On Application Design for Manycore Processing Systems in the Domain of Neuroscience

Over applicatieontwerp voor manycoresystemen in het domein van neurowetenschappen

Thesis

to obtain the degree of Doctor from the
Erasmus University Rotterdam
by command of the
rector magnificus

prof. dr. R.C.M.E. Engels

and in accordance with the decision of the Doctorate Board.
The public defence shall be held on Tuesday 8 December 2020

at 9:30
by

Georgios Chatzikonstantis
born in Athens, Greece.

Erasmus University Rotterdam

Doctoral Committee:

Promotors:	prof. dr. C.I. de Zeeuw prof. dr. D. Soudris
Other members:	dr. Z. Gao dr. ir. Z. Al-Ars dr. D. Pnevmatikatos
Copromotor:	dr. C. Strydis

Abstract

In recent years, the rapidly growing field of human neuromodelling has undergone significant changes. Neuroscientists have been taking impressive steps in unveiling the elaborate functionality of the human brain. In doing so, complex mathematical models have been the focus of efforts for describing detailed electrochemical processes that govern the human brain's behaviour. Such efforts require tremendous computational power in order to render, simulate and analyze in traditional computing systems. As such, the field of computational neuroscience presents an imposing challenge that the realm of high performance computing is tasked with meeting.

The evolution of our understanding and mapping of the human brain has been accompanied by a steady increase in the processing power made available in manycore processors. Processors such as Intel's Xeon Phi line of products have grown to incorporate more advanced computing capability over the years. Due to their nature, they also provide traditional parallel coding tools, which can significantly impact the ease at which applications can be developed, tested and marketed. As a result, manycore CPUs are presented as an attractive alternative to other high-performance computing fabrics, such as GPUs and FPGAs.

In this Doctoral thesis, we investigate the impact that manycore processors can have in the domain of computational neuroscience, specifically from the viewpoint of high-detail neuromodelling. By identifying a lack of research efforts in high-performance, large-scale, detailed neuronal simulations, the thesis presents the development of a simulator rich 'in biophysical detail in manycore x86-based processors. Furthermore, the simulator acts as a means to study how manycore processors have evolved in architecture and behaviour, as well as highlight their strengths and drawbacks, in an effort to understand the role that they can play in the landscape of high-performance neuromodelling.

This Doctoral thesis presents the complete development effort of the aforementioned simulator. The effort commences with an application specifically designed for the earliest, experimental manycore processors. We meticulously re-configure the simulator and its implementation design in order to take advantage of the continuously evolving architecture of manycore processors. Through this process, the simulator incorporates more modelling options, supports a wider range of simulation parameters and operates on a scalable, modern manycore system. The end product of this thesis is a simulator that constitutes an efficient solution for studying demanding neuronal models, in terms of both performance and energy. The thesis starts with a design that can simulate an average network of $50k$ neurons and $2million$ synapses in 40 minutes for every second of simulated brain activity; the final design on a modern, small cluster of manycore

processors vastly improves on this design by simulating $2million$ neurons and $2billion$ synapses in under 10 minutes for every second of simulated brain activity.

Our point of focus and contributions lie in the analysis of manycore processor performance when tasked with demanding neuromodelling workloads. Through the proposed simulator, we highlight how the significant wealth of neuromodelling parameters affects simulation in different manycore processors. The thesis will provide a clear map on matching the correct amount, and type, of hardware to different network simulation configurations. As such, we take an important step towards defining proper utilization of high-performance hardware in order to match simulation challenges imposed by the domain of computational neuroscience.

Furthermore, significant effort is expended in incorporating the simulator in a larger, collaborative framework aimed at serving as an online resource for high-performance neuromodelling simulations. The designed framework, named "BrainFrame", leverages a heterogeneous ensemble of accelerators, namely manycore processors, FPGAs and GPUs, in order to provide efficient solutions for different modelling and network configurations. We provide a proof of value in the framework by identifying different use cases where a switch in the underlying accelerator hardware yields significant gains in performance, thus reinforcing the value of heterogeneity in high-performance neuromodelling. In particular, the framework further highlights the value of matching the accelerating hardware of choice with the computational workload at hand by indicating differences in performance by orders of magnitude between the different accelerators examined in different network configuration scenarios.

Περίληψη Στα Ελληνικά

Στα πρόσφατα χρόνια, το ταχύτατα εξελισσόμενο πεδίο της μοντελοποίησης του ανθρώπινου εγκεφάλου παρουσίασε σημαντικές εξελίξεις. Οι νευροεπιστήμονες ανά τον κόσμο σημείωνουν εντυπωσιακά βήματα στην χαρτογράφηση του λεπτομερούς τρόπου λειτουργίας του ανθρώπινου εγκεφάλου. Στην προσπάθεια τους αυτή, αναπτύχθηκαν πολύπλοκα μαθηματικά μοντέλα τα οποία επιτρέπουν την περιγραφή και μελέτη λεπτομερών ηλεκτροχημικών διεργασιών που διέπουν τη συμπεριφορά του εγκεφάλου. Τέτοιες προσπάθειες συνοδεύονται από εξαιρετικά μεγάλα υπολογιστικά φορτία προκειμένου να προσομοιωθούν και να αναλυθούν τα απαραίτητα δεδομένα, συνήθως δε σε απλά υπολογιστικά συστήματα. Ως εκ τούτου, το πεδίο της υπολογιστικής νευροεπιστήμης παρουσιάζει επιβλητικές προκλήσεις που η επιστήμη της υπολογιστικής επεξεργασίας υψηλής ισχύος καλείται να απαντήσει.

Η εξέλιξη της κατανόησης και της χαρτογράφησης του ανθρώπινου εγκεφάλου έχει συνοδευθεί από μια σταθερή αύξηση στη διαθέσιμη υπολογιστική ισχύ που προσφέρουν οι πολυπύρηνοι επεξεργαστές. Επεξεργαστές όπως ο Xeon Phi της Intel εξελίσσονται διαρκώς ώστε να διαθέτουν αυξανόμενη υπολογιστική ικανότητα. Χάρη στον σχεδιασμό τους, προσφέρουν την ικανότητα προγραμματισμού με παραδοσιακά εργαλεία λογισμικού παράλληλης επεξεργασίας. Η δυνατότητα αυτή επηρεάζει σημαντικά την ευκολία με την οποία οι εφαρμογές λογισμικού μπορούν να αναπτυχθούν, να ελεγχθούν για την ποιότητά τους και να προωθηθούν στην αγορά. Ως αποτέλεσμα, οι πολυπύρηνοι επεξεργαστές αποτελούν μια ενδιαφέρουσα εναλλακτική οδό σε σύγκριση με άλλα εδραιωμένα συστήματα υψηλής επεξεργαστικής ισχύος, όπως οι κάρτες γραφικών και τα FPGA.

Η παρούσα διατριβή ερευνά την αποτελεσματικότητα των πολυπύρηνων επεξεργαστών στα προβλήματα που απαντώνται στον τομέα της υπολογιστικής νευροεπιστήμης, κυρίως στο κομμάτι της μοντελοποίησης νευρώνων με μεγάλο βαθμό λεπτομέρειας. Μετά την τεκμηρίωση μιας ανιχνευμένης έλλειψης έρευνας σε προσομοιώσεις μεγάλων, λεπτομερών δικτύων νευρώνων σε υψηλής επεξεργαστικής ισχύος συστήματα, η διατριβή παρουσιάζει την ανάπτυξη ενός προσομοιωτή με έμφαση στη μοντελοποίηση βιοφυσικής λεπτομέρειας σε πολυπύρηνους επεξεργαστές αρχιτεκτονικής x86. Ο προσομοιωτής επίσης δρα ως αντικείμενο μελέτης για το πώς οι πολυπύρηνοι επεξεργαστές έχουν εξελιχθεί σε αρχιτεκτονική και συμπεριφορά, καθώς και βοηθά στην ανάλυση των δυνατών και αδύνατων σημείων των πολυπύρηνων επεξεργαστών, ώστε να διερευνηθεί ο ρόλος που μπορούν να έχουν στον τομέα της μοντελοποίησης δικτύων νευρώνων με μεγάλη υπολογιστική ισχύ.

Η διατριβή παρουσιάζει ολοκληρωμένα την ανάπτυξη του προαναφερθέντος προσομοιωτή. Το έργο ξεκινά με μια εφαρμογή ειδικά σχεδιασμένη για τους πρώτους πειραματικούς πολυπύρηνους επεξεργαστές. Καθώς η πολυπύρηνη αρχιτεκτονική εξελίσσεται, επαναπροσδιορίζουμε τις παραμέτρους και τον σχεδιασμό του προσομοιωτή με ακρίβεια προκειμένου να εκμεταλλευθούμε τις προόδους της εξελισσόμενης τεχνολογίας. Καθ' όλη τη διαδικασία, ο προσομοιωτής ενσωματώνει περισσότερες επιλογές μοντελοποίησης, υποστηρίζει μεγαλύτερο εύρος παραμέτρων προσομοίωσης και τελικά, λειτουργεί σε ένα σύγχρονο, κλιμακώσιμο

πολυπύρηνο επεξεργαστικό σύστημα. Το τελικό προϊόν αυτής της διατριβής είναι ένας προσομοιωτής που αποτελεί μια αποτελεσματική λύση για τη μελέτη απαιτητικών μοντέλων νευρώνων, τόσο από άποψη υπολογιστικής επίδοσης αλλά και καταναλώσης ενέργειας.

Οι προσφορές μας στην επιστήμη του τομέα εστιάζονται στην ανάλυση των επιδόσεων των πολυπύρηνων επεξεργαστών όταν καλούνται να επεξεργαστούν μαθηματικά μοντέλα νευρώνων. Μέσω του προτεινόμενου προσομοιωτή, αναδεικνύουμε πώς το ευρύ φάσμα των παραμέτρων μοντελοποίησης νευρώνων επηρεάζει την προσομοίωση σε πολυπύρηνους επεξεργαστές. Ως εκ τούτου, κάνουμε ένα σημαντικό βήμα προς την αποτελεσματική αξιοποίηση υπολογιστικών συστημάτων υψηλών επιδόσεων με σκοπό την αντιμετώπιση των προκλήσεων που επιβάλλονται από τον τομέα της υπολογιστικής νευροεπιστήμης.

Σημαντικό κομμάτι της διατριβής ασχολείται με την ενσωμάτωση του προσομοιωτή σε μια ευρύτερη, συνεργατική, διαδικτυακή πλατφόρμα που στοχεύει στην εκτέλεση προσομοιώσεων μοντέλων νευρώνων με υψηλή επεξεργαστική ισχύ. Η πλατφόρμα που παρουσιάζεται, με την ονομασία "BrainFrame", αξιοποιεί ενα ετερογενές σύνολο από επιταχυντές, συγκεκριμένα πολυπύρηνους επεξεργαστές, κάρτες γραφικών και FPGA, προκειμένου να δώσει αποτελεσματικές λύσεις για διαφορετικές περιπτώσεις μοντελοποίησης και παραμέτρων δικτύου νευρώνων. Αποδεικνύουμε την αξία της πλατφόρμας μέσω της ανίχνευσης διαφορετικών περιπτώσεων προσομοίωσης όπου μια αλλαγή στον επιταχυντή που εκτελεί την προσομοίωση προσφέρει σημαντικά κέρδη στην ταχύτητα προσομοίωσης, υπογραμμίζοντας έτσι την αξία της ετερογένειας στην προσομοίωση μοντέλων νευρώνων με μεγάλη επεξεργαστική ισχύ.

Acknowledgements

The entirety of my journey to the depths of academic research has been supervised by my professor Dimitrios Soudris, without whom not a single page of this document would have been made possible.

Three distinct people have aided me tremendously in the past 6 years of my research. Dr. Dimitrios Rodopoulos, Dr. Harry Sidiropoulos and Dr. Christos Strydis have been the best mentors and advisors anyone could ever ask for. They are also pretty cool people to hang out with.

I've had the pleasure of working closely with a good number of collaborators for my thesis. Soon-to-be-Dr. George Smaragdos, Prof. Mario Negrello, Konstantinos Katsantonis and Sotiris Panagiotou have been exemplary colleagues and helped shape this Doctoral thesis, particularly for our joint project of BrainFrame. Colleagues Alexandros Neofytou and Ioannis Magkanaris have contributed significantly in our collaborative efforts; it was a joy to help them graduate. Colleague Sofia Nomikou has helped me lay the groundwork on top of which I built my thesis.

I would like to acknowledge many people from both NTUA Microlab and Erasmus MC. A few distinct people amongst many are Prof. Chris I. De Zeeuw, Dr. Maro Baka, Dr. Dimitris Anagnostos, Dr. Robert Seepers, Dr. Georgios Zervakis, Dr. Sotiris Xydis, Konstantina Koliogeorgi and Zefi Skini.

I owe my eternal gratitude to my mom Despoina, my dad Theodoros, my brother Antonis, my grandfather Georgios and my beloved, sweet late grandmother Barbara. Thank you for supporting me for 30 years and loving me unconditionally.

May my furry best pal Homer forever chase after balls and bark at vacuum cleaners in dog heaven.

Finally, no man walks alone. I would like to close this section with the names of my dearest friends who have helped me shape who I am as a person and supported me emotionally through this tough journey: Dimitris, Vasilis, Thodoris, Mary, George, Ira, Kostas, Konstantinos, Julia and Daphne.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Computational Neuroscience

1.1.1 Challenges

Computational neuroscience is an interdisciplinary field which encompasses studies of the brain's functionality and cognitive operations in conjunction with the advances of modern computer science. The field represents the advancement of knowledge concerning the world's most crucial phenomena, such as the human brain, through the aid of modern technology.

The last decade has witnessed a great amount of advances in the field of computational neuroscience. Neuroscientists have been gradually unveiling details of neuron operation. Using this knowledge, there is a wide research interest in studying the behaviour of single-neurons, as well as small networks of neurons and eventually brain-sized populations of neurons. Simulating these neuronal networks on various platforms is an active field of research; a major challenge is the sheer computational complexity that many of these neuron models entail.

Mapping the human brain, discovering its functionality and replicating its behaviour are all endeavours that entail multiple challenges and obstacles to overcome. The human brain is a particularly large and complicated organ. Compared to the rest of our biological organs, human knowledge is still in its infancy regarding how our brains operate. These factors lead to the existence of multiple mathematical approaches in an effort to model brain functionality. Each modelling effort has its own merits and shortcomings and all share a common trait: they require massive amounts of computations in order to simulate a meaningful portion of the brain's operations.

Massive computing power is necessary to calculate neuronal processes and their interactions, since our brain is comprised of billions of neurons [6]. The amount of information

that is exchanged within our formed neuronal networks is also vast due to the large amount of synapses, the "bridges" that connect the aforementioned neurons. This great volume of data needs to be modelled and interpreted in order to gain a grasp on how the human brain functions at a low, neuronal level.

The massive volume of calculations that are undertaken by the human brain is not the sole challenge that renders the neuroscientific field difficult to navigate. As science delves deeper into the complex nature of the human brain, hopes of replicating parts of its activity arise; and so does the need to design simulation systems that can respond within logical time limits. Ideally, the tools used to probe brain functionality would operate within the same time scale as our brains in real life. Such tools would lessen experimentation setup times and aid neuroscientists in designing complex, in-silico experiments that help unravel how the brain responds to real-time stimuli. In the future, designing simulation systems that survey and process the conditions under which a brain operates in real time can be a crucial step to achieving implantable monitoring devices.

Furthermore, computational neuroscience is a relatively young field of science. The first book produced by the field was only published in 1990 and edited by Eric L. Schwartz [7]. As such, it is a rapidly evolving field full of potential, with new approaches to new obstacles being invented frequently. Neuroscientists have a lot of options in their arsenal when tasked with studying a certain phenomenon and choosing which avenue to pursue for their given task and circumstances can be a daunting problem. This is a common problem in the discipline of mathematical modelling, where different degrees of complexity and scale can expose or obscure details of the studied phenomenon. Thus, the goal of exploring each modelling option's benefits and shortcoming becomes increasingly relevant in the young field of computational neuroscience.

While computational neuroscience is a young and emerging field, the domain of neuroscience itself has a long and rich history. Neuroscientists have amassed an impressive amount of knowledge concerning the human nervous system, its physiology, anatomical details and biology. This vast knowledge needs to be updated and translated to formats that are suitable to utilize in today's processors. The challenge of porting the existing wealth of resources to modern, cutting-edge processing systems is far from a trivial task. It is, however, a necessary task in order for the field of computational neuroscience to be able to exploit the technological advances of high-performance computing.

1.1.2 Motivation

The intersection of two different domains, neuroscience and computer science, is aptly named "Computational Neuroscience" and aims at revolutionizing what is possible in the field of human brain studies.

In-vitro or in-vivo neuroscientific experiments are very costly, time-consuming and require testing on animals. They are also highly complex, often difficult to reproduce and offer limited access. Hence, constructing and exploring realistic simulations of biological neural networks on computing platforms has come across as a very viable and useful alternative for neuroscientists in recent years [8, 9]. The resulting discipline of computational neuroscience comprises a powerful tool in the hands of the community. It helps elucidate fundamental brain mechanisms underlying many obscure neurological maladies and guide possible new therapies. In-vivo and in-vitro experiments, while being traditional and powerful experimentation tools for neuroscience, inherently present the possibility of the experimental data to become contaminated (from factors such as the effects of anesthesia). Many of the complex brain-system dynamics that define biological behavior are hypothesized and many in-vivo or in-vitro techniques are not always able to provide the means to validate them. Such challenges further increase the value of in-silico experimentation.

For decades, scientists have been fascinated by the methods and computational capabilities of the biological brain. The US National Academy of Engineers has listed the simulation of the human brain as one of the Grand Engineering Challenges. Inspired by the scientific effort, engineers began to copy computational concepts found in the brain, which led to the creation of the first Artificial Neural Networks (ANNs) with the creation of the perceptron. ANNs do not execute commands sequentially like the typical Von Neumann computer, but each node (or neuron) in a neural network is a separate set of functions and they are all evaluated concurrently during execution. The relation between input and output is defined largely by the network size, topology and interconnectivity of the neurons. Interconnectivity could eventually be adaptive, thus, mimicking the behavior of biological systems. Eventually, more advanced versions of neural-network models were developed, based upon greater understanding of the biological processes. Spiking Neural Networks (SNNs) do not abstractly mimic biological-neuron behavior but outright simulate the computational behavior of brain processes. True to their biological counterparts, SNNs have the ability to encode information using the transfer rate, amplitude and spike-train patterns, which gives them more capabilities than traditional ANNs. As a result, they are currently heavily used to model the complex behavior of biological-brain systems in neuroscientific research.

Regardless of the plethora of models and execution platforms, there has been reduced emphasis on systematically optimizing the model execution on a many-core platform. Given the capabilities of modern computing systems, the degrees of freedom that are available to the community allow an aggressive exploration of the performance vs. quality-cost trade-off. This enables cost-conscious execution of biologically accurate neural models on multi-/many-core systems. There is significant potential using such systematic approaches, given the magnitude and energy budget [10] of computing infrastructure employed for brain modeling [11].

Ultimately, computational neuroscience attempts to use neuronal network models of various complexities, the accuracy of which can provide predictive behavior and insight, and exploit them to guide further biological experiments. Long-term advancement in computational neuroscience can hopefully lead to improved medical treatment of brain-related health issues, novel artificial-intelligence applications and groundbreaking computer architectures.

Computational neuroscience can advance our understanding of human cognitive abilities and brain functionality, as well as reveal opportunities for evolving the nature of computer architecture. On one hand, mapping the human brain allows for detecting the cause of brain-related health issues and possibly, reveal methods for repairing damage caused to our nervous system. On the other hand, there have been emerging processing systems designed after certain aspects of our brains' method of operation. Artificial intelligence has been growing in popularity and applicability in different domains and novel computer architectures seek to emulate the human brain's high degree of intercommunication and multi-tasking.

Despite momentous achievements in the simulation of large scale neural systems, the path ahead is no less daunting. In the last decade, the computational neuroscience literature has seen the publication of brain scale models that include numbers of neurons comparable with those of biological systems, or patches of brain with high level of detail. Izhikevich and Edelman [12] simulated the whole thalamocortical system with quadratic 2D models and simple synapses, the Blue Brain Project has simulated detailed networks of a whole reconstructed cortical column with compartmental models and detailed synaptic models [13], as well as Erik De Schutter et al, who produced a highly detailed model of the cerebellar granular layer [14] . Going forward, it is the stated goal of the human brain project of expanding on the work of the Blue Brain Project and simulating a whole brain. Many other large scale reconstruction and analysis projects should be expected in the future, examining both larger neuron populations and more detailed neuron models [15].

The projects named above should be taken as isolated 'proofs of principle', and even if the authors have searched parameter spaces, the parameter space of possible networks has barely been scratched. The goal of computational neuroscience is not only to simulate a single column or even brain, but enormous classes of possible virtual brains. Making matters worse, it is likely that the future will demand that these brains be hooked to sensors and actuators and be required to function in real time and closed-loops.

This type of work pushes multiple boundaries of knowledge and technology. On the knowledge front, it commits the computational neuroscientist to a level of detail of the representation that exposes the free or unknown parameters of the system. This includes both the procurement of biological data, and the exploration of the gigantic parameter spaces. In fact, biological measurements of neuronal parameters can only take us so far, since neural network parameter spaces are far from convex [16, 17], hence simply

measuring biophysical properties of neurons will not be sufficient to recreate plausible neurodynamics. To make matters worse, biological neurons are in continuous change [18], and future brain models will need to tackle the problem of changeability as well, introducing yet another level of computational demands on the simulation.

A caveat of large scale simulations often put forth is that the correct level of detail for simulating brains is not known. This alone should be taken as justification for maintaining an agnostic view on the "a priori" required level of detail of the simulation. It is not inconceivable that future models will continue to biological detail that is relevant in particular scientific domains, and hence this agnosticism is commendable. The best means to define that required level of detail is in the simulation of large scale systems and the comparison with reduced version, to gauge the contribution of the extra amounts of detail. The work of reducing a model to its essentials, often passes through understanding the implications of more complex assumptions, and hence, to simplify one often has to complexify.

Hence, we should predict that the computational requirements for future neurocomputational models will demand ever increasing computational resources, particularly in the problems of parameter space exploration, large network homeostasis and real time embedding of brain sized simulations.

1.2 Manycore Computing Systems

1.2.1 Transition from Single-Core to Manycore

Traditionally, single-threaded programming has been the go-to programming model for most research-oriented applications, including modelling. Single-threaded is a relatively simpler programming paradigm which naturally fits the design of the modeller. Specifically in the realm of neuromodelling, GENESIS and Matlab have been extensively used to cater to the needs of the community [19–22]. Processors largely remained single-core as long as they were sufficient for the needs of the industry and academic research. However, with processors reaching their power consumption limits [23], it became increasingly apparent that continuous escalation of a single-core's clock frequency would be unsustainable and different avenues for gaining more processing power out of computing systems needed exploration.

Multi-core architectures have been a large step in the evolution of high-performance computing, as well as algorithm design. In 2001, IBM's Power 4 processor [24] is the first step for the well-established semiconductor chip maker in the realm of incorporating more than one core on a single die. In order to take advantage of having multiple cores on a chip, the paradigm of parallel processing was made necessary. While the concept of parallel programming greatly predates the design of industry-grade multicore

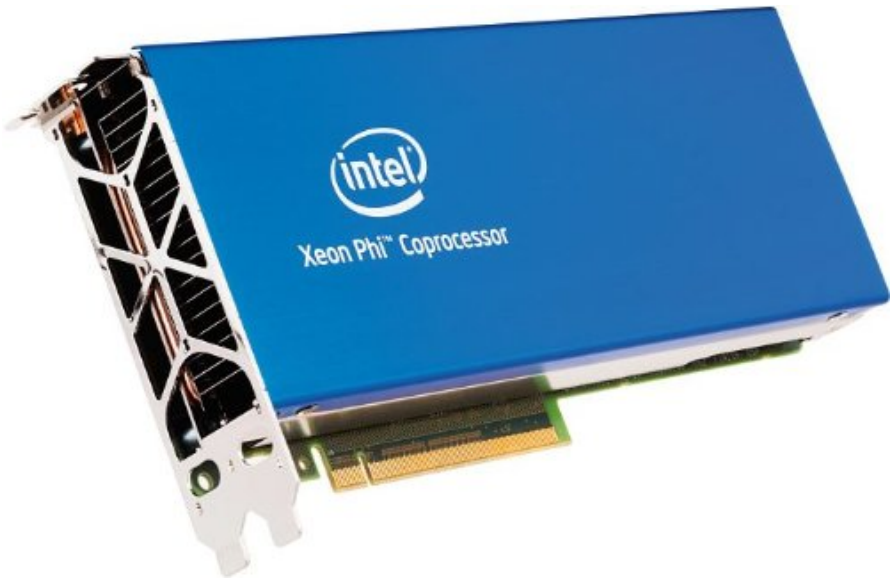


FIGURE 1.1: Image of the Xeon Phi Knights Corner Coprocessor, the first industry-grade manycore processor designed by Intel [1]. This particular model is a 7120p, a PCI-Express card that depends on a processor host in order to boot.

architectures [25, 26], a systematic approach to parallel coding and the corresponding shift in the coding paradigm was made more prevalent once keeping up with Moore's Law became an increasingly difficult goal to attend via single-core chips.

Multi-core processors focus on a mixture of single-threaded performance and a limited amount of independent cores available on the die for parallel processing. The cores remain relatively complex, integrating most of the technological advancements that decades of single-threaded processing have added to engineering knowledge. Multi-core processors continue to evolve and take up a large share of the modern processor market [27], increasing in processing power as well as energy efficiency [28].

As the necessity for massive parallel processing of unprecedented amounts of data grew in recent years in various fields, such as communication [29], finance [30] and life sciences [31], the industry responds with increased availability of computational resources on chip. As a continuation of Moore's Law in a different form, the amount of cores integrated in a single die has risen exponentially, with supercomputers making the Top 500 list (the "who-is-who list in the field of high performance computing" [32]) comprised of processors with continuously increasing cores-per-die. The term "manycore" computing has been coined for a sub-category of multi-core processors which place an emphasis on their high degree of parallel processing power, offering tens to thousands of independent processor cores and frequently stressing the quantity, over the complexity of their available computational resources.

1.2.2 Manycore Systems Progression

Manycore computing systems have continuously increased the density of their computational power over the years; the amount of CPUs that can "fit" in a single die has been an escalating number over the past decade. An interesting case in manycore systems evolution has been Intel's line of products in the manycore genre.

The semiconductor chip manufacturer Intel started an early project in 2007 code-named Intel Tera Scale Computing Research Program, which aimed at providing its customers with unprecedented, for its era, amounts of potential processing power. A first prototype that emerged out of this project was the 80-core Polaris chip [33] which was aimed towards research and allowing engineers to work with massively parallel applications in practice, rather than in theory. The die contained 80 routers for its core communication needs and produced great peak computational performance per watt.

However, not all early members of the Tera Scale Program were successful products. Larrabee was a manycore chip originally designed to act as a GPU [34]. Larrabee used multiple in-order x86-based CPU cores that featured vector processing units for single-instruction multiple-data (SIMD) commands. In 2010, the chip was discontinued shortly after its production, unsuccessful in its original intended purpose of rivalling general-purpose GPUs [35].

A significant entry in the list of early manycore processors is the Single-Chip Cloud Computer (SCC), announced in 2010 [36]. The 48-core processor held interesting features such as adjustable scaling of operating voltage and frequency levels, as well as the ability to code using familiar parallel computing paradigms. Since its cores did not share memory, a message-passing API code-named RCCE [37] was used that had significant similarities to the well-established MPI library [38]. The product was shipped to multiple research facilities and universities for experimental development and has been extensively used in many academic fields, long after the discontinuation of its production [39, 40].

The early products of the Tera Scale Program culminated in Intel's "Xeon Phi" line of manycore processors. The processors focus on the availability of parallel computing resources and their ease of programming. The Xeon Phi products are based on x86 cores, which support traditional coding tools, thus being able to run software that has been developed for regular, industrial CPUs with minimal effort. Their ease of usage, which increased as the products matured and more generations were produced, allowed for an attractive alternative to the high-performance-computing standard platform, the GPUs.

The prototype of Xeon Phi products was the Knights Ferry platform, produced in 2010 [41]. The chip is a derivative of the Larrabee prototype and was offered as a PCIe card with 32 in-order cores at up to 1.2 GHz with four threads per core, 2 GB GDDR5 memory

and 8 MB coherent L2 cache (256 KB per core with 32 KB L1 cache). It was built at a 45nm process and had a power consumption of approximately 300W. Despite supporting only single-precision floating point instructions, its high peak processing power at 750 GFLOPS allowed it to be used for research purposes by establishments such as CERN [42].

Following the Knights Ferry prototype, the Xeon Phi line of products officially recognizes the Knights Corner chip as the first generation of the Xeon Phi manycore processors [1]. The Xeon Phi Knights Corner is a manycore co-processor; it is designed as a Pci-Express card that requires a host in order to boot. However, after booting, the co-processor can act autonomously, is reachable via SSH protocols through its host and operates with a custom, down-to-basics lightweight Linux image. The focus of the co-processor lies in its wealth of computing resources, with 61 available cores and 4 threads available per core, as well as a 512-bit-wide vector processing unit for SIMD instructions. The processor becomes the object of experimentation for many research fields [43–45].

The most mature product in the Xeon Phi line of processors was the Xeon Phi Knights Landing [4]. This manycore processor was a stand-alone processor offering significant peak performance and energy efficiency. Variations of the processor as a Pci-Express accelerator card were also produced, but discontinued and not offered to consumers. The processor had a significant impact in the industry, making the top ranks of Top500 supercomputer lists [46]. The Knights Landing processor offered more resources than its predecessors, simplified inter-core communication schemes by using a well-established mesh interconnection pattern and its instruction set allowed for binaries compiled for any x86-based architecture to run on the Knights Landing without any modifications. Its ease of usage, its efficiency and peak performance were marketed as the platform's strong points, particularly against competing general-purpose GPUs.

The Xeon Phi line of products was officially announced to be discontinued by Intel in 2018 [47]. However, the manufacturer is expected to fabricate manycore platforms in its upcoming, at the time of writing, Xeon processor line. Manycore processors have already been made widely available to the market by AMD via their "Ryzen Threadripper" line of products [48], based on AMD's "Zen" architecture [49], processors integrating up to 32 CPU cores and 64 threads in a single chip. In an effort to maintain market share dominance, Intel's "Skylake" architecture products are projected to integrate 28 cores and 56 threads on chip. The products maintain architectural design aspects from the Xeon Phi Knights Landing, such as the interconnection fabric "Omni-Path" for low-latency inter-node communication in multinode systems, an AVX-512 instruction set and vectorization processing units [50]. With the volume of CPU cores offered by a single chip being being continuously driven higher by important chipset manufacturers, manycore processing will remain a valuable resource for high-performance computing in the future.

1.2.3 Modern Usage

Manycore processors are utilized in a variety of domains today. As the need for parallel processing of demanding datastreams continues to grow, manycore processors and GPUs offer an abundance of processing cores to be utilized in parallel fashion in order to satisfy the computational needs of such applications. There is a variety of manycore processor architectures being designed and used today.

An emerging category of specialized, manycore processors derive from neuromorphic chip design, a design philosophy that seeks to emulate the brain's functionality in order to solve problems in an efficient and reliable way. Such chips replicate the interactivity of thousands to millions of neural cells by integrating massive amounts of processing cores on a single die. A prominent example of this class of processors is IBM's TrueNorth [51, 52], a chip integrating more than 4,000 cores and a total of approximately 1 million programmable artificial neurons. The chip aims at great energy efficiency and has been successfully used for artificial-intelligence-oriented applications. Another manycore processor from the domain of artificial intelligence is Eyeriss [53], designed for the promising class of deep convolutional neural networks.

Supercomputers continue using manycore processors as their basis for achieving peak performance approaching the scale of Exaflops. One of the most dominant supercomputers at the time of writing, the Sunway TaihuLight [54] chinese supercomputer hosts a wide range of applications in its manycore RISC processors codenamed SW26010 [55–58]. The supercomputer's RISC processors allow for good performance per watt.

The Xeon Phi line of products, particularly the Knights Landing, is still used extensively in the industry. Since they support traditional coding tools, such as OpenMP [59] and MPI [38] libraries, their ease of usage marks the, now discontinued, platforms as attractive choices for researchers globally. Along with its processing power, the Knights Landing processors host simulators for various natural phenomena [60], complex mathematical calculations [61] and life-science-oriented applications [62].

For manycore processors, ease of usage plays an important role in their adoption rate. Today, a number of programming paradigms are available for manycore architectures. Specialized hardware, such as IBM's Truenorth which specializes in artificial-intelligence-related research, utilizes specific programming models designed for its intended usage [63]. Other manycore processors use general-purpose programming frameworks. Shared-memory applications favour OpenMP, whereas multinode, split-memory applications use message passing protocols between nodes, such MPI. Heterogeneous systems and applications that seek portability in different accelerator engines, such as both GPUs and x86-based manycore systems, encourage the usage of frameworks such as OpenCL [64] for compatibility across different accelerator architectures.

Given the advent of specialized accelerator fabric in modern High Performance Computing infrastructure, it is imperative that applications are well understood, especially in the context of the accelerating platforms that are utilized. Additionally, these applications are to be used in scientific research that is very dynamic and many times conducted by non-HPC experts. Moving forward, the goal should not be to over-optimize them, but keeping the programming effort moderate, resulting to short development times, while providing sufficient performance. These factors can be critical in achieving widespread usage of manycore accelerators and their potential processing power in research.

1.3 Contributions in Computational Neuroscience

Challenge I

Current approaches for experimentation in the field of neuroscience can be placed under two categories, each with significant challenges to overcome:
(i) traditional *in-vivo* (or *in-vitro*) experimentation, which is very time-consuming, costly and requires significant lab equipment, training and experience by the researcher and
(ii) emerging *in-silico* experimentation, which has been mostly focusing on simpler modelling due to the computational demands of large-scale neuronal network simulation.

Solution I

This dissertation shall prove that efficient neuromodelling of high complexity, scale and detail can be achieved, as long as enough engineering effort is provided. The proposed system achieves simulation of $1second$ of brain activity in $10minutes$ of execution time of a very demanding workload ($2million$ neurons with $1k$ synapses per neuron of a detailed, conductance-based model). Furthermore, engineering insights will be given in order to help navigate future neuromodelling applications towards more efficient implementations, stressing neuromodelling parameters that significantly impact simulation speed and efficiency.

Challenge II

Although there already exist multiple frameworks for neuromodelling, those who are focused more towards biophysically-accurate model simulation are older tools, which can be difficult to use without prior engineering and programming knowledge. This can prove challenging for users from the traditional neuroscientific domain.

Solution II

This dissertation ultimately builds towards a simulation framework that works as an online service, available to any scientist and requires no engineering effort since it is entirely hosted on cloud computing services. Experimentation via the framework is achieved through a Python-based modelling software package that is widely used in the neuroscientific community. The thesis provides a detailed synopsis of the proposed framework, a product of collaboration between the Microprocessors and Digital Systems Lab of the National Technical University of Athens and Neurasmus B.V.

1.4 Contributions in Manycore Computing

Challenge III

Application design in manycore processors meets a significant challenge generated by overheads due to thread-level and inter-node communication. For non-embarassingly parallel workloads, coherence-imposed delays can be prohibitive, particularly in the case of neuronal networks simulating rapid brain activity.

Solution III

In this dissertation, delays imposed by synchronization between computational resources in both single- and multi-node manycore systems are examined through numerous approaches, highlighting the most efficient solution for a category of workloads which are especially demanding and communication-heavy. Through a combination of methods, an efficient implementation is achieved that utilizes a small cluster of manycore processors in order to reach a level of performance previously achievable only by much larger and more expensive computing systems. In addition to simulating large, demanding networks, the design proposed in this thesis provides, throughout the generations of technology researched in this paper, a steady performance speedup ranging from 10× to 50×, depending on network configuration, when compared to simulating on more traditional processors with fewer multithreading capabilities and smaller core counts.

Challenge IV

There is a wealth of options for acceleration platforms that offer better potential performance (FPGAs) or have established a wider usage rate (GPUs) compared to manycore computing fabrics. For any engineer working on the field high performance computing, the challenge of picking suitable hardware for her application is significant and the role of manycore computing fabrics in this endeavour is relatively unclear.

Solution IV

In this dissertation, the benefits and hindrances of manycore processors are well-defined and elaborately compared to other widely-used accelerator options. By integrating manycore computing in an ensemble of different accelerator fabrics, their potential usage in the emerging world of heterogeneous computing is highlighted. Important speedup gains can be derived from switching between different hardware as computational platforms depending on the configuration of the simulated network, where in some cases a successful switch can result in more than doubling (2×) the execution speed of the requested simulation. Furthermore, the designed framework presented in this dissertation operates on a cloud computing platform which can act as a prime candidate for efficient heterogeneous computing.

1.5 Overview of the Doctoral Thesis

- In Chapter 1, the reader was introduced to the domain of computational neuroscience. An overview of the appearance and evolution of manycore computing was then provided. The contributions of this Doctoral thesis were summarized, both in the field of neuroscience and high-performance computing.

- In Chapter 2, a review of the available relevant literature in the domain of computational neuroscience will be provided. The Chapter will commence with an analysis of the landscape of human neuromodelling. It will then introduce pre-existing research in computational neuroscience by examining significant modelling efforts and the respective tools used. The Chapter will place an emphasis on the simulation frameworks currently used by the neuroscientific community and highlights their advantages and shortcomings.

- In Chapter 3, the development of a biologically complex neuromodelling simulator will be presented. A brief biological background of the studied model will be given before going into detail concerning its implementation on a research-grade manycore single-chip processor. After disclosing the experimental processor's assets and configuration options, a detailed evaluation of the implementation's performance and efficiency will be provided.

- In Chapter 4, the evolution of the designed simulator through different generations of manycore processors will be thoroughly presented. The Chapter will introduce the different implementation options of a more flexible and scalable simulator on a 1st Generation Xeon Phi processor, followed by the simulator's optimized version on the processor. The Chapter will continue with the simulator's porting on the second Generation Xeon Phi processor and provide an analysis on a multinode implementation. Through elaborate experimental evaluation, the Chapter will highlight parameters that affect the simulator's behaviour and scalability.

- In Chapter 5, an heterogeneous computational system for neuromodelling will be studied. The Chapter will commence with an introduction to heterogeneous computing and online cloud services. The Doctoral thesis will then present an online, cloud-backed heterogeneous platform for high-performance neuromodelling simulations, as well as analyze its design and different layers. A thorough performance evaluation will highlight how heterogeneity can support the strengths and weaknesses of manycore processors in the domain of computational neuroscience.

- Finally, in Chapter 6, a summary of this doctoral thesis will be presented and future directions will be given.

Chapter 2

Related Work

2.1 Human Brain Modelling

2.1.1 Overview

There is a wide range of phenomena occuring during neuronal activity. Scientific efforts in unveiling the human brain functionality has lead to the utilization of mathematical modelling methods for neuron representation. As neuroscientists continue to unveil more details in neuronal electrochemical synthesis and functionality, their developed mathematical models attempt to integrate such discoveries in their behaviour. As such, multiple models with vastly different behaviours are constantly being developed and shared with the scientific world via online libraries, such as the widely-used by the neuroscientific community modelDB [65]. Models do not invalidate one another; rather, each model focuses on displaying different aspects of neuron behaviour.

A significant factor that contributes to this differentiation between neuronal models is their level of complexity and accuracy in depicting the functionality of an individual neuron. There are different conceptual layers based on which the complex neuronal structure can be studied. The deepest level of analysis can be found in molecular-level models, in which the "building blocks" for the neuron model are the molecules of each chemical substance that make up the neuron [66, 67]. On the opposite side of the spectrum, models that are mostly concerned with replicating network dynamics and inter-neuron communication can depict neurons, or clusters of, as single points in a graph or network [68].

As such, different models are suitable for different use-cases. While simpler models cannot offer the complete electrochemical picture of a neuron or synapse during a studied phenomenon, their level of abstraction can simplify the study of network behaviour under different circumstances. A suite of models that are mostly concerned with the timings and propagation of signals within a neuronal network is the Integrate-and-Fire models,

whereas models that aim at depicting the neuron's electrochemical properties belong to the Conductance-based category of models. Both categories have been given considerable attention in the literature; however, due to their simplicity, Integrate-and-Fire models have been experimented with to a larger degree.

2.1.2 Integrate-and-Fire Models

Models that primarily focus on network behaviour and spike generation are broadly categorized as Integrate-and-Fire (I&F) models. I&F models primarily focus in simulating spike responses and exchanges between neurons in networks. They tend to treat neurons as "points" in a network that communicate via their synapses, which may or may not have complicated mechanisms for their operation.

I&F models are the simplest spiking neural network (SNN) models, primarily focusing on receiving a spike input and determining the neuron's response based on a voltage threshold. They are widely used due to their simplicity and extensibility, resulting in a large range of I&F variants in the literature (e.g. leaky I&F [69, 70] and exponential I&F [71] models). This model category allows for customization of their functionality and extensions to their behaviour, so that they can represent multiple phenomena studied by researchers. As such, they are widely used by the neuroscientific community.

The idea of Integrate-and-Fire models can be attributed to Lapicque back in 1907 [72]. In 1936, Hill [73] combined the model with a second equation in order to describe adaptation, i.e. the process of coupling threshold and sub-threshold voltage in the neuronal membrane. Fuortes and Mantegazzini introduced the concept of spike-triggered adaptation via an adjustable threshold for the Integrate-and-Fire models in their experimental work on neuronal refractoriness [74].

The Leaky Integrate-and-Fire model in its modern form is the combination of spike-related adaptation mechanisms introduced by Treves in 1993 [75] and sub-threshold-related formulas developed by Izhikevich in 2001 [76] and Brunel et al in 2003 [77]. The model has then been used in a wide category of studies [78–80].

If the equation of the Leaky Integrate-and-Fire model is replaced by a quadratic non-linear formula, known in the field of neuroscience as the canonical form of a neuron model with a continuous gain function (Type I neuron model) [81], one arrives at the model of Izhikevich [82]. The Izhikevich model bridges the gap between simpler I&F models and more complicated, biophysically accurate ones. This model is computationally simple and, although it does not expose the electrochemical details of biological neurons, Izhikevich neurons display biologically plausible input-output interactions. They feature computational complexity significantly smaller than that of biologically-accurate models. To put their differences in perspective, an Izhikevich neuron requires 13 floating-point operations to simulate 1 ms of its operation, whereas the widely-used Hodgkin-Huxley

neuron will need thousands of floating-point operations for the same amount of activity [83].

If an exponential non-linear equation is attached to the Leaky Integrate-and-Fire model in order to approximate the functionality of biologically-accurate models [84], the resulting model forms the Adaptive-Exponential Integrate-and-Fire model [85]. The Adaptive-Exponential Integrate-and-Fire model, also called AdEx, is a spiking neuron model with two variables, in the same fashion as the aforementioned Izhikevich model. The first equation describes the dynamics of the membrane potential and includes the mechanics for the membrane's activation and exponential spiking. The membrane's voltage levels are coupled to a second equation which describes the adaptation of the model. Both variables are reset upon spike triggering. The combination of adaptation and exponential voltage dependence forms the name Adaptive-Exponential Integrate-and-Fire model.

The Adaptive-Exponential Integrate-and-Fire model is capable of describing a wide range of observed neuronal firing patterns; for example it can be used to describe both delayed and fast spiking, as well as adapting and bursting spiking. The AdEx model builds on and combines features of previously studied integrate-and-fire models. Its flexibility renders it an important tool in the field [86–88].

2.1.3 Conductance-based Models

Conductance-based models lie on the opposite side of the spectrum, using complicated differential equations. They constitute models that aim to expose electrophysiological qualities in the human brain, due to their emphasis on modelling the neuronal cellular structure as an electronic circuit. They offer valuable insight into the electrochemical properties of the neuron and the ability to study its ion channels. However, the high modeling accuracy they offer comes at the cost of significant computational complexity, often deeming them too expensive to use in daily experiments.

While there are lower-level models with increased complexity and biological detail, conductance-based models are the simplest possible biophysical representation of an excitable neuron, in which its ion channels are represented by conductances and its bilayer membrane by a capacitor. Conductance-based models are based on an equivalent circuit representation of a cell membrane as first put forth by Hodgkin and Huxley in 1952 [89]. They utilized their model to explain the ionic mechanisms underlying the initiation and propagation of voltage spikes in the squid giant axon, the very large axon that controls part of the water jet propulsion system in squid. Their work still constitutes the most prominent example of the conductance-based models.

The Hodgkin–Huxley model can be thought of as a differential equation with four state variables that change with respect to time. The system is difficult to study because it is a nonlinear system and cannot be solved analytically. However, there are numeric

methods available to analyze the system. Euler-based solutions for the complicated system of ordinary-differential equations of the Hodgkin-Huxley model are used [90].

Abstractions and additions to the widely-used Hodgkin-Huxley model exist. The FitzHugh and Nagumo model [91] is a two-dimensional simplification of the Hodgkin-Huxley model, where the state variables are reduced to the voltage levels of the neuronal membrane and a recovery variable. The motivation for the FitzHugh-Nagumo model was to conceptually separate the essential properties of neuron excitation and spike propagation from the detailed neuronal electrochemical functionality. This approach simplifies the study of the aforementioned phenomena but limits which aspects of the neuronal functionality can be modeled [92].

Another important abstraction of the Hodgkin-Huxley model is the Morris-Lecar model [93]. The Morris-Lecar model is also a two-dimensional reduction of the Hodgkin-Huxley model that utilizes two non-inactivating voltage-sensitive conductances. The Morris-Lecar equations are particularly useful for modelling fast-spiking neurons. Morris-Lecar-type models may prove useful for studying scaling phenomena involving cell growth. Such problems on real systems involve more biological channels than those featured in two-dimensional models but the Morris-Lecar model may give some insight into the phenomena that allows for maintaining functionality stability while a real organism grows. A thorough classification of available neuron models and related simulators has been made by Brette et al. [94].

2.2 High Performance Computing in Neuromodelling

2.2.1 In-Silico Experiments

The 21st century has marked the domain of neuroscience by introducing a significant rise in *in silico* experimentation, i.e. computer-aided simulations and neuromodelling. The transition to utilizing computational resources in order to unveil mysteries surrounding the human brain functions lead to tremendous investment efforts in the field of computational neuroscience globally [31, 95, 96]. In silico experimentation attracts attention from both the academic and industrial world.

In the literature, there is a large amount of noteworthy experiments, both in magnitude and biological significance, which were conducted via computer simulation. A significant amount of early experimentation focused on the fine details in the biology of a sole neuron, using primarily conductance-based models. Seung et al. [97] utilized a biophysically-accurate representation of 15 neurons in order to study the persistent effects of transient events on the oculomotor system. Their study was based on the earlier work of Lee et al [98]. Otsuka et al. [99] used a conductance-based model in order to study the firing rate of the subthalamic nucleus, a part of the brain's basal

ganglia system of yet unexplored functionality. More recently, Tian et al. [100] developed a biomechanical model stemming from the Hodgkin-Huxley model that attempts to integrate mechanical detail and the impact of stretching forces into the existing electrochemical model. In a similar fashion to other past and recent works, their study focuses on verification and functionality of the model, rather than its behaviour when scaling to larger neuronal networks.

Simulations of larger network behaviour followed the study of single-neuron modelling, due to advances both in computer science and neuroscientific understanding. Simpler integrate-and-fire models have been the model of choice for such cases. One of the largest experiments to date, carried out on a cutting-edge supercomputer, managed to study a network which matched a small animal's brain in size and connection density [101]. Hines et al. have also used a Blue Gene supercomputer to simulate up to 4 millions of simple spiking neurons [102]. Fidjeland et al. [103] have successfully deployed densely connected neuronal networks on GPUs, reaching network sizes of $40,000$ Izhikevich neurons. Bhuiyan et al. [104] use various platforms to scale up to millions of Izhikevich neurons, coupled with 50 Hodgkin-Huxley neurons in a 2-level neuronal structure; it should be noted, however, that their work aims at utilizing the neuronal networks for character recognition rather than studying their biophysiological behaviour. Choi et al. [105] have developed 1,000 silicon spiking neurons on a Xilinx Field Programmable Gate Array (FPGA), based on the Izhikevich model.

Large-scale conductance-based modelling is scarcer in the literature. A significant effort was undertaken by partners of the Human Brain Project, utilizing GPUs and FPGAs for Hodgkin-Huxley modeling of the human cerebellum [106]. Their implementation scales up to 400,000 neurons. Their experiments, however, impose a significantly simple network interconnectivity pattern, partly due to the difficulty of representing a dense, biophysically-accurate neuronal network.

2.2.2 Neuronal Simulation Frameworks

The domain of computational neuroscience fields various simulation environments, tuned towards tackling different experimental workloads.

Traditionally in the domain of neuroscience, the most common methods for simulating neuron models and studying their behaviour were through widely-known mathematical software suites such as MATLAB [107]. While these tools have been used extensively to advance the field of computational neuroscience, simulating neuronal networks of realistic sizes in high detail remains a challenge. Traditional execution of accurate models on CPUs with generic mathematical computing tools, such as MATLAB, could take a prohibitive amount of time to complete.

A dedicated reference tool in this domain is NEURON [108], a flexible neural simulator with a rich library of neuron models [109] from Yale university. Since its creation, NEURON has supported neuroscientific research and continues being widely used today [110]. The greatest asset that NEURON can offer is the diversity in its model library. Due to its long history and its widespread usage, the tool has reached a level in maturity that few other neuromodelling suites can match. As such, it attracts the interest of a large part of the neuroscientific community, which continues to explore and modify the tool to suit each laboratory's unique experimentation needs. However, since its original implementation in 1997, the software has undergone a lot of changes in order to adapt to the modern age of high performance computing. It was not inherently designed to expose massive parallelism required by demanding neuronal networks, although such efforts exist in the literature [111].

Another staple in the field is the GENESIS software [112]. Development of GENESIS software was initiated in Caltech and then spread to various labs of U.S. and Europe. GENESIS (which stands for General Neural Simulation System) is a simulation environment for constructing realistic models of neurobiological systems at many levels of scale, from sub-neuronal processes to entire neuronal networks and systems. The tool focuses on the anatomical and physical detail of neuronal systems [113]. GENESIS is intended to quantify the physical framework of the nervous system in a way that allows for easy understanding of the physical structure of the nerves in question. As such, it has a very specific purpose, which it serves in a satisfactory manner.

NEST [114] is a lighter tool that mostly aims at simulating large networks of simpler neuron models. NEST allows MPI, multithreading and hybrid usage thereof as parallelization methods, thus providing another interesting alternative for high-performance neuron modeling that has been tested on high-end supercomputers [115]. NEST also focuses on recreating an environment that the experimenter is familiar with, by introducing devices which represent the various instruments (for measuring and stimulation) found in an experiment. The framework, however, is not the tool of choice for more complicated, biophysically-meaningful experimentations.

Another framework that enjoys significant usage is Brian [116]. It is written in Python, a programming language that is familiar to the neuroscientific community. It utilizes vectorisation techniques for accelerating the simulator, however Python is an interpreted language and as such, its speed is only "good enough"; Brian's developers aim at providing a tool that performs sufficiently and is easy to deploy, rather than offering a true high-performance computing tool. Its lack of intermediate scripting tool marks it as an attractive choice to the community, who can thusly avoid learning and using the specific scripting requirements of NEURON, GENESIS and NEST.

CARLsim [117], on the other hand, is a GPU-oriented library for spiking neural network simulation and model-testing. CARLsim, currently in its 4.0 release, allows execution of networks of Izhikevich spiking neurons with realistic synaptic dynamics. The tool

focuses on GPU implementations, although it supports running in any x86-architecture. It allows for rapid development of experimentation environments, with an emphasis on synaptic dynamics. However, the library limits itself to simple models of spiking neurons, as its intended usage was in the field of robotics [118].

A different approach is explored by the European research project FACETS [119], whereby instead of using software-based numerical methods, analog neuromorphic hardware directly simulates complex neuron models. The tool describes an accelerated hardware neuron capable of emulating the adaptive exponential integrate-and-fire neuron model. Furthermore, an FPGA toolbox for simulating spiking neural networks in hardware has been developed by Qingxiang et al. [120]. These approaches, although novel and highly interesting, face inherent limitations and are difficult to be adopted by the broader neuroscientific community.

The aforementioned tools have been used extensively to advance the field of computational neuroscience. However, simulating neuronal networks of realistic sizes in high detail remains a challenge; high-performance computing has been recognized as a viable means for coping with this obstacle [121–125]. Neuronal complexity and computational demands call for high processing power with high inherent parallelism. Thus, they are very much in-line with the multi- and many-core paradigm observed in modern computing infrastructure [126]. The processing power of such systems has already been harnessed in significant brain modeling projects [127].

Finally, it should be noted that lately, due to large projects such as the Human Brain Project [31], a need for software that takes care of not only the calculation, but also the visualization of important details in neuron functionality and networking has arisen. VIOLA [128] is designed for initial visual inspection of massively parallel data generated primarily by simulations of spatially organized spiking neuronal networks. Its 2D and 3D visualizations support the exploration of neuronal activity across space and time. Such a tool can be used alongside a high-performance, biophysically-detailed simulator for powerful analysis of brain activity.

2.2.3 Model and Network Description Tools

There is a necessity in developing a widely-accepted application programming interface (API) that unites how different laboratories across the globe develop a mathematical model for brain simulation and how it inter-operates with different models of other parts of the brain. The neuronal simulation frameworks described in Section 2.2.2 each have a different set of expected expressions for describing an experimental setup. As such, cooperation between different scientific groups can prove to be difficult without a consensus on a programming paradigm.

Since 2008 [8], an important attempt at providing a unified interface for simulation frameworks in computational neuroscience has been undertaken. PyNN is a Python Interface for Neuronal Networking that attempts to loosely describe such an aforementioned API. It is a simulator-independent language for building neuronal network models.

A neuroscientist can write the code for a model using the PyNN API and the Python programming language. He can then run the model without modification on any simulator that PyNN supports. NEURON, NEST and Brian, which were mentioned in Section 2.2.2, support PyNN's API, which is a major attraction for the interface since these simulators drive a large portion of neuroscientific research today.

The PyNN API aims to support modelling at a high-level of abstraction (for example, by offering the ability to describe neuronal networks as populations of neurons in layers, columns and the connections between them) while still allowing access to the details of even individual neurons and their synaptic electrochemistry, if required by the scientist. To this end, PyNN provides a library of standard neuron, synapse and synaptic plasticity models, which have been verified to work the same on the different supported simulators. The library continues to grow as more simulators provide support for the interface and more standard models are defined and used by the community.

In this way, PyNN works as a "middleman" between the neuroscientist and the developer of a framework. It sets a standard API the developer must adhere to and guarantees to the neuroscientist that basic principles she is familiar with, will be supported by the desired simulator (or alerts the scientist that a certain feature is missing from the simulator). It also allows for the developer to work independently from the scientific community and continuously implement features that are valuable to the community, leading to a steady and controlled growth in simulator maturity. These aspects mark PyNN as a very encouraging undertaking in the interdisciplinary field of computational neuroscience.

It should be noted that PyNN's design also introduces some limitations. By design, PyNN enforces the use of standard models, chosen from an expanding library. As such, research that uses custom, or heavily modified models, may not benefit from the utilization of PyNN. PyNN provides a low-level API with increased flexibility that allows for the description of any custom model. However, in such a case, a language designed for model description, such as NeuroML [129, 130], may be a better fit for the researcher's needs.

The NeuroML project focuses on the development of a description tool based on an extensible markup language (XML). It provides a common data format for defining and exchanging descriptions of neuronal cell and network models. The project uses XML schemas to define the model specifications. The biggest asset of NeuroML is that it focuses on models which are based on the biophysical and anatomical properties of real neurons. The described models can include details of neuronal morphologies, anatomical

connectivity and membrane conductances and their role in action potential generation; as such, NeuroML is a very good framework for describing a conductance-based model in a standard, well-defined manner. Other XML-based approaches exist in the field [131].

Today, these model-describing tools are widely in a variety of neuroscientific research topics and are continuously expanded to meet the ever-increasing needs of the community [132–136]. They can prove to be invaluable tools in achieving efficient cooperation between computer engineers and neuroscientists; as such, their usage has been included in this dissertation's work.

2.3 Summary

The field of computational neuroscience is relatively young and as such, there is much activity in order to discern the efficiency of each modelling approach towards the discovery of the human brain. There is a number of neuromodelling approaches, each focusing on different levels of detail. This chapter presented the most prominent and widely-used models. We note that an increased amount of research has been focused on models with less biological detail, treating neurons as simple nodes in a network and with reduced mathematical complexity for their functionality. Conductance-based models, such as those inspired by the work of Hodgkin and Huxley [89], provide more information concerning neuronal electrochemical activity and are the focus of this Doctoral thesis. Their computational complexity serves as an important benchmark for high-performance computational platforms.

While there is an increasing amount of modelling approaches to exposing neuronal anatomy, an attempt at presenting these models in a well-defined manner is present with widely-accepted modelling languages. The efforts presented in this chapter are a Python-based package named PyNN and an entity-description language named NeuroML. They offer a much-needed API that unifies how a neuronal model and its components are represented, simplifying their porting on computational platforms.

There is a significant amount of frameworks aimed at bringing efficient computing tools to the world of neuroscience. Some of the most widely-used, well-known and more traditional frameworks, such as Yale's NEURON, aim at including most known modelling efforts and accommodating to emerging modelling needs from new neuroscientific discoveries. Other, more modern approaches, such as NEST, focus on performance and scalability with computational resources. Table 2.1 offers a synopsis of the most important undertakings presented in this chapter, as well as present the scale and performance achieved in this Doctoral thesis. It relays an overview of the landscape to the reader, although an extensive survey is outside the scope of this engineering-focused Doctoral thesis.

Table 2.1: Prior Art

Reference	**Platform**	**Model Complexity**	**Problem Size (nrns×syn.)**	**Solution Cost (time×nodes)**	**Perf. Ratio (size / cost)**
Kunkel, 2014 [115]	Supercomputer K	Low	2.1e22	8.1e10	2.6e11
Hines, 2011 [102]	Supercomputer Intrepid BG/P	Low	1.0e18	1.5e6	6.8e11
Sripad, 2018 [137]	Multi-FPGA	Mid	4.0e7	2.0e-3	2.0e10
Beyeler, 2015 [117]	GPU	Mid	2.7e13	1.5e1	1.8e12
Hoang, 2013[138]	Multinode GPU	Mid	1.0e14	8.0e0	1.3e13
Ananthanarayanan, 2009 [101]	Supercomputer Dawn BG/P	Mid	8.1e21	1.1e7	7.4e14
Florimbi, 2016 [106]	GPU	High	1.3e12	4.8e3	2.8e8
Nguyen, 2015 [139]	GPU	High	8.0e12	4.0e2	2.0e10
Chatzikonstantis, 2017 [140]	Xeon CPU and Phi KNC	High	1.0e14	1.4e3	6.9e10
Chatzikonstantis, 2018 [141]	Multinode Phi KNL	High	4.0e15	4.8e3	8.3e11

Table detailing the efforts in the literature to simulate neuronal models of varying complexity in high-performance computing platforms. The Table attempts to extract information from the cited studies concerning each work's best-effort network simulation and respective simulation speed. Platform indicates the computational fabric used in the work. Model complexity is categorized as either Low for Integrate-and-Fire models, Mid for Izhikevich-based models and High for conductance-based models. In column problem size, the amount of neurons is multiplied by the amount of totals synapses present in the largest network simulation reported by the literature. Metric "solution cost" represents the resources used for solving the simulation problem by multiplying the amount of execution time (measured in seconds) required for simulating a second of brain activity by the amount of computational nodes employed in the simulation, either CPU, GPU or FPGA nodes. For ease of comparison, a ratio between these two metrics, named performance ratio, is supplied where problem size is divided by solution cost. The metrics are reported as pure numbers in scientific notation format.

Chapter 3

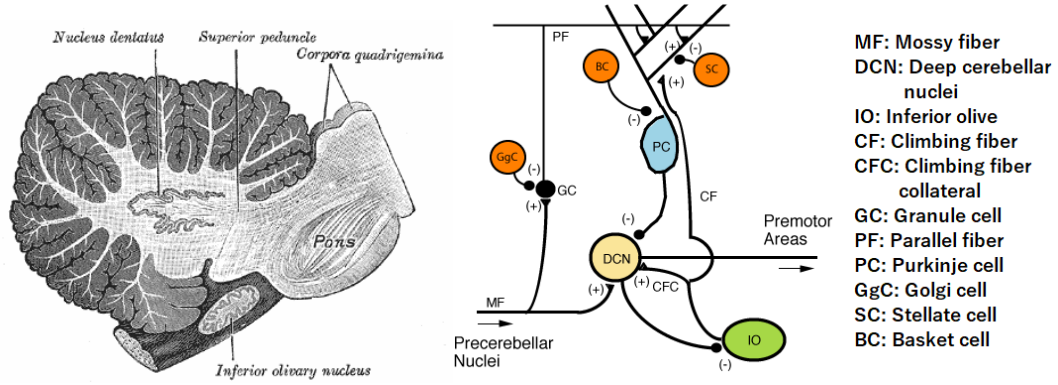
Neuromodelling on a Manycore Processor

3.1 Introduction

Chapter 2 has established the difficulty of challenges with modelling complex, computationally demanding neuronal models. As mentioned, most of the modelling effort in the community has focused primarily on simpler neurocomputing models which are easier to both implement and scale. Even existing neurocomputing software tools that offer the capability to perform biophysically-accurate simulations face significant obstacles with computational performance, workload scaling and ease of usage. As such, the existing literature has not defined a clear, go-to solution to the important challenge of neuronal simulations, particularly for complex networks of biological detail.

This dissertation will present the full development of such a simulator, from the perspective of a computer engineer and will address how the advent of manycore computing, as well as the evolution in the computing capabilities of available accelerating processors can help cross boundaries that have limited previous efforts in this field. It aims at presenting the reader with a full disclosure of years of development.

The presented simulator began with focusing on a particular region of the human brain, called the Inferior Olivary Nucleus. The first manycore platform used for its implementation was Intel's 48-core research manycore chip, named Single-Chip Cloud Computer. Both the modelled region, the implementation and the platform will be presented in this chapter, as well as the results of the initial effort. This chapter will serve as an establishing point for the evolution of the modelling effort, along with the evolution of the manycore processors, as detailed in Chapter 2.



(A) Sagittal section through right cerebellar hemisphere.

(B) Microcircuitry of the cerebellum.

FIGURE 3.1: Anatomical and circuit-level representations of the brain regions studied.

3.1.1 The Human Inferior Olivary Nucleus

The human inferior olivary nucleus (ION) is the region of interest for the studied model. It constitutes a sub-part of the human brain's cerebellum.

More specifically, the inferior olivary nucleus is a structure found in the medulla oblongata underneath the superior olivary nucleus [142]. In vertebrates, the ION is known to coordinate signals from the spinal cord to the cerebellum to regulate motor coordination and learning. These connections have been shown to be tightly associated, as degeneration of either the cerebellum or the ION results in degeneration of the other [143].

Patient studies revealed the connection between the inferior olivary nucleus ION and the cerebellum. Lesions in the ION impair the ability to learn higher level motility, such as performing a jumpshot in basketball [144]. Further investigation of the neuroanatomy confirmed the intimate connection between the ION and the cerebellum in motor coordination and learning [145, 146].

The ION sends signals to the cerebellum based on information sent from the spino-olivary tract. Regulation following this point is highly debated. The original hypothesis as to how the ION influenced the cerebellum involved Long Term Depression (LTD). In this scenario, granule cells from the cerebellum send GABA to inhibit ION activation. However, this hypothesis was brought in to question when multiple laboratories inhibited LTD and motor learning was intact.

More recent studies [147] suggest that encoding the timing of sensory input is the key component of these connections. The ION sends signals through different cell clusters. These signals vary in location and in frequency bundles and appear inconsistent. However, the temporal pattern of these signals is consistent. Therefore, ongoing research on

motor learning is investigating how these timed signals develop and their role in motor learning.

The inferior olivary nucleus expresses key enzymes involved in steroidogenesis required for neuroprotection and maintenance. The most crucial of these enzymes is aromatase, which is the enzyme that is necessary for the conversion of testosterone into estradiol. Without aromatase, the ION is unable to make estradiol, and cannot recover from injury properly.

Because the inferior olivary nucleus is tightly associated with the cerebellum, lesions in either the IO or the cerebellum results in degeneration in the other. There is little known about damage to the inferior olivary nucleus independent from the cerebellum. To date, the only known disorder which specifically targets the ION is an extremely rare form of degeneration called hypertrophic olivary degeneration (HOD).

Although the ION is not often investigated on its own, degeneration in the ION has been identified in disorders that are typically associated with the cerebellum. These disorders include supranuclear palsy, Leigh disease and SCA6, and there are several more. These disorders all involve motor coordination [148–150].

3.1.2 InfOli Simulator

The simulator that will be presented here aims at modelling the human inferior olivary nucleus via an extended version of the Hodgkin-Huxley conductance-based model; as such, it is named InfOli simulator. It is a product of a joint of collaborative effort between the National Technical University of Athens (NTUA) and the Erasmus Medical Centre of Rotterdam (EMC Rotterdam).

The importance of the simulator lies in the computational complexity of the model. The conductance-based model uses Ordinary Differential Equations (ODE) in order to simulate the behaviour of the ionic channels in each neuron. The ODE-system that describes the model is solved with the forward Euler method [151]. Due to the sensitivity of the model and the nature of its mathematical modelling, the simulator works in transient mode. In each step, the full suite of parameters for each neuron in the network is calculated accurately, without the possibility to "skip" any timestep even in the case of absence of activity in the network (either due to a lack of external stimuli or any internal disturbances). Put simply, even the small sub-threshold oscillations in the voltage levels of the neuron membrane layers are necessary to be calculated accurately in every timestep.

These aspects of the model in question cause the simulator to be demanding in both computational power and memory. Furthermore, these traits offer a biophysically-accurate, rich-in-detail simulator which can serve as a suitable case study for the capacities of the

high-performance manycore processors in the field of computational neuroscience and complex mathematical modelling in general.

As a result, the study of performance vs. quality-cost trade-off is particularly interesting. In particular, the manycore platform that was first used and will be presented in this chapter, the Single Chip Cloud computer, is a platform with various power-management and application-mapping capabilities. Among others, the target chip features Dynamic Voltage and Frequency Scaling (DVFS) and on-die message passing. The SCC experimental processor [152] is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research. The *InfOli simulator* essentially is a self-contained and highly realistic computational model of the inferior olivary nucleus. As such, this chapter will present a feasibility study of the IO simulator on a research-grade manycore chip, incorporating all elements of a full system realization.

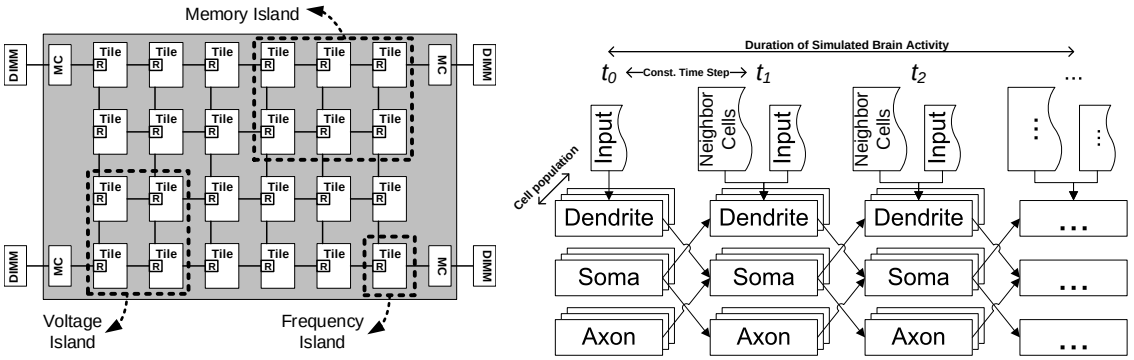
3.2 Target Architecture

In this thesis, we target manycore systems as the computational fabric of choice, specifically for their scalability and coding paradigms. The outline of the different generations of manycore processors has been relayed in Chapter 1, however a more detailed presentation of the first manycore processor used for the InfOli simulator, the SCC, will be given here.

3.2.1 The Single Chip Cloud Computer

In 2005, Intel launched single-chip manycore processors, initially designed towards research applications and manycore coding exploration. Intel developed the chip architecture based on cloud data centers, the cores being separated across the chip but able to directly communicate with each other.

The platform that was first used for designing the InfOli simulator is the Intel SCC [152]. It is a homogeneous, many-core chip consisting of 48 cores, organized in pairs called *tiles* that are interconnected through a mesh network organized in a 4×6 2D-fashion. Each individual core is of limited computing capability, based on IA-32 P54C cores; the chip focuses instead on the large amount of available hardware. An overview of the chip can be seen in Figure 3.2a. Each tile constitutes a separate *frequency island*. Pairs of tiles create an individual *voltage island*. Each core of the SCC has its own private memory space and also access to a low-level Message-Passing Buffer (MPB) that is used for the exchange of messages between SCC cores. In each tile, the two P54C cores share the 16 KB (8 per core) MPB which acts as a router. The MPB allows SCC cores to avoid sending information back to the main memory and re-routing it to other cores.



(A) The frequency, voltage and memory islands of the SCC.

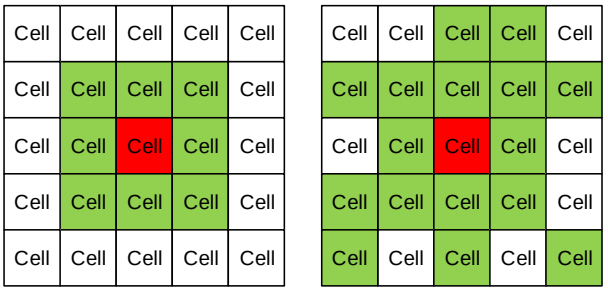
(B) Illustration of the IO simulator runtime.

FIGURE 3.2: High level view of the target platform and the target application that are discussed in this Chapter.

Each core in the SCC chip is booted with a custom Linux distribution. The chip is installed on a board that communicates with a Management-Console Personal Computer (MCPC) through Ethernet (for power monitoring) and PCIe (for disk access) connections. The `/shared` directory is common between the MCPC and each core of the SCC. That way, exchange of files and executables is made possible between the MCPC and SCC cores. Source code is written and cross-compiled on the MCPC and executed on the SCC. A special library called RCCE [153] is available on the MCPC, providing commands for message passing and dynamic voltage and frequency scaling. A set of software utilities on the MCPC allows dispatching of tasks across SCC cores and monitoring of performance metrics of the chip (e.g. measurement of the chip's power consumption etc).

The SCC contains 1.3 billion 45nm transistors that can amplify signals or act as a switch and turn core pairs on and off. These transistors use anywhere from 25 to 125 watts of power depending on the processing demand. Each quarter of the chip is assigned a Dual Inline Memory Module (DIMM) through a Memory Controller (MC). The four DDR3 memory controllers on each chip are connected to the 2D-mesh as well. These controllers are capable of addressing 64 GB of random-access memory. The DDR3 memory is used to help each tile communicate with the others. These controllers also work with the transistors to control when certain tiles are turned on and off to save power when not in use.

The SCC has two modes that it can operate under, processor mode and mesh mode. In processor mode, cores are on and executing code from the system memory. Loading memory and configuring the processor for bootstrapping (sustaining after the initial load) is done by software running on the MCPC. In mesh mode, cores are turned off. Only the routers, transistors and RAM controllers are on and exchanging large packets of data. Additionally there is no memory map.



(A) An 8-way connectivity is acceptable for IO neuron simulations.

(B) As we move away from the cell, connectivity drops by 50 %.

FIGURE 3.3: Illustration of IO neuron cell connectivity: the central cell is connected to the highlighted cells around it.

When proper coding is implemented, the aforementioned assets constitute a functional processor that was capable at the time of its manufacturing and energy efficient. Its coding paradigm and design that closely follows that of a network of cloud computers.

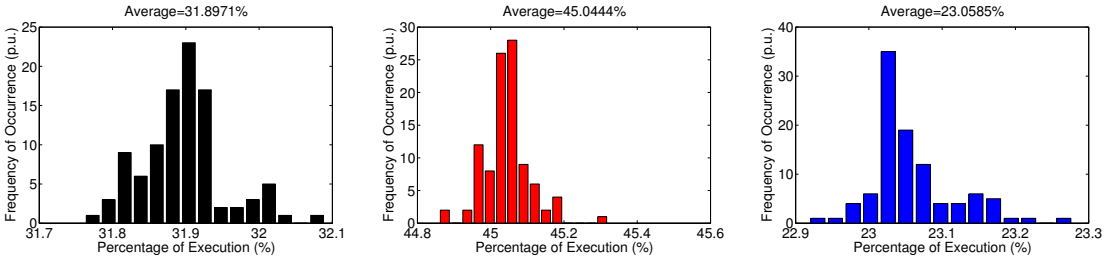
3.3 Implementation

The simulator is a C-based repository that aims at providing extensive support for various experimental parameters. It uses particular libraries, exclusive to the SCC, like the RCCE libraries mentioned in this Chapter's 3.2.1 that aim at passing data between the fabric's cores. We shall examine the application in question, namely the inferior olive neuron-network simulator.

3.3.1 Algorithm

For a given cluster of ION neurons, the simulator calculates the membrane potential of each cell as the latter is affected by a mix of external and internal input stimuli (currents). The simulator is **transient** with a constant step $\Delta t = 50\mu$s. The data flow during simulation steps $t_0, t_1, t_2, ...$ can be seen in Figure 3.2b.

The simulated cell population can be visualized as a two-dimensional mesh, similar to the ones presented in Figure 3.3. A very important element is the *interconnectivity* between the cells. In order to realistically calculate the potential of each cell, the simulator needs to consider the voltage levels of neighboring cells. For the experiments presented in this paper, we assume an 8-way cell interconnectivity (Figure 3.3a). Our implementation is parameterized to support more complicated schemes (e.g. Figure 3.3b) in order to more closely model different loci in the neural ensemble. In order to run a simulation, the user needs to specify the *size of the cell network*, the *duration of the simulated brain*



(A) Profiling of the dendrite compartment

(B) Profiling of the soma compartment

(C) Profiling of the axon compartment

FIGURE 3.4: Profiling of 100 iterations of the IO neuron cell model to extract timing information for each compartment

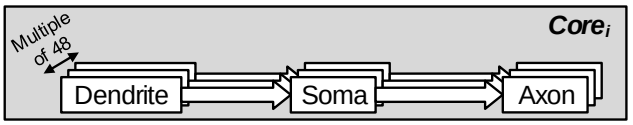
activity, the *external input currents to each cell* as well as the desired *interconnectivity scheme*. The simulator solves the model for each of the simulated neurons. This model is multi-compartmental and splits each cell into three segments:

1. The *dendrite* compartment is responsible for exchanging information with other connected IO cells and also receiving the input current. At the beginning of each simulation step, voltage levels of neighboring cells are recorded to be used in further computations.

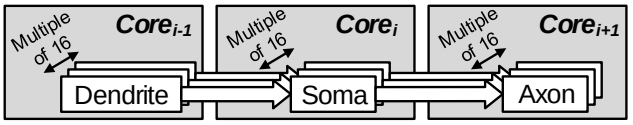
2. After information from the environment has been received, the *soma* compartment performs the most computationally intensive part of the neuron.

3. The *axon* component serves as the output stage of the neuron. Its so-called action potential is – in fact – the voltage output recorded for the entire cell.

The entire simulator is a system *with memory* and parameters of each compartment are always reused in the next iteration of the simulation. We start from a single-threaded implementation that includes these three cell compartments for an arbitrary cell network size, brain activity duration and connectivity scheme. Before exploring mapping options of the simulator on the SCC, we perform a profiling experiment to derive the workload distribution between the three compartments. The results are shown in Figure 3.4 for 100 executions of a 6-second simulation for a single IO cell. The UNIX function `getttimeofday` has been used to record execution times. Immediately, we verify that the soma compartment is the most computationally intensive part of the simulator's runtime. The axon compartment exhibits the shortest processing times. The dendrite component lies in the middle. However, given that it is highly related to inter-neuron communication, we can expect that it will occupy a large fraction of the overall execution time in case we employ a more complicated interconnectivity scheme (as in the case of Figure 3.3b) requiring communication with more than 8 neighboring neural cells.



(A) Data partitioning, where $i = 0, 1, 2, ..., 47$



(B) Combined data and task partitioning, where $i = 2, 4, ..., 46$

FIGURE 3.5: The two mapping options of the IO neuron cell simulator on the SCC chip that are discussed in this chapter.

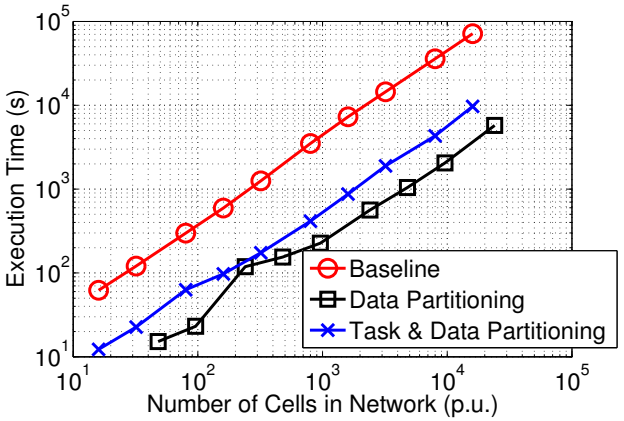
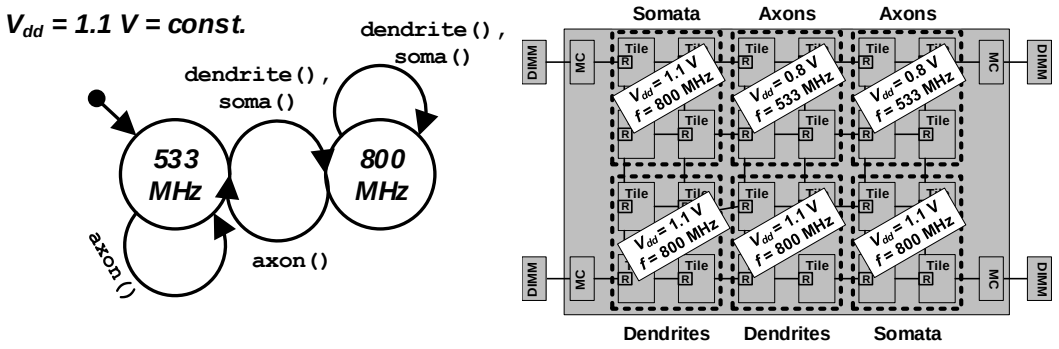


FIGURE 3.6: Initial benchmarking with the SCC trained at $V_{dd} = 1.2$ V with a 533 MHz clock. The baseline represents a single-threaded version of the simulator, running on a single SCC core with the above voltage and frequency.

If we imagine a plane of ION neurons, the most intuitive mapping option of the respective model would be to assign a subset of the cell population to each of the cores of a many-core system. This effectively represents *data partitioning* of the simulation and it will be the first mapping explored in this paper. That way, we submit a cell network size that is a multiple of 48 to the SCC chip, in order to maintain a homogeneous distribution of workloads across the SCC cores. This mapping option is illustrated in Figure 3.5a.

With respect to inter-cell communication, cells assigned to the same core communicate through the private memory of the core. Cells assigned to different cores communicate using the `RCCE_send` and `RCCE_recv` commands [153] of the SCC for inter-core communication through the MPB.

By inspecting the profiling information of Figure 3.4, we can approximate the execution-time ratios among the three neuron compartments. Based on this information, it is further interesting to explore the splitting of the simulation with respect to neuron compartments. Assigning the simulation of each compartment to a different core effectively



(A) The FSM of the DFS power management strategy for the case of the data partitioned IO neuron simulator

(B) SVFS of the SCC cores for case of the task and data partitioned partitioned IO neuron simulator

FIGURE 3.7: Two power management schemes, tailored to the data and combined task and data partitioning mapping schemes

constitutes *task partitioning*. With a total of 48 available cores, we can additionally perform data partitioning of the simulation, thus implementing *combined task and data partitioning* of the IO neuron simulator, a concept that is illustrated in Figure 3.5b.

More specifically, we assign each compartment to a single core. Thus, an entire IO neuron requires a triplet of SCC cores. That way, each core will take up the simulation of a single compartment of $N \div 16$ cells, where N is the size of the cell network that is simulated (preferably a multiple of 16, to avoid workload imbalance across SCC cores).

In Figure 3.6, we can see the execution times for the data- and combined task- and data-partitioning schemes. For both small and larger cell populations the data-partitioning mapping is the faster option. We achieve a constant speedup of about an order of magnitude compared to the single-threaded execution (Baseline). The combined data- and task-partitioning mapping exhibits a somewhat lower speedup in comparison, which was expected due to the increased inter-core traffic required for intra-cell, inter-compartment communication. However, this combined partitioning enables more efficient power-management strategies with significant energy savings. Finally, it is important that both proposed mappings have been verified against the baseline and show no accuracy degradation.

3.3.2 Platform Considerations

The Single-chip Cloud Computer (SCC) provides interesting capabilities in regulating its voltage and frequency levels of operation in real time.

As transient neuron simulations contain an increasing number of cells, their demands in terms of computational resources are also increasing over time. This leads to a subsequent increase in the energy budget required to complete such simulations. This

is a known problem in existing supercomputing ensembles used for large-scale brain-simulation (and other) applications [154–156]. In order to make the IO-simulator mapping as efficient as possible on the SCC, we can utilize power-management strategies for the data-only- (Figure 3.5a) and the combined task- and data-partitioning case (Figure 3.5b).

Based on the data-partitioning option as outlined in Figure 3.5a, a power-management strategy should be applied that is global across the SCC chip. The reason is that there is no difference between cores on the basis of executed workload. As the profiling information of the IO-neuron simulator has revealed (Figure 3.4), the dendrite and soma compartments each require roughly twice the execution time taken up by the axon compartment. The dendrite component is a bit faster in comparison to the soma, however given a more complicated interconnectivity scheme, its computational overhead is bound to increase.

If we were to be conservative with the clock frequency of each core, an interesting opportunity would arise: Given the Dynamic-Frequency-Scaling (DFS) capabilities of the SCC (at tile granularity), we can manipulate the frequency of the cores depending on the compartment that they are simulating in each case. We can alternate between the 533 MHz and 800 MHz clock frequencies, by properly setting each tile frequency divider. The Finite-State Machine (FSM) of Figure 3.7a is illustrating this concept. Since both frequencies need to be supported at runtime, the voltage supply is set at the minimum allowed value of 1.1 V. We avoid voltage manipulation at runtime, since the SCC voltage regulators are less responsive than the frequency dividers [157].

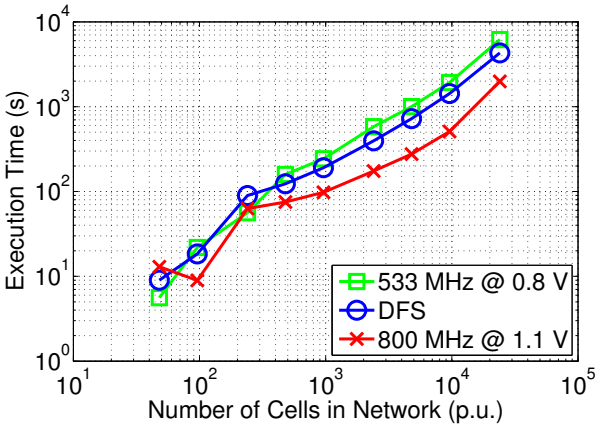
3.4 Evaluation

3.4.1 Experimental Setup

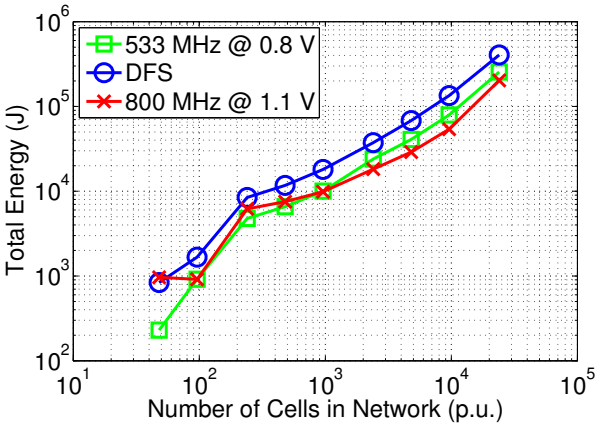
In order to evaluate this version of the simulator on the experimental manycore processor SCC, a set of experiments has been conducted. A variety of different voltage and frequency management scenarios were tested. Each scenario is evaluated for increasing network sizes, and consumed energy and total execution time are measured.

We test alternating frequencies of CPU operation according to the FSM of Figure 3.7a. We also test two other power-management schemes for this mapping option. The first involves a constantly high clock frequency (800 MHz) along with the lowest user-selectable voltage (1.1 V). The other scheme involves a constantly low frequency (533 MHz) with the associated V_{dd} setting (0.8 V). Simulation accuracy remains unaffected by the various power-management strategies applied.

These voltage and frequency scenarios are tested in two different mapping schemes. As mentioned in Section 3.3, the three different compartments of neurons, soma, axon



(A) Logarithmic sweep of cell inventory



(B) Logarithmic sweep of cell inventory

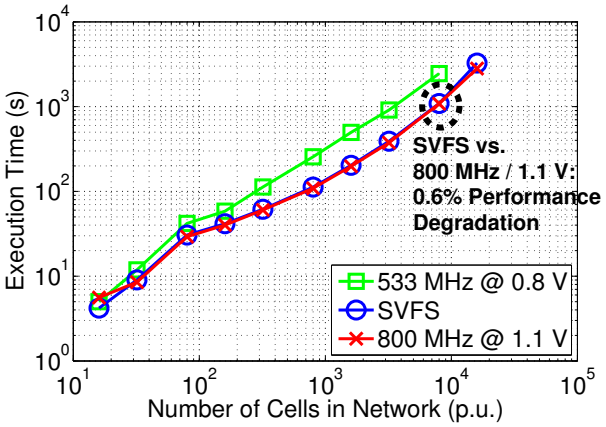
FIGURE 3.8: Performance vs. quality cost assessment for different power management schemes of the strictly data partitioning mapping, for various neuron network sizes

and dendrite, can either be mapped indiscriminately on the SCC cores, or they can be divided in groups of cores, according to their processing needs. We test both options by evaluating mapping of strictly data partitioning, as well as a mapping strategy where both data and task partitioning is utilized.

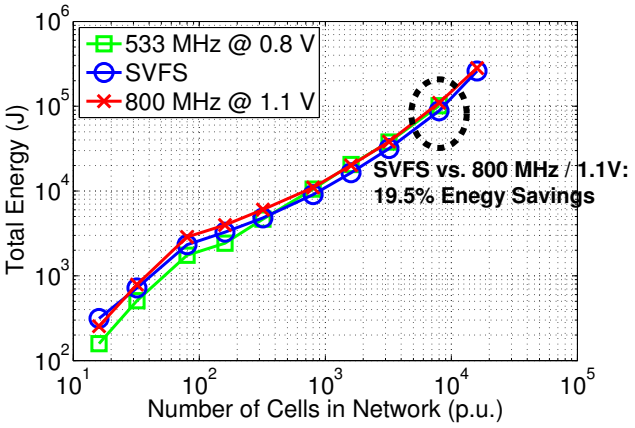
For the measurements, and without loss of generality, simulator interim output messages are deactivated, which reduces simulation time significantly. Thus, the simulator reports only on its starting and ending network state (at simulation termination).

3.4.2 Results

In Figures 3.8a and 3.8b we present the performance and energy budget of each power-management strategy for various neuron-network sizes of the data-partitioning case. It turns out that a constant, high frequency is a preferable choice, especially as the neuron population increases. DFS is not a good solution since it consumes the most energy with a very mediocre acceleration, in comparison to the low-frequency case. This is to be



(A) Logarithmic sweep of cell inventory



(B) Logarithmic sweep of cell inventory

FIGURE 3.9: Performance vs. quality cost evaluation for different power management schemes of the task *and* data partitioning mapping option, for various cell network sizes

expected, given the overhead of signaling the frequency divider for every transition of the FSM of Figure 3.7a. In case the voltage regulator could be of the same responsiveness as the frequency divider, we could witness higher energy benefits, assuming that V_{dd} could follow clock-frequency alterations.

In the case of combined task and data partitioning (depicted in Figure 3.5b), we can have different power-management strategies per core since the cores are executing different tasks. We isolate tasks that simulate specific compartments and map them to separate voltage islands of the chip [158]. Based on the profiling information collected previously, voltage islands simulating somata or dendrites need to operate at twice the frequency of voltage islands simulating axons. Dendrites are assigned to high-frequency execution due to the potential for intensive inter-core communication, in case complex interconnectivity schemes are simulated.

Hence, cores that need to run "fast" will be set at 800 MHz and "slow" cores will run at 533 MHz. Frequency scaling needs to be performed only once before the simulation begins. That way, we have the chance to (statically) "train" the SCC system to the

appropriate voltages. "Fast" voltage islands will be configured at 1.1 V (minimum voltage setting to support a 800 MHz clock) and "slow" voltage islands will run at 0.8 V (minimum voltage that supports a 533 MHz clock). We refer to this power management as Static Voltage and Frequency Scaling (SVFS), shown in Figure 3.7b.

We also test two extra power-management schemes: a static high-frequency (800 MHz / 1.1 V) and a static low-frequency (533 MHz / 0.8 V) option. The performance and energy-budget evaluation is presented in Figures 3.9a and 3.9b, respectively. In terms of acceleration the static, low-frequency option is less desirable. The SVFS and static high-frequency options appear to perform similarly with larger neuron populations. However, for a network of 8,000 neurons, SVFS saves 19.5% energy for a negligible 0.6% loss in performance, in comparison to the static high-frequency setting (see annotation in Figure 3.9b). Similar observations can be made for other network sizes and for the comparison of the SVFS option with the low-frequency setting. In general, SVFS is a very good power-management strategy for the combined task- and data-partitioning mapping of the IO simulator. It yields significant energy savings which are important if we consider the energy bills generated by large computing infrastructure when running large brain-scale simulations, such as the IO simulator.

Given the different frequencies and voltages used by the SVFS setting, it is important to average out the stressing of the cores of the SCC (i.e. avoid hot spots or heterogeneous aging across the many-core chip). In regards to addressing this issue, effective solutions have been already proposed in the literature [159].

The variety of design-time choices that can be made when mapping the IO simulator on the SCC create a set of *design points*. Each point comes as a different combination of task/data partitioning and represents a specific power-management strategy. Each point can also be evaluated in terms of certain *cost metrics*, such as execution time or consumed energy. This set of points creates a *design space* which can be treated in a formal way in order to derive the subset of *Pareto-optimal* [160] design points. In this formal treatment, the cell-network target size, the interconnectivity scheme and the duration of simulated brain activity are left as independent variables. The reason is that these three parameters have an adverse effect on the performance and quality cost of any neuron-simulator implementation.

Figure 3.10 illustrates the Design-Space Exploration (DSE) steps that lead to optimal mapping of the IO-neuron-simulator on the SCC. Initially, the target cell-network size, the assumed cell interconnectivity and the duration of brain activity are defined. Then, a set of n available design points is identified. Each design point P_i is evaluated in terms of certain cost metrics $x_j\,(P_i)$, where $j = 1, 2, ..., m$. Examples of such include execution time, average power or total energy consumption. The front of Pareto optimality is a set V containing all the design points P_i that satisfy Equation 3.1, assuming n design points and m cost metrics:

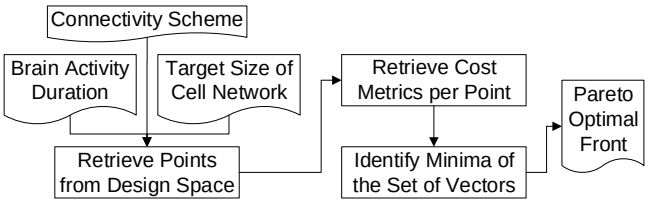
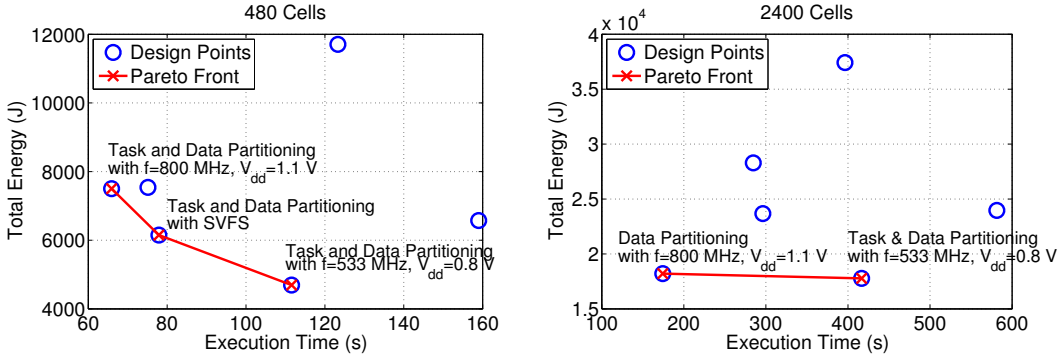


FIGURE 3.10: DSE flow for optimal IO cell simulation



(A) (B)

FIGURE 3.11: DSE for two different neuron network sizes; mapping and power management details are given only for each point belonging to the Pareto front

$$\begin{gathered} x_j\left(P_i\right) \leq x_j\left(P_k\right), \\ \forall\ k = 1, 2, ..., n \text{ and } \forall\ j = 1, 2, ..., m \end{gathered} \tag{3.1}$$

The derivation of the Pareto front can be automated using an algorithm that calculates minima of a set of vectors [161]. To accelerate exploration, the Pareto front can be pruned even further if constraints are imposed on any of the cost metrics, as in the case of *real-time* IO-neuron simulation (where any execution time exceeding the duration of simulated activity is not allowed). Obviously, to perform the aforementioned methodology, knowledge of the performance of the various IO-simulator mapping options is required. This can originate from a extensive benchmarking session at design time, where possible mappings and power-management strategies are evaluated. Obviously, the degree of detail achieved in this benchmarking session will affect the accuracy of each derived Pareto front.

In order to substantiate the proposed DSE methodology, we identify the Pareto front of optimal IO-simulator mappings for two different cell populations: 480 and 2,400 neurons. The DSE results are displayed in Figures 3.11a and 3.11b respectively. In both cases, the simulated duration of brain activity is set to 6 seconds and 8-way neuron interconnectivity is assumed. This means that two of the three inputs to the DSE methodology are kept constant, for the sake of simplicity. From the DSE results, we identify that *mapping and power-management options that are Pareto-optimal for a*

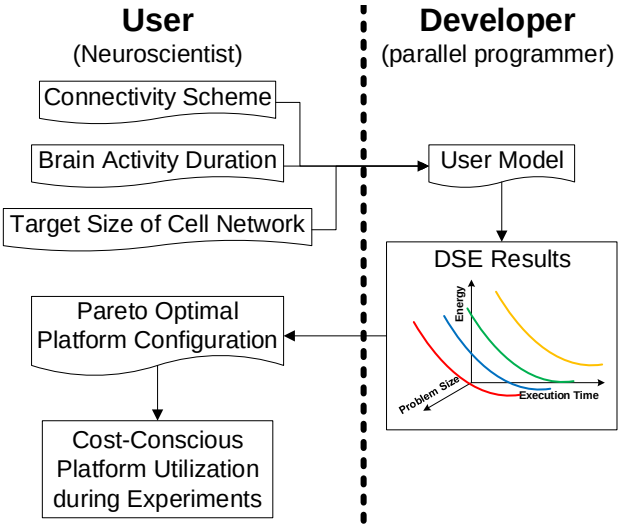


FIGURE 3.12: Exploitation of the DSE results

certain size of neuron network, are not necessarily Pareto-optimal for other network sizes. This is a very important observation, since we prove that the front of Pareto optimality is changing based on the parameters of the IO simulation problem (neuron-network size in this case).

3.5 Summary

In view of the results presented in this Chapter, we strongly motivate that for the proposed simulator to optimally occupy the resources of the SCC platform, we need to exploit the DSE results for the specific simulator. This exploitation can be generalized to any neuron simulator and any many-core platform. The overall concept is illustrated in Figure 3.12. For the case of the IO simulator and the SCC, the methodology employed should be as follows: The *user* (i.e. a neuroscientist) needs to specify the target neuron network size, interconnectivity and duration of the simulation. Based on the DSE results, the *developer* (i.e. the parallel programmer of the platform) needs to adapt the provided *user model* to a Pareto-optimal configuration for the target many-core system (i.e. adjust the mapping and power-management options). *Using this arrangement, any experiments of the user are bound to occupy the target platform in a Pareto-optimal way.* Given the variations found in the Pareto space (e.g. around 50% variation in energy as illustrated in Figure 3.11b for the SCC–IO-simulator combination), we believe that exploitation of a thorough DSE campaign can greatly improve the energy efficiency [10] of larger systems running biologically accurate brain simulations [11].

We have presented a thorough design-space exploration for the mapping of a biologically accurate neuron simulator on an industrial grade many-core platform. The simulator of our choice is based on a transient model of the inferior-olive neurons which are of major importance for human sensorimotor control. The target platform is the

Single-Chip Cloud Computer, developed by Intel Labs. We explored different mapping options, including data partitioning and combined task and data partitioning. Also, we exploited the power-management options of the chip, implementing both Dynamic Frequency Scaling and Static Voltage and Frequency Scaling. Combinations of mapping and power-management options create a design space, which is formally treated in order to derive Pareto-optimal executions of the simulator on the target platform. The achieved energy savings, along with the sensitivity of the Pareto space in problem parameters, motivate a close collaboration between the user (neuroscientist) and the developer (platform programmer) of this neuron-activity model. Given the importance of energy efficiency for many-core systems, this collaboration paradigm is strongly motivated in order to achieve optimal and cost-conscious utilization of computing resources used for biologically accurate neuron modeling.

This initially presented version of the simulator will act as a starting point for the development of a more generic and powerful tool developed for the purposes of the field of computational neuroscience.

Chapter 4

Scaling the Neuromodelling Application

4.1 Introduction

The SCC, as presented in Chapter 3, is a research-oriented manycore processor which acted as a starting point for the generation of manycore processors. The transition to the more industrial-focused generation of Xeon Phi processors was deemed beneficial; the main drive behind the transition lies in the level of maturity behind the Xeon Phi products, as compared to the SCC.

From a technological standpoint, the Xeon Phi processors offer superior computational power for workloads that can be processed by multiple, parallel threads. As a commercial and mature product, the Xeon Phi processors are significantly more powerful processors than the SCC. A design point that limits the SCC's processing power is its older, simpler-design cores. As mentioned in Chapter 3.2.1, the SCC uses P54C cores. As the design of the simulator expands to contain more elaborate and difficult-to-process modelling detail, the single-threaded performance of each P54C core becomes increasingly inadequate. The Xeon Phi line of products upgrade their processor cores, particularly in the case of 2nd generation Knights Landing manycore processors, where their Atom cores also offer efficient, low-energy-cost computing.

On top of superior quality in processor cores, Xeon Phi products offer a larger quantity of computational resources. The manycore processors that succeeded the SCC integrated a larger count of processing cores on die (approximately 60 versus the SCC's 48 cores). Furthermore, the Xeon Phi cores are able to process multiple threads simultaneously, increasing the amount of parallel processes running during execution. Finally, each computational resource can take advantage of the Single Instruction Multiple Data (SIMD) paradigm by utilizing their Vectorization Processing Units (VPU) and AVX instruction sets.

There are further attributes to the Xeon Phi line of products that allow them to be considered more mature and industrial-grade products. The SCC chip required a host PC to manage and oversee its function. A suite of tools specifically made for monitoring of SCC's status, as well as its voltage and frequency scaling, are required by the user of the management console in order to utilize the SCC. This design complicates the usage of the SCC, especially when aiming at utilizing multiple SCC processors for the parallel processing of a demanding task, such as large network simulation. Multi-SCC usage is further complicated by the existence of the RCCE library, as mentioned in Section 3.2.1, which excludes the usage of more traditional HPC tools, such as the MPI library [38]. These concerns are lifted when moving to the Xeon Phi line of products, whose coding paradigm is simpler to manage and support native code execution. In particular, the 2nd generation of Xeon Phi processors, the Knights Landing, is a standalone manycore processor which operates like a regular x86-based processor.

These aspects motivated the porting of the simulator, as described in Chapter 3, to the Xeon Phi line of manycore processors. The simulator grew in the number of features supported, particularly in the domain of network connectivity patterns, as well as in modelling detail. As such, the more capable and easily programmable and maintainable Xeon Phi platforms were preferred for hosting the simulator.

4.2 Porting to Intel Xeon Phi 1st Generation

4.2.1 Platform Architecture

Intel Xeon Phi Knights Corner belongs to the Many Integrated Core (MIC) architecture; the processor model used for the mapping of the previously presented simulator features 61 cores with multithreading capabilities and VPUs, allowing Single Instruction Multiple Data execution of FP operations.

The Intel Xeon Phi Knights Corner co-processor is primarily composed of processing cores, caches, memory controllers, PCIe client logic, and a high-bandwidth bidirectional ring interconnect [2]. Each core in the KNC co-processor is designed to be power efficient while providing a high throughput for highly parallel workloads. The core uses a short in-order pipeline and is capable of supporting 4 threads in hardware. Each core comes complete with a private L2 cache that is kept fully coherent by a global-distributed tag directory. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the co-processor and the PCIe bus, respectively. All these components are connected together by the ring interconnect.

The Intel Phi card is treated as an accelerator and requires a Xeon host to boot a Linux image on it; however it can also be thought of as a standalone processor, executing any series of instructions independently from the main processor. In contrast to the SCC, the

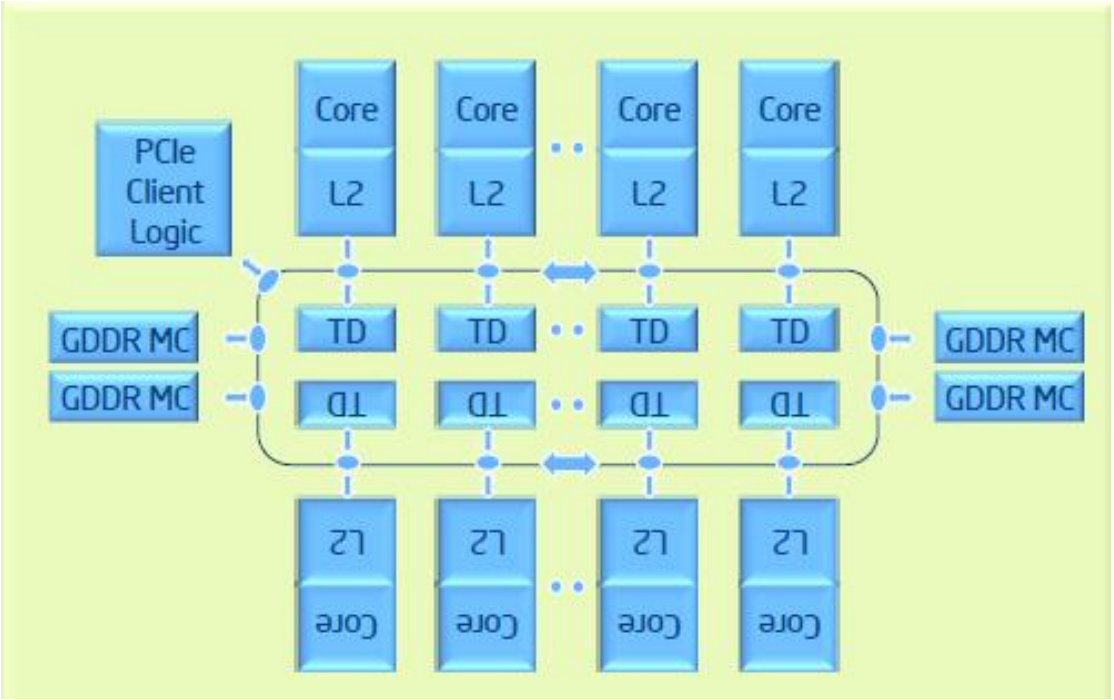


FIGURE 4.1: The Knights Corner die organization [2]. The reader can notice the bidirectional ring that constitutes the communication avenue for the cores of the Knights Corner co-processor [1]. Each core is accompanied by a private L2 cache that is kept fully coherent by a global-distributed tag directory (TD). The bidirectional ring also connects to the PCIe bus and to the GDDR5 memory via respective controllers.

Linux image on the Knights Corner allows to directly operate in the manycore processor natively, usually through an ssh connection that passes through the Xeon host.

Multiple Intel Xeon Phi co-processors can be installed in a single host system. Within a single system, the co-processors can communicate with each other through the PCIe peer-to-peer interconnect without any intervention from the host. Similarly,the co-processors can also communicate through a network card such as InfiniBand or Ethernet, without any intervention from the host.

A developer can use the Phi to code via traditional tools of parallel programming, such as the OpenMP library [59]. This fact differentiates it from other acceleration platforms such as GPUs by allowing the programmer to avoid using specialized libraries, such as CUDA [162] or OpenCL [163], allowing for rapid code development. This benefit is partially tempered when attempting to exploit the platform's SIMD instructions, a task which is not trivial for complicated codebases. To that end, there exist tools that aid with the use of the VPUs and enhance vectorization [164].

The VPUs are an especially important asset for the manycore processors examined in this thesis. For the KNC, the VPU features a 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions (Intel IMCI). Thus, the VPU can execute 16 single-precision (SP) or 8 double-precision (DP) operations per cycle. The VPU also

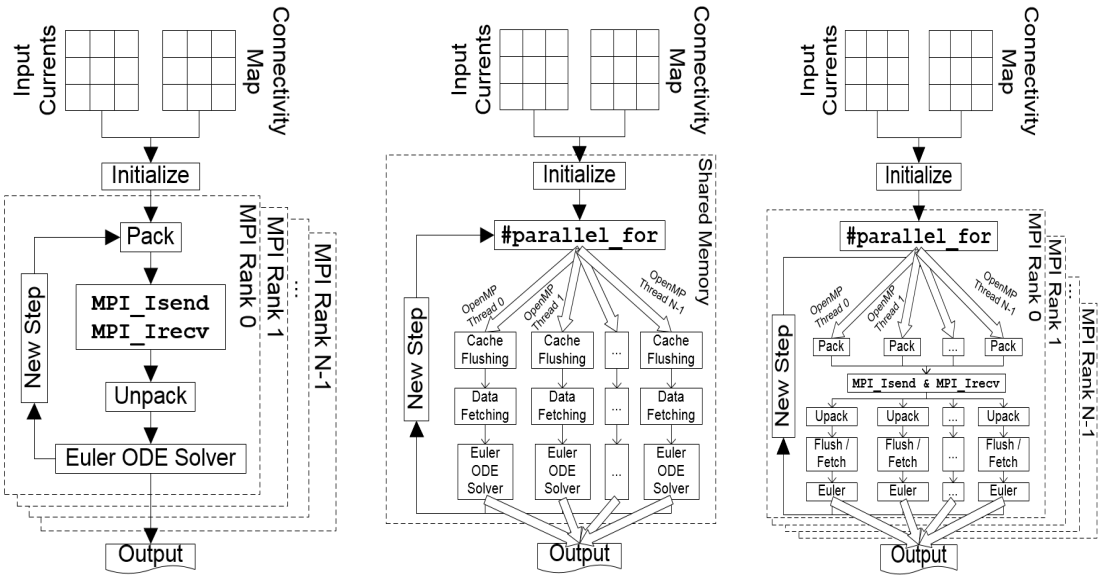
supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers. Vector Processing Units are designed to be power efficient for HPC workloads. A single operation can "package" multiple functionalities and does not incur energy costs associated with fetching, decoding, and retiring many instructions.

In order to support such "tightly-packed", power-efficient SIMD instructions, adjustments were made to the KNC architecture. A mask register supports the VPU to allow per lane predicated execution. This helps in vectorizing short conditional branches, thereby improving the overall software pipelining efficiency. The VPU also supports gather and scatter instructions, which are non-unit stride vector memory accesses, directly in hardware. Thus, for codebases with sporadic or irregular access patterns, vector scatter and gather instructions help in keeping the code vectorized. The VPU also features an Extended Math Unit (EMU) that can execute transcendental operations such as reciprocal, square root, and log, thereby allowing these operations to be executed in a vector fashion with high bandwidth. The EMU operates by calculating polynomial approximations of these functions.

For the purposes of evaluating the benefits and costs of exploiting the platform to its fullest, two varieties of implementations will be presented. Initially, implementations that are generic enough to be seamlessly portable on both the processor and the coprocessor will be evaluated. Later on in this Section, the most promising generic implementation for the neuronal simulator will be enhanced with fine-grain vectorization in order to take full advantage of the underlying platform. The process of increasing the effectiveness of vectorization can be applied to any program running on a VPU-equipped manycore processor; however, the details of each processor's architecture may force recompilation, or tweaking of the codebase in order to attain the best results specifically for each processor.

4.2.2 Application Mapping

The simplest method a programmer can employ to parallelize a neuron modeling (or any other) workload is to assign different parts of the workload to different cores. Each core computes independent parts of the workload and communicates with other cores in order to complete operations that require input from them. Our InfOli model is data-partitionable in the above way, with communication imposed by gap junctions. These constitute the biological mechanism through which a neuron receives input from connected neurons based on the voltage differential of the respective dendritic membrane potentials. Thus, in each step of the simulation, collection of the dendritic membrane potential from all neurons that connect to any of the core's neurons, is required. The connection is determined by a user-defined *connectivity map*. This task requires inter-core communication that can be achieved by using several different programming paradigms,



(A) MPI (B) OpenMP (C) Hybrid

FIGURE 4.2: Flowchart of the implementations discussed in this chapter [3].

such as MPI, OpenMP or a hybrid combination of the two. The InfOli simulator has been ported to the KNC according to these configurations, as presented in Figure 4.2. These three alternatives are explored in the following Subsections.

4.2.2.1 MPI Implementation

MPI is a library for distributed-memory systems where coordination between cores is achieved by message-passing through fast-memory buffers. While both the Xeon host and the Phi co-processor share memory between their cores, the MPI implementation is still useful for evaluating the shared-memory performance and serves as a baseline towards the hybrid method. It is also a well-supported and continuously updated tool which can aid in multi-node-system implementations [165, 166]. In such multi-node systems, message passing is achieved over TCP or Infiniband. In a single-node system, shared memory is used instead.

The primary unit of execution of this implementation is the MPI rank, with a one-to-one correspondence between ranks and cores. Each rank handles a subset of the neuronal network as well as its data-exchanging needs; in order to properly model gap junctions, neurons exchange states before simulating each time-step. One approach to this task is to perform neuron-to-neuron communication based on user-defined connectivity and the respective MPI commands. Assuming a neuron population N, the worst-case number of `MPI_Isend` and `MPI_Irecv` pairs executed is $(N-1)^2$.[1]

[1]Note that the MPI functions for transmitting and receiving data used in this work are asynchronous. The non-blocking nature of these functions facilitates irregular core communication. This trait was taken advantage of, since the InfOli simulator aims at supporting any network interconnection pattern, causing unpredictable core-communication schemes.

Furthermore, we explore an alternative data-grouping technique, whereby data is exchanged in buffers. Each MPI rank consolidates all dendritic membrane potentials that are to be sent to another core into a single buffer. *Packing* is the procedure of determining which values need to be sent over and then, bundling them together in one buffer, designated for the recipient core. *Unpacking* is the procedure of analyzing data in the received buffer so as to distribute the values to the neurons that need them based on the the neuron connectivity map. Assuming k MPI ranks, the worst-case number of `MPI_Isend` and `MPI_Irecv` pairs is $(k-1)^2$, whereas each bundle contains at the most N/k more data than the naive neuron-to-neuron case.

Both packing and unpacking take place in each iteration of the InfOli model. During the first simulation step, each core "marks" neurons that are necessary for a core-data-exchange; packing marked data into the buffer can then be performed efficiently. Unpacking, however, requires each neuron to extract marked data from the buffer, which is done in a sequential manner since no OpenMP threads are employed in this implementation. Thus, while packing is completed in each step with little computational effort, unpacking imposes a non-negligible overhead. However, this bundling technique is more efficient than issuing MPI calls for every neuron that needs to communicate.

Figure 4.2a describes the MPI implementation. The simulator starts by initializing the neuron states and processes the connectivity map. The neuron population is divided and assigned to MPI ranks. The execution then proceeds to the main loop which lasts for a fixed, user-defined number of simulation steps. In each step, the neurons receive input stimulus current. The cores then pack their data in $k-1$ buffers. These buffers are exchanged via asynchronous MPI communication and unpacked. Each core checks for the completion of all relevant MPI communication functions to ensure that neurons have access to updated data regarding their connections to other neurons. Only then can the neuronal network be updated to its new state while avoiding stale data-propagation. Each core performs a set of calculations for each neuron it handles and stores the neuron's new state values locally. The simulation step then ends and the cycle begins anew.

4.2.2.2 OpenMP Implementation

OpenMP uses `#pragma omp` directives to designate parallel regions of code to the compiler. We mainly use OpenMP's `#parallel_for`, which flags the iterations of a `for` loop as eligible for parallelization. Since data-exchange is transparent in OpenMP and does not involve manual coordination of message-passing, there is no need to pack and unpack data. This makes the OpenMP implementation much simpler in terms of coding effort.

The primary unit of execution is the OpenMP thread. The main loop of the InfOli model is divided between threads with `#parallel_for`. Each thread handles a different part of the network, much like the MPI ranks do in the message-passing implementation. Since

memory is shared between the primary units of execution, each unit can freely access another unit's data, thus allowing the dendritic-voltage exchange to be a completely local and independent process. Pure computation (i.e. solution of the respective ODE) is also carried out locally, allowing the whole loop to be parallelized efficiently. On the other hand, since data is shared between different cores, preserving cache coherence introduces MESI-protocol-related overheads. This may cause the implementation to slow down considerably; this behaviour becomes particularly prominent when the network solver operates on a small-sized network and is saturated with too many OpenMP threads. Determining the optimal number of threads to alleviate the burden of such overheads is important for the OpenMP implementation.

4.2.2.3 Hybrid Implementation

An OpenMP implementation, as proposed in the previous Subsection, may appear to be the most intuitive parallelization strategy for the InfOli model. Given both implementations' strengths and weaknesses, it may be interesting to explore a hybrid implementation, combining both MPI and OpenMP. This course of action is even more compelling for the Phi, since it combines the platform's multithreading capabilities and the option to distribute the workload across multiple Phi cards. The hybrid implementation developed stems primarily from our MPI implementation (Subsection 4.2.2.1). While the primary unit of execution is an MPI rank, each MPI rank further spawns OpenMP threads to create a hybrid porting. These threads are used to boost packing and unpacking, as well as the main computation process.

In Figure 4.2c, we organize the cores of a platform into *groups*. Within each group, all cores communicate over shared memory. They spawn OpenMP threads to perform and accelerate computations. Each group is perceived as a single MPI rank in the MPI environment. Within the group, one "master" core handles MPI communication and sends necessary data from the entire group to another shared-memory group on every simulation step. The packing and unpacking of this data is performed by the OpenMP threads spawned by the entire group. Only the actual MPI calls are performed in single-threaded fashion by one core per group.

This implementation treats any single-node system (with processor and co-processor) as a potential multi-node one. It aims at dividing its computing resources (hardware cores and instruction streams) in standalone islands of shared memory that communicate with each other via message passing. This method is logically extensible to multi-node platforms, assuming that computing resources of different nodes belong to different shared-memory islands. Thus, it serves as a bridge from single- to multi-node systems. The granularity of the hybrid implementation shall be expressed as the ratio of MPI ranks to OpenMP threads spawned by each rank. Similar to the pure OpenMP case,

this ratio needs to be fine-tuned in order to minimize the overheads of OpenMP threads (stemming from maintaining cache coherence) and of implementing message-passing.

4.2.3 Experimental Evaluation

Thorough experimentation has been undertaken in order to evaluate the simulator's performance on the KNC. This Section focuses on the evaluation of the initial implementation of the simulator on KNC, where vectorization has been enabled in the Intel compiler (icc), but not fine-tuned specifically for the platform. Each implementation, as presented in Figure 4.2, will be evaluated in its dedicated subsection.

4.2.3.1 Experimental Setup

All experiments performed involve a simulation of $5s$ of brain time. This time interval is sufficient for our measurements since the InfOli simulator represents a deterministic workload of highly predictable behaviour - which is typical of time-driven simulators. The simulator has been set up to operate with a constant step δ= 50 μs due to modeling-accuracy requirements. Thus, the entire simulation ends after 10^5 simulation steps. Neuronal networks simulated in this work are represented as a three-dimensional (3D) mesh. Furthermore, network topology is important, since it dictates the coordinates of each neuron based on which a pseudo-distance between any two neurons is calculated. For these calculations, the model does not take into account the geometrical properties of individual neurons. The simulator treats each neuron as a point in the 3D-space. The functionality of each gap junction is unaffected by the neurons' spacial orientation and size, thus this level of detail is assumed to be sufficient for the model.

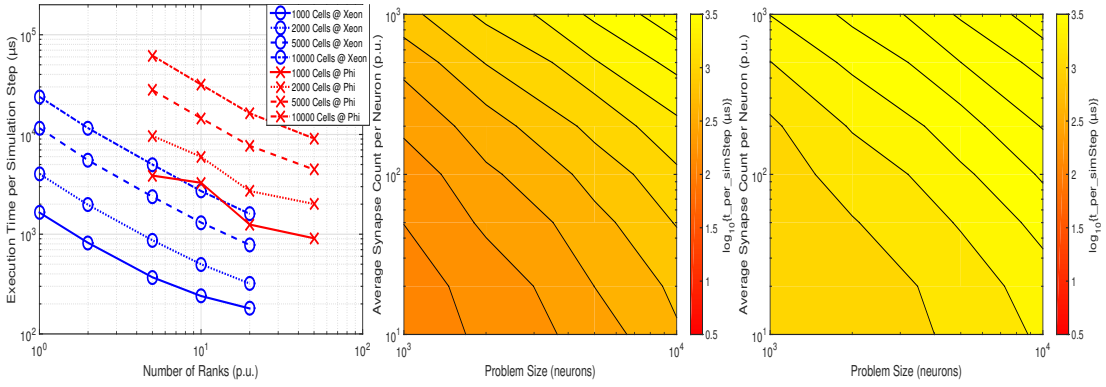
As far as network connectivity is concerned, we employed two different connectivity patterns. Firstly, we made an assumption reasonable for the inferior-olivary nucleus that the closer neurons are to one another, the more likely they are to form connections (i.e. gap junctions) and exchange information. Thus, we employed a probabilistic connectivity pattern where the probability of connection between two neurons depends on their cartesian distance and is calculated using the formula of a normal distribution, with a standard deviation of 5 neurons. This deviation results in an average of $50 - 200$ connections formed per neuron, varying with network size, which is an adequate connectivity density for the purposes of testing the implementations' scaling capabilities. Alternatively, we also explored set amounts of connections per neuron, which allows a more direct control over the network's density. Experiments carried out using this connectivity pattern can provide insight into how each implementation handles increased network traffic and data-storage needs.

Connections are created at a pre-processing stage (based on the aforementioned patterns) and are stored in the connectivity map of the simulation, represented as an adjacency

matrix. To decrease input file sizes, sparse-matrix formats are used for large neuronal networks ($\geq 10,000$ neurons).

The entirety of our experiments has been carried out in the Blue Wonder cluster, at the Hartree Center of the Science & Technology Facilities Council (STFC) in the United Kingdom. Access to Xeon/Xeon Phi systems (one single node) is enabled over `ssh`. Each node contains an Intel Xeon E5-2697v2 processor (dual-socket arrangement) with 64 GB of RAM, operating at 2.7 GHz and one Intel Phi 5110P accelerator. All measurements presented in the current paper have been taken from execution of the target application on a single node of the cluster. The Intel MPI compiler (Intel MPI-5.0.3) is used for the MPI and hybrid implementations. The Intel C compiler (Intel compiler 15.0.2) is used for the OpenMP implementation, as well as the related Intel OpenMP runtime library (following the OpenMP 4.0 standard). Performance measurements of small-scale runs (maximum network size of 10,000 neurons) have been performed using the Intel VTune performance analyzer (Intel VTune Amplifier XE 2016). Since VTune collects hardware events during execution, its overhead (both in execution time and disk space) is prohibitive for larger simulations; the Linux default `time` command (GNU time 1.7) has been used in large-scale simulations instead. We present performance measurements in execution time per InfOli simulation step, so that we mitigate transient effects at simulation start/end, as well as depict the simulator's performance in a manner more easily comparable to related work in the literature.

4.2.3.2 Evaluation



(A) MPI

(B) MPI Network Density Exploration - Xeon

(C) MPI Network Density Exploration - Phi

FIGURE 4.3: Depiction of MPI measurements on the host and the Phi.

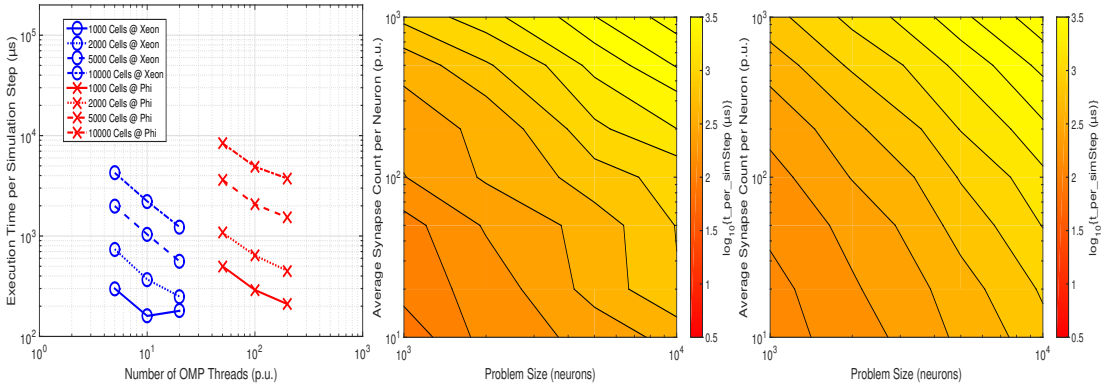
The three implementations have been tested natively both on the Xeon host and the Phi co-processor, creating a total of six different evaluations. We explore the granularity of each implementation and evaluate a variety of problem sizes (neuron populations, network density). Findings are reported next.

In Figure 4.3, we present results for the pure MPI implementation. Figure 4.3a explores the granularity of the implementation by varying the number of MPI ranks. We used up to 20 ranks for the Xeon processor and up to 50 ranks for the Phi co-processor, since it offers more hardware cores than the Xeon processor. The figures reveal that the Phi is outperformed by the dual-Xeon processor host. One of the primary reasons for this is the inability of a strictly MPI-based implementation to take advantage of the entirety of the Phi's computing resources. The Phi accelerator cards feature cores that base much of their processing power on their multithreading capabilities, capable of supporting up to 4 instruction streams in parallel. The MPI implementation however, only uses a single thread per rank.

From the figure, we also observe a difference in performance gains as we employ more MPI ranks. There is an irregularity in efficiency when simulating relatively small networks of 1,000 to 2,000 neurons on the Phi device. On the Xeon host, there is reduced performance gain for a network of only 1,000 neurons as we reach 20 MPI ranks. This behavior suggests that such problem sizes pose relatively small workloads that do not fully exploit the computational resources of each platform, especially the Phi. The aforementioned trend disappears for problem sizes increase to at least $5,000$ neurons. When solving for such networks, we get a near-linear performance gain when we increase the number of MPI ranks employed. This behavior is interesting since, as explained in Subsection 4.2.2.1, an increase in MPI ranks increases communication overheads in data exchange as well as in data packing and unpacking. Linear performance gains, on the Phi device in particular, indicate the following: given that the MIC architecture focuses on high memory bandwidth, it can handle the scaling message exchanging demands imposed by as many as 50 communicating MPI ranks, as long as the workload per rank is large enough.

This statement is further supported by data in Figures 4.3b and 4.3c, where the MPI implementation is tested with the maximum amount of MPI ranks available on both platforms, for networks representing varying degrees of communication activity. The Xeon host consistently remains the better choice out of the two computational fabrics for the MPI implementation. However, a variation in the performance differences is observed with varying degrees of network density and size. For sparse and small networks ($1,000$ neurons with $10-20$ synapses each), the Xeon host outperforms the Phi by a margin of 10-20$\times$. As networks grow denser and larger, the performance difference becomes less pronounced, down to a range of 3-4$\times$ for networks of $10,000$ neurons, each with $1,000$ synapses.

In Figure 4.4, we illustrate the performance assessment of the OpenMP implementation. For network sizes between 1,000 and 10,000 neurons the shared-memory implementation works better on the Phi device compared to the purely MPI equivalent. Particularly for smaller networks, OpenMP runs for a fraction of the execution times reported for MPI. By design, Phi supports many more threads (up to 240 when fully utilizing 60 cores)



(A) OpenMP

(B) OpenMP Network Density Exploration - Xeon

(C) OpenMP Network Density Exploration - Phi

FIGURE 4.4: Depiction of OpenMP measurements on the host and the Phi.

than the Xeon processor. Thus, in the OpenMP paradigm, we can exploit the accelerator's resources much more aggressively. This leads to a performance improvement when compared to MPI in the case of the Phi. Besides, the Xeon host can be expected to have similar performance between MPI and OpenMP, given that the number of threads that can be supported is smaller.

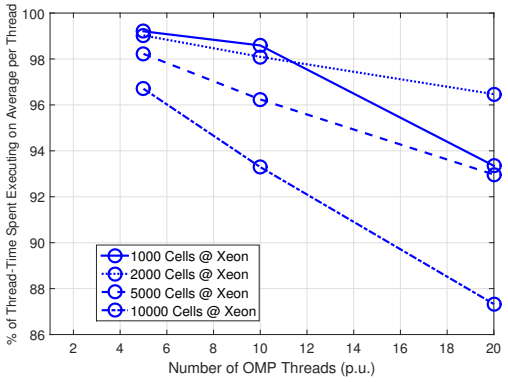


FIGURE 4.5: OpenMP thread activity on the Xeon host.

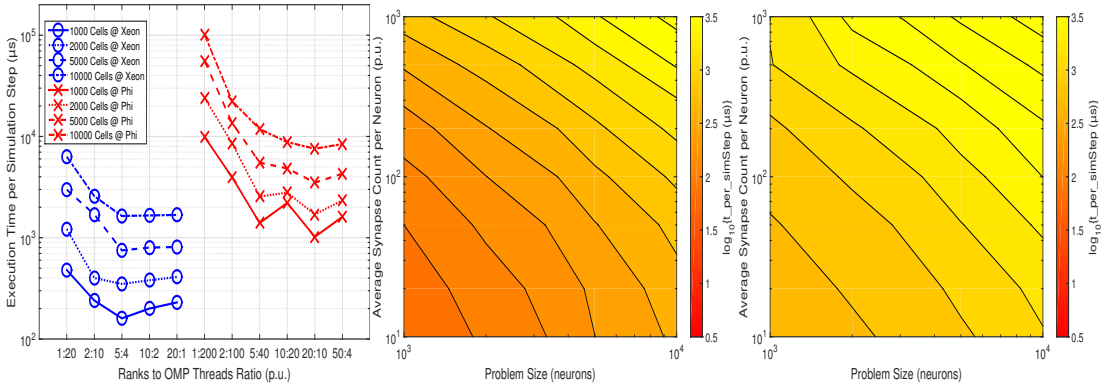
According to Figure 4.4a, the Phi accelerator's performance increases in a near-linear fashion with the amount of OpenMP threads invoked. This is an expected observation if we consider the design of the Phi as a platform for massive multithreading. In contrast, the Xeon host does not exhibit consistent scaling as more OpenMP threads are added for small populations of simulated neurons. More specifically, after using half of its resources, employing more OpenMP threads does not speed the processor further up.

We can generally form the following hypothesis for non-linear scaling of small neuron populations on the Xeon host: The *benefit* of the OpenMP implementation is that, for sufficiently large problem sizes, increasing the available thread count reduces the computational burden assigned per thread. In other words, more threads means less simulated neurons per thread. On the other hand, the *cost* of an OpenMP implementation is

related to the overhead of shared-memory operations. More OpenMP threads on the same platform results in thread concurrency taking a larger hit due, for instance, to race conditions on shared resources. Conclusively, and as the Xeon host's performance in 4.4a indicates, small problem sizes can be efficiently tackled with a small number of threads. On the contrary, when the problem size is sufficiently large, initializing more OpenMP threads is beneficial, the coherency penalties notwithstanding.

The Xeon host's OpenMP performance issues for small workloads are further supported by data in Figure 4.5. By using Intel VTune to analyze the application performance on the host, we collected data concerning the OpenMP threads CPU time, which is defined as: "the amount of time a thread spends executing on a logical processor and, for multiple threads, the CPU time of the threads is summed" [167]. By dividing the collective CPU time with the number of threads employed by an application, we thus calculate the time spent executing on the processor, averaged across all threads. We then compare this time against the real elapsed time of the workload to calculate the percentage of time spent executing on the processor, averaged across all threads. It is then, demonstrated that, for $1,000$ neurons, using 20 threads drastically decreases the average time of thread activity. When not executing on the processor, the threads are idle, as would be the case of waiting for thread synchronization. This idleness appears to be prevalent when spawning multiple OpenMP threads for small workloads.

In addition, in Figure 4.4b, the increase in execution time per simulation step of the OpenMP implementation is erratic when examining smaller and sparser networks. This observation builds further upon the statement that the Xeon host's performance is relatively inefficient when using the maximum number of OpenMP threads for small workloads. On the contrary, for larger and denser networks, the Xeon host performs in a more predictable manner. Furthermore, the Phi accelerator, in Figure 4.4c, features a more linear increase in execution time as the workload increases.



(A) Hybrid

(B) Hybrid Network Density Exploration - Xeon

(C) Hybrid Network Density Exploration - Phi

FIGURE 4.6: Depiction of Hybrid measurements on the host and the Phi.

In Figure 4.6, we present the performance assessment of the hybrid implementation. We examine a range of ratios between the number of MPI ranks and the number of OpenMP threads each rank spawns. For this implementation, multiplying the number of MPI ranks employed by the corresponding number of OpenMP threads a rank utilizes yields the total amount of OpenMP threads spawned across the platform. In all measurements presented for this implementation, this number will always be equal to 20 for the Xeon host and 200 for the Phi accelerator. In this manner, the implementation takes advantage of each platform's assets in a consistent manner throughout the ratio-sweep. In general, measurements shown in 4.6a indicate that a "middle-of-the-road" ranks-to-threads ratio yields the best performance.

In the case of the Xeon host, spawning 5 MPI ranks, with each rank using 4 OpenMP threads, offers the best results. Using more MPI ranks does not offer any additional benefit. A similar behavior is observed on the Phi co-processor. A configuration of 20 MPI ranks, each spawning 10 OpenMP threads, offers the best performance. Additionally, simulations of reduced neuron populations using the hybrid implementation on the Phi exhibit performance unpredictability beyond the 5:40 rank-to-thread ratio. In general, we observe that both platforms perform better with a moderate balance between MPI ranks and OpenMP threads. Using these ratios, extensive measurements for networks of varying size and complexity are depicted in Figures 4.6b and 4.6c.

Moreover, the hybrid implementation appears to be performance-bound by the two previous ones (strictly MPI or OpenMP): On the one hand, when too many ranks are employed, the implementation behaves more or less like the purely MPI codebase. Apart from the message-passing overhead, a slight performance drop is attributed to OpenMP thread creation and maintenance. When few MPI ranks are deployed and shared-memory threads are emphasized, the application behaves more or less like the OpenMP implementation. Apparently, a balanced configuration distributes the burden of message exchange between a reasonable number of core-groups, while keeping the workload of each group big enough in order to near-maximally utilize computational resources for OpenMP thread maintenance. Thus, for the hybrid implementation as a whole, balanced configurations appear to minimize the combined message-passing and shared-memory overheads.

Having swept the parameters of the discussed implementations on the Xeon host and Phi co-processor, we attempt to scale the best of them to neuroscientifically-relevant problem sizes, namely beyond half a million neurons [168]. From each of the discussed implementations, we isolate the configurations behaving optimally in Figures 4.3a, 4.4a, and 4.6a and increase the simulated neuron populations. The results of this final set of experiments are illustrated in Figure 4.7. As expected from Figures 4.3a, 4.4a, and 4.6a, the Phi accelerator cannot compete with the host for native execution of these implementations. Clearly, the only way for potentially gaining more performance from

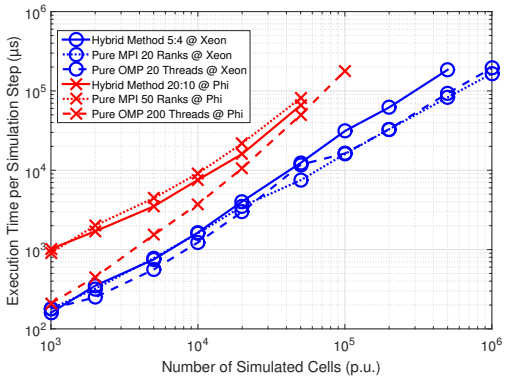


FIGURE 4.7: Comparing the best implementations on host and accelerator, before manual AVX-oriented optimizations.

the MIC accelerator is by performing source code vectorization after identifying the implementation of choice for the Phi.

After examining each implementation independently, we observe that the Phi behaves significantly better under a shared-memory programming paradigm. For smaller networks, it achieves execution times that are comparable to those of the Xeon host, even without manual code vectorization. However, as network size increases, OpenMP implementations display a steeper performance curve. This causes MPI-based methods to catch up with the shared-memory implementation when solving for networks of more than 20,000 neurons. At this point, the message-passing-based porting methods, that aim at dividing the network in groups, become attractive. Furthermore, MPI-based implementations are the only viable option for carrying out large simulations on a multi-node system with Phi cards.

Comparing the two message-passing methods on the accelerator, the hybrid implementation outperforms the pure MPI method by drawing on more of the platform's resources. A small performance gain is created by spawning OpenMP threads in each MPI rank, that slowly grows as network size increases. The hybrid approach was expected to improve on the MPI implementation by a wider margin. The overhead of spawning OpenMP threads on each simulation step appears be a limiting factor to its efficiency.

On the Xeon host, all implementations perform comparably to each other. In a similar fashion to the Phi platform, OpenMP is the implementation of choice for smaller networks. When the problem size scales to very large networks of more than $50,000$ neurons, we can observe a trend where pure-MPI and OpenMP implementations outperform their hybrid utilization. On the Xeon host, hybrid coding is not an improvement over the strictly MPI method since the platform does not support multithreading.

All three implementations can use the entirety of the Xeon computing capacity and the larger workloads demand a pure approach, rather than a hybridized one. There is no significant difference between MPI and OpenMP when aiming at simulations of more

than 10^5 neurons. We discern a slight inclination for MPI to outperform OpenMP when the network reaches the 1 million neurons barrier.

4.2.4 Vectorized Implementation

In order to exploit the Phi device to its fullest potential, extensive micro-optimizations, as well as code transformations are needed. Section 4.2.3.2 showed clearly that the Phi's performance is inadequate without spending development time in optimizing the code-base. The initial study of the un-optimized code presented evidence that the OpenMP programming paradigm provides the most efficient porting solution. Thus, the efforts of drawing the platform's resources were focused on the shared-memory version of the simulator. This decision was reinforced by the fact that the MPI message-exchanging functions are incapable of using VPUs for acceleration.

4.2.4.1 Basics of Vectorization

VPUs are units that enable fine-grain parallelism and are present on Intel's Xeon architecture - both on Xeon processors and Phi cards. At their core, VPUs are registers that allow the execution of a specific instruction set, named Advanced Vector Extensions (AVX) [169], which is an extension to the well-known x86 instruction set. AVX originated in an older extension to the x86 instruction set designed to support Single Instruction Multiple Data (SIMD), named Streaming SIMD Extensions (SSE) [170]. AVX features multiple versions and varying width in the vectorization registers, with current Intel Phi models (Knight's Corner - KNC) supporting AVX2 and future models (i.e. Knight's Landing - KNL), supporting AVX512.

The Knight's Corner Phi that is under evaluation in this work utilizes 512-bit wide VPUs. They allow up to 16 single-precision or 8 double-precision operations to be carried out simultaneously by each of the 240 hardware threads of the device. There already are multiple case studies taking advantage of VPUs and AVX instructions to significantly boost the performance of evaluated applications from a variety of scientific fields [171, 172].

In practice, the application developer should picture the VPUs as an effort to unroll and parallelize the iterations of a loop, whereas they would otherwise be executed sequentially. This level of parallelism requires that the loop's iterations can be executed independently from each other and in any sequence. This is not always possible; for example, loop iterations may present Read-After-Write (RAW) and Write-After-Read (WAR) dependencies when reading and storing data in the same memory addresses.

The developer is assisted in vectorizing his code by the compiler's optimizations, which circumvent some of these limitations. In other cases, regions of code are designated as not

vectorizable, due to dependencies that cannot be avoided by the compiler in an automatic fashion. In this work, we used the Intel C Compiler (ICC) to compile our application for the Phi architecture and enabled the compiler's built-in vectorization assistance by compiling with the `-vec-report` flag for Linux Operating Systems. Furthermore, there are various documents detailing guidelines for efficient vectorization particularly on the Phi [173, 174].

In our study, a number of steps was taken in order to vastly improve the efficiency of AVX instruction implementation. Each step introduces some form of modification of the codebase. We classify these modifications under two general categories. There are steps that should be taken into consideration by any developer that aims at porting an application on an AVX-compliant computing fabric, regardless of the application's nature. We also performed transformations that fit the particular algorithm used for this simulator and can be of use in other codebases that follow similar patterns. For ease of reference, we term the former as *generic modifications* and the latter as *specialized transformations*.

4.2.4.2 User-assisted Dependency Disambiguation (DD)

In order to ensure correct program functionality, the compiler assumes a conservative approach when determining the existence of a dependency. If the limits of data structures cannot be calculated with certainty, which is often the case for dynamically-allocated data, then the compiler is forced to assume that segments of memory appointed to different structures may overlap. The case of accessing the same memory address under two different names, such as by using two pointers with the same value, is called *aliasing* and it forces conservative compilation without SIMD-operations in order to protect an application's coherence.

In Figure 4.8, the compiler may be unable to determine whether pointers a and b refer to entirely separate memory regions, particularly if there is no pre-compiling information regarding the region sizes. However, when a developer is certain that assumed dependencies and aliasing-caused precautions can be ignored, the compiler may be instructed to override its assumptions and produce vectorized loops.

In the case of the icc compiler, this can be achieved using the `#pragma ivdep` directive, as demonstrated in algorithm 4.8. This is a *generic modification*; in this particular example, the developer is aware that proper coding ensures there is at least $4 * upper\_bound$ bytes worth of memory space separating the values of pointers a and b; thus, there are no memory accesses in this loop for pointer a that could interfere with pointer b's and vice versa. Declaring pointers using the `restrict` keyword is also recommended, acting as a way to communicate to the compiler that the developer guarantees exclusive memory accessing.

```
1: int * restrict a, * restrict b;
2: ...
3: #pragma ivdep
4: for i = 0 to upper_bound do
5:    b[i] = a[i] * constant_k;
6: end for
```

FIGURE 4.8: An example of preventing aliasing.

```
1: int *a = _mm_malloc(upper_bound*sizeof(int), 64);
2: int *b = _mm_malloc(upper_bound*sizeof(int), 64);
3: ...
4: #pragma ivdep
5: for i = 0 to upper_bound do
6:    b[i] = a[i] * constant_k;
7: end for
```

FIGURE 4.9: Using `_mm_malloc`.

4.2.4.3 Inline Expansion (IE) and Memory Alignment (MM)

By vectorizing a loop, memory accesses that would otherwise take place in different iterations of the loop happen in parallel. Hence, it is imperative that when cache lines are fetched from the main memory, all simultaneous memory accesses are satisfied. To this end, data structures need to be aligned with cache lines; this essentially means that each data allocation for a structure begins in an address that is also the beginning of a cache line. Vectorized accesses to the memory space of an aligned data structure coincide with a single cache-line-fetching. Memory alignment is a crucial step that avoids latency in the execution of SIMD-instructions due to multiple cache-line-retrievals for a single instruction. Since this applies to any application regardless of its nature, this is a *generic modification*.

In order to ensure aligned memory allocations, the developer is encouraged to avoid using standard C `malloc` function calls and opt instead for Intel's `_mm_malloc`. This icc-compatible function allows the developer to ensure that data allocation will begin at an address divisible by the size of the platform's cache line. As shown in Figure 4.9, for the Phi (KNC) architecture with cache-line size of 64 bytes, data structures need to be allocated at an address that is divisible by 64, whereas the Xeon host has 32-byte alignment.

In addition, using function calls in a vectorized loop is discouraged. When vectorizing a loop, an identical instruction pattern must be maintained across all iterations so that their execution can be parallelized. Instructions that alter the flow of a program, such as conditional instructions and function calling, can pose obstacles for efficient vectorization. Function inlining is a practice extensively researched [175] and automatically performed by the compiler in many cases; however, in our work, manual inline expansion proved beneficial in regions of code where the compiler did not intervene.

```
1: for i = 0 to NW_Size do
2:    #pragma ivdep
3:    for j = 0 to Synapse_Count do
4:       incoming_current[i][j] = calculate_synapse(i, j);
5:    end for
6:    calculate_new_state(incoming_current[i], i);
7: end for
```

FIGURE 4.10: Nested Loop example.

```
1: for i = 0 to NW_Size do
2:    #pragma ivdep
3:    for j = 0 to Synapse_Count do
4:       incoming_current[i][j] = calculate_differential(i, j);
5:    end for
6: end for
7: #pragma ivdep
8: for i = 0 to NW_Size do
9:    calculate_new_state(incoming_current[i], i);
10: end for
```

FIGURE 4.11: Split Loop example.

4.2.4.4 Vectorization-Driven Loop Splitting (LS)

As mentioned before, vectorizing a loop involves using SIMD operations in order to execute iterations of a loop in parallel. In the case of nested loops, the compiler always chooses the innermost level of the loop to vectorize. This decision is supported by the fact that vectorized loops include as few alterations in the execution of each iteration as possible. Should the compiler vectorize the outer level of a nested loop, the produced vectorized code would include the conditional branching instructions of the inner loop, which would hamper the performance of the program.

The described behaviour also forces the compiler to ignore any other instructions contained in the nested loop outside its innermost layer. Nonetheless, a program's performance may be affected by operations outside the inner loop. In Figure 4.10, an example from the InfOli simulator presented in this paper is given. The algorithm operates in two phases: for every neuron in the network, the simulator calculates the effect each synapse has on the neuron, represented in the inner loop, and then evaluates the changes in the neuron's state. While the former phase claims a large portion of the total workload, the latter phase features a large amount of exponential functions due to neuron-channel calculations.

In order to produce vectorized code for both phases of the algorithm, we split the simulation loop in two dedicated loops. The first is a two-layer loop for the synapse-evaluation phase whereas the second is a single-layer loop estimating the changes in the neuron's state. The two loops are then vectorized separately; this also allows for exclusive

modifications in each loop's code. This *specialized transformation* is presented in Figure 4.11. The example contains function calls for the sake of clarity and compactness; however, as stated in Subsection 4.2.4.3, the functions' code has been inlined in the main loop.

4.2.4.5 Data Restructuring (DR)

Subsection 4.2.4.3 discussed the importance of matching the execution of vectorized regions of code with cache-line-aligned memory accesses in order to maximize the effectiveness of VPU usage. However, allocating data structures in aligned memory addresses does not guarantee optimal memory-access patterns. It is equally important to ensure that data is accessed in a serial, unit-stride manner across all iterations of the vectorized loop. Unit-stride memory references ensure memory is accessed in a sequential and continuous manner, which is important in the case of vectorized code, since memory accesses happen in parallel.

Since unit-stride memory accesses are paramount to obtaining good performance, data structures need to be designed accordingly. In Figure 4.12, a data structure is used that contains all relevant information for the main object under examination in this work, a neuron. While the structure presents the data in a meaningful and compact way, its usage in a vectorized algorithm proves to be problematic.

In the main loop, each of the neuron's channels is accessed and processed in a sequential manner. For the unvectorized code, data should be allocated in the memory in such a way that each neuron stores the entirety of its data, such as channel states and membrane voltage levels, as compactly as possible. In this case, the data structure presented in Figure 4.12 is beneficial to use. In order to generate a network of such neurons, the programmer would allocate an array of the **struct** Neuron.

However, in the case of vectorized code, the order of data accesses changes. Since there are parallel iterations of the loop, data from different neurons will be accessed simultaneously; the processor computes each of the model's parameters for the entire network in parallel. This order of memory accesses points towards storing each parameter's data from the entirety of the network as compactly as possible. In this case, a **struct**, such as Neuron, is unsuitable. In order for memory accesses to happen in unit stride, it is advisable to represent each of the model's parameters as an array that stores data for the entire network, as shown in Figure 4.13. These arrays can then be packed, if desired, in a different struct that represents the neuron network, rather than each neuron individually. This technique is an Array-of-Structs (AoS) to Struct-of-Arrays (SoA) *specialized transformation* and it, along with other data-structure transformations, has been extensively used in the literature, in multiple fields of HPC and SIMD computing [176, 177].

```
1: struct Neuron {
2: float Na;
3: float K;
4: ...
5: };
6: ...
7: #pragma ivdep
8: for i = 0 to NW_Size do
9:    calculate_channel_Na(Neuron[i].Na);
10:   calculate_channel_K(Neuron[i].K);
11:   ...
12: end for
```

FIGURE 4.12: Data represented as a struct.

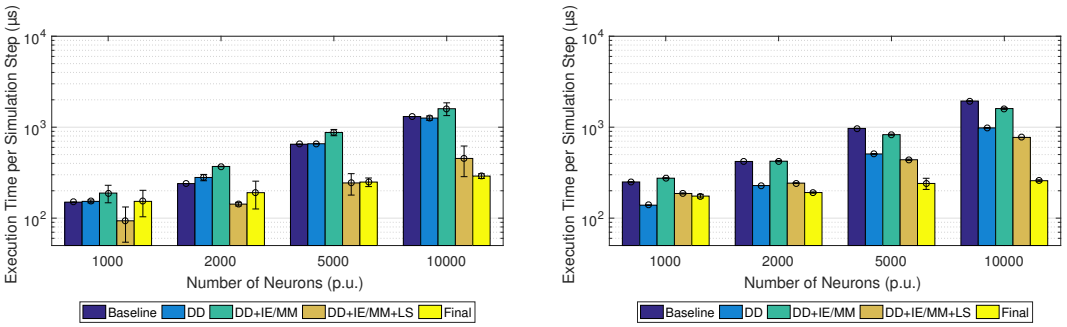
```
1: float *Na = _mm_malloc(NW_Size*sizeof(float), 64);
2: float *K = _mm_malloc(NW_Size*sizeof(float), 64);
3: ...
4: #pragma ivdep
5: for i = 0 to NW_Size do
6:    calculate_channel_Na(Na[i]);
7:    calculate_channel_K(K[i]);
8:    ...
9: end for
```

FIGURE 4.13: Data represented as multiple arrays.

4.2.4.6 Evaluation

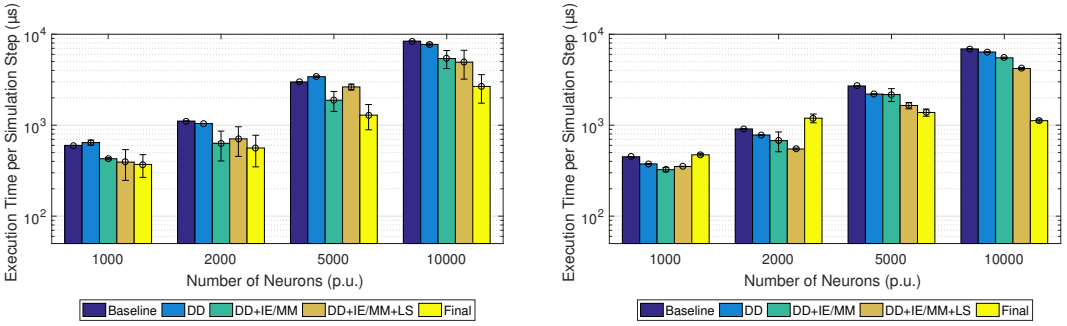
The simulator's codebase features a baseline version, which largely ignores the AVX instruction set due to the compiler's conservative strategy concerning assumed dependencies. The techniques mentioned in Subsections 4.2.4.2, 4.2.4.3, 4.2.4.4 and 4.2.4.5 are successively applied to this version, revealing a steady increase in the efficiency with which the simulator uses the platform's resources. The application's performance is then measured and the contribution of each modification is evaluated. In Figure 4.14, networks that are both sparsely and densely connected are tested, on both the host and the co-processor. Multiple measurements of each test are conducted and the mean value, along with an error margin corresponding to a confidence interval of 95%, is plotted. Different behaviour patterns are observed based on network complexity.

In the case of sparse networks, performance gains are highly dependent on the vectorization technique used, as well as the size of the network. Small and sparse networks present both platforms with a small workload. Particularly for the Xeon host, using the AVX instruction set does not guarantee a boost in performance. It is observed that the aforementioned techniques of Section 4.2.4, such as memory alignment and loop splitting, are mandatory in order to attain an improvement in performance; prior to applying them, small-workload-processing is not accelerated via vectorization. In this case, the Xeon host performs better without taking advantage of the VPUs.



(A) Sparse network on Xeon host (10 synapses per neuron)

(B) Sparse network on Phi (10 synapses per neuron)



(C) Dense network on Xeon host (500 synapses per neuron)

(D) Dense network on Phi (500 synapses per neuron)

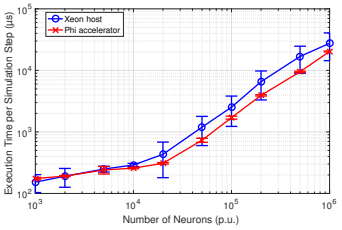
FIGURE 4.14: Effects of vectorization on networks of varying size and complexity.

This behaviour can be explained by reflecting on the trade-offs made when using the AVX instruction set. Compared to the scalar instruction set, the average vector instruction requires an increased amount of clock cycles until completion. The reward of using vector instructions lies in processing multiple data simultaneously. As Figure 4.14 demonstrates, this gain diminishes when there is a relatively small amount of neurons to be simulated per OpenMP thread, as the amount of data processed by each thread is insufficiently large to fill the VPUs. Suitable memory-alignment of aforementioned data also plays a critical role in performance. As a result, scalar instructions may outperform an improperly vectorized codebase, particularly for the Xeon which features better scalar performance than the Phi.

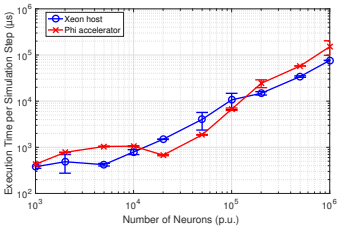
On the other hand, dense networks provide a larger workload and thus, a better opportunity to take advantage of the platform's resources. Cases of not-fully-optimized code outperform the unvectorized codebase, even for the Xeon host. It can be assumed that, in cases where the workload is sufficiently heavy (due to the large amount of calculations required by neuronal synapses), vector instructions are largely "safe" to use. In Figure 4.14d, the increase of neuron populations alters execution time only slightly for properly-vectorized code. This is an indication that, with proper manual vectorization, the accelerator can utilize its assets efficiently and can handle increases in workload well, until the entirety of its computational resource pool is expended. Thus, vectorization

can potentially yield significant boosts in performance, with larger benefits observed for the Phi accelerator due to a larger amount of available resources.

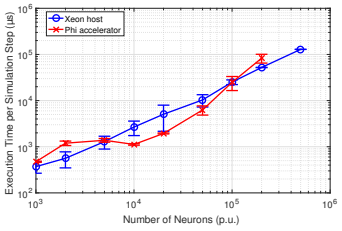
Figure 4.15 evaluates the properly-vectorized code when solving for scaled-up networks, on both platforms. In Figure 4.15a, there is a small and stable performance gap between the Phi device and the Xeon host. The accelerator outperforms the host in a predictable manner. On the contrary, denser networks in Figures 4.15b and 4.15c depict a more complicated behaviour. There is a range of neuron populations where the Phi accelerator outperforms the host. Furthermore, as network size increases, approaching populations of realistic, human inferior olivary nucleus' numbers, the performance gap between the two computing fabrics diminishes. For larger populations, the Xeon host may outperform the Phi.



(A) Networks of 10 Synapses/Neuron



(B) Networks of 100 Synapses/Neuron



(C) Networks of 500 Synapses/Neuron

FIGURE 4.15: Scaling up the best and properly vectorized implementation on the host and the accelerator.

These observations can be justified by network density being closely correlated to how parallelizable the code is. From a programmer's point of view, the synapse count per neuron signifies the degree by which the simulator will differ from an embarrassingly parallel application. The neuron's dendritic compartment forces OpenMP threads to sync due to shared-memory accesses and MPI ranks to exchange data via messages. In addition, it imposes irregularities in memory access patterns due to the fact that the network connectivity matrix is unknown before the simulation begins and thus, data required by each neuron cannot be stored in a sequential, unit-stride manner. This holds especially true for simulations that would require the connectivity matrix to be constantly changing during a run, in order to study the constantly changing connections forming in the human brain. Non-unit stride accesses have an adverse effect on the efficiency of vectorization, as presented in Subsection 4.2.4.5.

The results illustrated in Figure 4.15 can be traced back to the model's shifting behaviour based on network connectivity. Sparse networks can be parallelized and vectorized efficiently; the Phi accelerator will outperform the Xeon host due to an increased amount of available resources, for any non-trivial neuron population. Dense networks, on the other hand, cannot be accelerated as efficiently. The Xeon host is the superior platform for small networks of high connectivity due to its better single-threaded performance, as well as the fact that small networks present less opportunities for the Phi accelerator to utilize its available threads and larger VPUs. As the network size increases, the accelerator can use more of its assets. A point is reached where the Phi outperforms the host by meeting workload demands with aggressive usage of its computational assets. Once the Phi's computational resources are working at maximum capacity, a saturation point is reached; in Figure 4.15c, the performance gap between the two platform gradually narrows for populations beyond $10,000$ neurons, whereas this point is reached at $20,000$ neurons in Figure 4.15b. From then on, the Xeon host's superior single-threaded performance handles the application in a better manner, outperforming the accelerator for human inferior-olive numbers ($\geq 100,000$).

4.3 Porting to Intel Xeon Phi 2nd Generation

Intel's second generation of Xeon Phi processors introduced several architectural differences with respect to its predecessor, designed as a more mature and easier to use manycore processor. The departure from mandatory PCIe connections to host PCs and the re-design as a standalone processor (although there have also been Knights Landing models that follow the co-processor paradigm) which runs the same binaries as other x86-based processors significantly simplifies development for the Knights Landing. Most importantly, it encourages utilization of HPC methods which have been used in other processors and are well-researched.

Furthermore, the manycore processor was a technological step-up from its previous generation, offering more cores per processor, higher bandwidth, superior single-threaded performance and significantly better peak FLOPS performance [178]. As such, at the time of its launch, the Knights Landing manycore processor was the best-suited processor for the growing needs of a biophysically-accurate neuroscientific simulator that focuses on the functionality of massive, complex neuronal networks.

4.3.1 Platform Architecture

Knights Landing (KNL) is a standard Intel Many-Integrated Core (MIC) Architecture standalone series of processors that can boot stock operating systems and connect to a network directly via common interconnects such as Infiniband, Ethernet, etc. They feature an x86-based many-core architecture that specializes in servicing demanding HPC

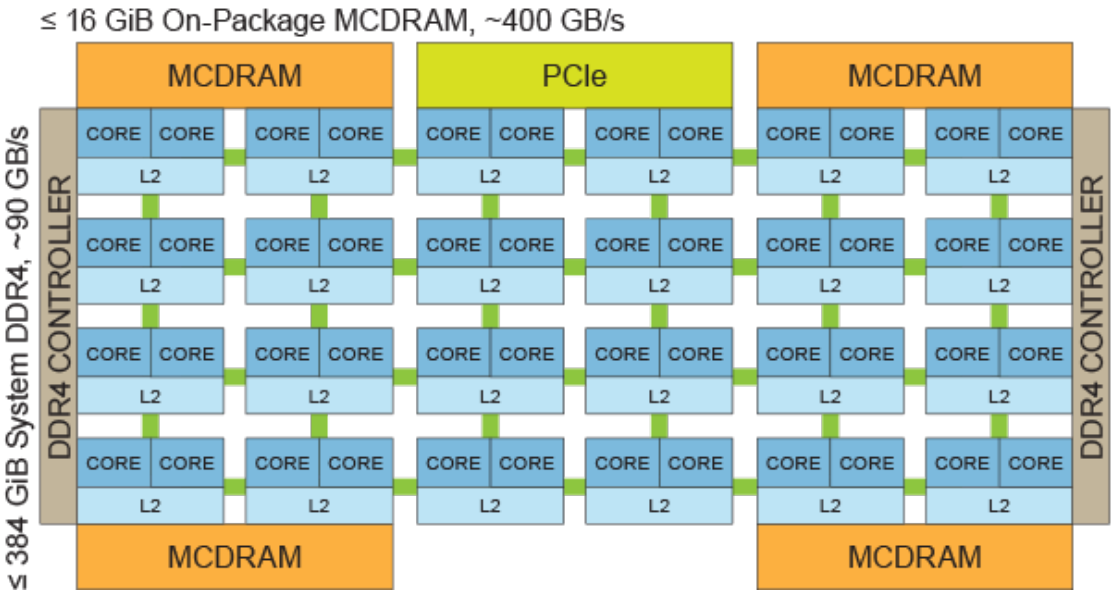


FIGURE 4.16: The Knights Landing die organization [4]. Each tile consists of 2 cores that share an L2 cache. Inter-core communication is orchestrated as a mesh, in contrast to the previous generation (Knights Corner) which employed bidirectional rings [1].

applications. The specific Knights Landing processor model examined in the present thesis features 64 14-nm Airmont cores. Each core utilizes two 512-bit-wide vectorization processing units, as opposed to the single VPU per core present in KNC models, which enable AVX-512 instructions for parallel data processing. Furthermore, best practices indicate that each Knights Landing core can support the execution of up to four software threads in parallel [179].

In total, the KNL processor examined offers up to 64 Airmont cores, each capable of hyperthreading for a total of 256 threads, and 144 VPUs. Communication between its 32 tiles is achieved through an on-die 2D mesh interconnect (also indicated in Figure 4.16) which replaces the bidirectional ring bus used on the KNC co-processor.

The cores of the KNL processor each have access to a private 32KB L1 cache and pairs of cores have a 1MB L2 cache shared between the two cores. Via the L2 caches, the tiles are connected to each other in a mesh fashion. There are options available to the KNL user concerning the mode of operation followed by the processor's cache hierarchy. These options are referred to as "cache clustering modes", are configured at boot time and determine how the memory address space is distributed across the chip. The KNL features four modes: all-to-all, hemisphere/quadrant and sub-NUMA cluster modes of cache operation.

In all-to-all clustering mode, memory addresses are uniformly distributed across all of the tiles' tag directories. In hemisphere clustering mode, the 36 tiles of the KNL are divided into two spacial halves called hemispheres, ensuring that messages can be constrained within the hemisphere. The quadrant clustering mode follows the same mentality as the

hemisphere but partitions the die's tiles in four spacial parts instead of two. Finally, the sub-NUMA cluster (SNC) modes are Non-Uniform Memory Access extensions of the hemisphere and quadrant cache operation modes; they are divided in SNC-2 and SNC-4, respectively. In our research, symmetrical networks act as well-balanced workloads evenly distributed throughout the KNL's cores. As such, we treat the KNL processor as a symmetrically-distributed multiprocessor and opt for quadrant mode of cache operation.

Another feature of the KNL processor aimed at reducing memory-access latency is the 16GB multi-channel dynamic random access memory (MCDRAM). This is an on-package high-bandwidth memory spacially located next to the processing cores that can deliver significantly higher (more than 400GB/s) bandwidth than the chip's 384GB DDR4 RAM (approximately 90 GB/s bandwidth). It also comes with three modes of operation chosen at boot time. When the MCDRAM operates in "flat" mode, it serves as a high-speed extension of the DDR4 memory. Alternatively, it can be configured to serve in "cache" mode, where it is treated as a last-level cache (LLC). Finally, it can be set up in "hybrid" mode where a pre-determined part of the memory is used in flat mode, while the remaining MCDRAM serves as an LLC. We utilize the MCDRAM entirely in cache mode, since some of the larger neuronal networks explored in this paper cannot be allocated on 16GBs of "flat" MCDRAM; additionally, "cache" mode is the most generalizable configuration for any other type of model we choose to port to the KNL and it bears resemblance to shared LLCs present in Xeon processors.

The aforementioned assets and the combination thereof hint on the massive potential parallelism, data access speed and peak computational performance present on the processor. As such, codebases operate best on Knights Landing processors if they feature high degrees of parallelism, vectorization and ideally, well-designed accesses to memory. Our simulator is designed with these aspects in mind, marking the KNL as a suitable platform for our implementation.

4.3.2 Application Mapping

The application has been designed with other x86-based systems in mind, in order to increase portability in other architectures. Due to this fact, only small amounts of alterations were necessary in order to transition from the first (KNC) to the second (KNL) generation of Xeon Phi manycore processors. In addition, the micro-optimizations used to boost simulator performance are also beneficial to other x86 systems.

"KNL-exclusive" hardware assets are configured in a fashion that can be encountered in other processors as well; for example, we use a special low-latency on chip memory called "MCDRAM" as shared last-level-cache, which is a commonly found in Intel Xeon processors. As such, we refrain from limiting our conclusions to the KNL family of processors and ensure effective portability to other platforms.

Like previous iterations of the simulator, in each simulation step, the simulator has the task of updating the status of each neuron in a pre-defined network. The neurons are based on an elaborate, Hodgkin-Huxley model of the human neuron. The model is tri-compartmental: the dendrite, the soma and the axon.

The dendritic compartment holds the important task of communicating with the rest of the network; it features a set of ordinary differential equations (ODEs) that simulate current exchange with other neurons of the inferior olivary network. This exchange happens between dendrites that have formed Gap Junctions (GJ), i.e. the electrotonic connections or synapses among them. Each dendritic compartment forms multiple such electrical synapses, allowing inter-neuron communication and introducing, for denser networks, a major source of computational complexity and multiprocessing synchronization overhead.

The somatic compartment is the main body of the neuron, where most calculations for the neuron's membrane and ionic channels take place. These channels are crucial to evaluating the neuron's state in each simulation step. In sparser networks, the floating-point operations demanded by each somatic compartment dictates the majority of the simulation's computational workload.

Finally, the axonal compartment acts as the output port of the neuron (specifically, in our application, of the Inferior Olivary neuron) to other parts of the brain (such as, the cerebellum). It features less floating-point operations than the soma and its simulation is less complex than the other compartments of the neuron.

In each step, the simulator processes the current flow in the GJs of the network and then, re-calculates the states of the three compartments of each neuron. This is achieved by solving the Ordinary Differential Equations (ODE) governing the model via the Euler forward method [180]. Each neuron may also receive an external stimulus by its environment, in each step of the simulation.

In order to boost simulation speed, OpenMP [59] has been employed to parallelize the application. This implementation has proven effective in KNC's case as well, as exhibited in Section 4.2.3.2. The network is divided in equal parts and assigned to different OpenMP threads, ensuring a balanced distribution of workload.

4.3.3 Experimental Evaluation

4.3.3.1 Experimental Setup

The measurements presented in this section have been carried out using two different generations of Intel Xeon Phi. The Knights Corner co-processor's model is 3120P, featuring 57 cores at 1.1GHz, each supporting up to 4 threads running concurrently via

multithreading technology. Cores run at 300W thermal design power (TDP). The application is designed to run natively on the co-processor, thus excluding any impact from its Intel Xeon host on its measured performance. Specifically, after compiling and transfering via Secure Copy Protocol (scp) all necessary binaries to the co-processor, the host remains idle throughout the experiment. The Knights Landing processor's model is 7210, with 64 cores at 1.3GHz and similar multithreading capacities. Its TDP is noticeably lower at 215W.

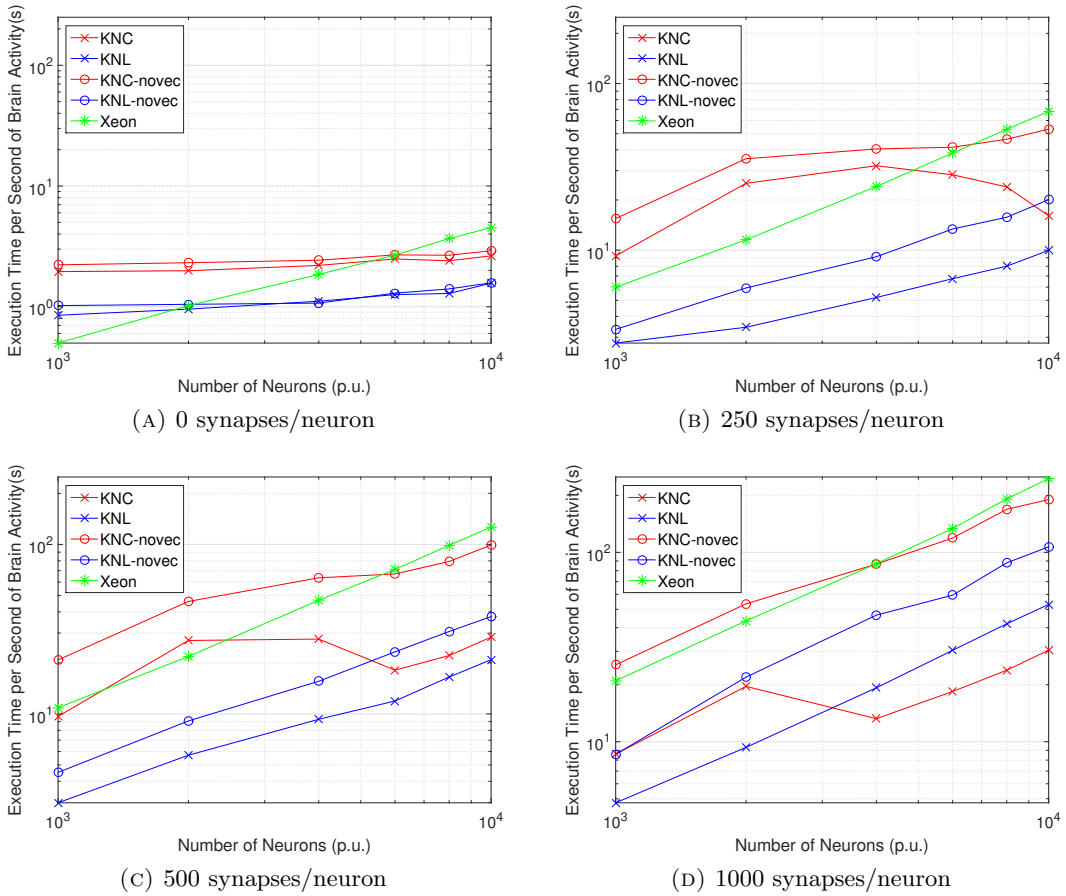
For the power measurements in this section, different methodologies have been followed for the two platforms. For the Knights Landing processor, the processor's power consumption was sampled via Intelligent Platform Management Interface (IPMI) [181] via a script running concurrently with each experiment's execution. Polling frequency was set to approximately 1 Hz. Energy consumption for each experiment was then calculated by integrating the power samples over the simulation's duration. On the other hand, power measurements on the Knights Corner co-processor is achieved by accessing the host's logs of information and errors regarding the co-processor. These logs are attained via a built-in tool named micrasd which can track the KNC's power in intervals of 5 milliseconds. The reports are generated from the beginning of the simulation and by summation of each report until the end of the experiment, an accurate estimation of total energy consumption can be attained.

In each experiment, a network of neurons connecting to each other via the Gap Junction mechanism, as explained in the application description of Subsection 4.2.2 for the Knights Corner processor, is generated. The neuron connections are generated randomly, with each pair of neurons given a chance to form a bond regardless of their position on the neuronal grid. This chance is calculated based on the amount of connections each neuron is designed to have for each experiment, as well as the total neuronal network size; a division of the two variables calculates the network's average connection density, which, in turn, directly leads to the chance of a pair of neurons forming a bond.

4.3.3.2 Evaluation

In Figure 4.17, we can observe obtained simulation speed of each platform for networks of varying connectivity density. The measurements explore varying network sizes, where each neuron has a fixed average amount of connections to the rest of the network.

All experiments in Figure 4.17 have been carried out using approximately the maximum amount of threads available to each platform. For the KNC, we used 220 threads, whereas the KNL offered 256 threads. On average the KNL platform outperforms the KNC platform by 2.4× in terms of execution time. The maximum speed-up is 6×, while in some cases the KNC comes in front with up to 1.6× speed up over the KNL. More specifically, we can observe that, in the cases of low connectivity density, which translates to a low amounts of workload per thread, the KNL shows a superior performance to



(A) 0 synapses/neuron

(B) 250 synapses/neuron

(C) 500 synapses/neuron

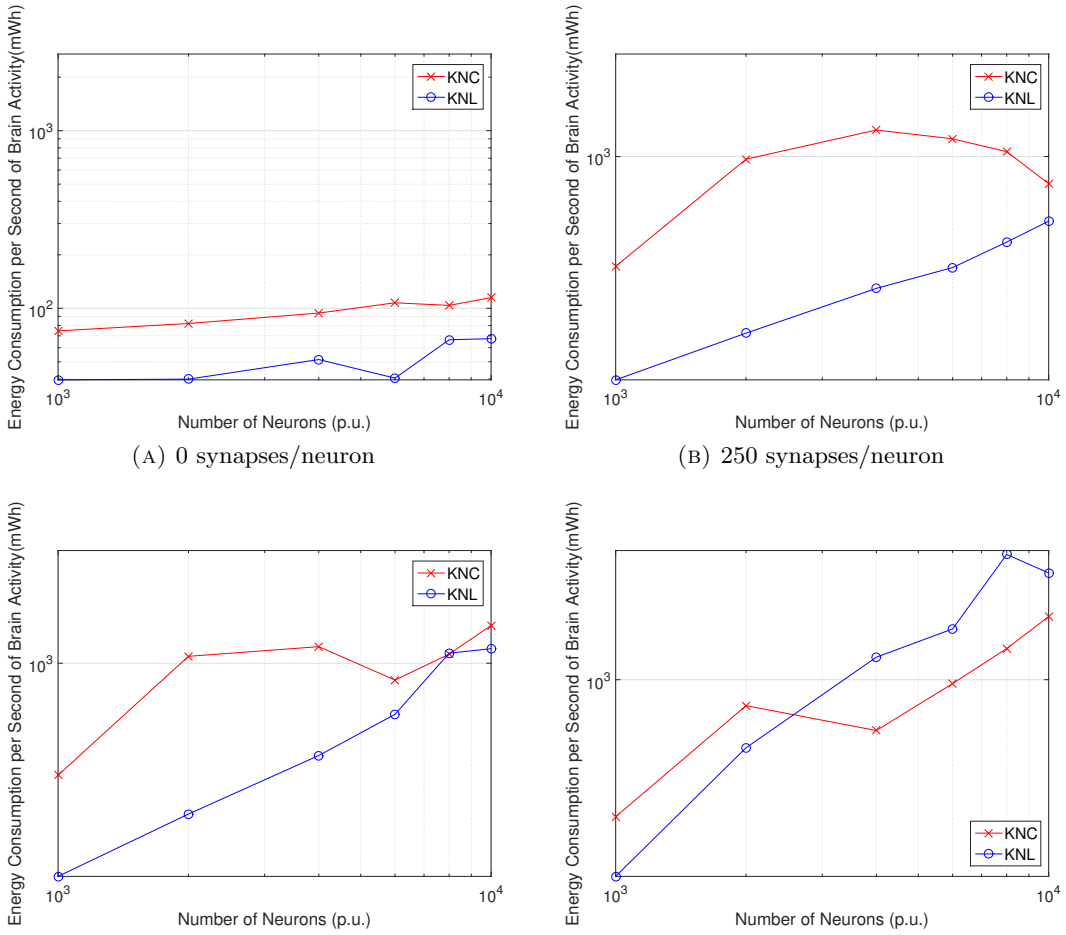
(D) 1000 synapses/neuron

FIGURE 4.17: Execution Time per second of simulated brain activity, comparison between KNC and KNL on different Simulator configurations

the KNC. In cases of small workloads, the efficiency in usage of parallelization assets is diminished, thus single-threaded performance becomes much more important for overall simulation speed. The KNL demonstrates a considerably stronger single-threaded processing power and overtakes the KNC by a fair margin.

On the other hand, as the computational workload assigned to each thread increases for denser networks, the KNC performs significantly better. The performance gap between the two platforms lessens as the KNC can use its assets with increasing efficiency, since the application has been optimized with the KNC architecture in mind. For workloads of more than 4,000 neurons, each forming approximately 1,000 synapses, the KNL is outperformed by the KNC.

It should be noted, however, that in terms of performance predictability, the KNL is heavily favoured. Its performance is linear and very predictable. On the contrary, the KNC's performance is harder to anticipate. The platform's capability to take advantage of its computational resources increases with the supplied workload. Because of this behaviour, it forms a plateau, during which simulation time for larger networks remains stable or even lessens. Beyond a certain point in network sizes, which differs based



(A) 0 synapses/neuron

(B) 250 synapses/neuron

(C) 500 synapses/neuron

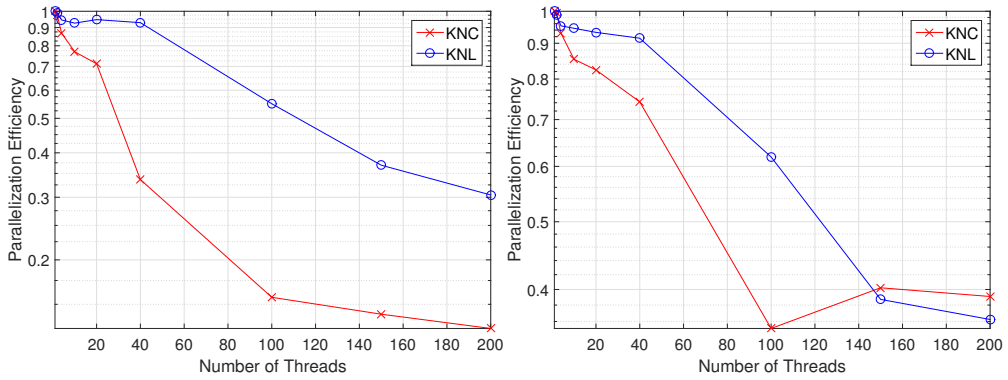
(D) 1000 synapses/neuron

FIGURE 4.18: Energy Consumption per second of simulated brain activity, comparison between KNC and KNL on different Simulator configurations

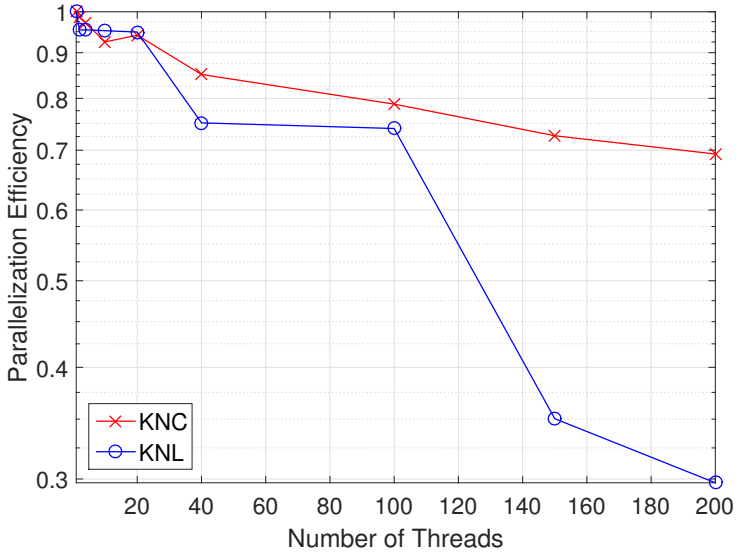
on how dense the network is, "this plateau" ceases to exist and its performance curve resumes its linear nature. The existence of such "plateaus" impacts the performance predictability of the KNC, whereas the KNL does not exhibit similar behaviour.

In Figure 4.18, we present information regarding the energy required by each computing fabric in order to simulate a second of brain activity, measured in mWh. The Figure is directly linked to Figure 4.17, since energy consumption is dependent on execution time needed for simulation of each second of brain activity. As such, we can observe similar patterns between the two Figures. On average we have to note that the KNL consumes 48% less energy than the KNC. Because of the KNL's lower TDP and better performance for light workloads, there is a significant reduction in energy consumption when computing for small networks. To put this claim into perspective, whereas the simulation of one second of brain activity in a network of 4000 neurons, with a density of 250 synapses per neuron, requires over $1200mWh$ for the KNC, the KNL consumes under $300mWh$ for the same workload, improving on energy efficiency by a factor of $4\times$.

On the contrary, due to the KNC's smaller execution times for larger, denser networks, it is preferable from a power consumption standpoint to the KNL for such workloads. A network of 10,000 neurons, each forming 1,000 synapses with the rest of the network, requires 27% less energy on the KNC ($1600mWh$ per second of simulated time) than on the KNL ($2200mWh$ for the same amount of activity).



(A) 1000 neurons (B) 4000 neurons



(C) 10000 neurons

FIGURE 4.19: Threading Efficiency on the KNC and the KNL, for different Simulator configurations

In Figure 4.19, information regarding the efficiency with which each platform manages its OpenMP threads is displayed. In HPC, the efficiency with which an application utilizes the underlying platform's resources can be calculated as the speedup yielded by employing said resources, compared to a single-threaded performance, divided by the amount of resources used, such as the number of processors used to run the application, or the number of threads spawned by the application. In our case, we calculated the efficiency metric by dividing execution speedup with the number of OpenMP threads spawned, with a range of OpenMP threads utilized from 1 to 200, on both platforms.

In each subfigure, network density has been set to 1,000 synapses per neuron and we explore networks of different size.

For the KNL, we can observe that the efficiency of utilizing up to approximately 50 threads remains at satisfactory levels. In these cases, each core spawns either one or two threads (due to the selected balanced thread affinity) and, in contrast to the KNC, the KNL's cores operate significantly better when operating with only one thread [4]. The KNL maintains a reliable efficiency for low degrees of threading regardless of the simulated network's size, whereas the KNC's efficiency suffers for small workloads, such as for networks of 4000 neurons.

Larger networks, however, offer better opportunities for the KNC to utilize its computational assets efficiently, maintaining a speedup-to-threads ratio above 70% even for 200 threads. The KNL's threading efficiency sharply declines when employing massive degrees of parallelism, dropping below 40% when using more than 150 threads. The application's inability to utilize the entirety of KNL's assets efficiently to tackle demanding simulations explains the performance gap between the two platforms for larger workloads. This inability is mostly attributed to the fact that the simulator has been fine-tuned to the KNC environment and has been tested "out-of-the-box" on the KNL.

4.4 Multinode Implementation

A significant step in the process of scaling a demanding neuromodelling application is the efficient transition to multinode implementations. In this venture, an advantage of using Intel Xeon Phi Knights Landing 2nd generation processors as a high-performance computing fabric over its predecessor, the KNC, as well as other accelerators, is the ease of employing a multinode implementation. The Xeon Phi line of products supports traditional parallel coding paradigms, such as MPI and OpenMP for task-level parallelism. These tools have been well-studied and are constantly improved upon, significantly reducing the difficulty and time-to-market of a scalable, highly-parallel implementation of the simulator's algorithm.

A multinode implementation of the simulator allows taking advantage of a more computational resources, as well as data storage. This is especially important in the class of models studied in this thesis - the conductance-based, biophysically accurate models, as briefly presented in Section 2.1.3. These models place particular emphasis on the accurate restructuring of the mechanics in inter-neuron communication. In large and dense networks, which are usually the object of study in significant experiments that attempt to recreate real phenomenons occurring in the human brain [31], the existence of billions of inter-neuron connection points is common occurrence. Whether the modeled mechanism for such connections is an electrochemical synapse, or a more complicated

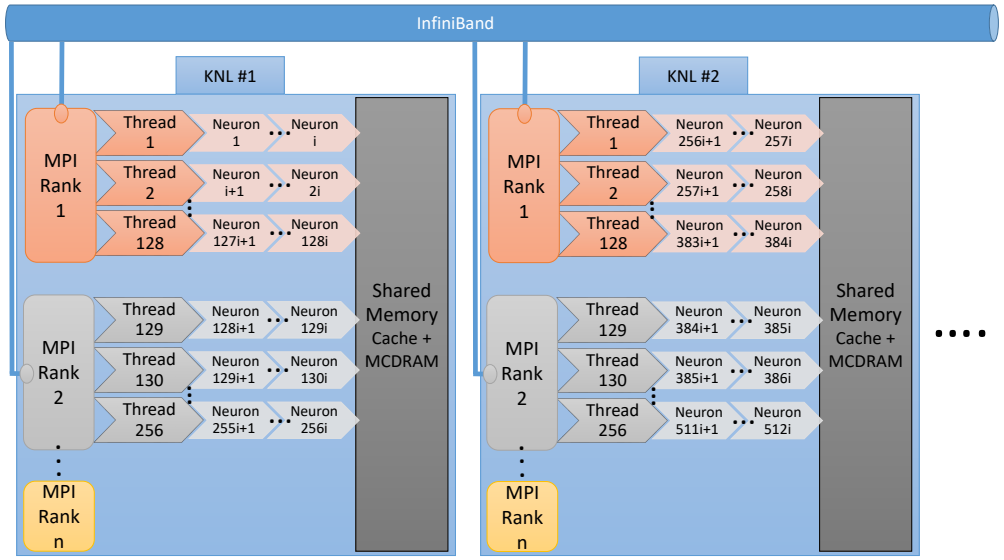


FIGURE 4.20: Schematic of the simulator multinode implementation on the Knights Landing. In this example, each KNL processor operates at maximum capacity, meaning all of its $64 \times 4 = 256$ threads are employed, while a variable n amount of MPI Ranks are spawned per platform. It should be noted that in our work, we opted for spawning $n = 4$ MPI Ranks per KNL platform. A number of i neurons is assigned to each thread in this simulation, totalling a simulated network of $l = i \times 512$ neurons over two KNLs. The implementation schema can be extended to include as many KNL machines as necessary and available.

electrotronic gap junction, the massive amounts of data used to model these connections can feasibly be handled by a computing system of equally massive computational assets.

In order to achieve such levels of computing power, a supercomputing, distributed system is the most obvious choice. In this class of systems, a strong case can be made for supercomputing clusters based on manycore, x86-processors, since they utilize traditional parallel processing tools. These tools are particularly effective for describing and implementing efficiently the unpredictable behaviour of massive neuronal networks and respective communication patterns. On the contrary, GPU-based systems follow a coding paradigm that is better suited for algorithms with less frequent context switches between tasks and less conditional jumps in the code, thus leaning towards more predictable network behaviour. After the aforementioned conclusions, I believe that a KNL-based multinode system can be a suitable candidate for implementing the simulator and scaling it further in order to suit significant neuronal network experiments.

4.4.1 Programming Model

In order to utilize multiple Xeon Phi manycore processors, message-passing (MPI) libraries are used for inter-node communication in multi-KNL systems. In each processor, OpenMP threads parallelize the computation of each part of the network assigned to

the processor. Multiple MPI ranks spawn per node; each MPI process spawns a number of OpenMP threads, such that the total number of OpenMP threads spawned across all MPI ranks of each node equal to the maximum amount of threads capable of running concurrently on the KNL (256, as discussed in Section 4.3.1). Figure 4.20 describes the design of a hybrid implementation on such a system.

On an algorithmic level, OpenMP threads operate on different parts of the neuronal network. Each neuron in the network is assigned to a single thread in order to be processed. Each thread handles an equal number of neurons, in order for the computational workload to remain balanced. Each neuron in the network is connected to others (except for special cases of zero connectivity) via the modeled Gap Junctions. This mechanism necessitates the usage of MPI collective communication in order to exchange data between processors that do not share memory. The amount of communication traffic between MPI ranks, whether on the same or on different KNL processors, depends on the amount of neurons in the network and the network's density, which indicates the average number of GJs each neuron has established.

During the simulation of any given neuronal network, each MPI rank is responsible for the message-passing needs of its assigned subnetwork, which is processed in parallel by 64 threads. This procedure can be divided in two sub-processes: sending and receiving MPI messages. In each simulation step, Gap Junctions need the dendritic membrane voltage levels of the participating neurons in the connection in order to be computed. The MPI rank satisfies the other ranks' needs by packing the necessary values in a buffer after OpenMP thread calculations. The buffer is then distributed by using MPI's broadcast function (**MPI_Bcast**). The upper limit for this data-exchange instance happens when each MPI rank needs to broadcast voltage values for each neuron they handle.

The specific amount of threads spawned by each MPI rank (64), in conjunction with the amount of MPI ranks spawned on each node (4), was determined after a brief evaluation of the optimal configuration on a KNL. Figure 4.21 depicts the exploration of these parameters. The simplest implementation would spawn one rank in each KNL die and attribute all of the 256 available OpenMP threads to the single MPI rank. However, a brief design space exploration reveals that this is not the optimal configuration point for the KNL. While multiple configurations where the number of MPI ranks per die are kept low are viable, spawning 4 MPI ranks per die is suitable for all types of networks tested in this thesis. In particular, a single MPI rank exhibits slightly worse performance than spawning 4 MPI ranks per die when handling larger workloads, which are naturally more demanding; as such, 4 MPI ranks per die becomes the selected configuration point for the simulator.

After the MPI rank completes its **MPI_Bcast** function, it receives the other MPI ranks' broadcasts. The contents of each received buffer are processed by spawning 64 OpenMP threads which operate on the buffer in parallel. In the worst-case scenario of 100% connectivity density, each of the 64 threads needs access to the full content of the

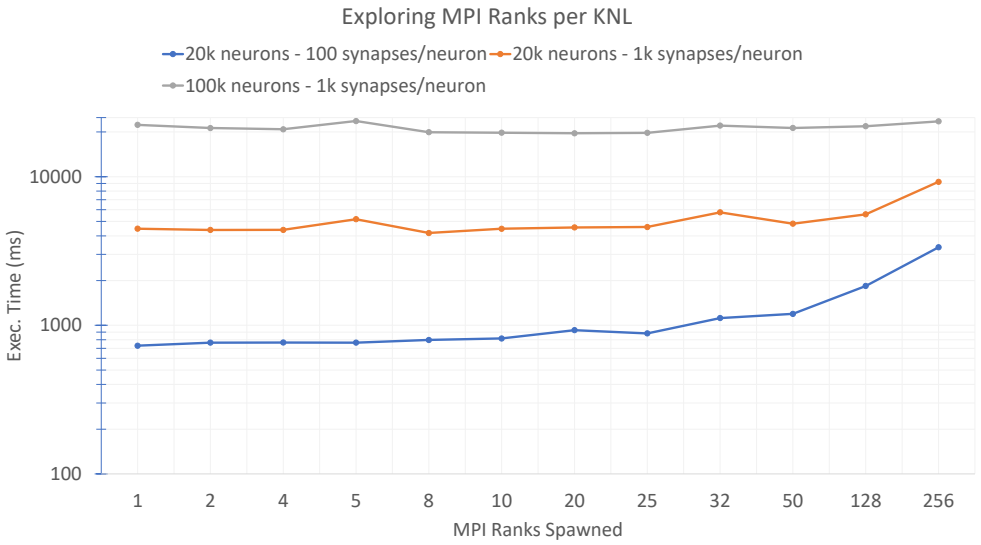


FIGURE 4.21: Exploration of KNL's performance under different configurations of hybrid MPI-OpenMP clustering granularity. Three different networks of varying degrees in neuron population size and density are examined for 100 milliseconds of simulated brain time. We alter the amount of MPI ranks spawned on a single KNL processor. Configurations employing a small amount of MPI ranks exhibit superior performance. In particular, using 4 MPI ranks spawning 64 OpenMP threads offers good, reliable performance for all neuronal networks.

received buffers; in this case, each rank gets updated on the entirety of the rest of the network in every simulation step. Following the processing of the received data buffers, the calculation of Gap Junctions, as well as the neuron compartmental states, can be carried out by the threads. Upon completion of these calculations, the OpenMP threads are joined, thus ensuring that the network state update is complete and ready to be processed in the next simulation step, which begins with a new **MPI_Bcast** function from the MPI rank.

4.4.2 Scaling Considerations

Multinode manycore systems are complicated. Analysis and pattern-detection for a heavy data-exchanging application, such as a Hodgkin-Huxley-model-based neuron simulator, is a challenge on such a system. We will first discuss impact factors that heavily influence the simulator's performance under different workloads and configurations. We will, then, discuss our experimental results with these factors in mind.

There is a price all manycore systems pay for utilizing their resources in parallel. Spawning and joining software threads and/or tasks, via the usage of libraries such as the

OpenMP, requires an amount of preparation and core-time that constitutes a non-negligible overhead. In addition, unless examining an embarassingly parallel application, parallelization resources of the manycore platform require synchronization at certain "checkpoints" in the algorithm. Simulations of biological neuronal networks entail exchange of bio-signals, which invariably result in some way of thread communication when using a manycore system with a shared memory hierarchy. Increasing the detail and complexity of the model scales the amount of such bio-signals the application simulates; as such, the biophysically realistic model studied in this paper is highly demanding in synchronization when employing a complicated, dense neuronal network.

In addition, contemporary manycore processors feature a wealth of parallelization resources for threading and vectorizing code. The KNL, for instance, by utilizing AVX-512 instructions by all of its available threads, can potentially execute more than $10,000$ floating-point operations in parallel. This parallelization potential requires a suitable workload in order to be properly utilized. Since the simulator's unit of operation is the single neuron, a network's population size is bound by a lower limit; simulations under this size limit cannot be expected to utilize all of the manycore's assets, especially when investigating multinode systems.

As a result, under-utilization of the platform's resources can severely hinder the platform's performance during a biophysically-complex simulation. The manycore processor's parallelization assets go under-used, while still causing overheads of spawning/joining tasks. Even if the simulation is large enough to feature high degrees of asset utilization, stiff models, such as the one examined in this paper, enforce data synchronization between threads in every simulation step, further reducing the efficiency with which the processor's hardware is employed. In conclusion, in order to attain acceptable efficiency when using manycore processors such as the KNL, each of its threads need to be assigned with the computation of a suitably large workload.

As far as data exchange between nodes is concerned, MPI-like communication between the nodes in a multinode system is materialized through Infiniband. This type of communication poses a significantly heavier overhead than intra-node synchronization processes do. As such, locality of data exchange between neurons in the simulator is particularly important. Real neurons in the brain exchange current (data) by being physically approximate to each other; this translates well for locality in the hardware. By partitioning the network in clusters of neurons which are physically close to one another, most messages between those neurons stay intra-node, avoiding using MPI functions to other cores or processors.

As a result, simulations that do not allow for an efficient partitioning of the network in local sub-clusters will exhibit significantly less scaling potential. When examining different distributions for the network's connectivity map, it becomes evident that the overhead of inter-node communication is a limiting factor for utilizing multiple processors

TABLE 4.1: Parameter Space

Variable Name	Value Range
Network Size	1,000 - 2,000,000 nrns
Network Density	0 - 1,000 syn/nrn
Synaptic Pattern	Uniform and Gaussian
KNL Nodes Used	1 - 8 nodes

Range of explored parameters in this paper. The Table details network configurations considered, as well as the amount of hardware used during simulation.

if connections are spread out throughout the network. These types of networks can be hard to partition in an effective manner.

4.4.3 Experimental Evaluation

In order to evaluate the performance of the proposed simulator, a number of simulated runs in neuronal networks were conducted, ranging widely in connectivity patterns, density and size. The goal of this evaluation is to highlight the factors that significantly impact the scaling capacity of the multinode implementation. To this end, a wide range of network configuration points were evaluated. It will be shown that, for multinode manycore configurations, an aspect of the network that has been mostly unmentioned thus far in this thesis, as well as in the existing literature, is of significant impact on performance: the distribution of connections throughout the network. This particular parameter will shape how the experiments are structured. The setup of the multinode implementation evaluation follows in the next Subsection.

4.4.3.1 Experimental Setup

We organize neurons in a 3-dimensional grid. For exploring the impact of network topology, we explore two different (and naturally occuring) distributions: a uniform distribution of synapses in the network; and a Gaussian distribution of synapses where neurons in proximate positions on the 3D grid are significantly more likely to form a bond. The differences of these distributions are visualized (in a 2D grid, for ease of reference) in Figure 4.22.

These distributions represent different patterns of connectivity in the biological brain; neurons may exhibit local synaptic connectivity, as in the case of neocortical pyramidal neurons [182], while long-range synaptic patterns can also play an important role in neuron functionality [183]. Moreover, by exploring different connectivity patterns, it will be evident that synapse distribution affects performance in a definitive manner.

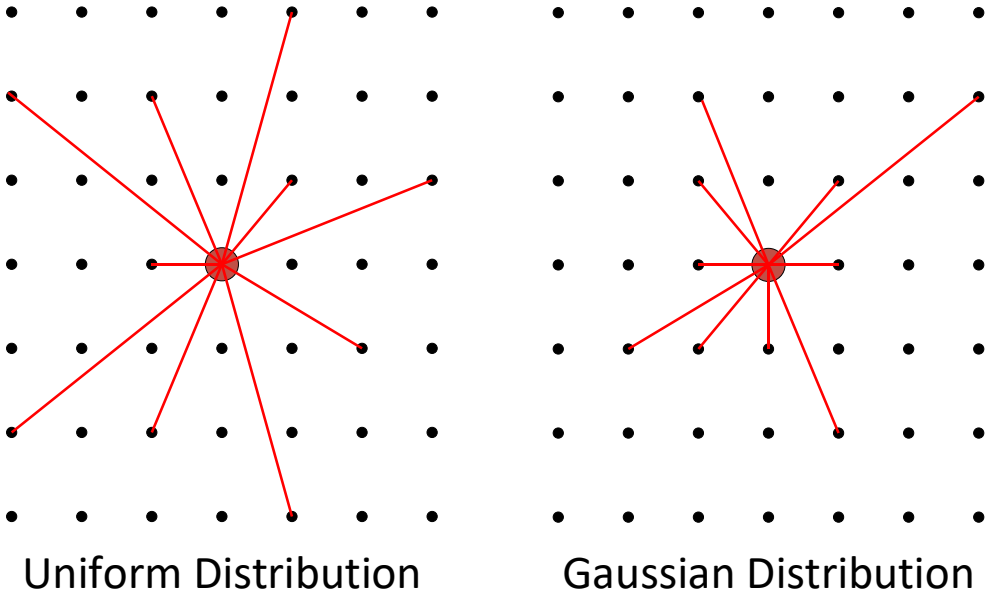


FIGURE 4.22: Depiction of two different 7x7 2D neuron-meshes. In each case, the neuron in the center of the mesh forms 10 connections; the leftmost mesh follows a uniform distribution, whereas the rightmost features a Gaussian distribution. Uniform distribution creates spread-out connections, whereas the Gaussian distribution keeps the connections closer to their point of origin (i.e. neuron in the center of the mesh).

Networks are tested on varying degrees of size, as summarized in Table 4.1. The smallest networks evaluated are formed of 1,000 neuron populations, whereas the largest are comprised of 2 million neurons. For exploring the impact of connectivity density, various (fixed) amounts of synapses per neuron have been used; configurations of no-connectivity, 10, 100 and 1,000 synapses per neuron are tested. These particular configuration points match (and surpass) connectivity as encountered in biological inferior olivary nucleus and aim at revealing the simulator performance trends under increasing network density.

The measurements utilize the standard **gettimeofday()** C-function in order to evaluate execution time for the simulation of the network after it has been set up. In these measurements, input and output have been restricted to a minimum in order to measure pure simulation execution time. The experiments simulate $100ms$ of brain time. Since this is a time-driven simulator with a steady, incompressible time-step of $50\mu s$, brain activity during simulated brain time is not relevant to the simulator's performance, in contrast to event-driven simulators, whose performance is affected by neuronal spike generation frequency.

In addition to differing network sizes, connectivity policies as well as densities, we perform scalability experiments by employing multiple KNL nodes (1, 2, 4 and 8), with hardware assets as described in Section 4.3.1 and configured as in Section 4.4.1. A detailed discussion on the scaling behaviour of the multi-KNL implementation is thus, also included in this evaluation.

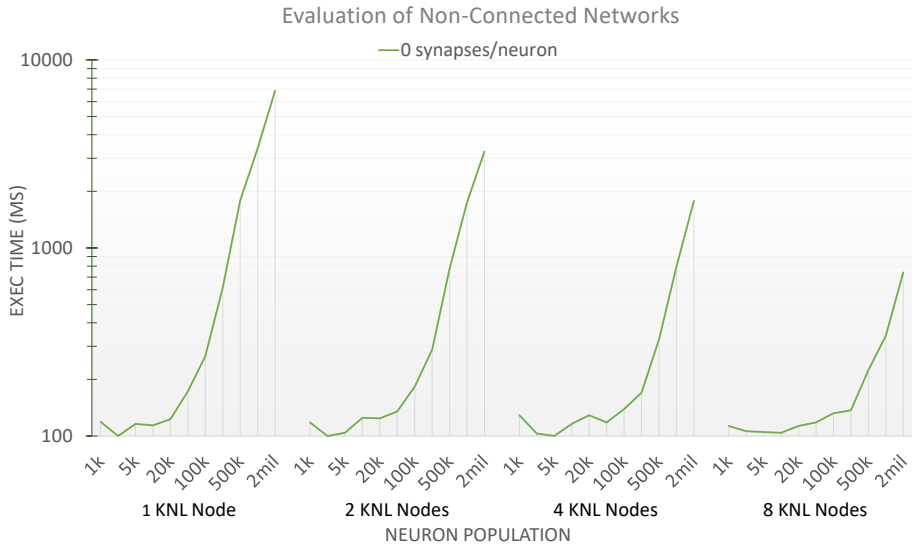


FIGURE 4.23: Special use case of the simulator operating on non-connected networks. The neurons oscillate in a solitary environment. Due to the absence of communication between the cores' assigned subnetworks, this use case can be considered as one of the best cases for parallel processing from a scaling perspective. Utilizing increasing amounts of hardware scales simulation speed in an efficient manner; network simulation for 2 million isolated neurons requires execution time that is within the same order of magnitude as real time.

4.4.3.2 Experimental Evaluation

We will present the results of the experiments carried out for this paper and assess the simulator's performance. We will analyze the behaviours exhibited in each case by referring to the factors impacting the manycore processors' performance, based on the discussion relayed in Section 4.4.2.

Figure 4.23 depicts the special case of networks without the forming of GJ connections. In these cases, neurons operate in isolation to each other in the network. The absence of GJs relaxes communication needs as it translates to a lack of need for synchronization between OpenMP threads and communication between MPI ranks. Furthermore, the special conditions for these types of simulations permits the KNL to utilize its low-latency memory assets without overheads from the MESIF cache coherency protocol. Finally, there is also a considerable reduction in computational needs since the processor skips the calculation of the GJs in each simulation step, which would otherwise take up a major portion of CPU time.

These factors combined lead to overall low execution times which differ from real time by less than two orders of magnitude even for populations of 2 million neurons. Each performance curve in Figure 4.23 exhibits similar trends. The initial part of the curve, which corresponds to low-population networks, is flat, since these simulations are "low-effort" and under-utilize the hardware's assets. This trend extends to higher-population

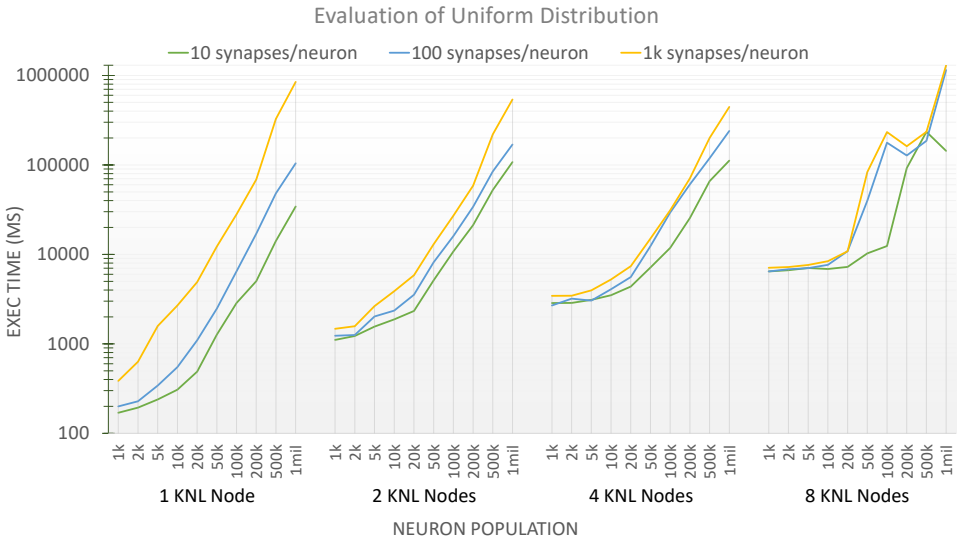


FIGURE 4.24: Study of simulated networks following a uniform distribution of Gap Junction bonds. In this scenario, neurons show no preference over which neuron they form a bond with, resulting in GJ bonds being uniformly distributed across the entirety of the network. In this fashion, the application's performance and scalability is hindered due to data messages being exchanged between cores, especially between those belonging to different KNL machines. As such, only a small degree of speedup is attained by employing two KNL nodes instead of one. Furthermore, no further gains are observed when scaling to more hardware, particularly for heavier workloads.

networks as more KNL processors are added to the simulator. On the other hand, when simulating larger neuronal networks, there is a linear increase in simulation speed as the number of KNL processors used grows. These observations are consistent with how an application with minimal communicational needs should behave.

The uniform distribution of connections in the network, depicted in Figure 4.24, is the worst-case scenario for the simulator. In this method of distributing each neuron's connections over the network, every neuron pair has a uniformly equal chance of being created. When examining a network of n neurons, each forming g connections, a neuron pair, regardless of its location in the network, has a probability $p = g/n$ of being formed. Furthermore, if the network is simulated by c cores, then each core is tasked with simulating n/c neurons and $g \times (n/c)$ GJs. Due to the uniform distribution of these Gap Junctions, the core stores data locally concerning only n/c neurons, thus lacking data necessary for the computation of $(g-1) \times (n/c)$ GJs. This scenario causes the simulation to be very "heavy" on utilizing MPI collective functions for data exchange. Memory accesses degrade the simulator's performance further, since L1 and L2 caches are unlikely to hold necessary data, forcing processor's cores to search in non-local caches.

This information explains the unsatisfactory performance exhibited by the simulator in Figure 4.24. The application scales poorly, particularly when utilizing 8 processor nodes. Due to the system's lack of scalability, measurements of only up to 1 million neurons are depicted. Larger network populations cannot be simulated effectively, regardless of

the amount of hardware utilized. The performance curves of 8 KNL nodes in Figure 4.24 demonstrates that for larger networks, execution times show a sharp increase and 8 KNLs perform worse than a single-node system, rendering the option of adding further hardware to the system ineffective.

The application's performance curves are significantly erratic and hard to interpret in this distribution case. A critical factor that determines simulation speed is the overhead of MPI collectives imposed during inter-node communication, as mentioned before in Section 4.4.2; this factor grows more dominant as the amount of machines employed during a simulation run increases. For any experiment consisting of a network of l GJs run on k different KNL machines, each processor needs to simulate the functionality of l/k GJ. The processor holds data capable of completing the calculation of a GJ without inter-node communication for l/k^2 GJs. Thus, the ratio of "expensive" inter-node communication versus "cheaper" intra-node data exchange directly correlates to the amount k of processors used in the case of uniform distributions of neuron connections.

In addition, Figure 4.24 shows a qualitative difference between the performance curves of dense networks with 1,000 GJs per neuron versus sparser networks. Dense networks, which exhibit a naturally heavier workload than sparser networks, depict a better tendency to benefit from using 2 KNL nodes over opting for single-node implementation. There is a small, but noticeable speedup for million-neuron dense networks, which is absent for similar in size, but sparser in connectivity populations.

This behaviour can be attributed to the fact that in our simulator, data exchange between MPI ranks takes place with collective communication functions. MPI ranks exchange bundles with relevant dendritic voltage data concerning their respective subnetworks. In each simulation step, the MPI rank "builds" the bundle with data from neurons in its assigned subnetwork. A neuron in said subnetwork will be added to the bundle as long as there is a *single* GJ calculated by another MPI rank which needs this datum. Thus, in the case of uniform distribution, the probability of a neuron being added to the bundle grows quickly with the average amount of GJs formed by each neuron and "caps off" to 100% even for sparsely connected networks. When this probability reaches 100%, each MPI rank exchanges all of its subnetwork's dendritic data in each simulation step. In these cases, the maximum amount of data exchange between MPI ranks is achieved and, as explained, these cases are present even for networks of sparser density.

In conclusion, both sparse and dense networks must circulate large amounts of GJ-related data through the KNL's communication channels, both intra- and inter-node. However, denser networks have significantly more operations to perform in order to calculate GJ states, after acquiring all of the necessary data. These calculations happen in parallel, thus benefiting from employing more hardware and ultimately favour 2-KNL implementations over single-node. This benefit is "hidden" when employing more than 2 nodes due to "heavier" penalties to performance from communication-related

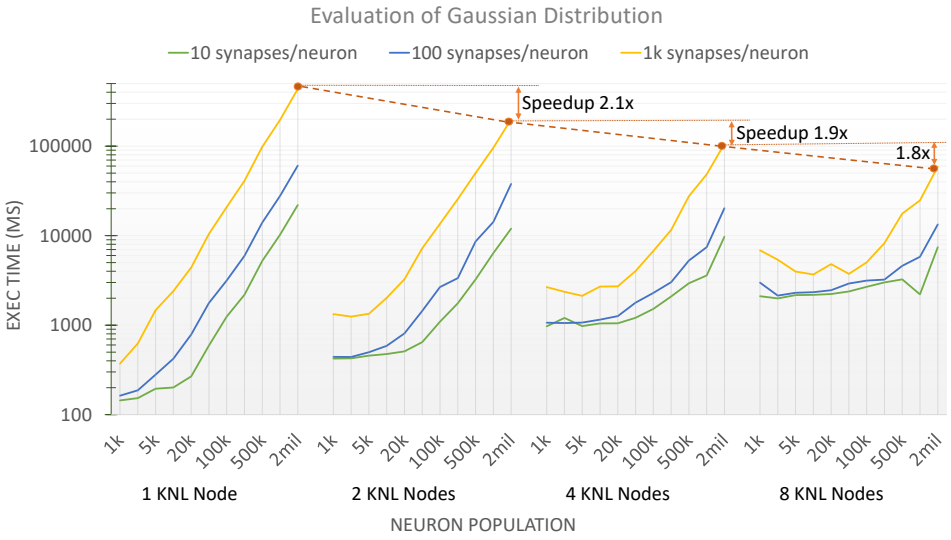


FIGURE 4.25: Evaluation of the simulator's performance when computing networks of varying size and density. The network's connectivity map follows a Gaussian distribution. Neurons are imagined in a 3D space and form Gap Junction bonds between them. The likelihood of a neuronal pair forming is based on their proximity in the 3D space. The simulator's performance is evaluated when utilizing 1, 2, 4 and 8 KNL machines. Scalability is boosted by a significant factor due to the greater data locality. For large enough workloads, simulation speed increases in an almost linear fashion with the amount of hardware employed. Smaller speedups are attained for sparser, smaller networks.

overheads. It should be noted, however, that the performance curves of both sparse and dense networks follow the same trends when moving to 4-KNL simulations and that sparser networks show worse degradations in performance than the heavier experiments.

The most realistic case of network connectivity, based on how real neurons in the inferior olivary region band together to form Gap Junction connections, is evaluated in Figure 4.25, where neuronal proximity plays an important role in synapse forming according to a Gaussian distribution. A quick observation of the logarithmic Y-axis in the figure reveals that this scenario displays a decrease in overall execution times by nearly an order of magnitude when compared to the worst-case scenario of uniform distribution in Figure 4.24.

In this use case, a satisfactory amount of locality in message exchange is achieved by clustering neurons according to their coordinates in the 3D-mesh. Neurons within a small range of Cartesian distance are assigned to the same core. According to the Gaussian distribution, this allows the core to calculate most of its GJs without referring to external data, since most (but not all) of its neuron connections link to other neurons handled locally by the same core. Hence, we limit the amount of messages exchanged between cores intra- and inter-node, as well as reduce memory access latency by maximizing local cache usage.

Due to the favorable distribution, utilizing a multinode implementation yields positive results. There is a considerable speedup by adding more Knights Landing processors to the larger simulations. High efficiency is maintained for workloads that approach the 100k neuron-population mark in the case of dense network with 1k synapses per neuron. On the other hand, smaller networks do not exhibit favorable results when moving from single-node to multinode implementations. More specifically, Figure 4.25 shows that there is a clear slowdown when employing 8 KNL nodes for relatively small networks of 5k neurons or less, when compared to the single-node's performance curve. Furthermore, an 8-KNL implementation for small and dense networks shows an improvement in execution speed when increasing the neuronal network size from 1k to 10k neurons.

These findings can be attributed to the factors mentioned in Section 4.4.2. When using a group of 8 manycore processors and spawning a large number of threads per processor, each capable of executing vectorization instructions, underutilization of the hardware assets causes considerable overheads that deteriorate performance based on how underutilized the processors are. This causes the simulator to execute *larger* neuronal networks *faster*, up to the point where the hardware's assets are utilized efficiently. The point at which the system's resources are *saturated* depend on the amount of processors used, as well as the network's density. Denser networks show a clearer, more impactful saturation point, as shown by comparing the performance curves of 100 versus 1k synapses per neuron. Furthermore, saturation is reached earlier when employing less manycore nodes due to less available resources to the system. When examining the performance curves of the densest network configurations in Figure 4.25 (as noted with a golden yellow line), 2 KNL nodes retain stable execution times until the 5k neurons mark, whereas 8 KNL nodes show a true increase in execution times only past the 50k neurons mark.

Another point of interest is a super-linear speedup when moving from a single-node system to a 2-KNL configuration for 2 million neurons and 2 billion synapses. This behaviour can be attributed to an increase of available low-latency assets. When using additional nodes of computational fabric, in addition to enhancing the system's potential parallel processing power, its total cache space (as well as the KNL's MCDRAM in our particular setup) is also expanded. By allowing a larger, if not whole, part of the network to be allocated in low-latency memory space, super-linear speedup can be observed in manycore multinode systems.

The multinode implementation allows the simulation of up to 2 million Hodgkin-Huxley-based neurons and 2 billion Gap Junctions for $100ms$ within two minutes. As such, even in the case of the heaviest workload tested in this paper, the simulator exhibits a simulation speed that differs from real time by two to three orders of magnitude. In addition, networks of 5k neurons and 500k Gap Junctions, which represent sizable experiments in neuroscientific research, can be simulated in a single node at a rate that approaches 30-50% of a real brain's operational speed. Thus, the simulator can

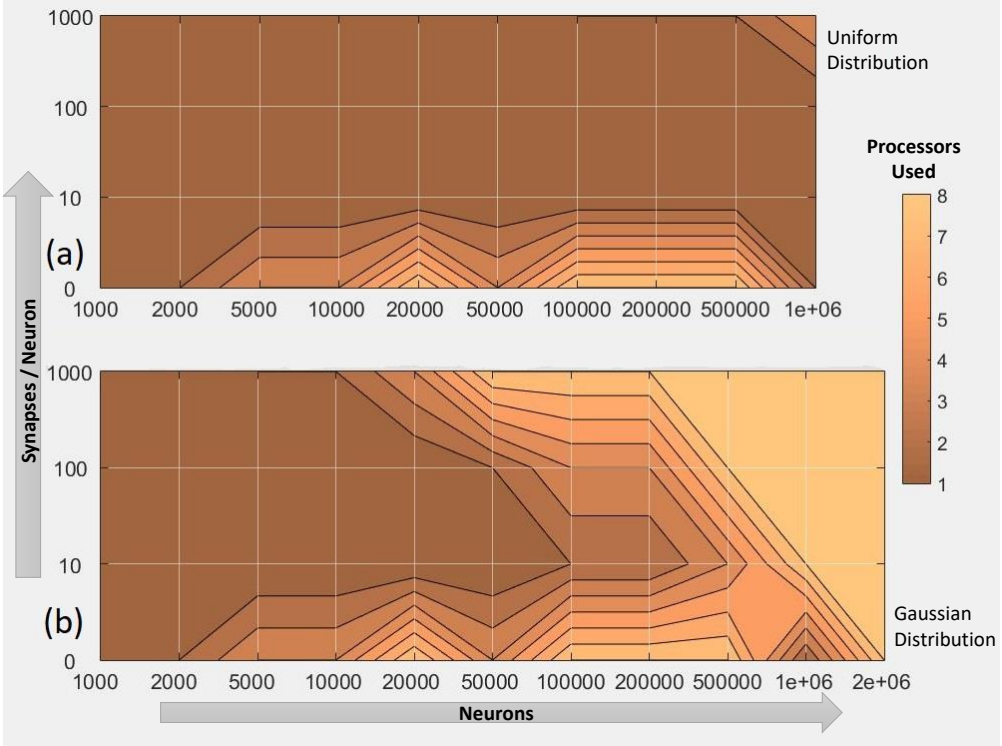


FIGURE 4.26: Footprint of suggested implementation for simulated networks of varying sizes and connectivities. The colourmaps depict the amount of processors providing the best simulation speed for networks of uniform (panel **a**) and Gaussian (panel **b**) synaptic patterns. Single-node implementations dominate networks with uniform synaptic distributions due to poor scalability. In contrast, the case of Gaussian synaptic distributions varies with network density: highly dense networks require the maximal number of KNL nodes, whereas lower densities can be tackled with less nodes.

calculate workloads both light and heavy at satisfactory speeds; the single-node approach is recommended for smaller workloads, while multinode implementations are preferred for demanding networks.

4.4.4 Resource Allocation

One of the focal points in this paper is the concept of matching hardware utilization to the workload that requires calculating. The suggested amount of hardware to deploy for each network simulation varies according to network size and its corresponding connectivity map. By using data collected from the experiments presented in Section 4.4.3, with parameter ranges described in Table 4.1, Figure 4.26 depicts a general guideline for allocating the minimal KNL instances necessary for achieving the best possible performance for workload instance. The Figure exhibits a number of interesting patterns.

Figure 4.26a depicts suggestions for networks with **uniformly distributed** connection patterns. We observe that uniformly-distributed connectivity maps force the simulator to become completely communication-bound, due to model complexity. These types of

network benefit mildly from 2-node implementations, while employing more hardware often yields no improvement. Figure 4.26 shows that only networks with populations larger than 500,000 neurons and connectivity patterns denser than 100 synapses per neuron (thus, totalling more than 50 million synapses in the network) have an optimal configuration point of 2 to 4 processor nodes. In other cases, single-node implementations are recommended due to poor scalability.

Figure 4.26b maps networks with connection patterns following a **Gaussian distribution**. We come to the following general conclusions. The notion that neighbouring neurons are more likely to form bonds leads to significantly more scalable network configurations. Connectivity maps based on the Gaussian distribution expose data locality better and support utilizing multiple KNL nodes. Figure 4.26 shows multiple network configuration points where the maximal tested amount of processors is optimal for simulation speed. In this paper, up to 8 KNL processor nodes have been employed due to availability (as noted in Table 4.1); a larger amount of processors may yield further boosts to simulation speed.

Furthermore, network density directly correlates to the prevalence of cases where multinode allocations are recommended. For example, in Figure 4.26b, networks featuring 10 synapses per neuron are suited for single-node solutions when population count is less than 100,000 neurons. On the other hand, when network density approaches 1,000 synapes per neuron, network sizes of more than 10,000 neurons merit multinode configurations. This behaviour can be attributed to the fact that network density affects the amount of floating-point instructions issued per simulation step; by increasing network density, the computational workload becomes heavier and thus, can be calculated more effectively by employing larger amounts of computational resources.

Both panels in Figure 4.26 show that when network sizes are large while network synaptic count is low, the neuromodeling problem becomes an embarassingly parallel use case and utilizing a high amount of processors is recommended. Small, dense networks benefit from single-node allocations, otherwise computational resources are effectively wasted and simulator performance suffers.

Networks featuring low synaptic connectivity maps behave in a similar fashion, since there is negligible communication overhead for the simulator. In both panels of Figure 4.26, multi-node configurations are encouraged when simulating less than 10 synapses per neuron in the network. This claim is challenged, to a degree, when simulating very high population counts (more than 500,000 neurons), since even a small amount of synapses per neuron can impose a non-trivial communication overhead.

4.4.5 Workload Parameters

It is clear that fully understanding the performance patterns exhibited by a biologically-accurate multinode simulator is not a trivial task. The simulator presented in this paper works on x86-based processors, which are very well-documented and have been extensively studied. Furthermore, in this paper, the parameter space we explored relates to network size, density and connectivity distribution, as described in Table 4.1. In this strictly-defined parameter space, the simulator behaviour, as depicted in Figures 4.23, 4.24, 4.25 and 4.26, clearly shows that even small changes to its parameters can have a large impact on performance. Furthermore, this phenomenon is exacerbated by increasing the amount of available computational resources.

Since predicting simulator behaviour in any given parameter space is hard, one is encouraged to create maps similar to the one featured in Figure 4.26, in order to discern emerging trends. Such maps aid in choosing simulator configuration for future research in related areas. This map generation process can be efficiently deployed in a Cloud setting. Cloud services lend themselves to performing parameter-space explorations by offering processing resources that can be otherwise difficult to access [184, 185]. In addition, the resources can be scaled to match problem size in a cost-efficient manner.

Furthermore, when mapping simulator behaviour, one is encouraged to increase the scope of parameter exploration as much as resource availability allows. In this manner, the generated map is more effective at conveying hints related to simulator behaviour trends. As an example, panel **a** of Figure 4.26 partially resembles the image that panel **b** depicts for networks of 1,000 to 20,000 neurons. It is possible that by further increasing the network size, trends that are already visible for Gaussian-distributed connectivity maps become manifest in uniformly-distributed maps as well. This could be attributed to the fact that uniformly-distributed networks face larger inter-node communication penalties; as such, they would require computing heavier workloads before additional computational resources prove to be beneficial.

A fundamental problem with extending parameter size is that heavier workloads demand larger execution times to be calculated. This, in turn, implies longer simulation times for evaluating optimal simulator configurations (here: number of nodes). Given that, for this type of cycle-accurate models, simulator behaviour remains largely stable after a small amount of warm-up steps is performed. Thus, it can be beneficial to reduce the amount of simulation steps and increase the range of parameters explored.

4.5 Summary

This Chapter describes the modernization of the simulator work on state-of-the-art manycore processors. We have ported the biophysically-accurate simulator of the inferior

olivary nucleus, previously exhibited on the experimental single-chip cloud computer, on a single-node Xeon CPU and Xeon Phi system. The selected simulator serves as a significant benchmark for parallelization and scaling of biologically-plausible neuron modeling workloads. We have evaluated three native implementations on the target system: an MPI-based, an OpenMP-based and a combination of both.

MPI is consistently the worst choice for the Phi accelerator. Its poor performance was expected due to the implementation's inability to utilize the platform's valuable multithreading capabilities. OpenMP exhibits the best performance for any problem size. The hybrid implementation is an improvement over MPI and for larger networks ($\geq 10^4$ simulated neurons), its performance approximates OpenMP's results. Since this porting method is designed as easily scalable to multi-node systems, this is an interesting finding when aiming at large network simulations.

On the Xeon host, small differences across implementations were observed. OpenMP remains a more suitable choice for small networks. However, its performance can vary wildly depending on network size and, when simulating more than 10^5 neurons, an MPI implementation is preferred. The hybrid implementation offers little benefit and a strictly MPI or OpenMP porting option is advisable here. Before manual vectorization, the Xeon host offered better performance than the Phi co-processor and successfully scaled up to networks of a million inferior olivary nuclei with normal distribution of inter-neuron connections.

Since the shared-memory implementation was proven to be the most effective option, it was subsequently further optimized, with an emphasis on vectorization performance. A combination of pragma directives, function inlining and specific memory allocation functions, specialized for cache line alignment, was used. These techniques are applicable to any codebase and form the basis of vectorizing any application. Following these alterations, modifications that were specifically aimed at the simulator's algorithm were employed.

A sizeable increase in attainable simulation speed was achieved for workload sizes that were eligible for vectorization. Overall, these techniques were beneficial for both the accelerator and the host. In particular, for networks that are large and densely-connected enough to saturate the Phi's assets, the difference in performance between manually vectorized code and un-optimized code that relies solely on the compiler is an order of magnitude. After applying these techniques, however, the platforms perform differently depending on network connectivity density. Sparse networks are a good candidate for acceleration via the Phi co-processor's large pool of computation resources. Furthermore, dense networks feature a range of populations between $5,000$ and $50,000$ neurons where the co-processor can use its computational resources to outperform the host. On the other hand, the host's focus on single-threaded and scalar performance is a better fit for dense networks outside this range due to their less well-parallelizable nature.

Thus, we can endorse the usage of these optimizations on codebases that resemble the algorithm, connectivity patterns and problem sizes encountered in the InfOli modeling application; it is also worth noting that significant development time was necessary to draw out the computational effectiveness of the Xeon Phi.

The application was then ported to the second generation of Xeon Phi, the KNL, a more commercially mature product. The simulator's performance was tested using a range of workloads, from small, unconnected neuronal populations to larger, dense networks. The results were evaluated from both a simulation-speed and a power-efficiency standpoint. On average KNL offered a speed up of 2.4× while consuming 48% less energy. Smaller workloads, by taking advantage of the KNL's superior single-threaded performance, exhibited very significant gains in both speed and, even more so, energy consumption, with specific experiments demanding 75% less Wh of energy per second of simulated brain activity on the KNL. On the other hand, OpenMP-thread efficiency suffered when running on the KNL, causing the simulator to handle more demanding networks poorly, relatively to the extensively optimized KNC version. Furthermore, throughout the whole range of experiments, it has been shown that the KNL offers a more robust, dependable performance curve with little variability.

Concerning the multinode implementation of the simulator on the KNL, a system setup of 8 processors was chosen to evaluate performance. The work highlighted that efficient usage of a small cluster of manycore processors, such as the system used, was able to achieve satisfactory performance even when facing a very demanding mathematical model of the human neuron, in network and synaptic sizes numbering in the millions and billions, respectively. It constitutes an efficient solution for studying demanding neuronal models in a pursuit of attaining deeper understanding of the human brain's intricate details.

Furthermore, it has been demonstrated that a biologically-accurate simulator exhibits performance patterns that are dictated by problem size and the nature of each network's connectivity map. A point of focus particularly in this Doctoral thesis was the system's multinode scalability; the system is highly sensitive to simulation parameters and as such, careful steps need to be taken in order to discern trends in performance behaviour.

Chapter 5

Heterogeneous Neurocomputing

5.1 The Case for Heterogeneity Today

The world of neuroscience has nurtured a vast variety of different computational workloads to tackle. As seen in Section 2.1, there is a significant amount of different approaches to modelling neurons and their respective networking behaviour. Furthermore, neuronal experimentation operates on different scaling levels. There are potential users of neuronal simulation software who are primarily interested in real-time simulation of smaller networks. On the other hand, different projects also concern themselves with large-scale simulation of brain-wide phenomena, as well as the elaborate challenge of connecting neuronal networks modelling different parts of the human brain.

Even though modern HPC platforms can often deal with such challenges, the vast diversity of the modeling field does not permit for a homogeneous acceleration platform to effectively address the complete array of modeling requirements. As this thesis has established thus far, different workloads require different approaches. As such, a case can be made for heterogeneous platforms being a more suitable solution to the problem of designing a widely-used, multi-purpose neuronal simulator, particularly for the demanding class of complex, biophysically meaningful models.

The point of heterogeneity in computational platforms can also be argued for challenges of different domains. Python-based mathematical packages exist for heterogeneous systems supporting linear algebra [186], offering good performance for a system based on an interpreted language. In graph theory, hybrid methods utilizing multicore CPUs and GPUs according to the nature of the explored graph have been proposed [187]. Tree-structured index search in databases is a challenge that has been attempted to be resolved in a combination of CPU and GPU systems [188]. More recently, the fundamental problem of rigid body simulation in physics engines for games and animation, has been approached via a solution utilizing both the assets of the CPU and the GPU of desktop computers [189]. In the field of pore-scale modeling, flow dynamics can be

simulated and evaluated using distributed, hybrid computational systems of CPU and GPU [190]. Hybrid systems involving FPGAs, as well as multicore CPUs and GPUs, have been utilized in the domain of online security [191], cryptography [192] and deep learning [193].

With these points in mind, it stands to reason to claim that heterogeneous computing is a well-established approach that can help tackle the wide variety of workloads present in the world of computational neuroscience. As such, it is important to further develop further the simulator presented in Chapters 3 and 4 in order to be a more inclusive, heterogeneous tool capable of handling experimentation needs in the field of neuroscience.

5.1.1 Challenges

The adoption of heterogeneous computational platforms introduces difficulties that have not been met in a definitive manner yet. As an ensemble of radically different hardware, each computing fabric requires a different approach and coding paradigm. This fact leads to obstacles in designing an application that can run on the entirety of the heterogeneous system and utilize its assets efficiently. In neuroscience, another factor that impedes the design of an all-purpose neuronal simulator is the wide range of still-developing modeling efforts, which impose different constraints on performance based on the nature of the workload.

Depending on the desired model characteristics, we identify two general types of simulations that are relevant in neuroscientific experiments. The first one has to do with highly accurate (biophysically accurate and even accurate to the molecular level) models of smaller-sized networks that requires *real-time* or close to real-time performance. These kinds of experiments can be used with artificial real-time set-ups or brain-machine interfaces (BMI) and are closely related to brain-rescue studies (TYPE-I experiments). The second type involves the simulation of large- or very large-scale networks in which accuracy can often be relaxed. These experiments attempt to simulate network sizes and connection densities closely resembling their biological counterparts (TYPE-II experiments) [124] [194]. This, in combination to the variety of models commonly used, makes for a class of applications that vary greatly in terms of workload, while also, depending on the case, requiring high throughput, low latency or both. A single type of HPC fabric, either software- or hardware-based cannot cover all possible use cases with optimal efficiency.

A better approach is to provide scientists with an acceleration platform that has the ability to adjust to the aforementioned variety of workload characteristics. This heterogeneous system, integrating multiple HPC technologies, instead of just one, would be able to provide such flexibility. In addition, a framework for a heterogeneous system using a popular user interface for all integrated technologies can also provide the ability to select a different accelerator, depending on availability, cost and performance desired.

Such a hardware back-end must overcome additional challenges to be used in the field. It requires a front-end which should provide two crucial features: an easy and commonly used interface through which neuroscientists can employ the platform, without the constant mediation of an engineer and a front-end that can reuse the vast amount of models already available to the community.

Developing and executing experiments with neuronal network models is a very rigorous process since experimenting with the models presupposes their careful fitting to experimental data. The neuroscientist should be able to interface with the acceleration platform directly, which is not a standard practice today and incurs significant delays in the research process. Lastly, the ability to program the accelerator platform in commonly used coding languages and the portability of legacy code, is essential for wide adoption of the HPC technologies by the community.

5.1.2 HPC Projects for Heterogeneity

A number of funded HPC projects attempt to provide frameworks for easing the adoption of heterogeneous systems, in order to combat the challenges mentioned in the previous Section. Vineyard [195] has the main goal of increasing the performance and energy efficiency of data centres by utilizing the advantages of heterogeneous ensembles of hardware accelerators. An important contribution to the cause is the development of a high-level programming framework and big-data infrastructure for allowing easy utilization of heterogeneous computing systems [196, 197]. Overall, the project focuses on the integration of FPGAs, which lack in ease of adoption, in data centers and computing clusters and promotes software for managing and coding on heterogeneous infrastructures, including FPGAs.

TANGO [198] uses an approach that simplifies the creation and operation of next-gen software by hiding the complexity between the distributed/parallel architecture level and the level of application/software. Furthermore, they expose performance, energy and other requirements of software applications to be incorporated into the overall development and deployment process, enabling programmers to code and resource managers execute being energy-aware. Overall, TANGO makes a compelling case concerning utilization of a heterogeneous system from an efficient power-consumption perspective and presents a framework for achieving energy efficiency at application construction, deployment, and operation.

OPERA [199] also directs its efforts towards hard constraints on low power consumption and proposes methods of modifying software to take advantage of a heterogeneous architecture in order to achieve efficient computation energy-wise. The proposed approach is to decompose existing software and run the decomposed tasks in parallel fashion. Each task component is analyzed for inter-dependencies with other tasks; resource requirements (processor speed, memory utilization) can then, be determined and

the component can be routed to the appropriate hardware of a heterogeneous system. This process aims at maximizing utilization of the system's hardware. The framework includes resource monitoring in order to perform real-time adjustments and reallocations in the heterogeneous system [200].

A newly developed project focusing on heterogeneity and exascale computing, the project of EXA2PRO [201] is developing a framework for the efficient deployment of applications in heterogeneous (pre-)exascale computing systems. It focuses in the productive application development and in enabling performance portability to address the diversity and the increasing heterogeneity of supercomputing centers. The project focuses on productivity when deploying highly parallel applications in exascale computing systems. They utilize a skeleton programming framework for hybrid execution on multicore CPUs and accelerators [202]. Additionally, the project also aims to provide various fault-tolerance mechanisms, both user-exposed and at runtime system level.

These projects, along with other unmentioned in this Section, have garnered a focus on heterogeneous computing. A trend towards efficient usage of accelerators and their integration in large computing systems and data centres can be observed. The frameworks described here are efforts to help meet the increase in demand of efficient development in such systems.

5.1.3 Cloud Solutions

Having access to a system with sufficient computational resources, especially in the case of a heterogeneous system, is not always easily achievable for a developer. Moreso in the field of neuroscience, where traditional *in-vivo* and *in-vitro* experimentation, rather than *in-silico*, is still the norm, research groups appreciate access to reliable, high-capacity computational systems which cannot be easily found in traditional neuroscientific labs. An option, that has been gaining traction over the last decade, for acquiring development time on such resources are cloud infrastructures.

According to NIST [203], "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction". While private networks can host a computational cloud for private usage, public providers of cloud services allow their clients the usage of their infrastructure, usually through a web interface that is intuitive and accessible to users specialized in field unrelated to computer science. These services can provide invaluable access to processing power, as well as other benefits.

Cloud infrastructures are becoming an increasingly effective solution for many users to access and develop on a large system, since launching applications on a cloud platform is becoming swifter and easier as technology advances, with automated tools [204] and

better-designed user interfaces lessening time-to-market. Furthermore, modern cloud services provides tools for management of application scaling, allowing increase or decrease of running computing resources "on the fly". This allows for swift capacity scaling, matching the size of computational workload at any given moment of execution. Furthermore, cloud services provide control over booting and restarting running instances of the application, thus providing invaluable tools for crash recovery and automation. These benefits are supported by containerization technology [205], which provides portable, light-on-overhead virtual environments equipped with a suite of tools tailored to each application's needs.

Cloud services are also a good option for gaining access to a system with diversity in available resources. Modern cloud providers offer a variety of multiple instance types, operating systems, and software packages. The variety in the underlying hardware allows a selection of memory, CPU, instance storage and boot partition size that is optimal for each application. Furthermore, cloud infrastructures have been able to provide access to accelerators, such as GPUs (for example, Google GPU Cloud) and FPGAs (for example, Amazon EC2 F1 instances). These factors combined make cloud services a prime candidate for gaining access to a heterogeneous system.

As a result, cloud infrastructures provide an effective solution for having access to and developing on large-scale systems. Cloud services, like the Amazon AWS [206], Oracle Cloud and Microsoft Azure, have made their appearance in modern literature of computational neuroscience. An EPFL-associated research group has undertaken the task of automating the process of fitting model parameters to experimental data via the processing power of cloud systems via NEURON packages [207]. Research effort has undergone into utilizing cloud computing to enable easier data sharing between neuroscientific groups globally [208]. Furthermore, significant effort has been spent on increasing security for Internet of Things (IoT) frameworks in neuroscientific research [209]. With various research groups working on building secure and accessible frameworks aimed at leveraging the processing power of the available cloud services for the neuroscientific field, it is high time for an attempt at providing the domain with a complete service for neuromodelling development and simulation.

5.2 BrainFrame: Bringing HPC to Neuroscience

In this thesis, BrainFrame is introduced, a heterogeneous acceleration platform that incorporates separate distinct acceleration technologies: Intel Xeon-Phi CPU, NVidia GP-GPU and Maxeler Dataflow Engine. The PyNN software framework (refer to Section 2.2.3) is also integrated into the platform. The framework is the fruit of collaborative work between the author of this dissertation, my colleagues in the National Technical University of Athens [210] and in the Erasmus Medical Centre, Rotterdam [211].

The BrainFrame framework is designed to transparently configure and select the appropriate back-end accelerator technology for use per simulation run. PyNN integration provides a familiar bridge to the vast number of models already available; the neuronal models will run as-is, independently of the underlying hardware fabric, without constant mediation by an engineer. Furthermore, the framework will be able to reuse a large amount of models already available and commonly used by the neuroscientific community. Finally, BrainFrame gives a clear roadmap for extending the platform support beyond the proof of concept, with improved usability and directly useful features to the computational-neuroscience community, paving the way for wider adoption.

As a challenging proof of concept, an analysis of the performance of BrainFrame on different experiment instances is presented, utilizing the state-of-the-art neuron model introduced in Chapters 3 and 4, a biophysically-meaningful, extended Hodgkin-Huxley representation. The model instances take into account not only the neuronal-network dimensions but also different network-connectivity densities, which can drastically affect the workload's performance characteristics. The performance analysis will aim at displaying that the model directly affects performance and all accelerator technologies are required to cope with all the simulation use cases.

5.2.1 System Overview

BrainFrame framework aims to provide neuroscientists with a service that will be able to adjust the target hardware fabric to be optimal to the characteristics of the simulation (neuron model, size of the network, network density, etc.). The scientist will have access to an easy graphical interface that he/she is comfortable (front-end) with, which will be connected to a heterogeneous system that integrates multiple HPC technologies (back-end). Handling the data provided by the scientist via the front-end and feeding it to the appropriate fabric in the back-end is achieved by utilizing PyNN (middle-ware); the generic Python-based modelling language allows for reconstructing the experimental simulation requested by the scientist. The proposed framework will be able to pick up the best accelerator to model under simulation such that it satisfies the cost and performance required.

This heterogeneous system is built with scalability in mind. By utilizing resources from a public cloud provider, such as Amazon AWS, the service can accomodate for multiple requests in parallel. Furthermore, as shown in Section 4.4.2, simulations can be further enhanced in efficiency and speed by utilizing more resources in each individual simulation run; multinode simulation runs are also feasible through cloud computing.

At this Section, a rundown of the system's layers will be presented, starting with the integrated platforms that constitute the heterogeneous back-end which accomodate the simulation runs. The middleware which utilizes PyNN, as well as the front-end interface for interacting with the scientist will then be described.

TABLE 5.1: Specifications of the accelerator fabrics used.

Specification	**Intel Xeon Phi CPU (5110P)**	**Maxeler DFE (Maia)**	**NVidia GPU (Titan X)**
On-Board DRAM	8 Gb	48 Gb	12 Gb
RAM bandwidth	320 Gb/s	76.8 Gb/s	336.5 Gb/s
Memory streams/channels	16	15	–
On-chip memory	30 Mb (L2 cache)	6 Mb (FPGA BRAMs)	3 Mb (L2 cache)
Number of chip cores	61	–	3072 CUDA Cores
Chip frequency	1.053 GHz	Depends on design kernel	1 GHz
Instructions set	64 bit	n/a	32 bit
Power consumption (TDP)	225 W	140 W	250 W
IC process	22nm	65nm	28nm

5.2.1.1 Integrated Platforms

Our heterogeneous platform incorporates three accelerator fabrics; a Maxeler *Maia* Data-Flow Engine (DFE) board [212], an Intel Xeon Phi 5110P CPU [1] and a Maxwell-based Titan X GPU by NVidia [213] (Table 5.1). All there boards are PCIe-based which is how they communicate with the host system. The three very different accelerators provide broad enough features to cover a variety of characteristics of neuronal network instances. Furthermore, the use of PCIe interfaces ensures that composition of BrainFrame-enabled machines can been easily tailored on a per-case basis depending on the availability of funds and hardware resources of a research laboratory. Different types and mixes of PCIe-based accelerators can be selected.

The architecture of the Xeon Phi implementation has been described in detail already in Section 4.2.2. Due to a single-node implementation that will be covered here, the design utilizes the OpenMP implementation that was relayed in Section 4.2.2.2 and 4.2.4, since it was proven to be most efficient option amongst other implementations in Section 4.2.3.2.

The Maia DFE is a Maxeler HPC technology based on reconfigurable hardware. Its tool flow is designed and optimized to accommodate the acceleration of dataflow applications; that is, applications with the bulk of their implementation using purely raw computations with the absence (partially or totally) of branching execution or feedback paths. The Maxeler tools can exploit the nature of dataflow applications to implement uniquely massive pipelines, maximizing the throughput and overall performance. The DFE boards also incorporate a high-bandwidth, multichannel, highly parallel, customizable interface to the onboard DRAM memory resources (up to 96 GBs) making it ideal for scientific applications. The DFE board used in our experimental setup is a 4th-generation Maia-DFE board implemented using an Altera Stratix V 5SGSD8 chip.

GP-GPUs have also been prominent in the HPC domain and in scientific computing in particular. The Titan X includes 3,072 CUDA micro-cores, which are used to parallelize computation execution, and 12 GB of on-board RAM. GPU implementations also benefit from the generally good adoption of the NVidia CUDA-library open environment that allows porting of applications with similar ease to the Phi OpenMP and OpenCL

frameworks. GPUs also come at a relatively lower cost than the other two accelerator types. However, as opposed to the the Xeon Phi, a GPU cannot act as its own host increasing communication delays between host and accelerator during execution.

It should be noted that the backends depicted here should be treated as a proof of concept. This setup has been used in order to provide a proof of concept for the value of the framework, particularly in that heterogeneity has a case for biophysically-complex simulation workloads, such as the one that has been developed in the previous Chapters (3 and 4) of this dissertation and will be used to evaluate the framework. The system is designed to utilize resources from a public cloud provider and is currently hosted on the Amazon AWS cloud service, with expanded computational resources available. Furthermore, the online system hosts a wide variety of models; only the InfOli model will be used in this thesis to evaluate the system.

Lastly, it must be noted that BrainFrame is to be used in scientific research that is very dynamic and fast-paced. The goal is not to over-optimize the different accelerator implementations, but to propose and maintain a balance between the programming effort and optimization needed, resulting in shorter development times for cutting-edge research tools. In real research, such development times should be kept short so as not to delay the scientific process.

5.2.1.2 Middleware

As mentioned in Section 2.2.3, PyNN [8] is a Python package that facilitates the interchangeability and the study of different simulation environments within the computational neuroscience community . It allows for simulator-independent specification of neuronal-network models and already supports many of the popular simulators mentioned in Section 2.2.2, like NEURON, NEST, PCSIM , Brian, and so on.

The PyNN API supports modeling at multiple levels of abstraction, both at the neuron level and the network level. It provides a library of standard neuron, synapse and synaptic-plasticity models and a set of commonly-used connectivity algorithms while also supporting custom user-defined connectivity in a simulator-independent fashion.

We integrated the three accelerator fabrics, described in Section 5.2.1.1, as back-ends on the BrainFrame system using PyNN as a front-end. The PyNN integration provides the neuroscientific community with easy access on the accelerators without constant mediation from the acceleration engineer while also providing an interface for the already established models to be used with the new heterogeneous acceleration back-end. These characteristics of PyNN can have decisive impact on the adoption of BrainFrame by the community.

As a proof of concept for the front-end of the BrainFrame platform, we have added the InfOli model the library of standard PyNN models. Following the PyNN paradigm,

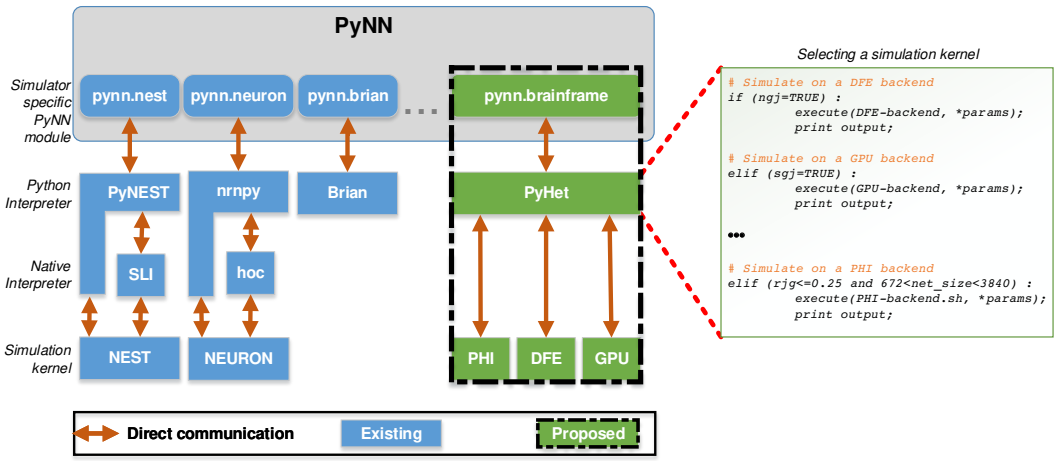


FIGURE 5.1: PyNN architecture and the proposed BrainFrame framework [5].

the user initially selects the simulator – in our case our BrainFrame simulator – and then proceeds to select the neuron model, in our case the Inferior-Olive model. A population of neurons using the chosen model is then generated, determining the inter-neuron connectivity type and, finally, a projection of the specified neuronal network is created.

The main difference between the proposed PyNN-backend substrate and the typical simulator back-ends within the PyNN environment is an additional selection step. In this step, a decision about which of the three alternative acceleration fabrics will be used for a specific experiment is made, based on the available hardware and the characteristics of the simulated neural network.

A conceptional view of the architecture of the PyNN BrainFrame module is shown in Figure 5.1. For the simulator kernels to communicate with the PyNN frontend, a intermediate BrainFrame-specific PyNN module (pynn.brainframe) is required that implements and extends common methods and objects like the neuron models, synapse models and projection methods and objects. In the case of the proposed BrainFrame module, we implemented objects and methods: i) for the initialization of the simulator, ii) for the description of the neuronal network in PyNN, and iii) for controlling the simulation execution. In some cases, an additional interpreter module is needed to translate these Python objects and parameters to each simulator's native parameters and language. For our system, we developed PyHet – the BrainFrame-specific Python interpreter – which serves the aforementioned role and also implements the accelerator selection.

5.2.1.3 Frontend and Automation

BrainFrame uses a combination of technologies in order to "tie" the underlying computational fabrics and modelling language into a reliable and expandable online platform.

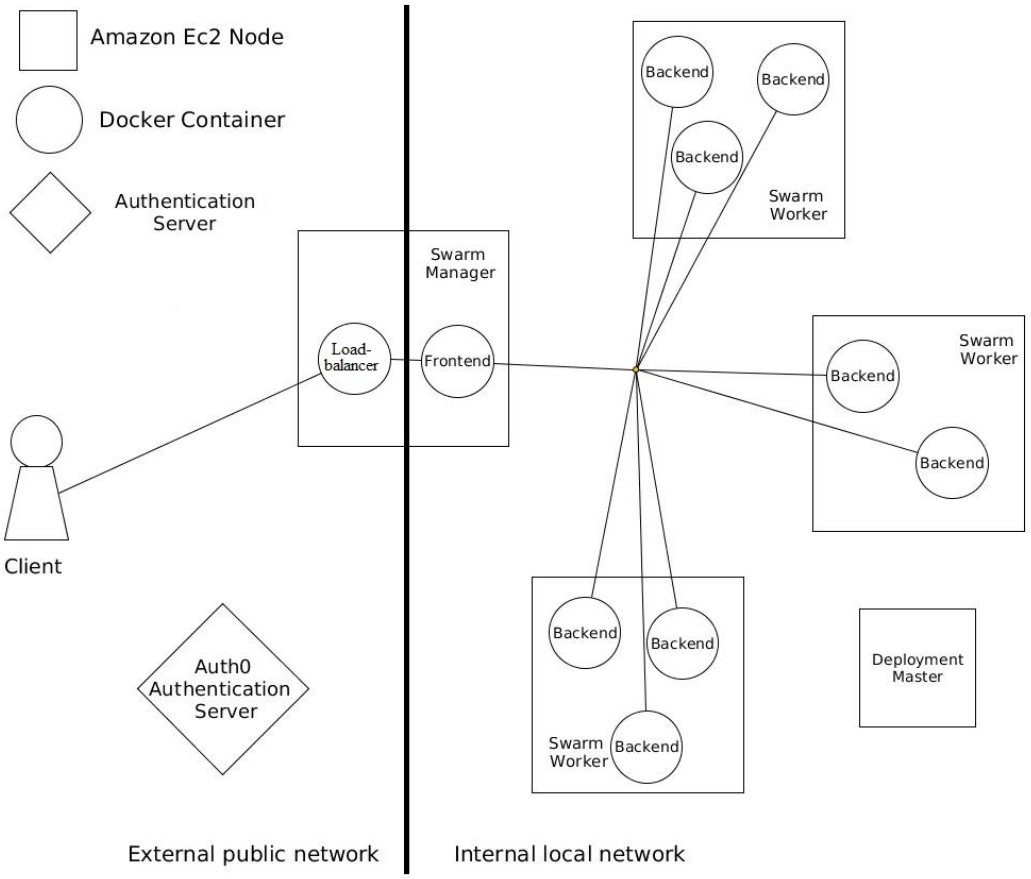


FIGURE 5.2: An abstract schema of BrainFrame's Architecture.

These technologies are necessary for designing the framework as a flexible cloud application, hosted on the Amazon AWS. They also aid with automation and recovering after service disruption. The front-end serves as an online portal for any scientist to interact with the framework without having prior engineering knowledge. The tools for achieving a reliable service will be briefly presented here.

Figure 5.2 demonstrates a snapshot of BrainFrame's state while functioning as an online service. In this particular snapshot, four computational nodes are employed, where three are acting as "swarm" workers and one is the "swarm" manager. In addition, ten Docker containers [214] have been spawned; one container acts as the **loadbalancer**, one container hosts the **frontend** service and the rest of the containers (eight) act as **backends**. The Figure's snapshot acts as an example. In this snapshot, there is only one frontend container. However, there can be multiple replicas "living" on any node. The framework's client connects to BrainFrame infrastructure through the loadbalancer. After a handshake, the loadbalancer forwards the incoming traffic to a frontend instance. The instance redirects the client to an **Auth0** authentication server in order to issue an authentication token. Finally, the Auth0 authentication server redirects the client back to the loadbalancer. The framework is then ready for handling the client's simulation requests, designed via the PyNN API mentioned in Section 5.2.1.2. Upon receiving one,

the loadbalancer can select an optimal platform amongst those available; an option exists for the client to request a specific computation platform. On the designated computation node, a random backend container is assigned to execute the simulation.

The technologies that enable the Figure's snapshot and the services described will be briefly mentioned here.

- Application level Technologies
  - Nodejs
  - Traefik
  - MongoDB
  - Amazon S3
- Deployment level Technologies
  - Amazon Ec2
  - Docker
  - Ansible

Docker is a containerization engine; containers can be described as "lightweight, standalone, executable packages of software that include everything needed to run an application: code, runtime, system tools, system libraries and settings". Each container is spawned based on a Docker container image, which are described in special files, called Dockerfiles, containing specific instructions for constructing the image.

Docker Swarm is a collection of computational nodes. Figure 5.2 showcases a swarm that consists of four nodes, one of which is the manager, which directs the functionality of the swarm, while the rest act as workers. Multiple managers can co-exist in any given swarm; furthermore, all managers can act as workers, while maintaining orchestration functionality.

Docker Overlay Network expands among the docker daemons of all the nodes of the swarm. The containers of each node use this network to communicate with the containers living on other nodes. In the current BrainFrame implementation, all containers are connected using a single overlay network codenamed **bf_net**. For security purposes, **bf_net** is not a public network but strictly an internal network.

Traefik is charged with feeding BrainFrame's Docker Overlay Network with traffic coming from the public network and acts as the framework's loadbalancer. Traefik is the only component of BrainFrame that has publicly exposed port, as all inbound traffic passes through Traefik. As a loadbalancer, Traefik balances the inbound traffic amongst multiple instances of backends, according to a specified loadbalancing strategy. In BrainFrame, the strategy used is named "session-affinity". This strategy guarantees that upon

user disconnection, the user will reconnect to the same container. Traefik also serves as the endpoint of TLS communication; all traffic between client and Traefic is encrypted. The SSL certificate used to achieve trusted and encrypted communication is issued by "Let's encrypt", a certificate authority trusted by all popular browsers [215].

Docker Service is a collection of identical containers. A service can have zero or more replicas of a container. A service deployed on a Docker swarm spreads its containers on the nodes of the swarm according to rules defined by the developer. A collection of different services is called a **Docker Stack**. Some of the Docker services employed by BrainFrame are the following:

- **load-balancer**: a service that consists of containers running a traefik image.
- **frontend**: a service that consists of containers assigning simulation tasks to the backend containers.
- **backend**: a service handling simulation tasks by the frontend service. There are multiple backend services instead of a single one. These services are classified according to the computing fabric they run on and its respective computing strength. For instance, one backend service runs on compute-optimized processors, whereas a different service runs on FPGA accelerators. In this manner, BrainFrame's design allows the integration of new computing fabrics like a new collection of CPUs or GPUs with ease, by simply spawning a different backend service.

Nodejs is a javascript framework for the creation of web applications. Nodejs instances run in BrainFrame's frontend containers. Nodejs integrates an application server. This server is not directry accesed by the clients; rather, traffic is forwarded to the Nodejs application server from Traefik. Nodejs uses npm as a package manager, which is currently the most module-rich manager available.

MongoDB is the database used for BrainFrame, which is necessary for storing data concerning clients and their requested simulation results. MongoDB is a NoSQL database.

Amazon S3 is the simplest data storage solution that Amazon AWS offers. User scripts and experiment results are stored on Amazon S3 in order to be accessible by all containers. Amazon S3 uses "buckets" to store data. BrainFrame creates a new bucket for each user, using a naming convention for the bucket based on an md5 hash of the user's email. In each user's bucket, her uploaded files are stored in a folder structure. Nodejs and Amazon S3 instances communicate via a custom wrapper of the S3 Amazon API.

Amazon Ec2 offers virtual machines of various specifications, most common of which are:

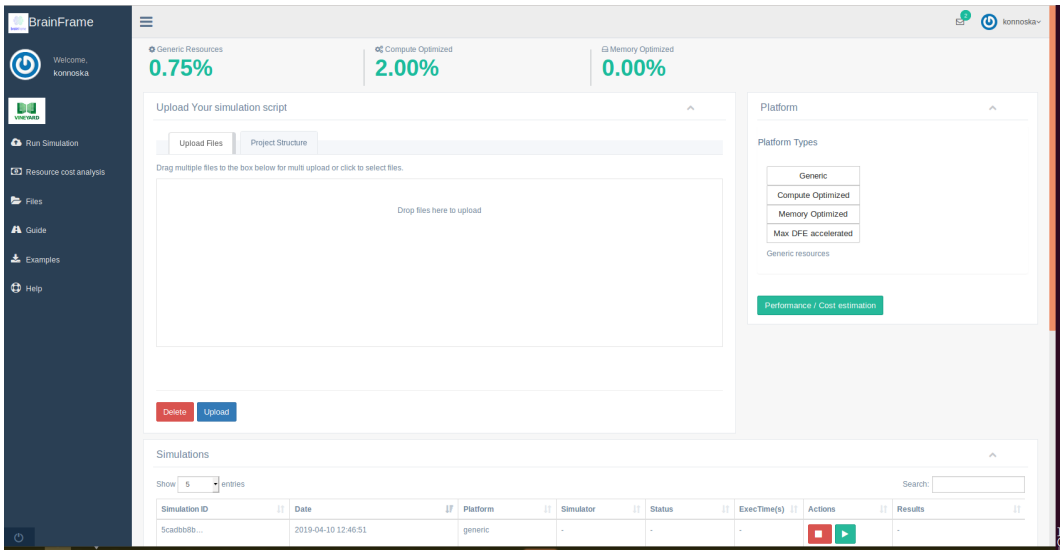


FIGURE 5.3: A screenshot of the BrainFrame online service.

- t2 and t3 which are general-purpose nodes.
- c4 and c5 which are compute-optimized nodes.
- r5, r4 and x1 which are memory-optimized nodes.
- p2, p3 and f1 which are accelerator nodes using either GPUs (p2, p3) or FPGAs (f1).

Finally, **Ansible** is a tool that automates processes, called "plays", which run on a specified set of nodes. A play is considered complete only when all defined tasks are carried out successfully. A set of plays can be written in a yaml-formatted file, which Ansible refers to as the "playbook". Ansible playbooks automate processes such as stopping the production environment of BrainFrame, reinitializing it once completely stopped from the ground up and performing live update of the production environment's image.

In order to put all of the aforementioned services into perspective, a screenshot of the online service is presented in Figure 5.3. The user may upload his simulation scripts, which are written in Python using PyNN, along with any auxilliary files necessary for his experiment. He may choose amongst the available computational resources provided by the computational cloud powering BrainFrame (which is hosted on the Amazon AWS in the current implementation), or allow the loadbalancer to make a decision based on the available resources and the performance cost estimator. The estimator gauges the best-performing computational fabric out of those available in the heterogeneous computation cloud. As it will be shown in the following experimental evaluation of the framework, the best-performing fabric heavily relies on the specifics of the simulation parameters.

It should be noted that the estimator also takes the monetary cost of utilizing a specific backend into consideration. In any heterogeneous cloud, virtual machines of different computational capacity feature significantly different hourly usage rates. As such, the estimator may prefer a slightly less powerful solution in terms of performance if the projected cost of the simulation is significantly lower. The user can view the estimator's projected simulation cost and execution time for each available backend and make an informed decision of his own.

5.2.2 Experimental Evaluation

In order to evaluate the performance of the presented framework, a series of experimental simulations have been made on a variety of workloads. The evaluation will be presented in this Section; it will be shown that significant gains in overall performance can be attained by utilizing an heterogeneous collection of hardware.

An early evaluation of the comparative strengths of different accelerators in the presented use case had been initially presented in the International Symposium on Performance Analysis of Systems and Software (ISPASS) of 2016 [216]. However, part of the measurements performed for the evaluation of large neuronal networks on Xeon Phi manycore processors and presented on that paper have been later found to be contaminated. This was caused by a bug in the algorithm generating the adjacency matrix for the networks simulated on the Xeon Phi, resulting in inaccurate performance measurements for the manycore processors, particularly for large networks. As such, the findings reported on the paper of ISPASS 2016 will be omitted from this dissertation. Instead, the section will focus on the evaluation presented in the Journal of Neural Engineering on 2017 [5], which gave a more mature and bug-free picture of the merits of the proposed heterogeneous framework.

5.2.2.1 Experimental Setup

In order to evaluate BrainFrame and provide a first proof of concept, we utilize the InfOli model presented so far in this disseration. The model has been shown to be particularly demanding in terms of processing power and inter-neuron communication and as such, constitutes a proper benchmark for the framework's potential. Furthermore, single-node evaluations will be presented; in particular, for the Xeon Phi manycore processors, a single Knights Corner processor has been used, according to the implementation shown in Section 4.2. While more advanced implementations have been depicted in Sections 4.3 and 4.4 and will be integrated in the framework in the future, a proof of concept can be attained nonetheless.

To validate the correct functionality of the separate accelerator implementations, we use a simple experiment that recreates a typical response that is found in the inferior-olive network (axon response). In this experiment, each cell produces a spike, from all simulated cells. 6 seconds of brain time are simulated, which translates to 120,000 simulation steps. The spike is produced by applying a small current pulse as input to all InfOli cells at the same instance after a programmed onset, for about 500 simulation steps (or 25 ms, in brain time). Despite being rudimentary, this experiment is easy to validate, provided all neurons are initialized with the same state, and also gives a good indication whether synchronization between neurons is correct, thus validating cell interconnectivity (when present).

We identify two distinct tracks that can be followed in conducting neuroscientific experiments, both covered in this evaluation. We perform one batch of measurements ranging from 96 to 960 neurons representing small-scale, real-time TYPE-I experiments, and a second batch ranging from 960 to 7,680 neurons representing larger-scale TYPE-II experiments. The neuroscientific community typically considers meaningful network sizes for experiments to start at approximately 100 neurons, thus our measurements for TYPE-I experiments begin at 96 neurons. Since the evaluation is restricted to the performance of single-node accelerators, a network-size cap is set by the smallest maximum network supported by each of the three accelerator fabrics: in this case, the DFE fabric limits network sizes to 7,680 cells.

The network connectivity is defined by an $N \times N$ *connectivity matrix* (where N is the network size) of floating-point weights signifying the weight of each connection. The weight value is used in the Gap Junction computations to calculate the connection impact on the neuron. The three use cases are focused around the biological complexity of the modeled Gap Junctions:

1. **Realistic Gap Junctions (RGJ)** – InfOli cells modeled with biophysically realistic GJ interconnectivity as presented in the original work of De Gruijl et al [217]. The highest amount of detail is included in the GJ modeling.
2. **Simplified Gap Junctions (SGJ)** – InfOli cells modeled with GJs replaced by simplified, passive connections. This constitutes a simpler connectivity in comparison to the previous use case.
3. **No Gap Junctions (NGJ)** – InfOli cells modeled without accounting for GJs and without any interconnectivity implementations. This is the simplest use case, whereby the neurons are modeled as independent computational islands.

Since BrainFrame aims at being a framework which accomodates for multiple levels of modelling detail, including a simplified version of the connectivity model in the evaluation is important. The level of detail as in the RGJ case is useful for many modeling

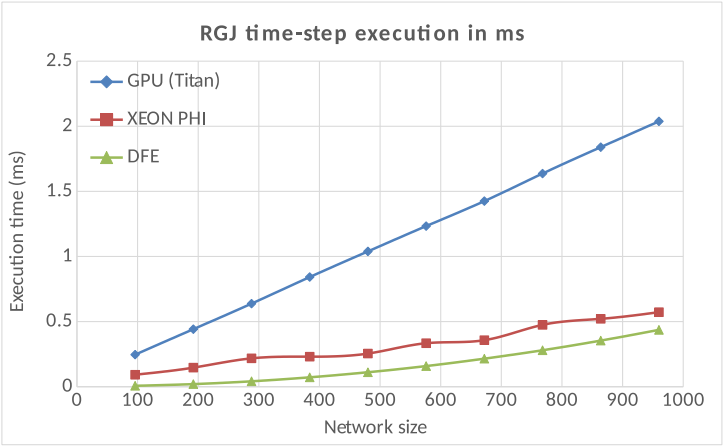
experiments; it is also an overkill in many other cases where simpler rudimentary connections are involved, like in simple synapses that accumulate inputs. As such, lighter workloads are represented by the SGJ case. As for the NGJ, it is the case where the application becomes purely dataflow and can achieve the greatest parallelism possible, representing the lightest workload with the smallest amount of floating point operations.

All performance measurements concerning the Xeon Phi have been carried out through the VTune Amplifier XE 2015 profiling and analysis tool by Intel. Timing measurements on the Maia DFE were taken by measuring the DFE-kernel time inlined within the host code using timestamps before and after the kernel call. Since, the host code (in the CPU) is blocking, only the DFE kernel is active during measurements. The time includes the kernel execution (processing and DRAM data-exchange delay) and the activation delay of the FPGA device. This activation takes about 1 ms, which is negligible compared to the overall execution time that takes several seconds to several minutes in our test experiment. GPU kernel-time measurements were taken using the CUDA Event API.

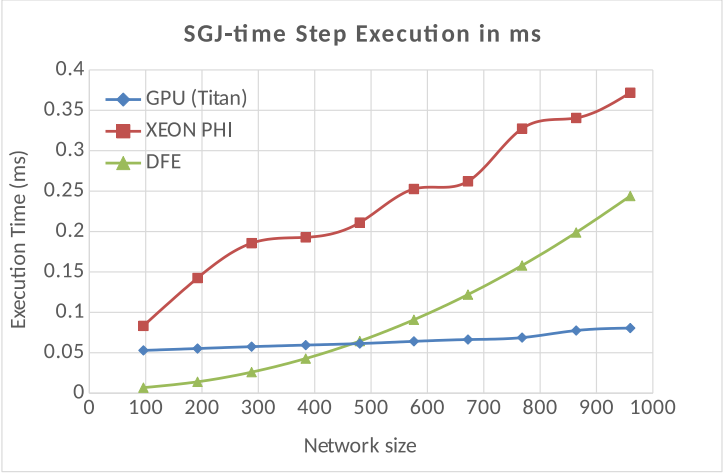
5.2.2.2 Results

Starting with the analysis for TYPE-I experimentation, in Figure 5.4a we plot the execution time of a single simulation time-step (50 μsec) for the most demanding use case, that of the RGJ with 100% connectivity density. Even though still not the most common case, a brain-simulation platform must support such high interconnectivity densities for certain TYPE-I experiments. The DFE exhibits the best performance for all tested network sizes. The Xeon Phi is a close second due to the local-memory delays and the less efficient use of its parallel threads: These network sizes are not large enough to provide sufficient parallelism for the Phi threads to be fully utilized. The GPU, on the other hand, has difficulties to cope with the computational intensity of the GJs, which involve mostly division and exponent FP calculations. Since each CUDA thread executes a single neuron, it cannot exploit any potential parallelism in the GJ calculation. This, alongside the fact that the CUDA threads are underutilized at such network sizes, impacts performance drastically.

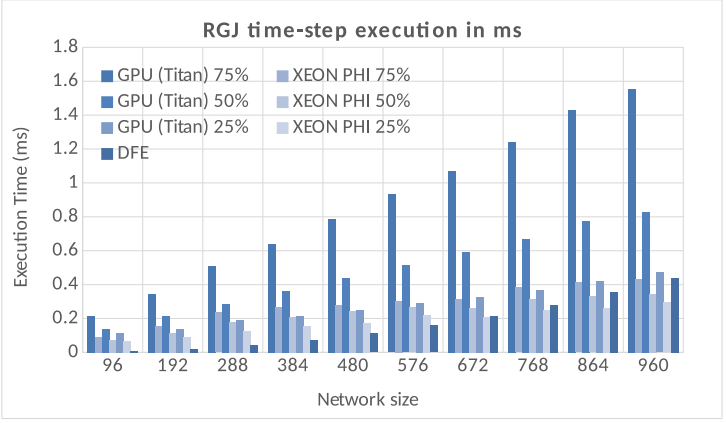
The inefficiency of the Titan X GPU in performing the realistic GJ computations is clearly revealed in the SGJ case, next (see Figure 5.4b). In this use case, that the most demanding GJ calculations are dropped, the GPU presents excellent scalability as the problem size increases, compared to the RGJ case. The Xeon Phi, on the other hand, still suffers from core-to-local-memory synchronization delays even though the actual calculations are much simpler now. The DFE needs to spend the same amount of operation ticks as in the RGJ case to evaluate the connection influence, even though it does enjoy gains in performance because of the simpler calculations involved (achieving higher operation frequencies, larger GJ computation parallelism and shorter pipelines). As a result, both latter accelerators show similar scaling properties to the RGJ case. In



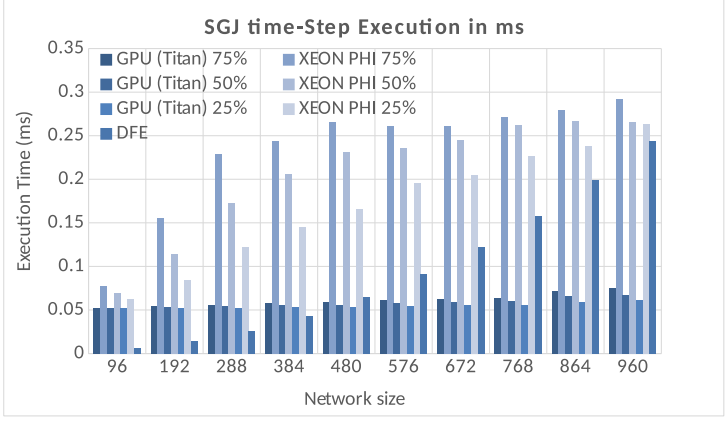
(A) RGJ execution time (TYPE I, 100% connectivity)



(B) SGJ execution time (TYPE I, 100% connectivity)



(C) RGJ execution time (TYPE I, <100% connectivity)



(D) SGJ execution time (TYPE I, <100% connectivity)

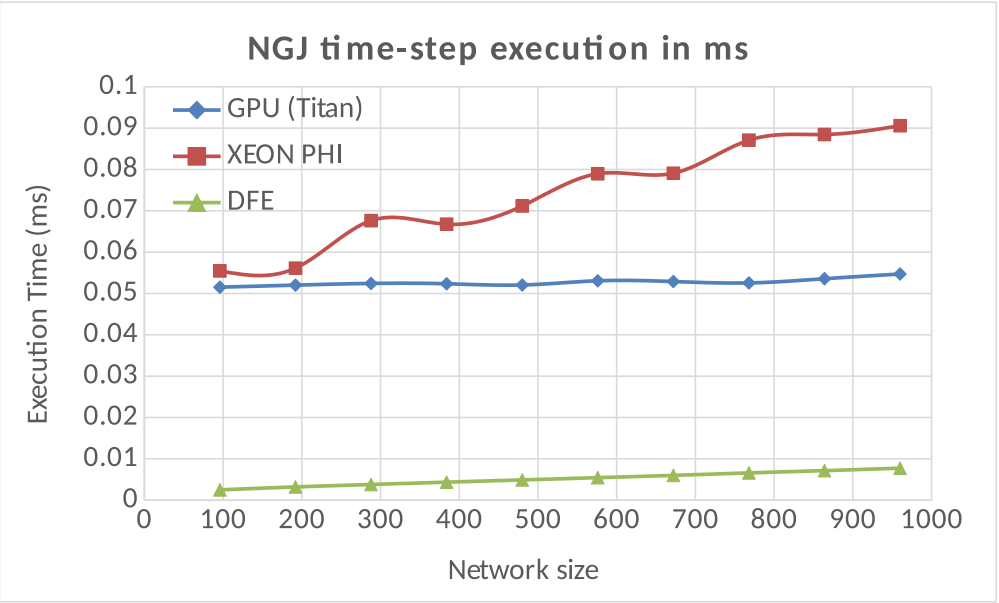


FIGURE 5.5: NGJ execution time (TYPE I, no connectivity).

contrast, the GPU scores performance benefits in the SGJ case compared to the robust DFE for network sizes above 480 neurons.

Next, it is interesting to evaluate the three accelerators for connectivities of lower than 100% density. Although not relevant for the DFE which maintains the same implementation for any connectivity density, smaller densities can influence the Xeon Phi and the GPU performance considerably. In Figure 5.4c, we plot the execution time of a single simulation time-step for 25%, 50% and 75% connectivity densities, under the RGJ case. The GPU delivers significant gains but the inefficient GJ execution still causes it to perform worse than DFE, even though the latter operates as in a 100%-density simulation. The Xeon Phi, on the other hand, manages to achieve enough performance gains to become faster than the DFE for sufficiently large problem sizes; that is, sizes $\geq$960 neurons for 75% density, $\geq$864 neurons for 50% density and $\geq$672 neurons for 25% density.

Under the SGJ use case (Figure 5.4d), we see similar trends as for the 100% SGJ use case: The GPU exhibits great scalability and is the best option for network sizes higher than 480 neurons. Besides, the DFE remains the most beneficial option for networks smaller than 480.

Under the NGJ case (no connectivity), for TYPE-I experiments, the results point to the DFE as the uniformly best option. In the complete absence of inter-neuron connectivity, the application becomes a purely dataflow workload, fully compatible for acceleration on a DFE, which is tailor-made for such cases, providing significant benefits over both the Xeon Phi and the GPU (see Figure 5.5).

Lastly, recall that for TYPE-I experiments, real-time speeds are often desired. The results show that, for real-time experimentation, the DFE accelerator is the best option

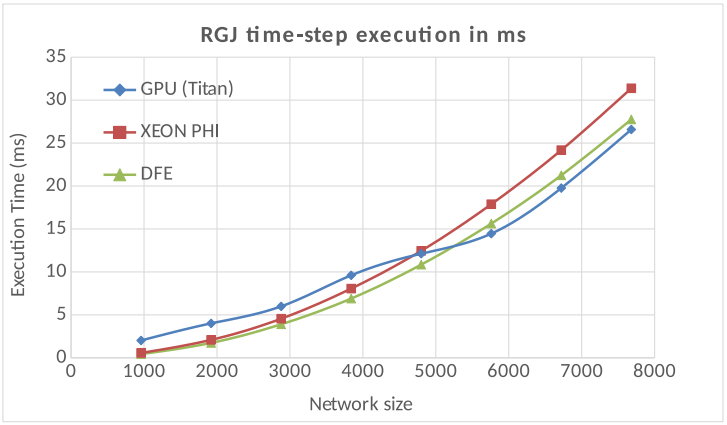
across the board. In contrast, and as mentioned in our previous analysis, the GPU and Xeon-Phi parallel threads tend to be underutilized at such small network sizes, even though most of the delays of using them are present. Thus the DFE – using fine-grain super-pipelined kernels – can achieve meaningful network sizes at real-time speeds under all use-case instances, according to the objective set in the introduction (≥ 100 cells). For low ($le50\%$) or zero densities, the GPU and Xeon Phi come close to the real-time objective, yet it is interesting to note that the DFE can even support real-time experimentation for TYPE-II experiments under the NGJ case.

For TYPE-II experiments, the trends under the RGJ case with 100% connectivity change significantly (see Figure 5.6a). Here, the massive explosion of the GJ computations begins to stress the parallelization capabilities of both the Xeon Phi and the DFE. The DFE's efficient parallelization of the GJs relies mostly on its ability to unroll the GJ loop on the FPGA hardware, allowing for more iterations to finish per operation tick. However, the achievable unrolling factor is limited by the available chip area. For network sizes above 1,000 neurons, the DFE compiler is forced to reuse a lot of resources in time (as the unrolling factor is reduced with increasing network sizes). In effect, the dataflow paradigm gradually degenerates to a sequential execution, making the application less scalable on the DFE. The Xeon Phi follows a similar trend, as the communication overhead between cores (which are interconnected through a moderately efficient ring topology [218]) increases, leading to similarly diminished scalability. Opposite to these accelerators, GPU scalability is largely improved. The GPU is underutilized until all CUDA cores are used (3,072) simultaneously, so for experiments over 3,000 neurons scalability is gradually improving. As a result, the GPU becomes the better performing solution (surpassing the DFE) for network sizes of 4,800 neurons and above.

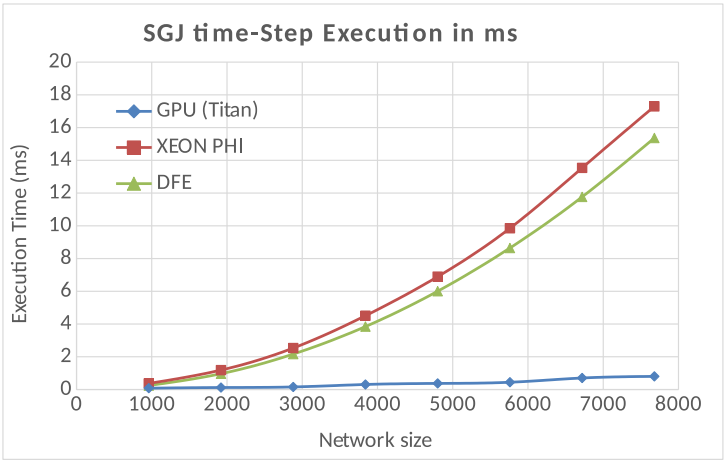
For lower connectivity densities under the RGJ case, we observe similar trends, although the Xeon-Phi scalability is slightly better because of the lower interconnectivity (see Figure 5.6c). Thus, the Xeon Phi retains the advantages it has for lower than 100% densities, compared to the DFE. Still, the effect of the inter-core communications is present allowing for the GPU to overtake the Xeon Phi for network sizes above 4,800 neurons (for densities of 50% and 75%) and above 3,840 neurons (for 25% density).

Under the SGJ case, the DFE and Xeon Phi follow similar trends, although they are less pronounced (see Figures 5.6b and 5.6d). As in the RGJ case, the GPU maintains its lead over the other two accelerator types for all tested network sizes and connectivity densities. Finally, in the NGJ case, the situation is the same as with TYPE-I experiments: The purely dataflow nature of the application allows the DFE to once more score the best performance across the board.

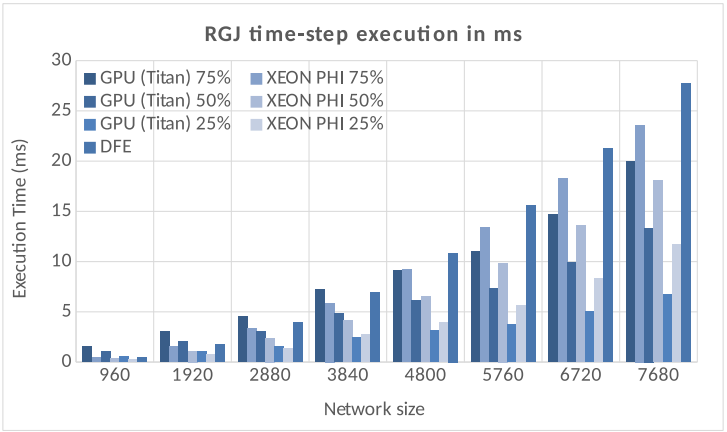
The performance analysis discussed above can now be used to formulate a simple accelerator-selection algorithm for BrainFrame automatically choosing the best-suited accelerator fabric based on the problem parameters: mainly, connectivity detail (biophysically realistic: RGJ, simple: SGJ and not present: NGJ), density and network size.



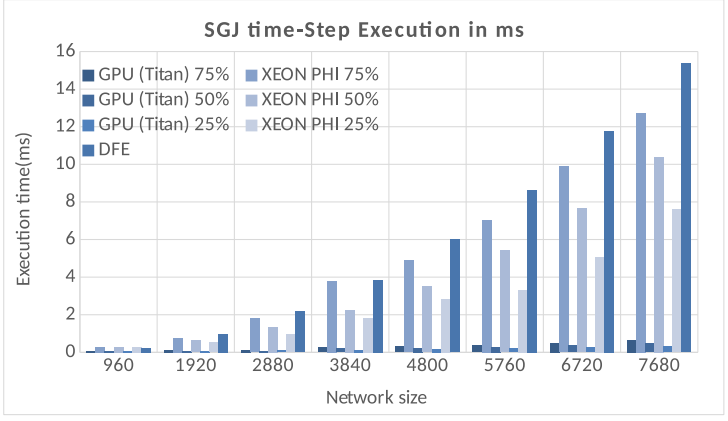
(A) RGJ execution time (TYPE II, 100% connectivity)



(B) SGJ execution time (TYPE II, 100% connectivity)



(C) RGJ execution time (TYPE II, <100% connectivity)



(D) SGJ execution time (TYPE II, <100% connectivity)

FIGURE 5.6: Type II experiments

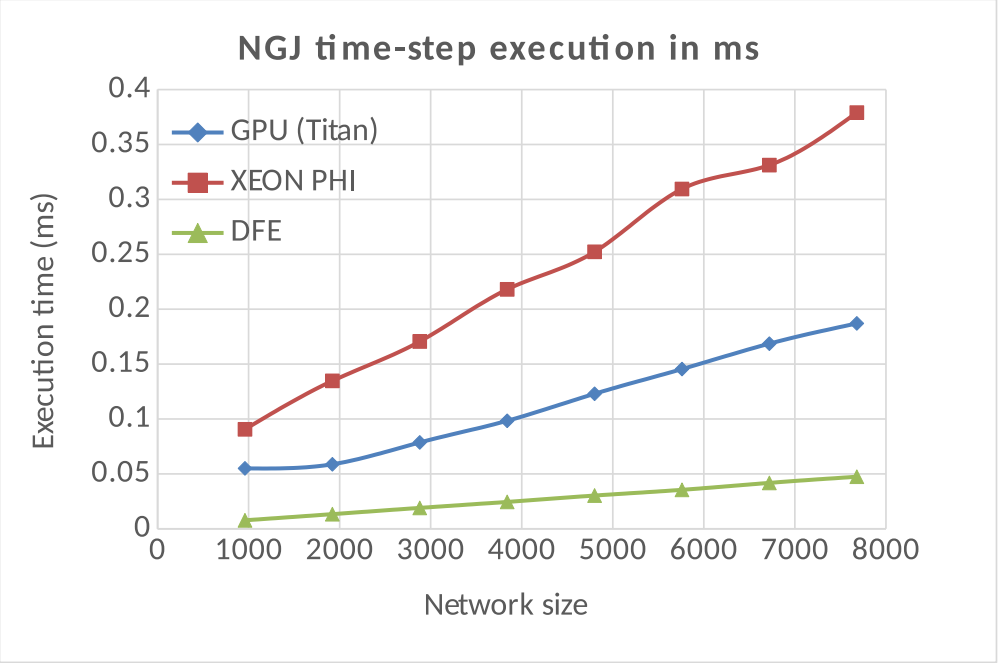


FIGURE 5.7: NGJ execution time (TYPE II, no connectivity).

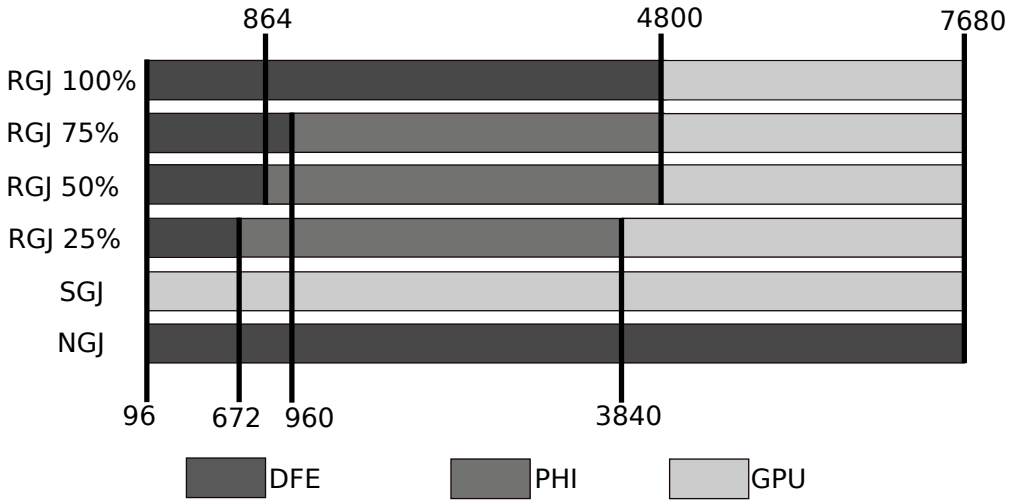


FIGURE 5.8: BrainFrame accelerator-selection map for TYPE-II experiments. Selection is heavily dependent on the experiment, involving all three accelerator fabrics. For TYPE-I experiments, the DFE is always the optimal choice (not shown).

Figure 5.8 shows the selection for our use-case instances. The RGJ case selection, which presents the most complex case in terms of accelerator choice, shifts between all three options depending on the connectivity density. For the SGJ case, the GPU is always the accelerator of choice, while for the NGJ case the DFE yields optimal results under all experiment parameters. Lastly, if the experiment is flagged as a real-time experiment, the algorithm exclusively chooses the DFE to accelerate the application, as it is the only clearly viable accelerator for real-time experiments.

5.2.3 Moving Forward

The evaluation presented in the previous Section 5.2.2 did not account for the entirety of BrainFrame's capability and potential; instead, it acted more as a proof of concept. Moving forward, incorporating more advanced implementations, already presented in this thesis, into BrainFrame is a significant priority. Incorporating more hardware fabrics and developing appropriate implementations (and the respective environments in Docker images) that utilize the underlying computing platform efficiently is a key factor in sustaining BrainFrame and having an up-to-date framework.

Multinode implementations, in particular, are a priority in BrainFrame implementation. As shown in Section 4.4.3, complex models can be accelerated efficiently in multinode implementations, provided specific conditions. As such, the scalability of the framework would be greatly increased by adding capability to handle simulations tasks by employing multiple instances of hardware in the cloud. It should be noted that due to how containerization technology operates, "housing" multiple hardware nodes in a single container, as well as inter-container communication during a simulation, is not a trivial matter; further research is needed in order to achieve an elegant design for multinode neuronal simulations in a containerized environment.

Finally, while accurate biophysical neuromodelling is the focus of this dissertation, large-scale modelling frequently uses simpler models for behavioural experiments. BrainFrame is designed with flexibility in mind and aims at incorporating more options in its available library of supported models. A first step has been made by supporting popular neuromodelling packages mentioned in Section 2.2.2, like NEURON and NEST. Moving forward, finely-tuned implementations of simpler Integrate-and-Fire models, such as the Izhikevich model mentioned in Section 2.1.2, will be able to better utilize the underlying heterogeneous ensemble of computational power that the computing cloud provides. Due to BrainFrame's scalable design, such modelling kernels can be progressively added to the existing library of models without disruptions in service.

5.3 Summary

A novel cloud-inspired methodology for supporting HPC-enabled neuroscientific experiments in massive scale was presented in this Chapter. A heterogeneous system of computational resources is effective for handling a large and varied stream of requests for neuronal network simulations. Depending on the characteristics of each simulation request, a different simulator can prove to be the most well-suited for the task. In order to have access to the computational resources to build such a system, cloud services are the option recommended in this Doctoral thesis.

The system of BrainFrame was thoroughly presented as an online service hosted on the Amazon AWS cloud platform. The system consists of three layers. A user-friendly and intuitive frontend accepts simulation requests by the user and spawns containerized jobs. These jobs contain user-submitted Python scripts based on a modelling tool, PyNN, that is widely used in the neuroscientific community. BrainFrame's middleware uses PyNN to properly set up each task for the respective accelerator to handle. The tasks are then executed by an heterogeneous ensemble of accelerators, consisting of manycore CPUs, Dataflow Engines (FPGAs) and GPUs. Depending on each task's simulation parameters, as well as the cost of utilizing each resource, a specific accelerator is chosen by the scheduler.

An evaluation of each accelerator was provided in order to highlight the significance of heterogeneous computing in this effort. We have focused our analysis on biophysically-accurate neuron models, like the InfOli model presented in the previous chapters of this thesis, but the BrainFrame system is designed with the goal of potentially supporting any modelling effort. The performance analysis of the system, employing use cases that take into account connectivity density and modeling complexity, revealed that all three fabrics (Xeon Phi, FPGA, GPU) are essential within the described simulation platform so as to optimally serve all possible experimentation cases. The platform, thus, achieved efficient large-network experiments as well as real-time performance for meaningful network sizes ($\geq$ 100 cells).

Chapter 6

Conclusions

6.1 Thesis Summary

This dissertation gave a complete presentation of handling the task of developing a neuroscientific simulator on manycore processors. A thorough insight into the world of computational neuroscience was given, briefly describing the nature of the computational workload. Existing frameworks are widely used by the neuroscientific community, however they are not without shortcomings; many simulation tools lack the option of detailed modelling and do not offer the level of detail many scientists require for their experimentation. Furthermore, other tools that are a staple in the field are relatively old software tools and impose significant difficulty in usage and in utilizing a high-performance computing environment.

The task of developing a modern and biophysically accurate neuron simulator commences by opting for a mostly research-grade manycore platform, the Single-chip Cloud computer (SCC). The simulator of our choice is based on a transient, time driven model for the inferior-olive neurons (hence named InfOli simulator), which are of major importance for human sensorimotor control. The selected InfOli simulator serves as a significant benchmark for parallelization and scaling of biologically-plausible neuron modeling workloads. We have presented a thorough Design Space Exploration (DSE); in this feasibility study, we have explored different partitioning schemes, based on data and combined task-and-data partitioning. Also, we explored the power-management options of the chip, implementing both Dynamic Frequency Scaling and Static Voltage and Frequency Scaling. Combinations of mapping and power-management options create a design space of different points. The quality cost of the simulation along with the sensitivity of the Pareto space in problem parameters, motivate a systematic treatment of this design space, in order to guarantee truly optimal utilization of the platform. A Pareto optimality problem has been formulated to extract such optimal platform configurations. The findings of the DSE reveal that a symmetric configuration, with identical

workload-per-core and a global power management policy is optimal for the mapping of the InfOli simulator on the SCC.

Following the findings on the research-oriented SCC, we ported the demanding neural-network simulator on a more industrial-level manycore processor: a single-node, dual-socket Xeon processor and Xeon Phi Knights Corner system. We initially tested three native implementations while letting the compiler's (icc) optimizations handle vectorization: an MPI-based one, an OpenMP-based one and a combination of both. The MPI implementation underperforms for the Xeon Phi accelerator since it does not utilize the coprocessor's multithreading resources, but performs well on the host, particularly for more than 10^5 neurons. The hybrid implementation improves on MPI's shortcomings on the accelerator, whereas in the Xeon host's case MPI already utilizes the platform efficiently.

OpenMP was the optimal choice on both computing platforms for smaller networks, albeit its performance did not scale linearly in all use-cases. On the Xeon host, OpenMP implementation can have significant variations in performance depending on network size. Furthermore, it was shown that the accelerator's hybrid implementation and the host's pure-MPI programming method rival OpenMP for large networks ($\geq 10^4$ simulated neurons) and should be considered due to them being extensible to multi-node systems. It should be noted that overall, despite the Knights Corner having a larger computational resource pool, the Xeon processor exhibited better overall performance than the Knights Corner, scaling up to a million inferior-olivary nuclei. Due to their gap in performance, a more elaborate and manual approach to fine-tuning the application, particularly with vectorization in mind, was deemed necessary.

The shared-memory implementation via OpenMP was manually tuned for the underlying platforms. A combination of pragma directives, function inlining and specific memory allocation functions, specialized for cache line alignment, was initially used. These techniques are applicable to any codebase and form the basis of vectorizing any application. Furthermore, modifications that are specifically designed for the simulator's algorithm, are employed. As a result, a sizeable increase in attainable simulation speed was achieved for workload sizes that are eligible for vectorization. Thus, we encourage the usage of these optimizations on codebases that resemble the algorithm, connectivity patterns and problem sizes encountered in the InfOli modeling application. Overall, the techniques presented in this paper were beneficial for both the accelerator and the host. In particular, for networks that are large and densely-connected enough to saturate the Phi's assets, the difference in performance between manually vectorized code and un-optimized code that relies solely on the compiler is an order of magnitude.

After fine-tuning the application, the platforms performed differently depending on network connectivity density. Sparse networks are a good candidate for acceleration via the Phi co-processor's large pool of computation resources. Furthermore, dense networks feature a range of populations between 5,000 and 50,000 neurons where the co-processor

can use its computational resources to outperform the host. On the other hand, the host's focus on single-threaded and scalar performance is a better fit for dense networks outside this range due to their less well-parallelizable nature.

The next step in increasing the performance scale of the simulator was to transition to the industrial-grade manycore processor Knights Landing (KNL) and work towards a multinode implementation. For this implementation in particular, the simulator has been designed with a broader manycore architecture in mind, since the KNL hardware assets are found on most x86-based manycore processors. This approach allows the portability of the simulator and the extraction of meaningful insight concerning the behaviour of similar workloads.

The InfOli simulator's performance was tested using a range of workloads, from small, unconnected neuronal populations to larger, dense networks. The results were evaluated from both a simulation-speed and a power-efficiency standpoint. On average KNL offers a speed up of 2.4× while consuming 48% less energy. Smaller workloads, by taking advantage of the KNL's superior single-threaded performance, exhibit very significant gains in both speed and, even more so, energy consumption, with specific experiments demanding 75% less watt-hours of energy per second of simulated brain activity on the KNL. On the other hand, OpenMP-thread efficiency suffers when running on the KNL, causing the simulator to handle more demanding networks poorly, relatively to the optimized version of the 1st generation Xeon Phi. Furthermore, throughout the whole range of experiments, it has been shown that the KNL offers a more robust, dependable performance curve with little variability.

The implementation was then scaled to work on a multinode manycore processing systems via a hybrid usage of the MPI and OpenMP libraries. The simulator was tested on a system of 8 Xeon Phi KNL manycore processors. The work has proven that efficient usage of even a small cluster of manycore processors is able to achieve satisfactory performance even when facing a very demanding mathematical model of the human neuron, in network and synaptic sizes numbering in the millions and billions, respectively. It constitutes an efficient solution for studying demanding neuronal models in a pursuit of attaining deeper understanding of the human brain's intricate details.

Furthermore, it has been demonstrated that a biologically-accurate simulator exhibits performance patterns that are dictated by problem size and the nature of each network's connectivity map. A focal point in our analysis was the system's scalability in multinode setups. It has been highlighted that the system is highly sensitive to simulation parameters and as such, careful steps need to be taken in order to discern trends in performance behaviour.

After achieving satisfactory performance in a very demanding class of models and identifying the shortcomings of the simulator that were tied to the nature of manycore processors, an effort was made to transition to heterogeneous systems. In particular,

the system integrated implementations in accelerators that have demonstrated potential in this particular domain of the neuromodelling field: an Intel Xeon Phi Knights Corner implementation, as it was presented in this dissertation, a Maxeler Vectis Data-Flow Engine (DFE) implementation and an NVIDIA GPU implementation. Both quality and quantity of inter-neuron connections acted as a means for evaluating the system under varying neuronal network parameters. In all cases, the target neuron simulator scaled gracefully in terms of DRAM utilization, with both Xeon Phi and DFE platforms exhibiting enough slack in DRAM timing overhead. Regarding overall performance, the Maxeler Vectis DFE was clearly optimal for small- and medium-scale, real time simulations. Executing a fixed synthesized implementation, the performance of the DFE was not affected by changes in neuron connectivity density. The Xeon Phi implementation, on the other hand, appeared more suitable for large-scale simulations, with many neurons and dense interconnectivity between them, as was expected from the manycore-focused research presented in this dissertation. The GPU implementation also was a strong candidate for the most demanding of the evaluated networks.

The evaluation was carried out via an online service that was presented in this dissertation. BrainFrame is an heterogeneous acceleration platform that serve computational neuroscience studies in conducting the variety of real experimentation often required for the study of brain functionality. It is hosted on a cloud computing service, which at the moment of writing is the Amazon AWS, in order to have access to a wealth of heterogeneous computing resources. By utilizing the fabrics presented in the evaluation, the platform achieves efficient large-network experiments as well as real-time performance for meaningful network sizes (100 cells).

The system has an intuitive user interface that any neuroscientist can interact with, demands no prior engineering knowledge and utilizes scripts of a Python-based modelling package (PyNN) which is widely used in the neuroscientific field. The PyNN front-end makes the heterogeneous platform immediately accessible to a multitude of prior modeling works, which is an essential strategy for the wide adoption of complex HPC platforms in the neuroscientific community. Furthermore, building on the elegant PyNN infrastructure, a simple accelerator-selection algorithm has also been integrated in BrainFrame for automatically identifying the most suitable HPC fabric (Xeon Phi, GPU, DFE) per neuroscientific experiment, as well as present the relative cost of utilizing each platform available on the cloud service. Overall, BrainFrame has been designed as a scalable, easy-to-use and flexible solution to utilizing high-performance computing fabrics in the demanding domain of complex neuromodelling.

6.2 Thesis Highlights

Overall, the goal of this Doctoral thesis was to research and design a tool for a realistic and detailed class of neuromodelling simulations that would be scalable and efficient enough to meet the computational demands of the most intense *in-silico* experiments carried out by neuroscientists.

The challenges for this endeavour focused around the limited prior research available for large-scale, rich-in-detail simulations and the difficulty of scaling a communication-heavy application in a manycore computing fabric.

With these tasks in mind, the following contributions were made in this Doctoral Thesis:

Contribution I
a simulation design efficient enough to be able to simulate high-complexity, very-large-size networks (millions of neurons, billions of synapses) with a satisfactory simulation speed (minutes of execution time per second of simulated brain time) has been proposed and evaluated.

Contribution II
through this simulator, a thorough analysis of the behaviour of manycore x86-based computing systems was provided for the class of biophysically-meaningful neuromodelling applications and the workload parameters impacting performance, scalability and efficiency thereof, focusing on how network connectivity patterns can influence inter-node communication delays by orders of magnitude.

Contribution III
building on this analysis concerning simulation in manycore hardware, developer insights are provided, concerning effective development and evaluation of neuromodelling tools for a vast amount of different workloads, as well as the challenges thereof.

Contribution IV
by identifying the challenge of supporting neuronal networks with vast differences in setup configuration, the Thesis introduces a collaborative, cloud-based, heterogeneous online service (BrainFrame) that offers a complete solution for the problem of detailed neuromodelling; by combining the strengths of different accelerators, we achieve design flexibility and scalability that, in some cases, can more than double the simulation speed of non-heterogeneous, single-accelerator setups.

6.3 Future Work

There are multiple directions in which the work presented in this Doctoral thesis can evolve. In particular, concerning the BrainFrame service, a number of features can be added. Furthermore, emerging new, cutting-edge technologies can aid in the difficult task of achieving mapping the human brain via accurate model simulation.

6.3.1 Framework Expansion

BrainFrame is designed as a fully-scalable solution that continuously integrates new models and hardware solvers. As such, one of the immediate future goals for this work is to add support for newer neuronal models to BrainFrame. As computational neuroscience is still a young domain, mathematical models are constantly being developed and adopted by the neuroscientific community. Keeping the service up to date with new demands is a necessity for maintaining a framework that offers efficient solutions for neuroscientific labs specializing in different aspects of brain mapping studies.

Several auxilliary tools can also improve BrainFrame's functionality. An important part of performing a study with a neuronal model is the task of analyzing and visualizing the collected data. Especially for the complex, biophysically-accurate models studied in this Doctoral thesis, the amount of generated data by a long simulation can be massive; as such, the task of properly utilizing the collected data is non-trivial. A set of tools to assist in this challenge can boost BrainFrame's adoption by the community. Data visualization tools can be added via standard Python libraries in the middleware; this way, instead of receiving raw data, a user of the BrainFrame service can opt for organized and well-defined graphs and other visual cues for comprehending the output of his requested simulation.

In addition, designing more advanced schedulers for the heterogeneous hardware ensemble can help better handle incoming traffic when the system has scaled enough to accomodate a large amount of simulation requests simultaneously. A number of parameters can affect the decision of which fabric is optimal for a particular simulation. While execution time is the most important, the cost of a simulation as well as its storage needs can also play a critical role. A scheduler would need to make a decision while the service is constantly online based on multiple factors for a number of different use-cases. In order to achieve this, acquiring a data set of successful simulation requests for different neuronal models and simulation parameters is required. Given a large enough data set, the scheduler would be able to make a selection amongst the available hardware for a multitude of different *in silico* experiments.

6.3.2 Emerging HPC Technologies

Currently, new technologies and trends in high performance computing can influence the domain of computational neuroscience. Neuromorphic engineering [219], also known as neuromorphic computing, is a concept developed in the late 1980s, describing the use of very-large-scale integration (VLSI) systems containing electronic analog circuits to mimic neuro-biological architectures present in the nervous system. Recent developments and research efforts have brought attention to the applicability of neuromorphic hardware in the domain of computational neuroscience [220]. While the technology is primarily used in areas more attuned to Artificial Intelligence, the concept has also been shown to lend itself to traditional neuromodelling applications [221].

The demand for data can grow to be very high for large-network simulations, particularly in the case of complex neuronal models. As cloud services continue providing increasing amounts of available RAM to meet the memory demands of the field, the emerging technology of in-memory computing becomes an attractive option. In-memory computing refers to using combinations of software and specialized hardware in order to enable data storage directly in RAM, distributed across multiple platforms and to allow data to be processed in parallel. The technology has been tested on simpler Spiking Neural Network models for event-driven simulation [222]; additional research is necessary for the application of in memory computing in the case of transient high-detail neuromodelling.

It should be also mentioned that the technology in manycore processors is constantly evolving. While the Xeon Phi line of products is discontinued, the Xeon processors are evolving towards more resources and heavier usage of the AVX instruction set. An interesting notion in the domain of processor architecture is the development of non-homogeneous processor chips; AMD has been reported to develop Zen-architecture multicore processors with different lithography figures for the I/O (14nm) and the cores (7nm) of the processor [223, 224]. The prospect of a single-chip heterogeneous system is interesting and ties in well with the proposed benefits of heterogeneity for neuromodelling simulations presented in this Doctoral thesis.

Bibliography

[1] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Elsevier, 2013. ISBN 9780124104945.

[2] George Chrysos. Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper*, 2014.

[3] George Chatzikonstantis, Dimitrios Rodopoulos, Sofia Nomikou, Christos Strydis, Chris I De Zeeuw, and Dimitrios Soudris. First impressions from detailed brain model simulations on a xeon/xeon-phi node. In *ACM Computing Frontiers*, 2016.

[4] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[5] Georgios Smaragdos, Georgios Chatzikonstantis, Rahul Kukreja, Harry Sidiropoulos, Dimitrios Rodopoulos, Ioannis Sourdis, Zaid Al-Ars, Christoforos Kachris, Dimitrios Soudris, Chris I De Zeeuw, et al. Brainframe: a node-level heterogeneous accelerator platform for neuron simulations. *Journal of neural engineering*, 14(6):066008, 2017.

[6] Aldo Fasolo. *The theory of evolution and its impact*. Springer Science & Business Media, 2011.

[7] Eric L Schwartz. *Computational neuroscience*. Mit Press, 1993.

[8] Davison A.P. and Hines M. and Muller E. Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience*, 3:3:374–380, 2009.

[9] Zaytsev Y.V. and Morrison A. Increasing quality and managing complexity in neuroinformatics software development with continuous integration. *Frontiers in Neuroinformatics*, 3:3:6–31, 2013.

[10] B. Subramaniam and Wu chun Feng. The green index: A metric for evaluating system-wide energy efficiency in hpc systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1007–1013, 2012.

[11] École Polytechnique Fédérale de Lausanne. The blue brain project – main components of the infrastructure, June 2012. URL http://bluebrain.epfl.ch/page-58110-en.html.

[12] Eugene M Izhikevich and Gerald M Edelman. Large-scale model of mammalian thalamocortical systems. *Proceedings of the national academy of sciences*, 105(9): 3593–3598, 2008.

[13] Henry Markram, Eilif Muller, Srikanth Ramaswamy, Michael W Reimann, Marwan Abdellah, Carlos Aguado Sanchez, Anastasia Ailamaki, Lidia Alonso-Nanclares, Nicolas Antille, Selim Arsever, et al. Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2):456–492, 2015.

[14] Shyam Kumar Sudhakar, Sungho Hong, Ivan Raikov, Rodrigo Publio, Claus Lang, Thomas Close, Daqing Guo, Mario Negrello, and Erik De Schutter. Spatiotemporal network coding of physiological mossy fiber inputs by the cerebellar granular layer. *PLoS computational biology*, 13(9):e1005754, 2017.

[15] Chris Eliasmith and Oliver Trujillo. The use and abuse of large-scale brain models. *Current Opinion in Neurobiology*, 25:1–6, 2014.

[16] Jorge Golowasch, Mark S Goldman, LF Abbott, and Eve Marder. Failure of averaging in the construction of a conductance-based neuron model. *Journal of Neurophysiology*, 87(2):1129–1131, 2002.

[17] Astrid A Prinz, Dirk Bucher, and Eve Marder. Similar network activity from disparate circuit parameters. *Nature neuroscience*, 7(12):1345, 2004.

[18] Eve Marder and Jean-Marc Goaillard. Variability, compensation and homeostasis in neuron and network function. *Nature Reviews Neuroscience*, 7(7):563, 2006.

[19] Dragos Calitoiu, Doron Nussbaum, and B. John Oommen. Large scale modeling of the piriform cortex for analyzing antiepileptic effects. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 599–608, San Diego, CA, USA, 2008. Society for Computer Simulation International. ISBN 1-56555-319-5. URL http://dl.acm.org/citation.cfm?id=1400549.1400644.

[20] JM Bower and D Beeman. The book of genesis: Exploring realistic neural models with the general neural simulation system 2nd edn (santa clara: Telos). 1998.

[21] Elliot D Menschik and Leif H Finkel. Neuromodulatory control of hippocampal function: towards a model of alzheimer's disease. *Artificial intelligence in medicine*, 13(1-2):99–121, 1998.

[22] David T J Liley, David M Alexander, James J Wright, and Mathew D Aldous. Alpha rhythm emerges from large-scale networks of realistically coupled

multicompartmental model cortical neurons. *Network: Computation in Neural Systems*, 10(1):79–92, 1999. doi: 10.1088/0954-898X_10_1_005. URL https://doi.org/10.1088/0954-898X_10_1_005.

[23] Gordon E Moore. Lithography and the future of moore's law. In *Integrated Circuit Metrology, Inspection, and Process Control IX*, volume 2439, pages 2–18. International Society for Optics and Photonics, 1995.

[24] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[25] John Gregory and R McReynolds. The solomon computer. *IEEE Transactions on Electronic Computers*, (6):774–781, 1963.

[26] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[27] Oded Lempel. 2nd generation intel® core processor family: Intel® core i7, i5 and i3. In *Hot Chips 23 Symposium (HCS), 2011 IEEE*, pages 1–48. IEEE, 2011.

[28] Hitoshi Oi. Energy efficiency study of ryzen microprocessor. In *SoutheastCon 2018*, pages 1–5. IEEE, 2018.

[29] Xiaohu Ge, Hui Cheng, Mohsen Guizani, and Tao Han. 5g wireless backhaul networks: challenges and research advances. *IEEE Network*, 28(6):6–11, 2014.

[30] Xiang Tian and Khaled Benkrid. Design and implementation of a high performance financial monte-carlo simulation engine on an fpga supercomputer. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 81–88. IEEE, 2008.

[31] Henry Markram. The human brain project. *Scientific American*, 306(6):50–55, 2012.

[32] Erich Strohmaier. Top500 supercomputer. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 18. ACM, 2006.

[33] Timothy G Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 38. IEEE Press, 2008.

[34] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27(3):18, 2008.

[35] Ryan Smith. Intel kills larrabee gpu, will not bring a discrete graphics product to market, May 2010. URL https://www.anandtech.com/show/3738/intel-kills-larrabee-gpu-will-not-bring-a-discrete-graphics-product-to-market.

[36] Justin Rattner. Single-chip cloud computer. *An experimental many-core processor from Intel Labs*, page 14, 2010.

[37] John-Nicholas Furst and Ayse K Coskun. Performance and power analysis of rcce message passing on the intel single-chip cloud computer. In *4th Many-core Applications Research Community (MARC) Symposium*, page 27, 2012.

[38] Marc Snir. *MPI–the Complete Reference: The MPI core*. MIT, 1998.

[39] Pawan Kumar Updhyay and Satish Chandra. An acceleration of improved segmentation methods for dermoscopy images using gpu. In *Recent Trends in Communication, Computing, and Electronics*, pages 261–269. Springer, 2019.

[40] Lisa Marie Dreier, Svilen Stefanov, David Schneller, and Alexander Ditter. Sc17 student cluster competition, team technical university of munich and friedrich–alexander university erlangen–nürnberg: Reproducing vectorization of the tersoff multi-body potential on the intel broadwell architecture. *Parallel Computing*, 78: 79–83, 2018.

[41] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus, gpus and intel mic architectures. *Intel Labs*, pages 77–80, 2010.

[42] A Nowak, Alfio Lazzaro, Julien Leduc, and S Jarp. The breaking point of modern processor and platform technology. Technical report, 2011.

[43] Jack Deslippe, Brian Austin, Chris Daley, and Woo-Sun Yang. Lessons learned from optimizing science kernels for intel's" knights corner" architecture. *Computing in Science & Engineering*, 17(3):30–42, 2015.

[44] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent smp systems: a case-study with xeon phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108. ACM, 2013.

[45] Tianyu Liu, X George Xu, and Christopher D Carothers. Comparison of two accelerators for monte carlo radiation transport calculations, nvidia tesla m2090 gpu and intel xeon phi 5110p coprocessor: A case study for x-ray ct imaging dose calculation. *Annals of Nuclear Energy*, 82:230–239, 2015.

[46] HPC Inside. Cray wins 174 million dollar contract for trinity supercomputer based on knights landing, 2014.

[47] Requiem for a phi: Knights landing discontinued, Jul 2018. URL https://www.hpcwire.com/2018/07/25/end-of-the-road-for-knights-landing-phi/.

[48] AMD Processor Specifications. Amd ryzen^{TM} threadripper^{TM} 3970x processor, 2019. URL https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x.

[49] R. T. Naik and Shivani Patel. First zen architecture based commercial processor. 2017.

[50] Hassan Mujtaba. Intel xeon scalable processor family detailed, takes on amd naples server chips – xeon platinum with 28 cores, 56 threads and 8s+ configuration, May 2017. URL https://wccftech.com/intel-xeon-skylake-scalable-processor-family-detailed/.

[51] Jeremy Hsu. Ibm's new brain [news]. *IEEE spectrum*, 51(10):17–19, 2014.

[52] Andreas G Andreou, Andrew A Dykman, Kate D Fischl, Guillaume Garreau, Daniel R Mendat, Garrick Orchard, Andrew S Cassidy, Paul Merolla, John Arthur, Rodrigo Alvarez-Icaza, et al. Real-time sensory information processing using the truenorth neurosynaptic system. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 2911–2911. IEEE, 2016.

[53] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[54] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[55] Haohuan Fu, Junfeng Liao, Wei Xue, Lanning Wang, Dexun Chen, Long Gu, Jinxiu Xu, Nan Ding, Xinliang Wang, Conghui He, et al. Refactoring and optimizing the community atmosphere model (cam) on the sunway taihulight supercomputer. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 969–980. IEEE, 2016.

[56] Jian Zhang, Chunbao Zhou, Yangang Wang, Lili Ju, Qiang Du, Xuebin Chi, Dongsheng Xu, Dexun Chen, Yong Liu, and Zhao Liu. Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Press, 2016.

[57] Yulong Ao, Chao Yang, Xinliang Wang, Wei Xue, Haohuan Fu, Fangfang Liu, Lin Gan, Ping Xu, and Wenjing Ma. 26 pflops stencil computations for atmospheric modeling on sunway taihulight. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 535–544. IEEE, 2017.

[58] Wenqian Dong, Letian Kang, Zhe Quan, Kenli Li, Keqin Li, Ziyu Hao, and Xiang-Hui Xie. Implementing molecular dynamics simulation on sunway taihulight system. In *2016 IEEE 18th International Conference on High-Performance Computing and Communications, IEEE 14th International Conference on Smart City, and IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 443–450. IEEE, 2016.

[59] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[60] Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. High order seismic simulations on the intel xeon phi processor (knights landing). In *International Conference on High Performance Computing*, pages 343–362. Springer, 2016.

[61] Azzam Haidar, Stanimire Tomov, Konstantin Arturov, Murat Guney, Shane Story, and Jack Dongarra. Lu, qr, and cholesky factorizations: Programming model, performance analysis and optimization techniques for the intel knights landing xeon phi. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.

[62] Mathias Jacquelin, Wibe De Jong, and Eric Bylaska. Towards highly scalable ab initio molecular dynamics (aimd) simulations on the intel knights landing manycore processor. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 234–243. IEEE, 2017.

[63] Arnon Amir, Pallab Datta, William P Risk, Andrew S Cassidy, Jeffrey A Kusnitz, Steve K Esser, Alexander Andreopoulos, Theodore M Wong, Myron Flickner, Rodrigo Alvarez-Icaza, et al. Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE, 2013.

[64] Khronos OpenCL Working Group et al. The OpenCL specification. 2008.

[65] Modeldb home. URL https://senselab.med.yale.edu/modeldb/.

[66] Eric J Nestler, Bruce T Hope, and Katherine L Widnell. Drug addiction: a model for the molecular basis of neural plasticity. *Neuron*, 11(6):995–1006, 1993.

[67] Peter G Gillespie and Richard G Walker. Molecular basis of mechanosensory transduction. *Nature*, 413(6852):194, 2001.

[68] Yanan Han and H Steven Colburn. Point-neuron model for binaural interaction in mso. *Hearing research*, 68(1):115–130, 1993.

[69] Ying-Hui Liu and Xiao-Jing Wang. Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron. *Journal of Comp. Neuroscience*, 10(1): 25–45, 2001. ISSN 0929-5313.

[70] Chacron, M. J. et al. Interspike interval correlations, memory, adaptation, and refractoriness in a leaky integrate-and-fire model with threshold fatigue. *Neural Comput.*, pages 253–278, 2003. ISSN 0899-7667. doi: 10.1162/089976603762552915.

[71] Brette, R. and Gerstner, W. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology*, 94(5), 2005. ISSN 0022-3077. doi: 10.1152/jn.00686.2005.

[72] Louis Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *Journal de Physiologie et de Pathologie Generalej*, 9:620–635, 1907.

[73] Archibald Vivian Hill. Excitation and accommodation in nerve. *Proc. R. Soc. Lond. B*, 119(814):305–355, 1936.

[74] MGF Fuortes and Francoise Mantegazzini. Interpretation of the repetitive firing of nerve cells. *The Journal of general physiology*, 45(6):1163–1179, 1962.

[75] Alessandro Treves. Mean-field analysis of neuronal spike dynamics. *Network: Computation in Neural Systems*, 4(3):259–284, 1993.

[76] Eugene M Izhikevich. Resonate-and-fire neurons. *Neural networks*, 14(6-7):883–894, 2001.

[77] Nicolas Brunel, Vincent Hakim, and Magnus JE Richardson. Firing-rate resonance in a generalized integrate-and-fire neuron with subthreshold resonance. *Physical Review E*, 67(5):051916, 2003.

[78] Nicolas Hohn and Anthony N Burkitt. Shot noise in the leaky integrate-and-fire neuron. *Physical Review E*, 63(3):031902, 2001.

[79] Shinsuke Koyama and Robert E Kass. Spike train probability models for stimulus-driven leaky integrate-and-fire neurons. *Neural computation*, 20(7):1776–1795, 2008.

[80] Mitchell A Nahmias, Bhavin J Shastri, Alexander N Tait, and Paul R Prucnal. A leaky integrate-and-fire laser neuron for ultrafast cognitive computing. *IEEE journal of selected topics in quantum electronics*, 19(5):1–12, 2013.

[81] Peter E Latham, BJ Richmond, PG Nelson, and S Nirenberg. Intrinsic dynamics in neuronal networks. i. theory. *Journal of neurophysiology*, 83(2):808–827, 2000.

[82] Izhikevich, E. M. et al. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[83] E.M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Trans. on Neural Networks*, 2004.

[84] Nicolas Fourcaud-Trocmé, David Hansel, Carl Van Vreeswijk, and Nicolas Brunel. How spike generation mechanisms determine the neuronal response to fluctuating inputs. *Journal of Neuroscience*, 23(37):11628–11640, 2003.

[85] Romain Brette and Wulfram Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of neurophysiology*, 94(5):3637–3642, 2005.

[86] Loreen Hertäg, Joachim Hass, Tatiana Golovko, and Daniel Durstewitz. An approximation to the adaptive exponential integrate-and-fire neuron model allows fast and predictive fitting to physiological data. *Frontiers in computational neuroscience*, 6:62, 2012.

[87] Loreen Hertäg, Daniel Durstewitz, and Nicolas Brunel. Analytical approximations of the firing rate of an adaptive exponential integrate-and-fire neuron in the presence of synaptic noise. *Frontiers in computational neuroscience*, 8:116, 2014.

[88] Chaitanya Medini, Bipin Nair, Egidio D'Angelo, Giovanni Naldi, and Shyam Diwakar. Modeling spike-train processing in the cerebellum granular layer and changes in plasticity reveal single neuron effects in neural ensembles. *Intell. Neuroscience*, 2012:7:7–7:7, January 2012. ISSN 1687-5265. doi: 10.1155/2012/359529. URL http://dx.doi.org/10.1155/2012/359529.

[89] A. L. Hodgkin and A. F. Huxley. Propagation of electrical signals along giant nerve fibres. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 140(899):177–183, 1952. ISSN 00804649.

[90] Jornt R De Gruijl, Paolo Bazzigaluppi, Marcel TG de Jeu, and Chris I De Zeeuw. Climbing fiber burst size and olivary sub-threshold oscillations in a network setting. *PLoS Comput Biol*, 8(12):e1002814, 2012.

[91] W. E. Sherwood. Fitzhugh–nagumo model. In D. Jaeger and R. Jung, editors, *Enc. of Comp. Neuroscience*, pages 1–11. Springer, 2014.

[92] Paul E. Phillipson and Peter Schuster. A comparative study of the hodgkin-huxley and fitzhugh-nagumo models of neuron pulse propagation. *I. J. Bifurcation and Chaos*, 15:3851–3866, 12 2005. doi: 10.1142/S0218127405014349.

[93] Harold Lecar. Morris-lecar model. *Scholarpedia*, 2(10):1333, 2007.

[94] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, et al. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23 (3):349–398, 2007. ISSN 1573-6873. doi: 10.1007/s10827-007-0038-6. URL https://doi.org/10.1007/s10827-007-0038-6.

[95] Thomas R Insel, Story C Landis, and Francis S Collins. The nih brain initiative. *Science*, 340(6133):687–688, 2013.

[96] Hideyuki Okano, Erika Sasaki, Tetsuo Yamamori, Atsushi Iriki, Tomomi Shimogori, Yoko Yamaguchi, Kiyoto Kasai, and Atsushi Miyawaki. Brain/minds: a japanese national brain project for marmoset neuroscience. *Neuron*, 92(3):582–590, 2016.

[97] H Sebastian Seung, Daniel D Lee, Ben Y Reis, and David W Tank. Stability of the memory of eye position in a recurrent network of conductance-based model neurons. *Neuron*, 26(1):259–271, 2000.

[98] DD Lee, BY Reis, HS Seung, and DW Tank. Nonlinear network models of the oculomotor integrator. In *Computational neuroscience*, pages 371–377. Springer, 1997.

[99] Takeshi Otsuka, Takafumi Abe, Takahisa Tsukagawa, and Wen-Jie Song. Conductance-based model of the voltage-dependent generation of a plateau potential in subthalamic neurons. *Journal of neurophysiology*, 92(1):255–264, 2004.

[100] Jin Tian, Guoyou Huang, Min Lin, Jinbin Qiu, Baoyong Sha, Tian Jian Lu, and Feng Xu. A mechanoelectrical coupling model of neurons under stretching. *Journal of the Mechanical Behavior of Biomedical Materials*, 2019.

[101] Rajagopal Ananthanarayanan, Steven K Esser, Horst D Simon, and Dharmendra S Modha. The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[102] Michael Hines, Sameer Kumar, and Felix Schürmann. Comparison of neuronal spike exchange methods on a blue gene/p supercomputer. *Frontiers in computational neuroscience*, 5:49, 2011.

[103] Fidjeland, A. K. et al. . Nemo: a platform for neural modelling of spiking neurons using gpus. In *IEEE ASAP*, pages 137–144, 2009.

[104] Bhuiyan, M. et al. Acceleration of spiking neural networks in emerging multi-core and gpu architectures. In *IPDPSW*, 2010.

[105] Choi, J. et al. Implementation of hardware model for spiking neural network. In *ICAI*, page 700, 2015.

[106] Giordana Florimbi, Emanuele Torti, Stefano Masoli, Egidio D'Angelo, Giovanni Danese, and Francesco Leporati. The human brain project: Parallel technologies for biologically accurate simulation of granule cells. *Microprocessors and Microsystems*, 2016.

[107] Pascal Wallisch, Michael E Lusignan, Marc D Benayoun, Tanya I Baker, Adam Seth Dickey, and Nicholas G Hatsopoulos. *MATLAB for neuroscientists: an introduction to scientific computing in MATLAB*. Academic Press, 2014.

[108] M. L. Hines and N. T. Carnevale. The neuron simulation environment. *Neural Computation*, 9(6):1179–1209, 1997. doi: 10.1162/neco.1997.9.6.1179. URL https://doi.org/10.1162/neco.1997.9.6.1179.

[109] William W Lytton, Alexandra H Seidenstein, Salvador Dura-Bernal, Robert A McDougal, Felix Schürmann, and Michael L Hines. Simulation neurotechnologies for advancing brain research: Parallelizing large networks in neuron. *Neural Computation*, 2016.

[110] Marianne J Bezaire, Ivan Raikov, Kelly Burk, Dhrumil Vyas, and Ivan Soltesz. Interneuronal mechanisms of hippocampal theta oscillation in a full-scale model of the rodent ca1 circuit. *eLife*, 2016. doi: 10.7554/eLife.18566.

[111] Michele Migliore, C Cannia, William W Lytton, Henry Markram, and Michael L Hines. Parallel network simulations with neuron. *Journal of computational neuroscience*, 21(2):119, 2006.

[112] James M Bower and David Beeman. *The book of GENESIS: exploring realistic neural models with the GEneral NEural SImulation System*. Springer Science & Business Media, 2012.

[113] James M Bower, David Beeman, and Michael Hucka. The genesis simulation system. 2003.

[114] Plesser, H. E. et al. Nest: the neural simulation tool. *Enc. of Comp. Neuroscience*, pages 1849–1852, 2015.

[115] Susanne Kunkel, Maximilian Schmidt, Jochen M. Eppler, Hans E. Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann, and Moritz Helias. Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics*, 8:78, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00078. URL https://www.frontiersin.org/article/10.3389/fninf.2014.00078.

[116] Dan FM Goodman and Romain Brette. The brian simulator. *Frontiers in neuroscience*, 3:26, 2009.

[117] Beyeler, M. et al. Carlsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks. In *IJCNN*, pages 1–8, 2015.

[118] Jeffrey L Krichmar. A biologically inspired action selection algorithm based on principles of neuromodulation. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012.

[119] Schemmel, J. et al. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *IEEE ISCAS*, May 2010.

[120] Wu, Q. et al. Development of fpga toolbox for implementation of spiking neural networks. In *CSNT*, pages 806–810, 2015.

[121] George Chatzikonstantis, Dimitrios Rodopoulos, Sofia Nomikou, Christos Strydis, Chris I. De Zeeuw, and Dimitrios Soudris. First Impressions from Detailed Brain Model Simulations on a Xeon/Xeon-Phi Node. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 361–364, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2903477. URL http://doi.acm.org/10.1145/2903150.2903477.

[122] H.A. Du Nguyen, Zaid Al-Ars, Georgios Smaragdos, and Christos Strydis. Accelerating complex brain-model simulations on GPU platforms . In *Design, Automation, and Test in Europe, DATE 2015*, March 2015.

[123] Georgios Smaragdos, Sebastian Isaza, Martijn Van Eijk, Ioannis Sourdis, and Christos Strydis. FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons. In *22nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2014.

[124] B. Glackin, J. A. Wall, T. M. McGinnity, L. P. Maguire, and L. McDaid. A spiking neural network model of the medial superior olive using spike timing dependent plasticity for sound localization. *Frontiers on Comput. Neurosci.*, 4(18), 2010.

[125] 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, 2013. ISSN 0893-6080. doi: 10.1016/j.neunet.2013.01.019. URL http://www.sciencedirect.com/science/article/pii/S0893608013000348. Computation in the Cerebellum.

[126] S.H. Fuller and L.I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011. ISSN 0018-9162.

[127] Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2): 153–160, 2006.

[128] Johanna Senk, Corto Carde, Espen Hagen, Torsten W Kuhlen, Markus Diesmann, and Benjamin Weyers. Viola-a multi-purpose and web-based visualization tool for neuronal-network simulation output. *arXiv preprint arXiv:1803.10205*, 2018.

[129] 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, et al. Neuroml: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS computational biology*, 6(6):e1000815, 2010.

[130] Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. Lems: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning neuroml 2. *Frontiers in Neuroinformatics*, 8:79, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00079. URL https://www.frontiersin.org/article/10.3389/fninf.2014.00079.

[131] Paul Richmond, Alex Cope, Kevin Gurney, and David J Allerton. From model specification to simulation of biologically constrained networks of spiking neurons. *Neuroinformatics*, 12(2):307–323, 2014.

[132] Thomas Sharp, Francesco Galluppi, Alexander Rast, and Steve Furber. Power-efficient simulation of detailed cortical microcircuits on spinnaker. *Journal of neuroscience methods*, 210(1):110–118, 2012.

[133] Nathan M Jordan, Kimberly B Perry, Nitish Narala, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. Design and implementation of an ncs-neuroml translator. In *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE 2012), Los Angeles, CA*, pages 13–19, 2012.

[134] Tim Busbice, Padraig Gleeson, Sergey Khayrulin, Matteo Cantarelli, Alexander Dibert, Giovanni Idili, Andrey Palyanov, and Stephen Larson. The neuroml c. elegans connectome. *Neuroinf. Abstract book*, pages 82–83, 2012.

[135] Padraig Gleeson, Sharon Crook, Volker Steuber, and R. Angus Silver. Using NeuroML and neuroConstruct to build neuronal network models for multiple simulators. *BMC Neuroscience*, 8(2):P1, Jul 2007. ISSN 1471-2202. doi: 10.1186/1471-2202-8-S2-P1. URL https://doi.org/10.1186/1471-2202-8-S2-P1.

[136] Adam Tomkins, C Luna Ortiz, Daniel Coca, and Paul Richmond. From gui to gpu: a toolchain for gpu code generation for large scale drosophila simulations using spineml. In *Front. Neuroinform. doi: 10.3389/conf. fninf*, volume 49, 2016.

[137] Athul Sripad, Giovanny Sanchez, Mireya Zapata, Vito Pirrone, Taho Dorta, Salvatore Cambria, Albert Marti, Karthikeyan Krishnamourthy, and Jordi Madrenas.

Snava—a real-time multi-fpga multi-model spiking neural network simulation architecture. *Neural Networks*, 97:28–45, 2018.

[138] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7:19, 2013.

[139] Nguyen, H. A. Du et al. Accelerating complex brain-model simulations on gpu platforms. In *DATE*, pages 974–979, 2015.

[140] George Chatzikonstantis, Dimitrios Rodopoulos, Christos Strydis, Chris I De Zeeuw, and Dimitris Soudris. Optimizing extended hodgkin-huxley neuron model simulations for a xeon/xeon phi node. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[141] Giorgos Chatzikonstantis, H Sidiropoulos, Christos Strydis, Mario Negrello, Georgios Smaragdos, CI De Zeeuw, and DJ Soudris. Multinode implementation of an extended hodgkin–huxley simulator. *Neurocomputing*, 329:370–383, 2019.

[142] Thomas A Woolsey, Joseph Hanaway, and Mokhtar H Gado. *The brain atlas: a visual guide to the human central nervous system.* John Wiley & Sons, 2017.

[143] Joseph L Gatlin, Robert Wineman, Bruce Schlakman, Razvan Buciuc, and Majid Khan. Hypertrophic olivary degeneration after resection of a pontine cavernous malformation: a case report. *Journal of radiology case reports*, 5(3):24, 2011.

[144] TA Martin, JG Keating, HP Goodkin, AJ Bastian, and WT Thach. Throwing while looking through prisms: I. focal olivocerebellar lesions impair adaptation. *Brain*, 119(4):1183–1198, 1996.

[145] Chris I. De Zeeuw et al. Spatiotemporal firing patterns in the cerebellum. *Nature Review Neuroscience*, 12(6):327–344, 2011.

[146] Nicolas Schweighofer, Eric J Lang, and Mitsuo Kawato. Role of the olivo-cerebellar complex in motor learning and control. *Frontiers in neural circuits*, 7:94, 2013.

[147] Tao Liu, Duo Xu, James Ashe, and Khalaf O Bushara. The specificity of inferior olive response to stimulus timing. *Journal of neurophysiology*, 2008.

[148] Tokiji Hanihara, Naoji Amano, Tatsuya Takahashi, Yoji Itoh, and Saburo Yagishita. Hypertrophy of the inferior olivary nucleus in patients with progressive supranuclear palsy. *European neurology*, 39(2):97, 1998.

[149] PS Bindu, AB Taly, K Sonam, C Govindaraju, HR Arvinda, N Gayathri, MM Srinivas Bharath, D Ranjith, M Nagappa, S Sinha, et al. Bilateral hypertrophic olivary nucleus degeneration on magnetic resonance imaging in children with leigh and leigh-like syndrome. *The British journal of radiology*, 87(1034): 20130478, 2014.

[150] Arnulf H Koeppen. The pathogenesis of spinocerebellar ataxia. *The Cerebellum*, 4(1):62, 2005.

[151] E Hairer, SP Norsett, and G Wanner. Solving ordinary differential equations i: Nonsti problems, volume second revised edition, 1993.

[152] Howard, J. et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 46(1):173–183, 2011. ISSN 0018-9200. doi: 10.1109/JSSC.2010.2079450.

[153] Mattson, T.G. et al. The 48-core scc processor: the programmer's view. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –11, nov. 2010.

[154] Wu chun Feng, Xizhou Feng, and Rong Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17–23, 2008.

[155] G.M. Shepherd, E. Rolls, A. G. Andreou, and M. Peitsch. Evaluation of the Blue Brain Project and Human Brain Project. Technical report, EPFL, Lausanne, March 28-31 2011.

[156] Lizhe Wang, Samee U. Khan, Dan Chen, Joanna Kolodziej, Rajiv Ranjan, Cheng zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems*, 29(7):1661 – 1670, 2013.

[157] The scc programmer's guide - revision 0.75. Technical report, Intel Corporation, October 2010.

[158] Using the rcce power management calls – revision 1.1. Technical report, Intel Corporation, September 2011.

[159] Ulya R. Karpuzcu, Brian Greskamp, and Josep Torrellas. The bubblewrap many-core: popping cores for sequential acceleration. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 447–458, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669169.

[160] Vilfredo Pareto. *Manual of Political Economy.* Augustus M. Kelley Publishers, 1971.

[161] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, October 1975. ISSN 0004-5411. doi: 10.1145/321906.321910.

[162] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838, May 2008. doi: 10.1109/ISBI.2008.4541126.

[163] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[164] Stephen Blair-Chappell and Andrew Stokes. *Parallel programming with intel parallel studio XE*. John Wiley & Sons, 2012.

[165] http://mpi-forum.org/.

[166] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[167] https://software.intel.com/en-us/node/471922.

[168] Renee Dambiermont Monagle and Harold Brody. The effects of age upon the main nucleus of the inferior olive in the human. *Journal of Comparative Neurology*, 155 (1):61–66, 1974.

[169] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.

[170] Srinivas K Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE micro*, 20(4):47–57, 2000.

[171] Ataru Tanikawa, Kohji Yoshikawa, Takashi Okamoto, and Keigo Nitadori. N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions. *New Astronomy*, 2012.

[172] Simon J Pennycook, Chris J Hughes, Mikhail Smelyanskiy, and Stephen A Jarvis. Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors. In *International Symposium on Parallel & Distributed Processing*, pages 1085–1097, 2013.

[173] M Deilmann. A guide to vectorization with intel c++ compilers. *Intel Corporation, April*, 2012.

[174] Michaela Barth, KTH Sweden, Mikko Byckling, CSC Finland, Nevena Ilieva, NCSA Bulgaria, Sami Saarinen, Michael Schliephake, Volker Weinberg, and LRZ Germany. Best practice guide intel xeon phi v1. *LRZ Germany March*, 31, 2013.

[175] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(4-5), 2002.

[176] Lazaros Papadopoulos, Christos Baloukas, and Dimitrios Soudris. Exploration methodology of dynamic data structures in multimedia and network applications for embedded platforms. *Journal of Systems Architecture*, 54(11), 2008.

[177] Tassadaq Hussain, Miquel Pericas, and Eduard Ayguadé. Reconfigurable memory controller with programmable pattern support. *HiPEAC WRC*, 67:95, 2011.

[178] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2): 34–46, 2016.

[179] Avinash Sodani. Knights landing (knl): 2nd generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[180] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge university press Cambridge, 1996.

[181] Gerald J Kaufman et al. System and method for application programming interface for extended intelligent platform management, June 21 2011. US Patent 7,966,389.

[182] Bernhard Hellwig. A quantitative analysis of the local connectivity between pyramidal neurons in layers 2/3 of the rat visual cortex. *Biological cybernetics*, 82(2): 111–121, 2000.

[183] Verena Senn, Steffen BE Wolff, Cyril Herry, François Grenier, Ingrid Ehrlich, Jan Gründemann, Jonathan P Fadok, Christian Müller, Johannes J Letzkus, and Andreas Lüthi. Long-range connectivity defines behavioral specificity of amygdala neurons. *Neuron*, 81(2):428–437, 2014.

[184] Keith R Jackson, Krishna Muriki, Lavanya Ramakrishnan, Karl J Runge, and Rollin C Thomas. Performance and cost analysis of the supernova factory on the amazon aws cloud. *Scientific Programming*, 19(2-3):107–119, 2011.

[185] Yuanzheng Shao, Liping Di, Yuqi Bai, Bingxuan Guo, and Jianya Gong. Geoprocessing on the amazon cloud computing platform—aws. In *Agro-Geoinformatics (Agro-Geoinformatics), 2012 First International Conference on*, pages 1–6. IEEE, 2012.

[186] Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, and Rosa M Badia. Executing linear algebra kernels in heterogeneous distributed infrastructures with pycompss. *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles*, 73:47, 2018.

[187] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE, 2011.

[188] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey.

Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[189] Kier Storey and Fengyun Lu. Hybrid, scalable cpu/gpu rigid body pipeline, January 3 2019. US Patent App. 15/636,973.

[190] Christopher Thiele, Mauricio Araya-Polo, Faruk Omer Alpak, Beatrice Riviere, et al. Distributed parallel hybrid cpu-gpgpu implementation of the phase-field method for accelerated high-accuracy simulations of pore-scale two-phase flow. In *SPE Reservoir Simulation Conference*. Society of Petroleum Engineers, 2019.

[191] Peng Liu, Shunbin Li, and Qingyuan Ding. An energy-efficient accelerator based on hybrid cpu-fpga devices for password recovery. *IEEE Transactions on Computers*, 68(2):170–181, 2019.

[192] Sokjoon Lee, Hwajeong Seo, Hyeokchan Kwon, and Hyunsoo Yoon. Hybrid approach of parallel implementation on cpu–gpu for high-speed ecdsa verification. *The Journal of Supercomputing*, Jan 2019. ISSN 1573-0484. doi: 10.1007/s11227-019-02744-6. URL https://doi.org/10.1007/s11227-019-02744-6.

[193] Xu Liu, Hibat Allah Ounifi, Abdelouahed Gherbi, Yves Lemieux, and Wubin Li. A hybrid gpu-fpga-based computing platform for machine learning. *Procedia Computer Science*, 141:104–111, 2018.

[194] Henry Markram et al. Reconstruction and Simulation of Neocortical Microcircuitry. *Cell*, 163(2):456–492, 2015. ISSN 0092-8674. doi: 10.1016/j.cell.2015.09.029. URL http://www.sciencedirect.com/science/article/pii/S0092867415011915.

[195] Christoforos Kachris, Dimitrios Soudris, Georgi Gaydadjiev, Huy-Nam Nguyen, Dimitrios S Nikolopoulos, Angelos Bilas, Neil Morgan, Christos Strydis, Christos Tsalidis, John Balafas, et al. The vineyard approach: Versatile, integrated, accelerator-based, heterogeneous data centres. In *International Symposium on Applied Reconfigurable Computing*, pages 3–13. Springer, 2016.

[196] Stelios Mavridis, Manolis Pavlidakis, Ioannis Stamoulias, Christos Kozanitis, Nikolaos Chrysos, Christoforos Kachris, Dimitrios Soudris, and Angelos Bilas. Vinetalk: Simplifying software access and sharing of fpgas in datacenters. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[197] Ioannis Stamelos, Elias Koromilas, Christoforos Kachris, and Dimitrios Soudris. A novel framework for the seamless integration of fpga accelerators with big data analytics frameworks in heterogeneous data centers. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 539–545. IEEE, 2018.

[198] Karim Djemame, Django Armstrong, Richard Kavanagh, Jean-Christophe Deprez, Ana Juan Ferrer, David Garcia Perez, Rosa Badia, Raul Sirvent, Jorge Ejarque, and Yiannis Georgiou. Tango: Transparent heterogeneous hardware architecture deployment for energy gain in operation. *arXiv preprint arXiv:1603.01407*, 2016.

[199] Alberto Scionti, Pietro Ruiu, Olivier Terzo, Joel Nider, Craig Petrie, and Niccolo Baldoni. Opera: A low power approach to the next generation cloud infrastructures. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 326–333. IEEE, 2016.

[200] Alberto Scionti, Klodiana Goga, F Lubrano, and Olivier Terzo. Towards energy efficient orchestration of cloud computing infrastructure. In *Conference on Complex, Intelligent, and Software Intensive Systems*, pages 172–183. Springer, 2018.

[201] Dimitrios Soudris, Lazaros Papadopoulos, Christoph W Kessler, Dionysios D Kehagias, Athanasios Papadopoulos, Panos Seferlis, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Samuel Thibault, Raymond Namyst, et al. Exa2pro programming environment: architecture and applications. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 202–209. ACM, 2018.

[202] Tomas Öhberg, August Ernstsson, and Christoph Kessler. Hybrid cpu–gpu execution support in the skeleton programming framework skepu. *The Journal of Supercomputing*, pages 1–19, 2018.

[203] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[204] Stephane H Maes, Rajeev Bharadhwaj, Travis S Tripp, Kevin Lee Wilson, Petr Fiedler, and John M Green. Cloud application deployment, March 20 2018. US Patent 9,923,952.

[205] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3): 24–31, 2015.

[206] Jinesh Varia. Best practices in architecting cloud applications in the aws cloud. In *Cloud Computing: Principles and Paradigms*, volume 18, pages 459–490. Wiley Online Library, 2011.

[207] Werner Van Geit, Michael Gevaert, Giuseppe Chindemi, Christian Rössert, Jean-Denis Courcol, Eilif B Muller, Felix Schürmann, Idan Segev, and Henry Markram. Bluepyopt: leveraging open source software and cloud infrastructure to optimise model parameters in neuroscience. *Frontiers in neuroinformatics*, 10:17, 2016.

[208] Paul Watson, Hugo Hiden, and Simon Woodman. e-science central for carmen: science as a service. *Concurrency and computation: Practice and Experience*, 22 (17):2369–2380, 2010.

[209] Mufti Mahmud, M Shamim Kaiser, M Mostafizur Rahman, M Arifur Rahman, Antesar Shabut, Shamim Al-Mamun, and Amir Hussain. A brain-inspired trust management model to assure security in a cloud based iot framework for neuroscience applications. *Cognitive Computation*, 10(5):864–873, 2018.

[210] Microprocessors and Digital Systems Laboratory. URL http://www.microlab.ntua.gr/?q=home.

[211] Brainframe @ Erasmus Brain Project. URL https://www.erasmusbrainproject.com/index.php/en/themes/brainframe.

[212] Oliver Pell, Oskar Mencer, Kuen Hung Tsoi, and Wayne Luk. *Maximum Performance Computing with Dataflow Engines*, pages 747–774. Springer New York, New York, NY, 2013. ISBN 978-1-4614-1791-0. doi: 10.1007/978-1-4614-1791-0_25. URL http://dx.doi.org/10.1007/978-1-4614-1791-0_25.

[213] NVidia Corporation. www.geforce.com.

[214] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[215] Antonis Manousis, Roy Ragsdale, Ben Draffin, Adwiteeya Agrawal, and Vyas Sekar. Shedding light on the adoption of let's encrypt. *arXiv preprint arXiv:1611.00469*, 2016.

[216] Georgios Smaragdos, Georgios Chatzikostantis, Sofia Nomikou, Dimitrios Rodopoulos, Ioannis Sourdis, Dimitrios Soudris, Chris I De Zeeuw, and Christos 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. IEEE, 2016.

[217] Jornt R. De Gruijl, Bazzigaluppi Paolo, G de Jeu Marcel T., and De Zeeuw Chris I. Climbing Fiber Burst Size and Olivary Sub-threshold Oscillations in a Network Setting. *PLoS Comput Biol*, 8(12), 12 2012.

[218] George Chatzikonstantis, Dimitrios Rodopoulos, Sofia Nomikou, Christos Strydis, Chris I. De Zeeuw, and Dimitrios Soudris. First Impressions from Detailed Brain Model Simulations on a Xeon/Xeon-Phi Node. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 361–364, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2903477. URL http://doi.acm.org/10.1145/2903150.2903477.

[219] Giacomo Indiveri and Timothy K Horiuchi. Frontiers in neuromorphic engineering. *Frontiers in neuroscience*, 5:118, 2011.

[220] Hanna Keren, Johannes Partzsch, Shimon Marom, and Christian G Mayr. A biohybrid setup for coupling biological and neuromorphic neural networks. *Frontiers in Neuroscience*, 13:432, 2019.

[221] 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(8):1943–1953, 2013.

[222] Indranil Chakraborty, Gobinda Saha, and Kaushik Roy. Photonic in-memory computing primitive for spiking neural networks using phase-change materials. *Physical Review Applied*, 11(1):014063, 2019.

[223] Paul Alcorn. AMD Ryzen 3000 Series CPUs: Rumors, Release Date, All We Know About Ryzen 3, February 2019. URL https://www.tomshardware.com/news/amd-ryzen-3000-everything-we-know,38233.html.

[224] Patrick Moorhead. AMD Surprises At CES 2019, January 2019. URL https://www.forbes.com/sites/patrickmoorhead/2019/01/15/amd-surprises-with-radeon-vii-at-ces-2019/#540d734c5816.

Appendix A

Appendix A. *Publications*

In this section a list of publications is presented that support the research work in this Thesis, categorized by type.

A.1 List of International Journal Publications

[J1] Chatzikonstantis, G., Sidiropoulos, H., Strydis, C., Negrello, M., Smaragdos, G., De Zeeuw, C. I., & Soudris, D. J. (2019). Multinode implementation of an extended Hodgkin–Huxley simulator. Neurocomputing, 329, 370-383.

[J2] Smaragdos, G., Chatzikonstantis, G., Kukreja, R., Sidiropoulos, H., Rodopoulos, D., Sourdis, I., Al-Ars, Z., Kachris, C., Soudris, D., De Zeeuw, C. I., & Strydis, C. (2017). BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations. Journal of neural engineering, 14(6), 066008.

[J3] Chatzikonstantis, G., Rodopoulos, D., Strydis, C., De Zeeuw, C. I., & Soudris, D. (2017). Optimizing extended hodgkin-huxley neuron model simulations for a xeon/xeon phi node. IEEE Transactions on Parallel and Distributed Systems, 28(9), 2581-2594.

A.2 List of International Conference Publications

[C1] Neofytou, A., Chatzikonstantis, G., Magkanaris, I., Smaragdos, G., Strydis, C., & Soudris, D. (2019, October). GPU Implementation of Adaptive Exponential Network Neuromodelling. In 2019 IEEE International Conference on BioInformatics and BioEngineering (BIBE). IEEE.

[C2] Sidiropoulos, H., Chatzikonstantis, G., Soudris, D., & Strydis, C. (2018, October). The VINEYARD framework for heterogeneous cloud applications: The BrainFrame case. In 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP) (pp. 70-75). IEEE.

[C3] Chatzikonstantis, G., Jiménez, D., Meneses, E., Strydis, C., Sidiropoulos, H., & Soudris, D. (2017, June). From Knights Corner to Landing: A Case Study Based on a Hodgkin-Huxley Neuron Simulator. In International Conference on High Performance Computing (pp. 363-375). Springer, Cham.

[C4] Smaragdos, G., Chatzikostantis, G., Nomikou, S., Rodopoulos, D., Sourdis, I., Soudris, D., De Zeeuw, C. I., & Strydis, C. (2016, April). Performance analysis of accelerated biophysically-meaningful neuron simulations. In 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 1-11). IEEE.

[C5] Chatzikonstantis, G., Rodopoulos, D., Nomikou, S., Strydis, C., De Zeeuw, C. I., & Soudris, D. (2016, May). First impressions from detailed brain model simulations on a Xeon/Xeon-Phi node. In Proceedings of the ACM International Conference on Computing Frontiers (pp. 361-364). ACM.

[C6] Rodopoulos, D., Chatzikonstantis, G., Pantelopoulos, A., Soudris, D., De Zeeuw, C. I., & Strydis, C. (2014, July). Optimal mapping of inferior olive neuron simulations on the single-chip cloud computer. In 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV) (pp. 367-374). IEEE.

[C7] Georgis, G., Reisis, D., Skordilakis, P., Tsakalis, K., Shafique, A. B., Chatzikonstantis, G., & Lentaris, G. (2014, July). Neuronal connectivity assessment for epileptic seizure prevention: Parallelizing the generalized partial directed coherence on many-core platforms. In 2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV) (pp. 359-366). IEEE.