with a fixed GPU count of 1 gpu per node. Compute platform selected is NSL-A

Table 6.5: Exploration space, Aris scalability

Parameter	Range
Model	IOmodel by DeGruijl [14]
Number of simulation steps	1000
Problem size	[8192,16384,32768,65536,131072,262144,524288, 1048576]
GJ connection density	[0,0.01,0.1,1]
GJ connection pattern	RandomBinary
GPU(s) per compute node	[2]
Compute node(s)	[1,2,4,8,16,32]
MPI Communication pattern	`MPI_allgather`

6.5. Communication Performance

One of the most important, but also the hardest metric, is the communication overhead. There is no computational platform that gives freedom to control every aspect of the networking between nodes. Also, the infrastructure of networking is most of the time unclear to the end-user. Because the interconnection architecture is unknown, one can not be sure that the nodes will be physically close to each other when picking two nodes. Because other running jobs potentially utilize the cluster network, it makes it very hard to get results that are comparable with each other.

Furthermore, without a clear understanding of the architecture and not having the possibility to monitor this network, it is impossible to determine the influence other jobs have. The best approach would be to run the same experiment multiple times during a particular period. Running jobs on the same list of nodes will exclude structural network differences.

Because of all these limitations, it is chosen to run each experiment multiple times, which should give an insight into network fluctuations. However, trends that occur are more interesting than perfect measurements. NSL-A and Aris are selected for experimentation. The most pressing research question is which MPI communication style is preferable when supplying a certain network configuration. It is expected that locality in the synaptic gap junction network will decrease MPI communications times when utilizing a tailored communication scheme. A more randomly distributed synaptic gap junction network will probably only add overhead, and all to all, sharing of the synaptic compartment potentials is preferred. Exploration space for the experiments is described in Table 6.6. Results for NSL-A are presented in Figure 6.7 and results for Aris in Figure 6.8. The gaussian synaptic connection pattern is expecting a variance and mean parameter. The mean will be fixed at zero, where the variance will be two times the network size—creating a gaussian distribution that still has a big chance of connecting

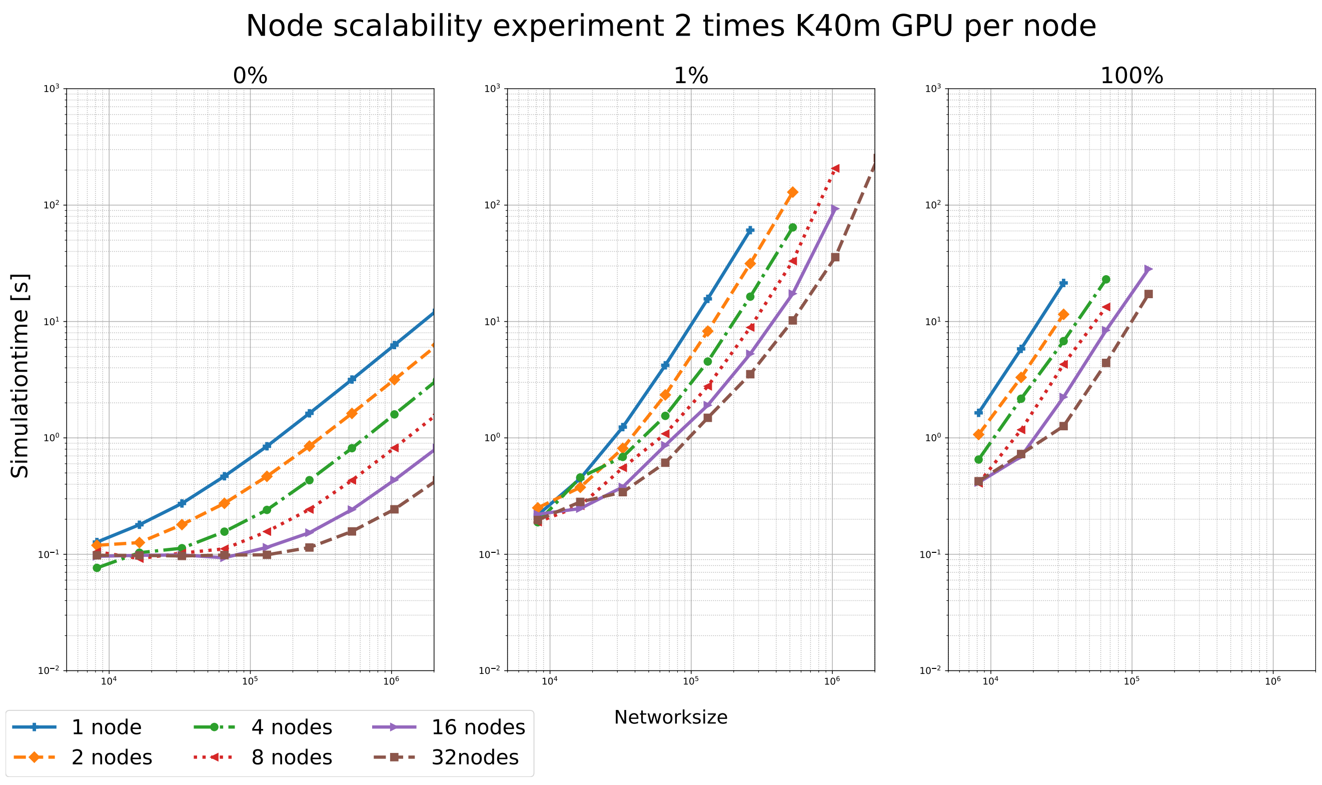


Figure 6.6: Experimental results to determine scalability over a varing size of nodes results are reported for 1,2,4,8,16 and 32 nodes with a fixed GPU count of 2 GPUs per node. Compute platform selected is ARIS

to a far node in the network. However, it will show a high degree of locality.

Table 6.6: Exploration space for MPI communication time experiments

Parameter	**Range**
Model	IOmodel by DeGruijl [14]
Number of simulation steps	1000
Problem size	262144(NSL-A) or 65536(Aris)
GJ connection pattern	[RandomBinary, Gaussian]
GJ connection density	[0.001,0.01,0.1]
GPU(s) per compute node	4 (NSL-A) or 2 (Aris)
Compute node(s)	[1,2,4,8] (NSLA-A) or [1,2,4,8,16,32] (Aris)
MPI Communication pattern	`Allgather` and `Alltoallv`

Observations Both Figures 6.7 and 6.8 are showing results that are expected. With high locality in the gaussian distribution network, an `alltoallv` approach is preferred over the `allgather` method for MPI communication. A possible next step could be to find the tipping point in terms of durations for both communication methods. Which communication method is preferred for which locality rate present in the gap junction network? This thesis left this question out of scope due to the additional challenge of quantifying the locality rate and designing generation schemes for such interconnection graphs. This was mainly due to the fact the focus is on building a usable simulator, and most neuroscientists will use their own created graphs.

The network fluctuation is visible but is not showing as much variance as anticipated. Both systems are running fat-tree topologies, which explains this observation. Because this experiment is focussing on comparing multi-node setups with each other, it can be argued that for the higher node counts, 8 and 32 respectively, most of the MPI communication timing is initialization and latency, which is a valid point. However, Figures 6.7 and 6.8 are selected because of the trend they show. Adding more nodes increases MPI communication duration. The duration increases in a linear fashion with the number of nodes selected, with a slope that shows the potential of scaling.

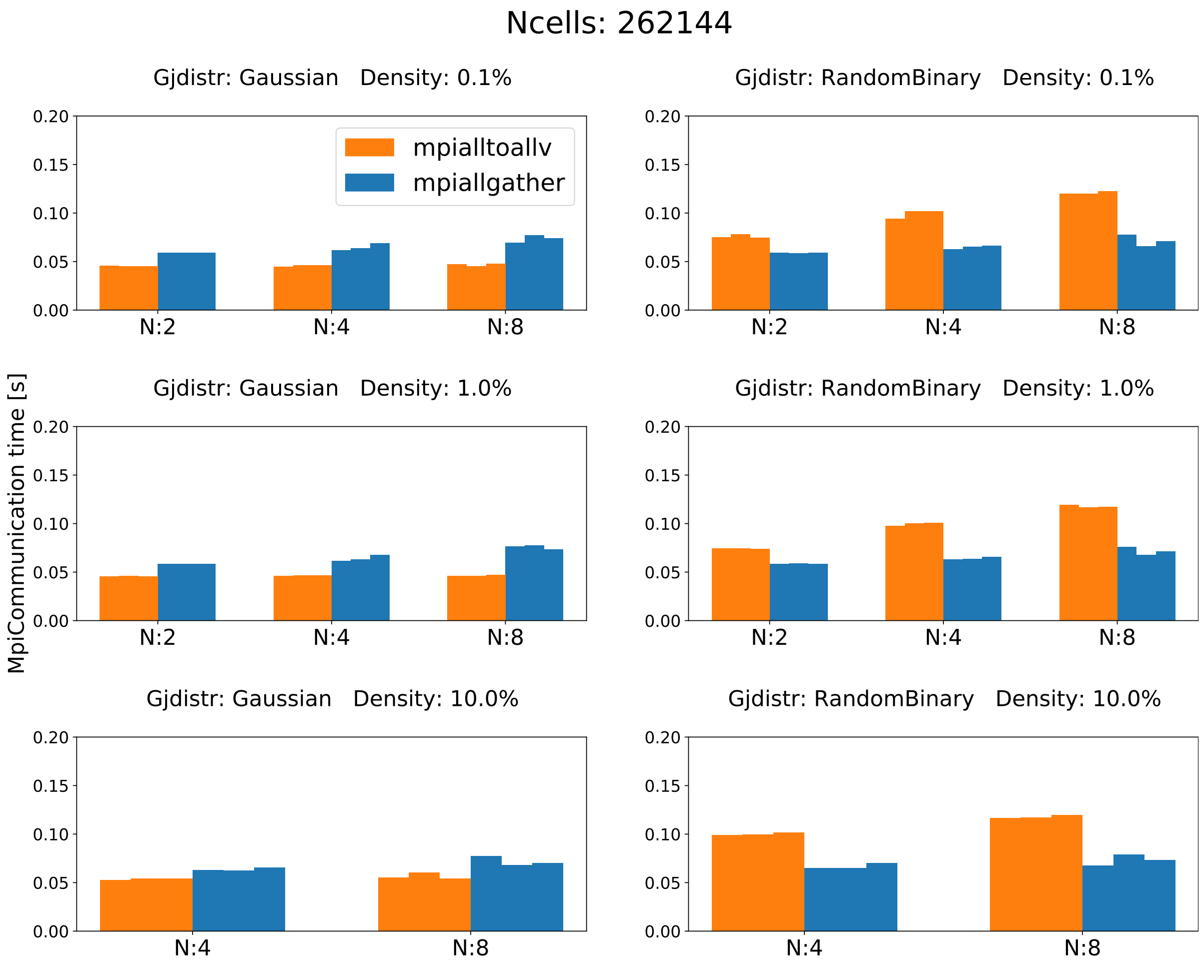


Figure 6.7: MPI communication results, at the NSL-A compute cluster, for the exploration space in Table 6.6. Three repetitions of each experiment are conducted and presented as a smaller bar of the same color, which shows network fluctuations between experiments. The results illustrate the mpi `allgather` method versus the mpi `alltoallv` method. It becomes clear the with a higher rate of locality (gaussian) the latter becomes favourable.

6.6. Comparison Against Related Work

In the related work, it became clear that quite a lot of work is trying to find the best solution for extended Hodkin-Hukley type networks. The most recent efforts at the neurocomputing Lab at the Erasmus MC were, flexHH, BrainGPU, and GenEHH. This work gives a multi-node multi GPU solution for the same problems these implementations are tackling. BrainGPU is included because it is the latest MultiNode capable GPU solution of the hardcode model. The exploration space is limited by flexHH, which only supports up to approximately 8000 neurons. flexHH, doesn't benefit from less density, and therefore, a fully connected graph is selected to compare against. The full exploration space is presented in Table 6.7. Results are visually presented in Figure 6.9. It was expected that mgpuHH would set some new standards. However, it is even competing with flexHH, which is implemented on a Data flow engine. flexHH still is the only implementation that gets simulation time the lowest for smaller sized networks. However, as soon as more than 2000 IO-model neurons are configured, mgpuHH is the clear new standard.

GenEHH vs mgpuHH While Figure 6.9 already gave some insight is this comparison network sizes only go up to 8K, which is not really representable for extensive size network experimentation. Therefore, a specific experiment for GenEHH versus mgpuHH is conducted. It is important to note that

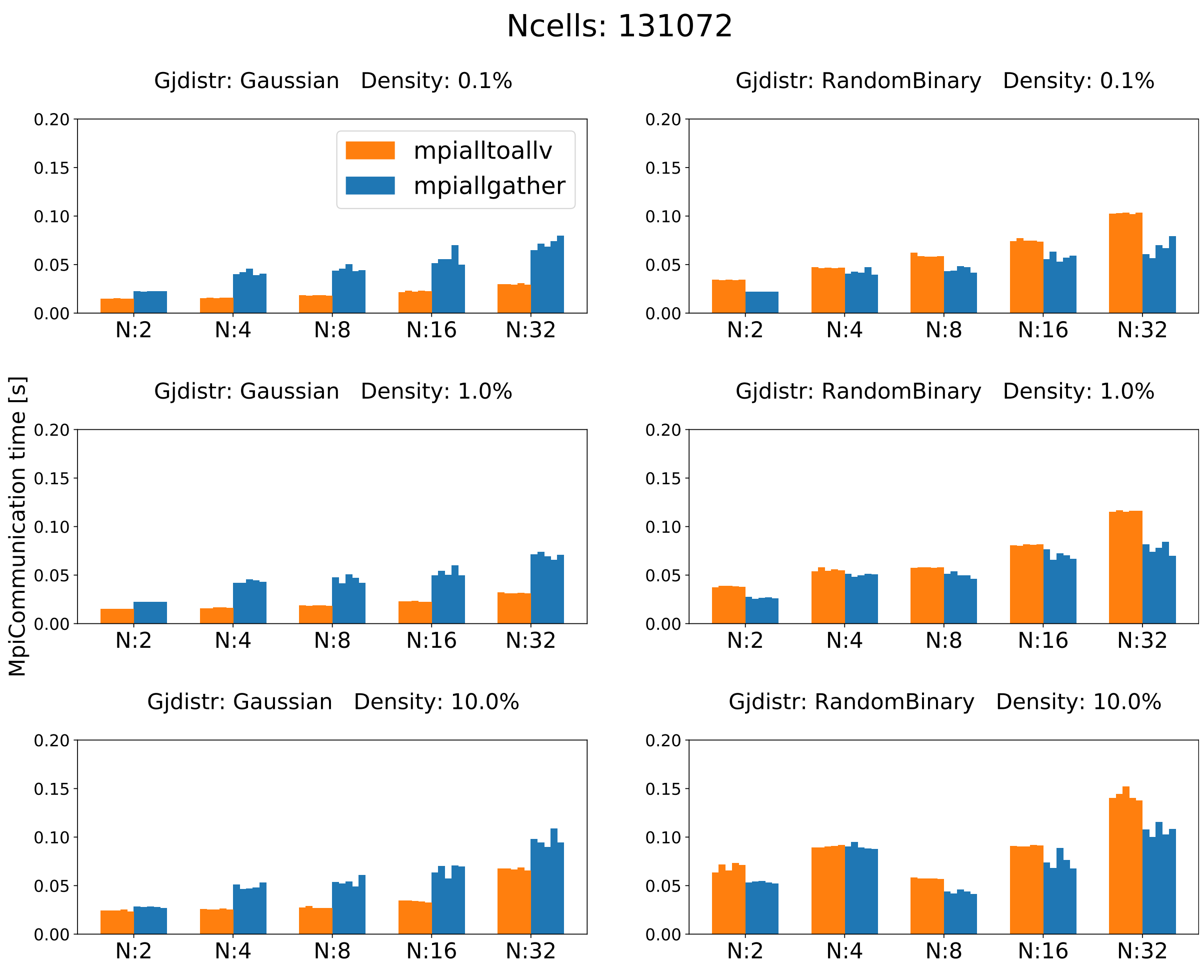


Figure 6.8: MPI communication results, from the ARIS compute cluster, for the exploration space Table 6.6. Five repetitions of each experiment are conducted and presented as a smaller bar of the same color, which shows networks fluctuations between experiments. The results illustrate the `MPI_allgather` method versus the `MPI_alltoallv` method. It becomes clear the with a higher rate of locality (gaussian) the latter becomes favourable.

Table 6.7: Exploration space, Coparison with related work

Model	IO-model by DeGruijl [14]
Number of simulation steps	100
Problem size	[166 , ... , 7808]
GJ connection density	1
Implementations	[flexHH, brainGPU, GenEHH, mgpuHH]

GenEHH is configured only to use one thread to show the potential of parallelization with mgpuHH.

Table 6.8: Exoloration space for the CPU vs GPU experiment, create to show the portential of parallelization

Parameter	**Range**
Model	IOmodel by DeGruijl [14]
Number of simulation steps	100
Problem size	[128,256,512,1024,2048,4096,8192,16384,32726,65536]
GJ connection density	[0,0.5,1]
GJ connection pattern	RandomBinary
CPU threads	1
GPU(s)	1

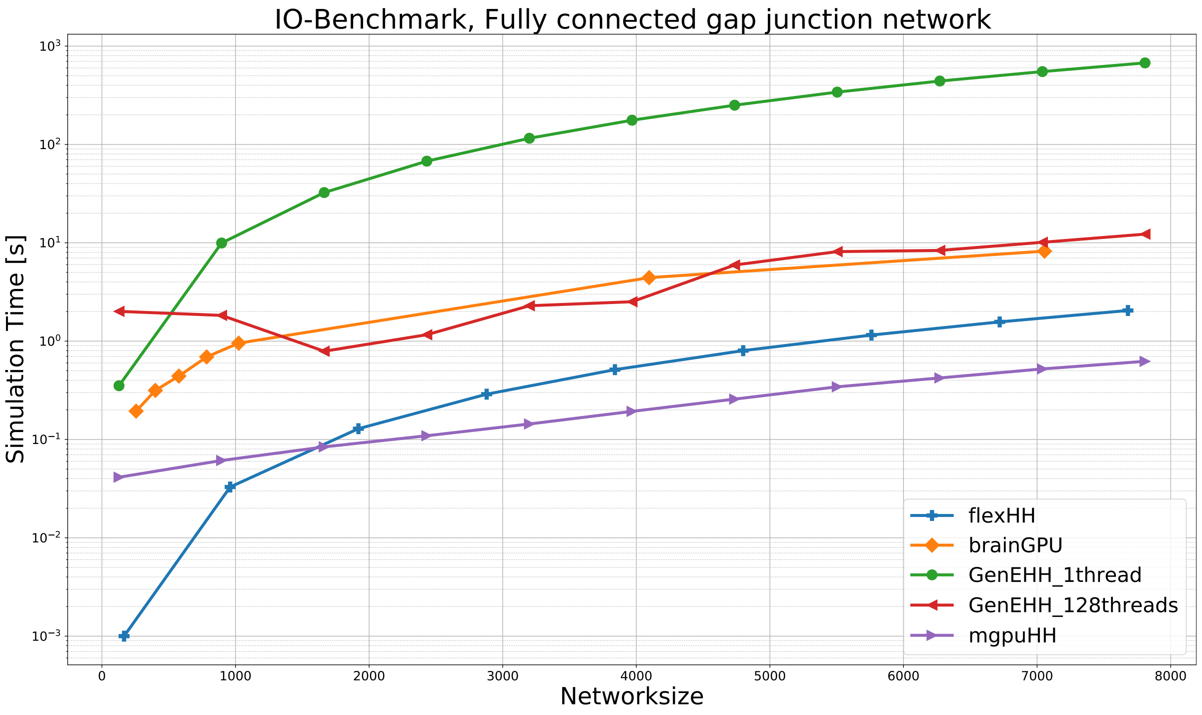


Figure 6.9: Comparison against related work. All experimentation is kept as close as possible to each other in terms of generated output and configuration. For flexHH, this work didn't have the freedom to run tailored experiments, and therefore results from [26] are taken. However, it is not clear what is included in terms of generated output for the simulation results. Predication is made that the most favorable results are presented in terms of output generation, and therefore it is estimated that the presented results are fair. BrainGPU and GenEHH source code and executables were accessible, giving the freedom to set all configurations in the same manner to create a fair comparison.

BrainGPU vs mgpuHH BrainGpu is a multi-node implementation containing a hardcoded IO-model. It was the inspiration that led to the research questions of this work. Therefore, comparison with BrainGPU is made to show that the difference in design choices taken is correct when trying to achieve higher performing simulators.

Table 6.9: Exploration space for BrainGPU vs mgpuHH

Parameter	**Range**
Model	IOmodel by DeGruijl [14]
Number of simulation steps	100
Problem size	[512,1024,2048,4096,8192,16384,32726,65536]
GJ connection density	[0,0.01,0.1,1]
GJ connection pattern	RandomBinary
GPU(s)	1

6.7. Use Case Evaluation

The design, as described in this thesis, is already deployed on servers by Neuroscientist to simulated their research. In collaboration, tweaks to the configurations are made and the way the program outputs data. Because this is a highly specific simulator for a particular use case, the codebase will stay under constant development to better suit particular experiments. Current research is in the field of dynamical clustering in the Inferior Olive is conducted, and the codebase functionality of the codebase is adapted to support this research.

An IO-model of 10 million cells with an average Synaptic connection density of 0.1% is simulated, setting new standards for what is possible in the large scale simulation of eHH type neurons.

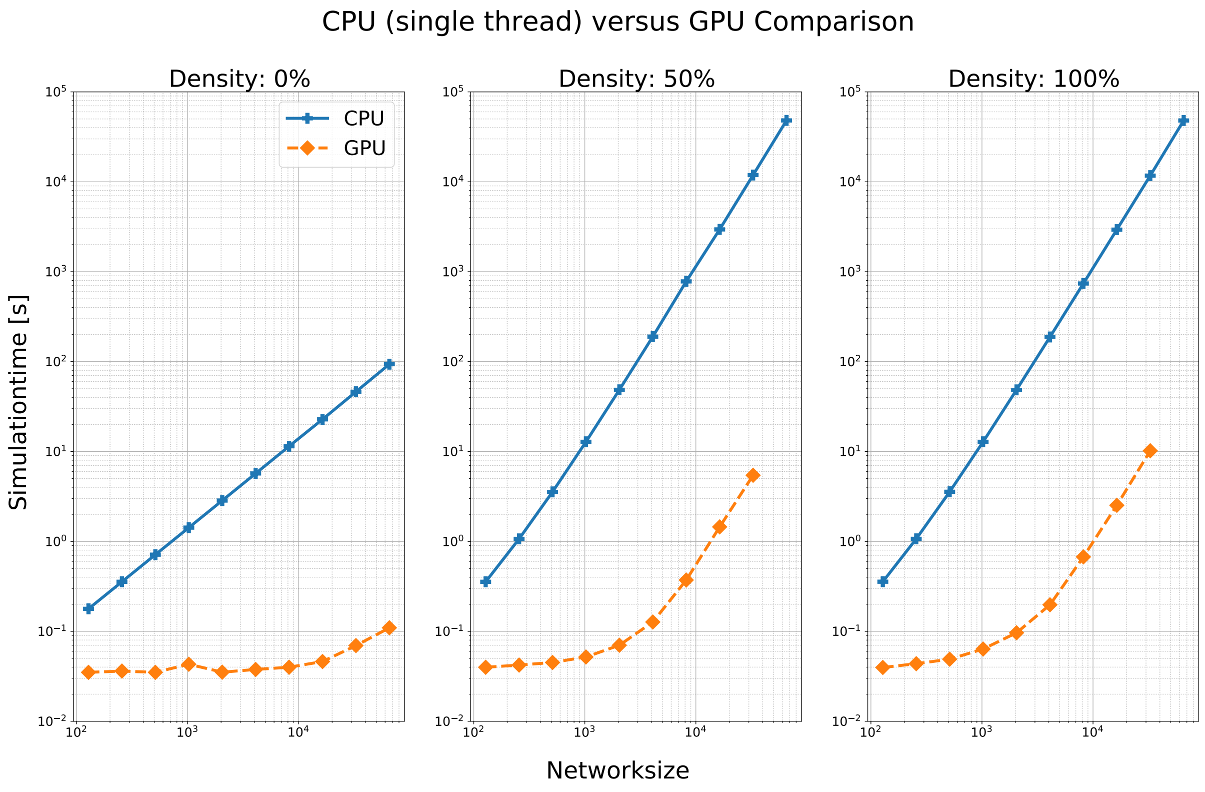


Figure 6.10: Results for the exploration space of Table 6.8. Presented results support the use parallelization for this simulator. As soon as the GPU is fully utalized it stabilized and is following the same trend as the Single Threaded CPU implementation showing increase in performance .

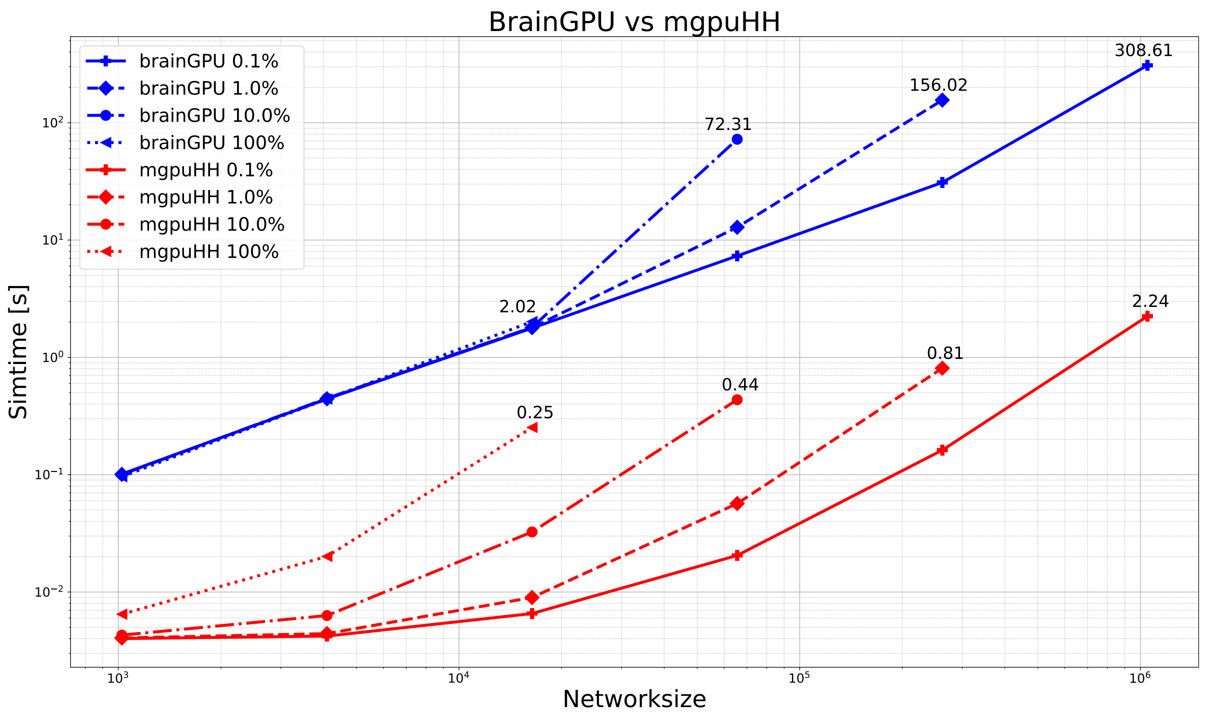


Figure 6.11: Comparison of braingpu vs mgpuHH for the IOmodel. Results show the improvement is made in terms of performance. BrainGPU is a hardcode version, and mgpuHH can support a sort of eHH models. The exploration space is presented in Table 6.9 and experimentation is done on the Computedev server at the NCL

7 Conclusions and Future Work

This work provides an addition to the work described in Miedema et al. [26] and Sotirios et al. [31]. The idea of building a flexible neural network simulator has proven to perform very well on DFE and CPU systems. This work has added a GPU implementation, which has proven to have outstanding performance and set the new standard for large scale IOmodel simulations. This work offers a multi-node multi-GPU simulator (mgpuHH) that offers excellent scalability performance due to how the simulator is constructed. Synaptic gap junction communication is done concurrently with calculations whenever the configuration allows it. It is proven to work well for the IOmodel form De Gruijl [14] where a simulation of up to millions of cells with high connection counts between neurons is possible. The memory footprint is designed with the focus on only storing the necessary information to be as small as possible. Intermediate results are offloaded to disk memory parallel with the simulation, making it possible to simulate a very long biological time with no system memory limitations on simulation time. Reported results go up to 32 Nodes with a total of 64 GPU cards. The design shows linear weak and strong scaling within the experimental setups for intra-node and inter-node scalability.

The synaptic gap junction calculations and communication are the critical path for simulation performance and the largest problem for the memory footprint needed to run specific neural networks. To calculate the Gap junction currents, a specialized kernel is designed, focussing on specific GPU functionality to increase performance and further parallelize computations. For communication, experimentation is done with two different communication scheme implementation. A share everything with everyone (`MPI_allgather`) method is compared against an implementation that only shares the necessary compartmental potentials with the nodes containing the neurons that rely on them (`MPI_alltoallv`). For Larger locality in the synaptic gap junction graph, the `MPI_alltoallv` method performs better than the `MPI_allgather`. For a complete uniformly distributed network (RandomBinary) `MPI_allgather` is performing better due to less computational overhead. Because of the random nature of generation synaptic connections, it is almost certain everything needs to be shared with every compute node anyway, so any added complexity will add to the performance.

Comparisons against related work on CPU and FPGAs have been conducted, a 100x speedup is acheived versus a single cpu threaded solution. Furthermore, a 2x speedup is acheived over a FPGA solution (flexHH) and 10 fold over a multithreaded CPU (GenEHH, with 128 threads) solution, both reported speedups are for a fully connected network with 7000 IO cells.

7.1. Contributions

- A new versatile, multi Node multi GPU eHH simulator is presented, which offers high performance with weak and strong scaling characteristics.

- This work sets the new standard for large scale simulations in the IO-model race as described in Chapter 3 and in simulating models of the eHH class.

- The work present a GPU optimize way to perform the Gap junction calculations present in the eHH class of neural models.

- A three kernel design is presented, which can overlap calculation with communication. (Independent of the numerical solver choice)
- A comprehensive performance analysis, including results up to 32 nodes and 64 GPUs, is presented.
- In collaboration with neuroscientists, it is made as usable as possible to aid neurocomputing research.
- Parameter randomization and description multiplications are added to reduce the memory footprint, increase performance and create more biologically accurate simulations.

7.2. Future work

Potential efforts that could be made to continue this project.

- Research into the influences of the Gap junction network's locality on Communication overhead would give better insight into preferred communication methods while varying the locality. The could pose interesting results that can influence the design choices for neural simulators. The research would be twofold. Fristly design a Graph Generation algorithm with controllable and quantifiable locality (e.g., Gaussian type), which can reliably satisfy the requested density. Secondly, implement and benchmark a sweep over this quantified locality with different communication methods (e.g., `mpi_allgather` versus `mpi_alltoallv`).
- General improvements in implementation are still possible. With more time and/or knowledge for CUDA design, it is estimated there is still performance to gain, as shown by the roofline model in Figure 6.2.
- The simulator presented in this thesis can still be extended with more features to support a wider range of experiments. Examples could be supporting a more extensive range of externally applied currents, support different compartment arrangments instead of only supporting compartments in a chain arrangement, and many more.
- Different solver types could be incorporated into the design. Where the simulator now only supports a Forward Euler solving method. It could be extended with higher-order solvers, which can possibly better balance compute and communication time for growing to compute node counts.
- In cooperation with neuroscientists, the first simulations of a human-size IOmodel could be created.
- Research in grouping clustered neurons together, and positioning them in memory in a way that reduces communication overhead, is a fascinating field of research.
- The JSON input configuration files are heavily based upon the NeuroML description language. Efforts could be made to parse NeuroML to the JSON style configuration input this work uses, increasing the usability even more of this work.

Bibliography

[1] CUDA C++ Programming Guide. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Online; accessed 18 December 2020.

[2] NVIDIA GPUDirect. URL https://developer.nvidia.com/gpudirect. Online; accessed 21 October 2020.

[3] TOP500. URL https://www.top500.org/lists/top500/2020/11/. Online; accessed 27 November 2020.

[4] Slurm Workload Manager. URL https://slurm.schedmd.com/. Online; accessed 21 October 2020.

[5] SPEC CPU2006 Results. Internet:https://spec.org/cpu2006/results/, 2018 [Oct. 18 2018].

[6] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. Arbor — a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb 2019. doi: 10.1109/empdp.2019.8671560. URL http://dx.doi.org/10.1109/EMPDP.2019.8671560.

[7] K. Asanovic, B. C. Catanzaro, D. Patterson, and K. Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. *EECS Department University of California Berkeley Tech Rep UCBEECS2006183*, 18:19, 2006. ISSN 00010782. doi: 10.1145/1562764.1562783.

[8] Christoph Börgers and Alexander R. Nectow. Exponential time differencing for hodgkin-huxley-like odes. *SIAM Journal on Scientific Computing*, 35, 2013. ISSN 10648275. doi: 10.1137/120883657.

[9] Rene Miedema CE-MS. flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations, 2020.

[10] 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*, 329:370–383, 2 2019. ISSN 18728286. doi: 10.1016/j.neucom.2018.10.062.

[11] Ting-Shuo Chou, Hirak J. Kashyap, Jinwei Xing, S. Listopad, Emily L. Rounds, Michael Beyeler, N. Dutt, and J. Krichmar. Carlsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2018.

[12] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 225–236, 2010. ISSN 10724451. doi: 10.1109/MICRO.2010.36.

[13] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL https://www.frontiersin.org/article/10.3389/neuro.11.011.2008.

[14] Jornt R. de Gruijl, Paolo Bazzigaluppi, Marcel T.G. de Jeu, and Chris I. de Zeeuw. Climbing fiber burst size and olivary sub-threshold oscillations in a network setting. *PLoS Computational Biology*, 8, 12 2012. ISSN 1553734X. doi: 10.1371/journal.pcbi.1002814.

[15] David A. Drachman. Do we have brain to spare?, 6 2005. ISSN 00283878.

[16] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[17] Padraig Gleeson, Sharon Crook, Robert C. Cannon, Michael L. Hines, Guy O. Billings, Matteo Farinella, Thomas M. Morse, Andrew P. Davison, Subhasis Ray, Upinder S. Bhalla, Simon R. Barnes, Yoana D. Dimitrova, and R. Angus Silver. Neuroml: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Computational Biology*, 6:1–19, 6 2010. ISSN 1553734X. doi: 10.1371/journal.pcbi.1000815.

[18] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006. doi: 10.1109/CLUSTR.2006.311904.

[19] Jan Hahne, Moritz Helias, Susanne Kunkel, Jun Igarashi, Matthias Bolten, Andreas Frommer, and Markus Diesmann. A unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. *Frontiers in Neuroinformatics*, 9, 9 2015. ISSN 16625196. doi: 10.3389/fninf.2015.00022.

[20] Suzana Herculano-Houzel. The human brain in numbers: A linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, 3, 11 2009. ISSN 16625161. doi: 10.3389/neuro.09.031.2009.

[21] A L Hodgkin and A F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol*, pages 500–544, 1952.

[22] Tammo Ippen, Jochen M. Eppler, Hans E. Plesser, and Markus Diesmann. Constructing neuronal network models in massively parallel environments. *Frontiers in Neuroinformatics*, 11, 5 2017. ISSN 16625196. doi: 10.3389/fninf.2017.00030.

[23] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15:1063–1070, 9 2004. ISSN 10459227. doi: 10.1109/TNN.2004.832719.

[24] Jakob Jordan, Tammo Ippen, Moritz Helias, Itaru Kitayama, Mitsuhisa Sato, Jun Igarashi, Markus Diesmann, and Susanne Kunkel. Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Frontiers in Neuroinformatics*, 12, 2 2018. ISSN 16625196. doi: 10.3389/fninf.2018.00002.

[25] Chris Mcclanahan. History and evolution of gpu architecture, 2010.

[26] Rene Miedema, Georgios Smaragdos, Mario Negrello, Zaid Al-Ars, Matthias Moller, and Christos Strydis. Flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations. *IEEE Access*, 8:121905–121919, 2020. ISSN 21693536. doi: 10.1109/ACCESS.2020.3007019.

[27] Kirill Minkovich, Corey M. Thibeault, Michael John O'Brien, Aleksey Nogin, Youngkwan Cho, and Narayan Srinivasa. Hrlsim: A high performance spiking neural network simulator for gpgpu clusters. *IEEE Transactions on Neural Networks and Learning Systems*, 25:316–331, 2 2014. ISSN 2162237X. doi: 10.1109/TNNLS.2013.2276056.

[28] H. A. D. Nguyen, Z. Al-Ars, G. Smaragdos, and C. Strydis. Accelerating complex brain-model simulations on gpu platforms. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 974–979, 2015. doi: 10.7873/DATE.2015.0071.

[29] Nikos Nikoloutsakos. Aris documentation, 2021. URL https://doc.aris.grnet.gr/.

[30] Eustace Painkras, Luis A. Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R. Lester, Andrew D. Brown, and Steve B. Furber. Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48: 1943–1953, 2013. ISSN 00189200. doi: 10.1109/JSSC.2013.2259038.

[31] S. Panagiotou, R. Miedema, H. Sidiropoulos, G. Smaragdos, C. Strydis, and D. Soudris. A novel simulator for extended hodgkin-huxley neural networks. In *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 395–402, 2020. doi: 10.1109/BIBE50027.2020.00071.

[32] Arleen Salles, Jan G. Bjaalie, Kathinka Evers, Michele Farisco, B. Tyr Fothergill, Manuel Guerrero, Hannah Maslen, Jeffrey Muller, Tony Prescott, Bernd C. Stahl, Henrik Walter, Karl Zilles, and Katrin Amunts. The human brain project: Responsible brain research for the benefit of society, 2 2019. ISSN 10974199.

[33] Nicolas Schweighofer, Eric J. Lang, and Mitsuo Kawato. Role of the olivo-cerebellar complex in motor learning and control. *Frontiers in Neural Circuits*, 4 2013. ISSN 16625110. doi: 10.3389/fncir.2013.00094.

[34] G. Smaragdos, G. Chatzikostantis, S. Nomikou, D. Rodopoulos, I. Sourdis, D. Soudris, C. I. De Zeeuw, and C. Strydis. Performance analysis of accelerated biophysically-meaningful neuron simulations. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 1–11, 2016. doi: 10.1109/ISPASS.2016.7482069.

[35] Georgios Smaragdos, Sebastian Isaza, Martijn F. van Eijk, Ioannis Sourdis, and Christos Strydis. Fpga-based biophysically-meaningful modeling of olivocerebellar neurons. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 89–98, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326711. doi: 10.1145/2554688.2554790. URL https://doi.org/10.1145/2554688.2554790.

[36] Georgios Smaragdos, Georgios Chatzikonstantis, Rahul Kukreja, Harry Sidiropoulos, Dimitrios Rodopoulos, Ioannis Sourdis, Zaid Al-Ars, Christoforos Kachris, Dimitrios Soudris, Chris I. De Zeeuw, and Christos Strydis. Brainframe: A node-level heterogeneous accelerator platform for neuron simulations. *Journal of Neural Engineering*, 14, 11 2017. ISSN 17412552. doi: 10.1088/1741-2552/aa7fc5.

[37] Marcel Stimberg, Dan F.M. Goodman, and Thomas Nowotny. Brian2genn: accelerating spiking neural network simulations with graphics hardware. *Scientific Reports*, 10, 12 2020. ISSN 20452322. doi: 10.1038/s41598-019-54957-7.

[38] Michiel A. Van Der Vlag, Georgios Smaragdos, Zaid Al-Ars, and Christos Strydis. Exploring complex brain-simulation workloads on multi-gpu deployments. *ACM Transactions on Architecture and Code Optimization*, 16, 2019. ISSN 15443973. doi: 10.1145/3371235.

[39] Tadashi Yamazaki and Jun Igarashi. Realtime cerebellum: A large-scale spiking network model of the cerebellum that runs in realtime using a graphics processing unit. *Neural Networks*, 47: 103–111, 11 2013. ISSN 08936080. doi: 10.1016/j.neunet.2013.01.019.

[40] Esin Yavuz, James Turner, and Thomas Nowotny. Genn: A code generation framework for accelerated brain simulations. *Scientific Reports*, 6, 1 2016. ISSN 20452322. doi: 10.1038/srep18854.

A

Overview of the IOmodel

The IOmodel described by [14] and alomst exclusivly used to benchmark this thesis is decribed by the following collection of formulas.

Dend

$$
\begin{aligned}
\frac{dV_{dend}}{dt} &= S\left[I_{app} - I_{gap} - I_{channels} - I_{interact} - I_{leak}\right] \\
V_{dend}\,(0) &= -60 \\
S &= 1 \\
g_{leak} &= 0.016 \\
V_{leak} &= 10 \\
I_{leak} &= g_{leak}\,(V - V_{leak}) \\
I_{interact} &= g_{int}\left(1 - g_{p,next}\right)(V_{dend} - V_{soma}) \\
I_{channel} &= I_{calcium\_channel} + I_{calcium\_controlled\_potassium\_channel} + I_{h\_channel} \\
g_{int} &= 0.13 \\
\left[Ca^{2+}\right](0) &= 3.7152 \\
\frac{d\left[Ca^{2+}\right]}{dt} &= 3I_{calcium_channel} - 0.075\left[Ca^{2+}\right]
\end{aligned}
$$

Calcium Channel

$$
\begin{aligned}
I_{calcium\_channel} &= g_c\,(V - V_c)\,y_r^2 \\
g_c &= 4.5 \\
V_c &= 120 \\
\frac{dy_r}{dt} &= (1 - y_r)\alpha_r - y_r\beta_r \\
\alpha_r &= \frac{0.34}{1e^{0.071942\cdot(5-V)} + 1} \\
\beta_r &= \frac{0.004(-8.5 - V)}{-1e^{-0.200000\cdot(-8.5-V)} + 1}
\end{aligned}
$$

Calcium Controlled Potassium Channel

$$
\begin{aligned}
I_{calcium\_controlled\_potassium\_channel} &= g_c\,(V - V_c)\,y_s \\
g_c &= 35 \\
V_c &= -75 \\
\frac{dy_s}{dt} &= (1 - y_s)\alpha_s - y_s\beta_s \\
\alpha_s &= \min\left(-0.000020 \cdot (0 - \left[Ca^{2+}\right]), 0.01\right) \\
\beta_s &= 0.015
\end{aligned}
$$

H Channel

$$
\begin{aligned}
I_{h\_channel} &= g_c\,(V - V_c)\,y_n \\
g_c &= 0.125 \\
V_c &= -43 \\
\frac{dy_n}{dt} &= \frac{\alpha_n - y_n}{\beta_n} \\
\alpha_n &= \frac{1}{1e^{-0.250000\cdot(-80-V)} + 1} \\
\beta_n &= \frac{1}{1e^{0.086000\cdot(-169.7674418604651-V)} + 1e^{-0.070000\cdot(26.714285714285715-V)}} + 1
\end{aligned}
$$

Soma

$$
\begin{aligned}
\frac{dV_{soma}}{dt} &= S\left[I_{app} - I_{gap} - I_{channels} - I_{interact} - I_{leak}\right] \\
V_{soma}\,(0) &= -60 \\
S &= 1 \\
g_{leak} &= 0.016 \\
V_{leak} &= 10 \\
I_{leak} &= g_{leak}\,(V - V_{leak}) \\
I_{interact} &= g_{int}/g_p(V_{soma} - V_{dend}) + g_{int}\left(1 - g_{p,next}\right)(V_{soma} - V_{axon}) \\
I_{channel} &= I_{calcium\_channel} + I_{sodium\_channel} + I_{potassium\_delayed\_rectifier\_channel} + I_{potassium\_channel} \\
g_p &= 0.25
\end{aligned}
$$

Calcium Channel

$$
\begin{aligned}
I_{calcium\_channel} &= g_c\,(V - V_c)\,y_k^3 y_l \\
g_c &= 0.68 \\
V_c &= 120 \\
\frac{dy_k}{dt} &= \frac{\alpha_k - y_k}{\beta_k} \\
\alpha_k &= \frac{1}{1e^{0.238095\cdot(-61-V)} + 1} \\
\beta_k &= 1 \\
\frac{dy_l}{dt} &= \frac{\alpha_l - y_l}{\beta_l} \\
\alpha_l &= \frac{1}{1e^{-0.117647\cdot(-85.5-V)} + 1} \\
\beta_l &= \frac{20e^{-0.033333\cdot(-160-V)}}{1e^{-0.136986\cdot(-84-V)} + 1} + 35
\end{aligned}
$$

Sodium Channel

$$
\begin{aligned}
I_{sodium\_channel} &= g_c\,(V - V_c)\,y_m^3 y_h \\
g_c &= 150 \\
V_c &= 55 \\
y_m &= \frac{1}{1e^{0.181818\cdot(-30-V)} + 1} \\
\frac{dy_h}{dt} &= \frac{\alpha_h - y_h}{\beta_h} \\
\alpha_h &= \frac{1}{1e^{-0.172414\cdot(-70-V)} + 1} \\
\beta_h &= 3e^{0.030303\cdot(-40-V)}
\end{aligned}
$$

Potassium Delayed Rectifier Channel

$$
\begin{aligned}
I_{potassium\_delayed\_rectifier\_channel} &= g_c\,(V - V_c)\,y_n^4 \\
g_c &= 9 \\
V_c &= -75 \\
\frac{dy_n}{dt} &= \frac{\alpha_n - y_n}{\beta_n} \\
\alpha_n &= \frac{1}{1e^{0.100000\cdot(-3-V)} + 1} \\
\beta_n &= 47e^{-0.001111\cdot(-50-V)} + 5
\end{aligned}
$$

Potassium Channel

$$\begin{aligned}
I_{potassium\_channel} &= g_c\,(V - V_c)\,y_x^4 \\
g_c &= 5 \\
V_c &= -75 \\
\frac{dy_x}{dt} &= (1 - y_x)\alpha_x - y_x\beta_x \\
\alpha_x &= \frac{-0.13(-25 - V)}{-1e^{0.100000\cdot(-25-V)} + 1} \\
\beta_x &= 1.69e^{0.012500\cdot(-35-V)}
\end{aligned}$$

Axon

$$\begin{aligned}
\frac{dV_{axon}}{dt} &= S\left[I_{app} - I_{gap} - I_{channels} - I_{interact} - I_{leak}\right] \\
V_{axon}\,(0) &= -60 \\
S &= 1 \\
g_{leak} &= 0.016 \\
V_{leak} &= 10 \\
I_{leak} &= g_{leak}\,(V - V_{leak}) \\
I_{interact} &= g_{int}/g_p(V_{axon} - V_{soma}) \\
I_{channel} &= I_{sodium\_channel} + I_{potassium\_channel} \\
g_p &= 0.15
\end{aligned}$$

Sodium Channel

$$\begin{aligned}
I_{sodium\_channel} &= g_c\,(V - V_c)\,y_m^3 y_h \\
g_c &= 240 \\
V_c &= 55 \\
y_m &= \frac{1}{1e^{0.181818\cdot(-30-V)} + 1} \\
\frac{dy_h}{dt} &= \frac{\alpha_h - y_h}{\beta_h} \\
\alpha_h &= \frac{1}{1e^{-0.172414\cdot(-60-V)} + 1} \\
\beta_h &= 1.5e^{0.030303\cdot(-40-V)}
\end{aligned}$$

Potassium Channel

$$\begin{aligned}
I_{potassium\_channel} &= g_c\,(V - V_c)\,y_x^4 \\
g_c &= 20 \\
V_c &= -75 \\
\frac{dy_x}{dt} &= (1 - y_x)\alpha_x - y_x\beta_x \\
\alpha_x &= \frac{-0.13(-25 - V)}{-1e^{0.100000\cdot(-25-V)} + 1} \\
\beta_x &= 1.69e^{0.012500\cdot(-35-V)}
\end{aligned}$$

B

JSON input-configuration-file example

A minimal example of a single compartment single channel single gate configuration file

```
{
  "Label": "test_5_infoli_gap",
  "SimTime":       10.00,
  "SimTimestep":   0.001,
    "GPU_enable": 4,
    "GPU_Time_kernels": true,
    "GPU_Use_UVA": false,
    "GPU_MPI_share_everything": false,
    "GPU_disable_peer_acces": false,
    "OutputMetadataFile": "metadata.dat",

  "OutputMonitor": {
    "Name": "./examples/doc/test_5_infoli_gap",
    "Type": "All",
    "Formatting": true,
    "WriteCompartPotentials": true,
    "WriteCalciumLevels": true,
    "WriteCurrents": true ,
    "WriteActivationVariables": true,
    "SaveInterval" : 10
  },

  "ConnectivityModel": {
    "Type": "RandomBinary",
    "Weight" : 0.1,
    "Density" : 1,
    "RandomSeed" : "time"
  },

  "StimulusModel":      {
    "Type" : "DcCurrentPulseModulo",
    "BaseCurrent": 0,
    "StartTime": 4,
    "Duration":  1,
    "Modulo": 20,
    "ModuloCurrent": 10
```

```
},

"CellPopulation":    {
  "Type": "Explicit",
  "NeuronModels": [
    {
      "Label": "infoli_full1",
      "NeuronModelSize": 256,
      "Compartments": [
        {
          "Label": "dend",

          "InverseCapacitance": 1,
          "PassiveLeakConductivity": 0.016,
          "PassiveLeakInversionPotential": 10,
          "InitialVoltage": -60,
          "ConductivityRatio": 0.13,

          "CalciumConcentration": {
            "Label": "caconc",

            "AlphaFormula": {
              "Type": "ExponentialByExponential",
              "NumVarOffset": 0,
              "NumVarScale": 0,
              "NumScale": 0,
              "NumOffset": 0,

              "DenVarOffset": 0,
              "DenVarScale": 0,
              "DenScale": 0,
              "DenOffset": 1,

              "Offset": 3
            },
            "BetaFormula": {
              "Type": "ExponentialByExponential",
              "NumVarOffset": 0,
              "NumVarScale": 0,
              "NumScale": 0,
              "NumOffset": 0,

              "DenVarOffset": 0,
              "DenVarScale": 0,
              "DenScale": 0,
              "DenOffset": 1,

              "Offset": 0.075
            },

            "CurrentGating": "GateVariable",
            "CurrentGatingExponent": 1,

            "AlphaGating": "CompartmentVoltage",
            "Dynamics": "ConcentratedCalcium",

          ”InitialState”: 3.7152
        },

        ”IonChannels”: [
          {
            ”Label”: ”calcium_channel”,

            ”LeakConductivity”: 4.5,
            ”LeakConductivity_Randomization_offset”: 0,
            ”LeakInversionPotential”: 120,
            ”LeakInversionPotential_Randomization_offset”: 0,
            ”IsCalciumChannel”: true,

            ”Gates”:[
              {
                ”Label”: ”r”,
                ”AlphaFormula”: {
                  ”Type”: ”ExponentialByExponential”,

                  ”NumVarOffset”: 0,
                  ”NumVarScale”: 0,
                  ”NumScale”: 0,
                  ”NumOffset”: 0.34,

                  ”DenVarOffset”: -5,
                  ”DenVarScale”: -0.071942,
                  ”DenScale”: 1,
                  ”DenOffset”: 1,

                  ”Offset”: 0
                },
                ”BetaFormula”: {
                  ”Type”: ”LinearByExponential”,

                  ”LinearVarOffset”: 8.5,
                  ”LinearVarScale”: -0.004,

                  ”ExponentialVarOffset”: 8.5,
                  ”ExponentialVarScale”: 0.2,
                  ”ExponentialScale”: -1,
                  ”ExponentialOffset”: 1,

                  ”Offset”: 0
                },

                ”CurrentGating”: ”GateVariable”,
                ”CurrentGatingExponent”: 2,

                ”AlphaGating”: ”CompartmentVoltage”,
                ”Dynamics”: ”Classical”,

                ”InitialState”: 0.0112788
              }
            ]
          }
        ]
```

```
                    }
                ]
            }
        ]
    }
}
```

C

Metadata output-file example

Example of a Real experiment metadata output file, from exepriments at CSCS AULTs development cluster

```
{
  ”ProcessorName”:  ”hsw226”,
  ”GPUcount”: 4,
  ”Timing”: {
    ”NetworkSetupTime”: 2.7946760654449463,
    ”NetworkSynapticNetworkGenerationTime”: 2.7946760654449463,
    ”MPISetupTime”: 39.625694990158081,
    ”OutputWriteTime”:  0,
    ”ComputeLaunchTime”:  0.023003101348876953,
    ”MPICommunicationTime”: 0.27750825881958008,
    ”LaunchWriteThread”:  6.4373016357421875e-06,
    ”Synchronizingtheloop1”:  2.2334461212158203,
    ”Synchronizingtheloop2”:  0.0056474208831787109,
    ”SimulationTime”: 2.5397908687591553,
    ”GPU0: Tesla V100-PCIE-32GB”: {
      ”time_Igap”:  2.2188638019561768,
      ”time_Icalc_gate”:  2.1898284549713134,
      ”time_Icalc_comp”:  0.0061534080244600772
    },
    ”GPU1: Tesla V100-PCIE-32GB”: {
      ”time_Igap”:  1.9870404148101808,
      ”time_Icalc_gate”:  1.9559014072418213,
      ”time_Icalc_comp”:  0.0058584639988839629
    },
    ”GPU2: Tesla V100-PCIE-32GB”: {
      ”time_Igap”:  1.9837053089141845,
      ”time_Icalc_gate”:  1.9539980487823487,
      ”time_Icalc_comp”:  0.0057299520149827008
    },
    ”GPU3: Tesla V100-PCIE-32GB”: {
      ”time_Igap”:  1.9889988174438478,
      ”time_Icalc_gate”:  1.9576277523040773,
      ”time_Icalc_comp”:  0.0057336320057511329
    }
  },
  ”Memory”: {
    ”GPU0: Tesla V100-PCIE-32GB”: {
      ”Total_global_memory”:  34089730048,
      ”Memory_used”:  19855835136
    },
    ”GPU1: Tesla V100-PCIE-32GB”: {
      ”Total_global_memory”:  34089730048,
      ”Memory_used”:  19855835136
    },
    ”GPU2: Tesla V100-PCIE-32GB”: {
      ”Total_global_memory”:  34089730048,
      ”Memory_used”:  19855835136
    },
    ”GPU3: Tesla V100-PCIE-32GB”: {
      ”Total_global_memory”:  34089730048,
      ”Memory_used”:  19855835136
    }
  },
  ”Connections”:  {
    ”nCells”: 1048576,
    ”nCells_local_node”:  524288,
    ”nCells_local_GPU ”:  131072,
    ”Totaal_Connections”: 4947834207,
    ”Xternal_Connections”:  524288,
    ”Avg_density”:  0.00900005828589201,
    ”Target_denisty”: 0.008999999612569809,
    ”Percentage_diff_denisty”:  6.5192584770557005e-06
  },
  ”V_check [0,0]”:  -60.012969970703125
}
```