Computational characteristics and hardware implications of brain tissue simulations

Thèse N° 9767

Présentée le 5 novembre 2019

à la Faculté des sciences de la vie
Groupe Schürmann
Programme doctoral en neurosciences

pour l'obtention du grade de Docteur ès Sciences

par

Francesco CREMONESI

Acceptée sur proposition du jury

Prof. F. C. Hummel, président du jury
Prof. F. Schürmann, directeur de thèse
Prof. U. Bhalla, rapporteur
Prof. M. Diesmann, rapporteur
Prof. B. Falsafi, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

I wish I did not think about
it so much, he thought.

— Ernest Hemingway, For Whom the Bell Tolls

To Ale, Leo, Mati,
and the rest of the next generation
yet to come.

Acknowledgements

In the last five years I have grown professionally and personally more than I could ever anticipate. For this, I owe thanks to countless people around me.

Felix Schürmann, you have guided my efforts and managed to remind me of the big picture every time I got lost in my own overthinking. Thank you for your scientific inputs and for your timely words of encouragement. Georg Hager, you've not only taught me everything I know about microprocessor architecture, you've succeeded in showing me the fun and challenging side of it. Michael Hines, thank you for teaching me all the subtleties in the methods for simulating neurons. Your patience and explanations have helped me navigate the world of neuron modelling, and your work in the NEURON simulator is an inspiration to me. Fabien Delalondre, even though you missed out on the last part of this wild ride, you laid its foundations. Without your confidence in me I wouldn't be here today.

My work has also significantly benefited from the practical help of several people. Pramod, thank you for your hard work in making the software tools usable, and for the uncountable discussions about computer performance. Jim, thank you for the whole simulation toolchain that you built and manage, and for teaching me about ancient scripting languages such as hoc. Judit, you've taught me the value of being organised and prepared and you've provided scientific help at moments when I really needed it. Omar, thank you for all the discussion about performance and for your help in NMODL development. Karin, you did your best in turning my stream of consciousness into acceptable scientific writing. Georg, Gerhard, Julian and the rest of the people at RRZE, thank you for your support in understanding the ECM model and your development efforts in Kerncraft. Dries and the rest of the core-services team, thank you for maintaining a running system, exploring bios options and fixing prefetcher configuration on the fly. Dace, Alina, Raquel, Julien, Hashini, Aleksandra and the rest of the administrative services team, I cannot thank you enough for your help in organising various aspects of my work and defence especially.

The one giant stroke of luck that defines my experience at Blue Brain is the unexpectedly high correlation of talent and niceness in the people around me. Taylor, we built our friendship revising cellular neuroscience facts together and developed it side by side in the trenches. You've shown me how to be a good scientist and how to make proper figures, but most of all you've taught me to be a skeptic and an intellectual. Francesco, when I had to interview you I immediately knew that your technical skills were a span above my own, but I never imagined you would be so effective at sharing your knowledge with me. Thank you for teaching me that when you want to understand something, you study night and day until you get it. Bruno,

we've been through this together and we supported each other when things got tough. You've taught me that a bit of asynchrony and chaos can lead to great improvements. Antonio, you've liberated a part of me that was locked in my childhood memories, and taught me that I can embrace it and still be cool. Tim, you've been my first friend in Switzerland and an unstoppable publishing machine. Giuseppe, your ironic view of PhD life got me through several sleepless nights. Nando, without your python lessons I'd still be stuck debugging triple-nested for loops in my analysis scripts. Berat, thank you for our discussions between jumps and hoops. Sam, thank you for reminding me that math is beautiful.

I've also made friends with some truly amazing people outside the office during these five years. Isa and Till, thank you for all the energy and happiness you've given me to make the days feel lighter. You managed to give me the support and relief I needed, even when I didn't realise I wanted it. #GoPhd! Mari and Fra, you've had the unique ability of providing me with technical discussions and comfort food simultaneously. Thank you for taking care of me when I needed it. Pierre and Nicolas, you've helped me maintain sanity by showing me another world made of notes and counterpoint. Dillenn, Nils, Josephine, John, Nico and the rest of the basketball team, thank you for making me feel welcome in a new city.

Finally, my thoughts go to people from my previous life before BBP. Mom and dad, thank you for teaching me to follow my passions, to use my imagination, to always do my best and to remain humble. To the rest of my family, thank you for your unwavering support. Ludo, you're the brother that I grew up with. Ale, your ambition and freedom in life are an inspiration to me. Also, *sono figo!* Nico and Jacopo, we've been through thick and thin, yet you still haven't stopped listening to my endless stories about my research, thank you. Miro, you are my oldest friend, and you've known the best and worst of me, thank you for always being there.

Silvia, there is no space here to describe the details of why and how much I'm grateful to you. I feel it with a clarity that I hadn't known before. Your determined, lovely smile is what inspired me in approaching this research every day. Let us go and build the next step together.

Genève, 3 Septembre 2019 F. C.

Abstract

Understanding the link between the brain's anatomy and its function through computer simulations of neural tissue models is a widely used approach in computational neuroscience. This technique enables rapid prototyping and testing of hypotheses, allowing researchers to bridge the scales of biological phenomena. Until recently, the constant trend of improvement in computational power has supported an exponential growth in the scale and level of detail of *in silico* experiments. However, a systematic characterization of the performance landscape has not yet been carried out.

In this work we intend to capture intrinsic computational properties of the existing modelling abstractions and answer questions about the intricate relationship between simulation algorithms and modern hardware architecture. Our first contribution is a novel set of hardware-agnostic metrics that enables us to bring focus to the heterogeneous landscape of brain tissue models. We develop a methodology able to capture subtle differences between cell-based models and quantify their impact on performance based on hardware features. We show that lumping simulation experiments together by referring to numbers of neurons and synapses without further detail hides fundamental differences in computational and hardware requirements across models. In addition to analysing different neuron representations, we investigate the impact of biological heterogeneity on the performance of a cortical microcircuit model.

Our analysis indicates that while general-purpose computers have until now sustained high-performance simulations of all brain tissue models, the next generation of *in silico* models will require hardware tailored to the underlying abstraction. We find that all formalisms saturate the memory bandwidth with a fairly small number of shared memory threads, but the reasons behind this are quite different: conductance-based models are dominated by the large memory traffic of clock-driven kernels, while current-based models are most affected by event-driven execution and memory latency. In distributed simulations the latency of the interconnect fabric is the root cause for a significant degradation in performance.

We argue that performance analyses such as ours are required to enable the next generation of brain tissue simulations – or else scientific progress risks being hindered by the presence of severe hardware bottlenecks. Our methodology provides a common tool to facilitate the communication between modellers, developers and hardware designers in order to sustain the larger memory and performance requirements of future brain tissue simulations.

Keywords Computational neuroscience, High-performance computing, Performance modelling, Blue Brain Project, Hardware modelling, Neuronal modelling, *in silico* neuroscience

Sommario

Comprendere il collegamento tra l'anatomia del cervello e il suo funzionamento attraverso l'utilizzo di simulazioni al computer è ormai una tecnica ampiamente utilizzata nell'ambito delle neuroscience computazionali. Questo metodo permette di formulare e testare rapidamente ipotesi scientifiche, dando ai ricercatori la possibilità di formare un collegamento tra le diverse scale fisiche del fenomeno biologico. Il trend costante di crescita delle capacità computazionali dei computer ha fino ad ora supportato una crescita esponenziale nel livello di dettaglio degli esperimenti *in silico*. Tuttavia una caratterizzazione sistematica del panorama delle prestazioni di queste simulazioni non è ancora stata condotta.

In questa tesi ci proponiamo di identificare le proprietà computazionali intrinseche ai modelli di neuroni e rispondere a domande sulla relazione tra algoritmi di simulazione e architettura dell'hardware. Il nostro primo contributo è un nuovo insieme di metriche agnostiche dell'hardware per analizzare il panorama eterogeneo di modelli del tessuto cerebrale. Sviluppiamo poi una metodologia capace di catturare sottili differenze tra modelli a livello cellulare e quantificare il loro impatto sulla prestazione a partire da proprietà dell'hardware. Dimostriamo che raggruppare esperimenti di simulazione riferendosi solamente al numero di neuroni e sinapsi senza fornire ulteriori dettagli nasconde differenze fondamentali tra i requisiti computazionali di modelli differenti. In aggiunta all'analisi di diversi formalismi investighiamo l'impatto della variabilità biologica sulle prestazioni di un modello di microcircuito corticale.

La nostra analisi indica che, mentre i computer d'uso generale hanno potuto sostenere fino ad ora simulazioni ad alta prestazione di tutti i modelli di tessuto cerebrale, la prossima generazione di modelli *in silico* richiederà hardware concepito su misura. Dimostriamo che tutti i formalismi per rappresentare un neurone saturano la larghezza di banda della memoria utilizzando un numero di thread a memoria condivisa relativamente basso, ma le ragioni alla base di ciò sono diverse: i modelli a base di conduttanza sono dominati da un grande traffico di dati, mentre nei modelli a base di corrente i fattori chiave di prestazione sono la latenza della memoria e l'esecuzione event-driven. Nelle simulazioni distribuite la latenza della rete di connessione è la causa principale di una perdita di prestazioni.

Sosteniamo che analisi delle prestazioni come la nostra siano necessarie per consentire lo sviluppo della prossima generazione di simulazioni del tessuto cerebrale, altrimenti il progresso scientifico rischia di essere impedito da gravi limitazioni dell'hardware. La nostra metodologia fornisce uno strumento comune per facilitare la comunicazione tra modellatori, sviluppatori e disegnatori di hardware per poter sostenere i requisiti maggiori di memoria e prestazioni delle simulazioni future di tessuto cerebrale.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Performance in brain tissue simulations

Cellular-level digital reconstructions and simulations of brain tissue electrophysiology have become a widespread tool for neuroscientific discovery. Even though *in silico* experiments cannot fully replace traditional *in vitro* or *in vivo* ones, it has been argued that they represent an indispensable contribution to the future of neuroscience (Einevoll et al., 2019; Fan and Markram, 2019). However, the memory footprint and performance requirements for simulating a brain represents an unprecedented challenge in the computational sciences. For example, it has been projected that simulating a human-scale whole-cortex model could require an exascale computer (Markram, 2012), while real-time simulations of such a system could even be impossible with the current hardware solutions (Zenke and Gerstner, 2014). Even a volume of cortical tissue as small as a cubic millimetre can contain several tens of thousands of neurons and glia cells, orders of magnitude more synaptic connections and several orders of magnitude more of proteins, neurotransmitters and signalling molecules. Therefore it is perhaps not surprising that computer performance has been one of the major limiting factors in the computational modelling and exploration of biological neural networks (Einevoll et al., 2019; Kotaleski and Blackwell, 2010; Markram, 2012). To overcome this challenge computational neuroscientists have routinely employed high-performance computing (HPC) techniques (Helias et al., 2012; Markram et al., 2015). These optimised approaches have enabled researchers to probe the electrochemical properties of the brain at an unprecedented scale (see e.g. Ananthanarayanan et al., 2009; Izhikevich and Edelman, 2008; Jordan et al., 2018; Nolte et al., 2018; Reimann et al., 2013). Several large-scale brain tissue models leveraging HPC hardware have been published (see Chapter 2 for a review), but achieving efficient, high-performance simulations still remains an open problem.

Much of the early increase in computational requirements of models and simulations have been supported by Dennard scaling (Dennard et al., 1974) and Moore's law (Moore, 1995), but with chip-manufacturing technology reaching its limit and the consequent rise of multi-core and heterogeneous architectures (Hardavellas et al., 2011; Simonite, 2016), computational

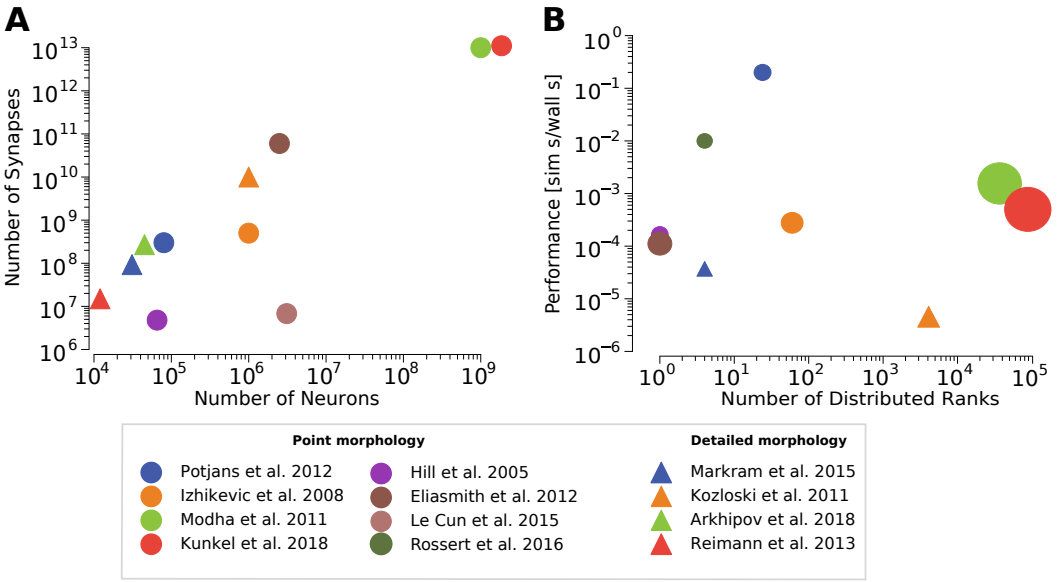

Figure 1.1 – State of the art cellular-level simulations and performance. Supercomputers and HPC architectures are begin used to conduct *in silico* experiments based on mathematical models of neurons. **A**: examples of state-of-the-art brain tissue simulations based on different underlying neuronal abstractions. Researchers typically report the number of neurons and synapses in their simulations as a proxy for complexity of the problem. **B**: cluster size and reported performance of published brain tissue simulations. Cluster size is measured in number of parallel distributed ranks. Performance is measured in simulated seconds per wallclock second to simulate the whole network. The marker size is proportional to the number of neurons in the simulated network.

neuroscientists have been forced to develop more efficient algorithms and software to be able to keep up with the increasing demands of modellers. Research efforts in the context of simulation neuroscience have investigated the efficient utilization of modern multicore processors (see e.g. Brette and Goodman, 2011; Eichner et al., 2009; Kumbhar et al., 2016, 2018), parallel computing (see e.g. Helias et al., 2012; Morrison et al., 2005; Ovcharenko et al., 2015), accelerators (see e.g. Brette and Goodman, 2012; Fidjeland et al., 2009; Knight and Nowotny, 2018) and brain-inspired hardware (see e.g. Benjamin et al., 2014; Indiveri et al., 2011; Painkras et al., 2013).

Typically researchers report the number of neurons and synapses as a proxy for the complexity of their model and compare it against the performance and amount of parallelism in the simulation experiment as shown in Figure 1.1. However this does not paint the whole picture. The wide variety of modelling scales and abstractions has led to a heterogeneous landscape of brain tissue models, making it difficult to compare brain tissue simulations. Moreover, the differences in hardware characteristics can lead to different interpretations of performance results. Finally, the underlying mechanisms and reasons supporting a certain performance are often overlooked or left unexplored.

It becomes clear that a detailed understanding of the performance properties of brain tissue simulations and models represents a fundamental milestone in achieving a better comprehension of the brain's electrophysiology and function. Some performance studies have been conducted, but the large variability between modelling abstractions that is observed in the neuroscientific literature is matched by an equally large variability in the simulation performance. Researchers have investigated memory efficiency (Knight and Nowotny, 2018; Kunkel et al., 2014), automatic generation of efficient kernel code (Kumbhar et al., 2019a; Yavuz et al., 2016), vectorisation (Brette and Goodman, 2011), efficient communication of spikes (see e.g. Ananthanarayanan and Modha, 2007; Hines et al., 2011; Navaridas et al., 2012), splitting of complex neurons (Hines et al., 2008; Kozloski and Wagner, 2011), asynchronous execution (Magalhães and Schürmann, 2019), and other optimisations. Simple performance models have also been proposed for distributed point neuron networks (Peyser and Schenck, 2015; Schenck et al., 2014) or for relevant sub portions of the simulation algorithm (Ewart et al., 2015).

Such work demonstrates the growing interest of the community in simulation performance, but suffers from a few important drawbacks: it is hardware platform and use case specific, and requires new benchmarks for every change in model or simulation configuration; it does not immediately explain differences in performance across *in silico* models and experiments; it provides very limited insight into which hardware features are the most relevant for simulation performance; it does not generalize to new architectures and does not allow exploration of new hardware solutions.

We have highlighted significant differences in the performance profiles of *in silico* models falling within the same category of cellular level representations, but such differences have not yet been thoroughly analysed and discussed. We believe that the situation calls for a better, deeper understanding of how hardware capabilities intersect with brain tissue simulation algorithms to determine their timely and efficient execution. This would empower computational neuroscientists to design more efficient *in silico* experiments and answer fundamental questions about the future of computing architectures by fostering a stronger collaboration between *in silico* modellers, developers and hardware specialists. We seek to fill the gap in our understanding of the performance and computational properties of *in silico* brain tissue simulations by developing an understanding of how neuronal abstractions, simulation algorithms and hardware interact to determine performance. We take inspiration from the field of performance modelling, which combines the representation of a simulation algorithm and a hardware platform to produce insight and predictions on the runtime performance and efficient utilization of resources (see e.g. Balsamo et al., 2004; Calotoiu et al., 2013; Pllana et al., 2007; Williams et al., 2009), as shown in Figure 1.2.

The goal of this thesis is to lay the foundations for a constructive discussion on the computational properties and hardware features that determine the performance of brain tissue simulations. Our first contribution is a structured review of the literature on brain tissue simulations by means of a set of novel hardware-agnostic metrics that capture the most salient

performance properties of a representative collection of *in silico* models and experiments. Our second contribution is the development of an analytic performance modelling framework able to combine a description of the simulation algorithm with hardware feature specifications to obtain a runtime prediction based on a concrete understanding of the execution of the simulation workflow on modern hardware. Finally, our third contribution is an analysis of three modelling strategies that encompass most of the state-of-the-art brain tissue simulation models based on the performance modelling method described before. Our analysis is conducted along two axes: a wide-angle view of differences between modelling strategies and their implications for simulation performance and a zoomed-in view of the effects of heterogeneity within a single model of a cortical microcircuit on the performance profile. Ultimately, our work represents an effort to enable and foster cooperation between the communities of neuroscientific modellers, software developers and hardware designers.

This thesis is structured as follows: Chapter 2 presents an extensive review of brain tissue simulations and their performance, and introduces a hardware-agnostic metrics framework to structure our understanding of the heterogeneous landscape of modelling and simulation strategies. Chapter 3 establishes the connection with hardware properties through performance modelling and explains the methods and challenges in understanding the performance of *in silico* brain models; Chapter 4 develops a detailed analysis of the performance characteristics and relevant hardware features of brain tissue simulations, with a focus on understanding the commonalities and differences across models; Chapter 5 investigates the heterogeneity in model parameters and computational properties within an *in silico* cortical microcircuit model, analysing its impact on the computational characteristics of the simulation; and finally, Chapter 6 summarises our main findings, discusses some limitations of our analysis and suggests future strategies to overcome them.

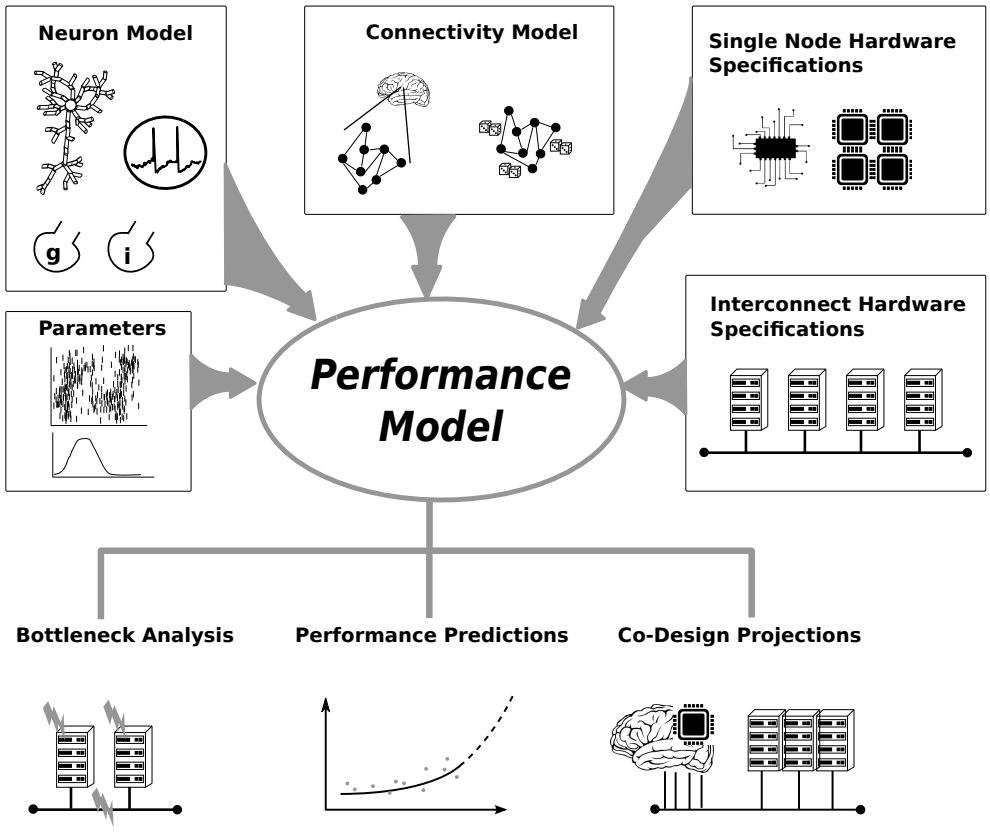

Figure 1.2 – A schematic view of a performance model applied to neuroscientific simulations. A performance model takes as input a parametrized representation of the neuron model, the connectivity, the simulation parameters, as well as a representation of the hardware platform and the interconnect fabric, and combines this information into an analytic expression to obtain insight on the performance bottlenecks and projections.

2 Overview of brain tissue simulations and performance

This chapter presents the state of the art in brain tissue simulations and performance modelling. The purpose of this chapter is to highlight fundamental concepts that will be useful in understanding and framing our work. To this end, we compile a list of representative simulation use cases and introduce a novel set of hardware-agnostic metrics that capture the most salient performance properties of brain tissue models. We use our framework to give a structured review of the literature of brain tissue simulations from the point of view of computational characteristics and performance implications on modern hardware. We conclude with an overview of performance modelling techniques and a detailed review of empirical performance studies applied to brain tissue simulations. While we do not provide an exhaustive review, our main contribution is the distillation of the heterogeneous literature in a structured form.

2.1 State of the art in brain tissue models and abstractions

Due to the multiscale nature of the brain, there exists an extremely large body of literature on models that can qualitatively and quantitatively reproduce different aspects of the brain. Modellers have addressed multiscale interactions in the brain by examining how molecular mechanisms and pathways affect cell-level behaviours such as vesicle fusion (Shillcock, 2013), synaptic plasticity (Kotaleski and Blackwell, 2010), and input sequence discrimination (Bhalla, 2017), how membrane potential and synaptic activity affect macro-scale behaviours such as voltage sensitive dye fluorescence (Newton et al., 2016), and finally how neural activity translates to cognitive function and complex behaviour (Falotico et al., 2017; Feldmeyer et al., 2013; Stewart et al., 2012). In addition to explicitly taking into account interactions between phenomena from multiple scales, there exists a multitude of modelling approaches focused on a unique level of abstraction. We distinguish here five different levels: molecular, sub-cellular, cellular, population and cognitive level. At the molecular level (Bhalla, 2014a), scientists use molecular dynamics simulation to understand how nanoscale forces affect the way in which proteins and other important molecules interact, for example in the process of vesicle fusion (Shillcock, 2013). At the sub-cellular level, researchers frequently use a simplification

process based on the well-mixed assumption to circumvent the explicit representation of individual molecules or ions and model instead their concentration. This level of abstraction was used to study the effect of inhomogeneous ion channel distribution on the neuronal membrane (Poirazi et al., 2003) and to understand how signalling pathways within synapses underlie long-lasting conservation of changes in synaptic efficacy at the heart of memory formation (Hayer and Bhalla, 2005). At the cellular level, the computational unit is the individual neuron and researchers study how synaptic connectivity and the membrane's bioelectrical properties affect the membrane potential of the cell. Within this formalism neurons can be modelled either as points (Gerstner et al., 2014) or with full morphological detail (Rall, 1962) and can be assembled in cortical microcircuits that represent a unitary portion of brain tissue (Calabrese and Woolley, 2015; Földy et al., 2005; Markram et al., 2015; Potjans and Diesmann, 2012). At the population level, assemblies of multiple neurons that share certain properties represent the modelling unit and only connectivity between populations is considered, grouping and simplifying synaptic connections between individual neurons. The population abstraction has been used to model cognitive functions and learning (Eliasmith et al., 2012), as well as in clinically relevant applications such as modelling brain tumour (Aerts et al., 2018), while its coarse level of detail makes it an ideal candidate for coupling with signal-averaging measurement methods such as EEG (Jansen and Rit, 1995; Wendling et al., 2003) and voltage sensitive dyes (Markounikau et al., 2010). Finally, at the cognitive level scientists are interested in modelling the causal relationship between observed stimuli and behaviour, often employing statistical tools that have no direct relationship with their biological substrate. Classical examples include Bayesian models (Penny, 2012), Markov models (Hintze et al., 2017) and artificial neural networks (LeCun et al., 2015; Schmidhuber, 2015). Figure 2.1 summarises the main scales of modelling.

This thesis was inspired by the work of Markram et al. (2015) and embedded in the Blue Brain Project, therefore it is framed in the context of cellular-level approaches. This level of abstraction allows for a faithful matching to a wide range of anatomical and electrophysiological data (see e.g. Hagen et al., 2016, 2018; Markram et al., 2015; Potjans and Diesmann, 2012; Pozzorini et al., 2015) and presents an important challenge in terms of understanding its computational properties, as the relationship between neuron models and performance has not yet been fully understood. Therefore the cellular-level approach represents an ideal ground to conduct our analysis. As a first step, we review in the following sections several strategies for modelling individual neurons as well as for integrating their evolution in time.

2.1.1 Cellular-level modelling abstractions

Point neuron models that can qualitatively and quantitatively reproduce the phenomenology of membrane potential fluctuations have been known for decades. One of the earliest representations to be formalized and mathematically analysed is the Lapicque – or leaky integrate and fire (LIF) – model (Lapicque, 1907), which treated the membrane as a resistor-capacitor circuit with a spike generation mechanism every time the potential reaches a certain

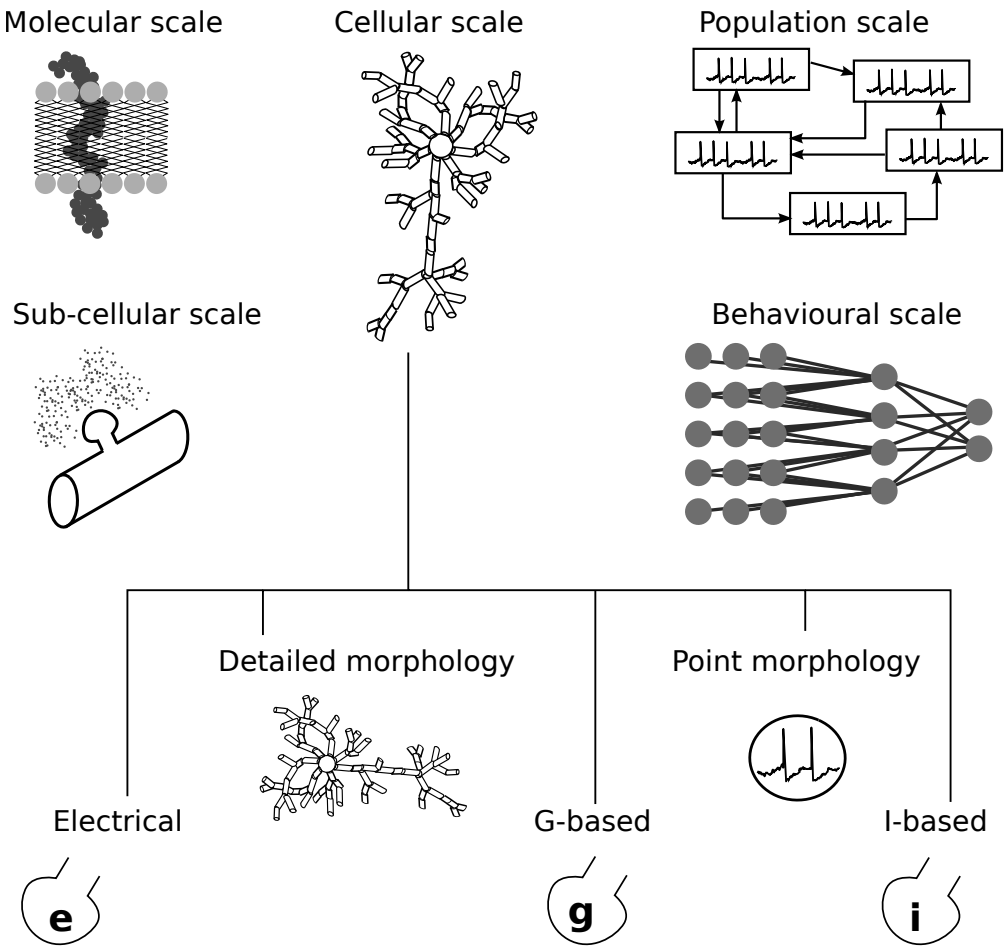

Figure 2.1 – Multiple scales of modelling and main abstractions for cellular models. At the molecular scale scientists study the effects of nano-interactions between molecules on neuronal activity. At the sub-cellular scale individual molecules are aggregated and the focus is on chemical reactions and diffusion of ions in the cellular and extracellular mediums. At the cellular scale synaptic connectivity and neurons' electrophysiology are investigated. At the population scale the main object of study are the firing dynamics of neuronal assemblies. At the behavioural scale the main objective is to reproduce and understand the brain's cognitive functions. In this thesis we focus on the cellular level of abstraction. Within it, multiple formalisms to represent neurons and synapses exist. Morphologically detailed models include information about a neuron's dendritic arborization, while point models focus on connectivity between neurons. G-based models use the electrical conductance of synaptic receptors as the most representative abstraction of a synapse, while I-based models directly use the synaptic current, and electrical synapses represent direct connections between the membrane locations of two neurons.

threshold. The LIF model has an extremely simple mathematical representation with a single linear ordinary differential equation (ODE), but has nevertheless enjoyed widespread use and applications (Gerstner et al., 2014; Tuckwell, 1988). The Hodgkin-Huxley model (Hodgkin and Huxley, 1952) takes a more complex approach. It uses a system of nonlinear ODEs with four unknowns to reproduce the phenomenon of rapid depolarisation and hyperpolarisation of neuronal membranes – known as action potential –, by modelling contributions from fast, depolarising sodium currents and slower, hyperpolarising potassium currents. A two-variable simplification was conceived a few years later, at the price of a cubic nonlinearity and the loss of interpretability of the individual current contributions (FitzHugh, 1961). More recently a simple two-variable system of ODEs with only quadratic nonlinearity was sufficient to replicate a wide range of spiking behaviours by careful tuning of its parameters (Izhikevich, 2003). Taking a similar approach, a study introduced the two variable adaptive exponential integrate and fire (AdEx) model (Brette and Gerstner, 2005) and another introduced the generalized integrate and fire (GIF) model (Pozzorini et al., 2013), which is a LIF model with a spike-triggered current and a moving threshold, both capable of reproducing an extremely wide range of observed spiking behaviours and amenable to automatic parameter fitting (Pozzorini et al., 2015).

Compartmental models allow computational neuroscientists to include the morphological details of dendritic arborization in their models (Bhalla, 2012). This field was pioneered by the seminal work of Rall (1962) who, inspired by the beauty and complexity of the drawings by Ramón y Cajal (1909), developed the mathematical formalism necessary to allow spatial discretisation of neurons along the axial dimension. Based on this work, models of individual fibres (Cooley and Dodge Jr, 1966), branched dendrites (Parnas and Segev, 1979) and whole neurons (Traub et al., 1991) have been proposed. Compartmental models represent a discretisation of the cable equation (Thompson and Kelvin, 1855) and the use of this equation in neuroscience has recently been revisited as an approximation of the Maxwell equations (Lindsay et al., 2004). As a consequence of the reliability of axonal transmission, especially in the cortex (Cox et al., 2000), a simplification is often made to simulate only the generation of an action potential (AP) in the axonal initial segment and avoid computing the actual transmission to downstream synapses by assuming a reliable transmission at a constant speed. This simplification allows reducing the computational burden by treating axonal transmission as a simple time delay, without the need to store and update the states of the axonal compartments. Citing scalability reasons, other groups have instead adopted the opposite strategy to simulate axonal compartments (Kozloski and Wagner, 2011).

2.1.2 Strategies for solving the temporal dimension

A review by Brette et al. (2007) presented a comprehensive analysis of the algorithmic and software simulation tools available to computational neuroscientists. They introduce several key concepts useful for characterising and classifying *in silico* models, namely in terms of the drivers for time-advancing, i.e. clock-driven or event-driven, and the current-based (I-based)

or conductance-based (G-based) formalisms for the integration of synaptic events. In clock-driven algorithms, operational kernels are performed regularly at fixed intervals of biological time; while in event-driven algorithms kernels are performed only if and every time a specific event occurs, typically in the form of receiving a spike. Differences in synaptic formalisms will be explained in Section 2.1.3. Figure 2.1 summarises the main modelling formalisms at the cellular level of abstraction.

Usually the analytical solution of the equations describing neuronal dynamics cannot be obtained analytically and numerical methods must instead be employed (Mascagni et al., 1989). For time-dependent ODEs comprising only of a linear and a constant term, the Exponential Euler method exploits the fact that an analytical solution can be obtained to design a first-order accurate time-stepping scheme (Bower and Beeman, 2012; MacGregor, 1987). Rotter and Diesmann (1999) introduced an extension of this scheme that enables exact integration of the temporal dynamics of a wide range of neuron models over a fixed timestep grid. For more general ODEs, the Runge-Kutta method has often been applied (see e.g. Gewaltig, 2015; Naud et al., 2008), but first order Euler schemes, both implicit and explicit, have also been used (Carnevale and Hines, 2006) as well as more modern and complex numerical schemes (see e.g. Hahne et al., 2015, 2017; Rempe and Chopp, 2006; Stewart and Bair, 2009). When using an implicit time-stepping scheme in solving the partial differential equations (PDE) associated with compartmental models, the resolution of a linear system is required because of the coupling between neighbouring compartments. The work of Hines introduced a scheme allowing to exploit linear-complexity methods to invert the corresponding matrix, enabling a generation of fast and numerically stable morphologically detailed models (Hines, 1984). To overcome accuracy and performance limitations of the fixed timestep schemes, adaptive timestep frameworks (Lytton and Hines, 2005) and hybrid explicit-implicit time stepping schemes have also been proposed (Rempe and Chopp, 2006).

2.1.3 Simulation algorithm and core properties

In the following paragraphs, we give a brief overview of the fundamental concepts related to the simulation algorithm underlying all *in silico* models and experiments. A detailed account of the algorithmic steps is given in Appendix A. While presenting a full account of the simulation workflows for all *in silico* models is out of the scope of this work, we focus on the similarities and differences that are instrumental to grasping the challenges and results in the remainder of this thesis, and refer the interested reader to (Brette et al., 2007; Carnevale and Hines, 2006; Gerstner et al., 2014) for more details. We present the algorithmic skeletons in Figure 2.2.

All the brain tissue simulations considered here fall under the category of cellular-level spiking neural network models. Within this modelling scale, the main biophysical phenomena of interest are the Action Potential (AP) and the transmission of individual spikes. An AP is a fast-dynamics event that is elicited when the membrane potential measured at the neuron's Axon Initial Segment (AIS) reaches a certain threshold. In modelling terms, the crossing of the

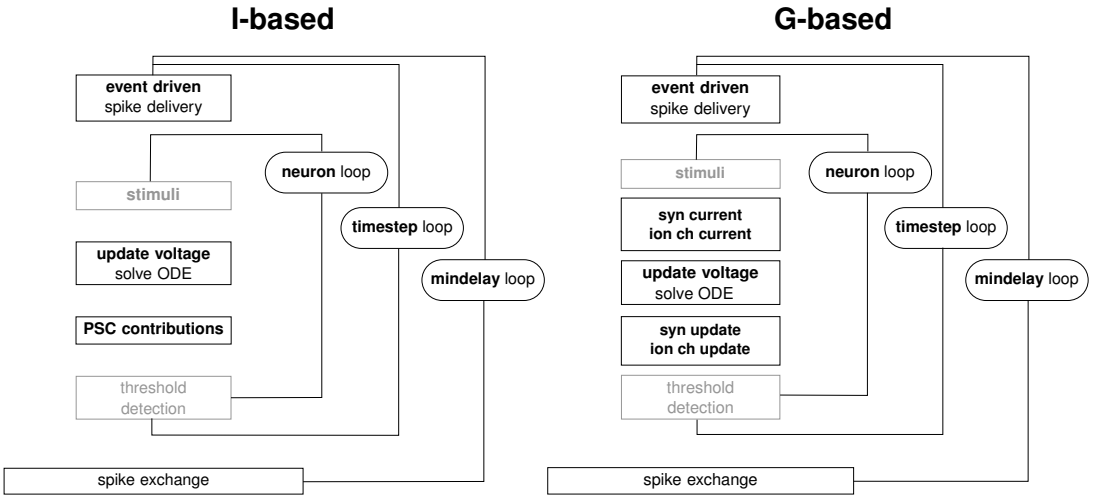

Figure 2.2 – Algorithmic skeletons for I-based *(left)* and G-based formalisms*(right)*. The figure highlights the three main loops in both formalisms: an innermost loop in which every single neuron is advanced by one timestep, a middle loop where time is stepped until a minimum delay boundary is reached, and an outermost loop that usually corresponds to a global synchronization timestep and interprocess spike exchange. Boxes in light grey are part of the simulation algorithm and are shown for completeness, but are not considered in our performance analysis as they do not constitute fundamental computational or communication kernels.

threshold is denoted as a **spike**. The AP is often conceptualized as a travelling standing wave moving along the axon, whose path crosses several synaptic locations. When the AP reaches a synaptic location in the presynaptic neuron, neurotransmitter is released in the adjacent extracellular volume. The neurotransmitter molecule binds to receptors in the postsynaptic neuron, determining the opening of specific channels that allow the influx or outflux of charged ions, ultimately causing a small variation in membrane potential at the postsynaptic neuron. Given the all-or-none nature of AP, spikes are often modelled using an event-driven formalism (Hines and Carnevale, 2004) mediated by a **synaptic delay**, although exceptions have been explored in the literature (Kozloski and Wagner, 2011). An important optimisation has been introduced by Morrison et al. (2005) when neurons communicate only via spikes: the global synchronization phase to exchange spiking information across neurons need only be performed every multiple of the **minimum network delay** period, denoted $\delta_{\min}$, while within this period the temporal dynamics of neurons may effectively be considered independent.

All the network models in our analysis can be interpreted as a graph, where nodes represent neurons, and edges represent weighted connections or synapses. We use K to denote the **fan-in** of neurons, i.e. the average number of incoming connections. For synapses there exist two main strategies for mathematical representation: current-based synapses (denoted hereinafter I-based) and conductance-based synapses (G-based). These correspond to the CUBA and COBA synapse formalisms (Brette et al., 2007). In current-based synapses, the

consequence of receiving a synaptic event is that a constant amount of current is injected in the postsynaptic neuron. In modelling terms this amount is called the **post-synaptic current** (PSC). In conductance-based synapses, on the other hand, the amount of current injected in the postsynaptic neuron is variable for different spikes and depends on the synaptic conductance through Ohm's law. Thus, when a synaptic event is received by a G-based synapse, the conductance – or a state variable from which the conductance can be computed – of that synapse is updated by a fixed quantity. In addition to the chemical synapses described above, neurons can also establish a direct connection between membranes known as an electrical synapse or a gap junction.

While there are significant differences across models in the way in which the membrane potential is modelled and integrated over time, all of the formalisms considered here have some sort of nonlinear dynamics that is able to, under the right conditions, reproduce the behaviour of the AP. However, different representations of neurons based on the level of morphological detail exist: point neurons do not include any information from the dendritic arborization of the neuron, while morphologically detailed neurons typically do. In terms of data representation, point neuron models typically require a few state variables to represent the membrane potential, while detailed models usually need at least one variable per compartment for the membrane potential, plus a few more for every ion channel type associated to that compartment – i.e. the spatial discretisation unit – making the total number of variables that represent a neuron quite large. There is also heterogeneity in the data representation of synapses depending on the underlying formalism: I-based synapses are represented by the total post-synaptic current I_{syn}, while G-based synapses by their conductance g_{syn}.

The heterogeneity in morphological representations and synaptic formalisms leads to some fundamental differences as well as some similarities across simulation strategies. Our analysis is focused on fixed timestep integration methods based on the bulk synchronous-parallel (BSP) formalism (Valiant, 1990). Figure 2.2 presents the main simulation algorithm for I-based and G-based formalisms. A detailed account of all the phases in the simulation workflow is provided in Appendix A. In the I-based simulation workflow, the effect of spikes is integrated at the beginning of each timestep, then the state of all neurons is advanced by one timestep by first numerically integrating the membrane equation and then updating the post synaptic current (PSC). When all neurons have been advanced to a time that is a multiple of the δ_{min}, a global synchronization point occurs and neurons exchange spiking information. In the G-based algorithm the integration and the communication of spikes happens in the same way, but the algorithm to update neuron states is different (Hines, 1984). First, a loop through all ion channel and synaptic types computes the current contributed by individual instances to the neuron's membrane equation. Then the membrane equation is solved numerically, and finally the state variables of all the ion channel and synapse instances can be advanced in time, once again through numerical integration. In the case of morphologically detailed neurons, solving the membrane equation usually requires the inversion of a matrix, for which Hines (1984) provided a linear-complexity algorithm.

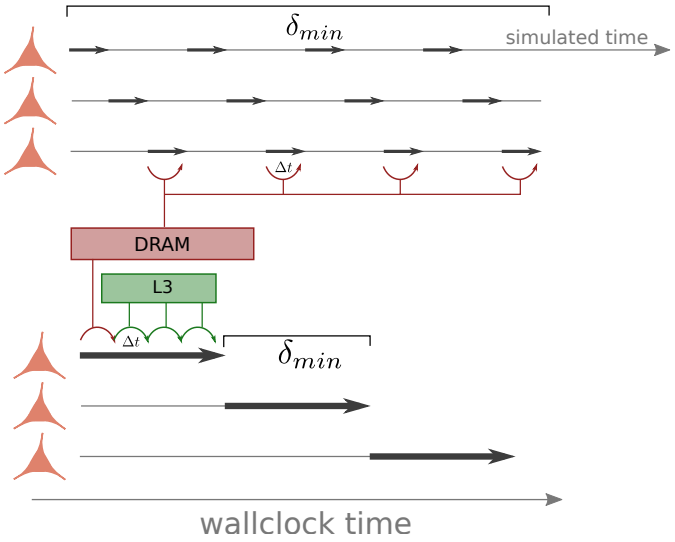

Figure 2.3 – Loop ordering optimisation. In the naïve implementation *(top)* the state of each neuron is advanced by one timestep, then the next neuron is processed and also advanced by one timestep, until all neurons have reached the same instant in time. In this implementation there is no possibility for cache reuse, and data is always fetched from main memory at every time iteration. In the loop ordering optimisation *(bottom)*, the state of a neuron is advanced by a whole minimum delay interval before the next neuron is processed. In this implementation, data must only be fetched from main memory at the first time iteration, and we assume data to be in the L3 cache during all subsequent iterations.

We conclude this section with the description of an important optimisation that promotes data reuse. It is possible to swap the timestep and neuron loops, by having a single neuron advance multiple timesteps until it reaches a minimum delay boundary, before the next neuron's evolution is integrated in time. By updating a neuron's state for multiple timesteps sequentially it should be possible to temporarily store data in caches instead of main memory, as long as a neuron's memory traffic requirements and the number of spikes to be integrated by that neuron are sufficiently low. We denote hereinafter this optimisation as the **loop ordering optimisation**. Figure 2.3 presents a visual representation of this technique, highlighting the source of data for each neuron's update step. In the naïve implementation each neuron is advanced only by a single timestep each time, therefore no reuse is possible because each neuron will overwrite the caches with its own data. In the optimised implementation a neuron is advanced by multiple timesteps, thus the relevant state data for that neuron is not overwritten from one timestep to the next, and can be stored in the cache. This optimisation was first introduced in Plesser et al. (2007) and is built-in in modern simulators such as NEST (Gewaltig and Diesmann, 2007) and Arbor (Akar et al., 2019b), while it can also be enabled through proper configuration in CoreNEURON (Kumbhar et al., 2019b). In terms of implementation effort, the loop ordering optimisation does not come for free. In particular, separate spike event queues must be maintained for each neuron to enable efficient integration of synaptic events.

2.1.4 Notable examples of high performance brain tissue simulations

Coupling modelling efforts with high-performance computing (HPC) has allowed to overcome limitations in the scale at which experiments are being conducted, such as measuring individual ionic currents and local field potentials from thousands of cells simultaneously (Reimann et al., 2013), isolating different sources of noise (Nolte et al., 2018), or perturbing individual spikes (Izhikevich and Edelman, 2008). As a consequence, HPC has enabled computational neuroscientists to test hypotheses and study protocols with a faster turnaround than ever (Jordan et al., 2018; Knight and Nowotny, 2018; Ovcharenko et al., 2015). Moreover, the process of identifying strategic data and principles governing the brain's organisation and function lies at the heart of the modelling effort. Such an endeavour represents a very effective way to understand the brain as a complex system, by inspecting the model when it fails to reproduce a certain behaviour of interest (Fan and Markram, 2019; Reimann, 2014). Finally, the existence of a validated computational model of the brain could allow, in the future, to reduce our reliance on animal experiments as well as develop and validate disease treatments *in silico* (D'Angelo, 2014; Frackowiak and Markram, 2015).

The underlying complexity of the brain entails that the best simulation attempts to date are the outcome of many years of collaborative efforts. Historically, one of the first simulations of the mammalian thalamocortical system was presented by Izhikevich and Edelman (2008). Despite being the first large-scale endeavour, this model was by no means simplistic and featured stylized neuronal morphologies, short and long term synaptic plasticity, and data-constrained random connectivity. Given the complexity of simulating brain systems, especially at the scale of brain regions, big science initiatives have been developed that have the capability to leverage unprecedented amounts of data made available by new technologies (De Garis et al., 2010; Kandel et al., 2013). The Blue Brain Project (BBP) is a research initiative aimed at building and simulating a biologically detailed digital reconstruction of the rodent brain (Markram, 2006; Markram et al., 2015). This endeavour is based on a data integration strategy consisting of collecting fragmented neuroscientific data. Specifications for this reconstruction are obtained by combining experimental data such as neuron morphologies and patch-clamp recordings with digital reconstruction heuristics for single-cell parameter fitting (Van Geit et al., 2008, 2016), cell connectivity (Iavarone et al., 2019; Reimann et al., 2015, 2019) and reconstructed thalamic inputs (Iavarone et al., 2019) to obtain a cortical microcircuit (Markram et al., 2015). The Human Brain Project is a European-wide initiative aimed at building the research infrastructure necessary to advance neuroscience, medicine and computing in the next decade (Markram, 2012). One of the main deliverables of this project is the Brain Simulation Platform, a collaborative platform for the reconstruction and simulation of brain models. The Allen Institute for Brain Science has recently developed BioNet (Gratiy et al., 2018), a python framework for describing large-scale neural network experiments using the NEURON simulator as backend that has been used in a model of layer 4 of mouse visual cortex (Arkhipov et al., 2018).

Computer simulations are also being used as a primary tool for neuroscientific discovery. The

transition from sleep to wakefulness (and vice versa) has been analysed using simulations of computational models (Hill and Tononi, 2005), and more recently by explicitly modelling neuromodulator release (Bazhenov et al., 2002). Simulations of neuromodulators have also been used to investigate the relationship between persistent spiking and cognitive tasks in a computational model of entorhinal cortex (Fransén et al., 2002). The microcircuit described by Potjans and Diesmann (2012) has been extensively used to study the computational properties of a cortical column with a focus on learning and neuroplasticity, both in its original version and as a basis for population models (Cain et al., 2016; Schwalger et al., 2017). Computer models of chemical reaction pathways have been used to investigate the role of specific signalling molecules in synaptic plasticity (Graupner and Brunel, 2007; Hayer and Bhalla, 2005), while multiscale models have been used to analyse the interactions of different levels of signalling (Bhalla, 2014b). In addition to publicly funded research institutions, private companies such as IBM have invested in developing functioning models of brain subsystems, both to study how synaptic plasticity regulates neuronal responses (Kozloski, 2016; Kozloski and Cecchi, 2010) and to develop the field of cognitive computing (Modha et al., 2011).

2.2 A structured view of the modelling landscape

2.2.1 A representative collection of *in silico* models and experiments

Several modelling abstractions and formalisms have been proposed in the literature of brain tissue simulations, and for each one there can be different algorithms and implementations, which have in turn been concretised in simulations on different hardware platforms. Upon close inspection we have identified a collection of representative use cases that captures the wide variety of cellular-level models in the literature. The identification and codification of recurring algorithmic patterns was demonstrated successfully in the HPC domain with Berkeley's dwarves of computing (see Asanovic et al., 2006, 2009). Their analysis identified and described twelve recurring computational patterns that can be considered the building blocks of almost all HPC applications in the literature. This methodology was extended to other computational fields such as big data (Fox et al., 2014), cloud computing (Phillips et al., 2011), accelerator-based computing (Krommydas et al., 2016) and automatic symbolic computation (Kaltofen, 2012), while a list of computational motifs in biological neural networks' architectures was also compiled (Marcus et al., 2014). The review by Brette et al. (2007) applied a similar approach to neuroscientific use cases, although their focus was on simulator software and target applications, while ours is a general analysis of the most representative and widely used models. We introduce below the *in silico* models and experiments at the basis of our characterisation of the performance landscape. We focus on short to medium length simulations of brain tissue, excluding long-term plasticity from our analysis. While long duration learning experiments have a significant interest in the computational neuroscience community, we narrow our analysis to simulations of the inference dynamics of neuronal circuits comparable to the work of Markram et al. (2015). This choice still allows us to investigate meaningful properties of neural network dynamics (see e.g. Brunel, 2000; del Mar Quiroga et al., 2016;

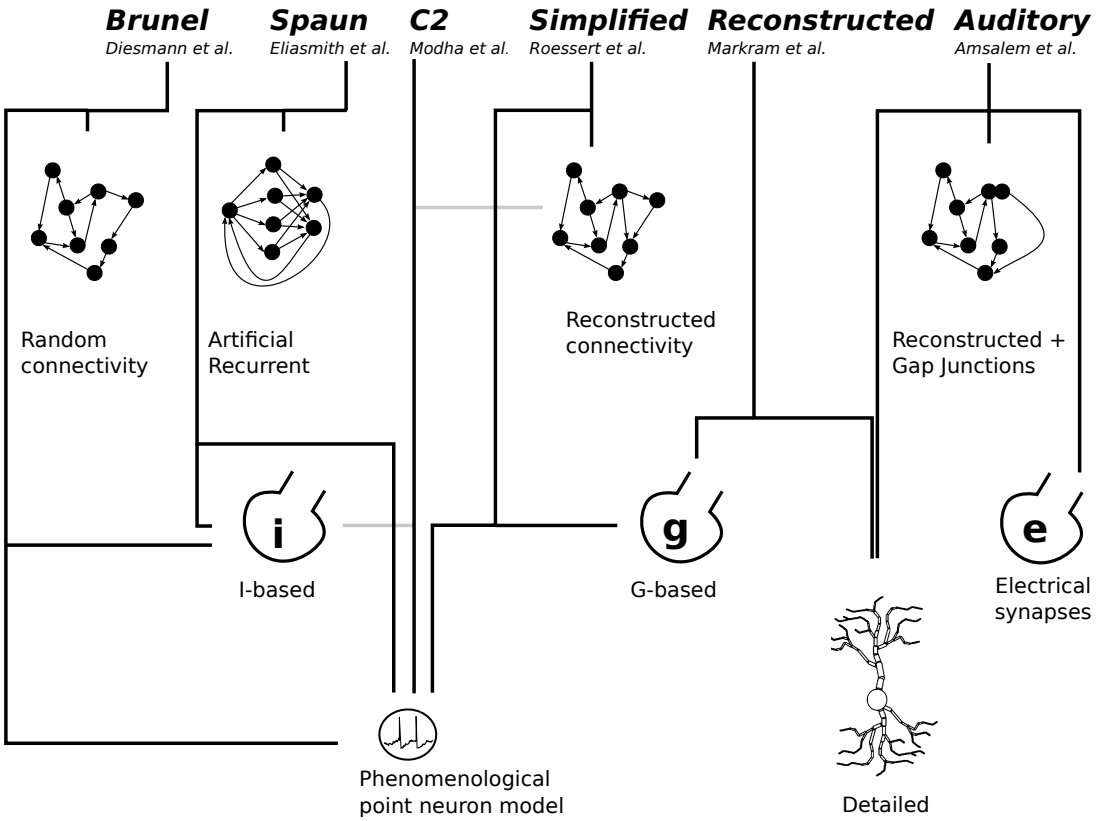

Figure 2.4 – Summary of *in silico* models and experiments. Lines connect a model's name with its underlying neuron model, synaptic formalism and network topology. Grey lines stemming from the C2 model were used to avoid intersecting lines that would have hindered the clarity of the figure.

Markram et al., 2015; Reimann et al., 2013) and circumvents the large degree of complexity added by the wide variety of long-term plasticity models and simulation experiments (see e.g. Citri and Malenka, 2008; Frémaux and Gerstner, 2016; Markram et al., 2012; Rennó-Costa et al., 2019). For each use case we highlight important modelling differences and similarities, and summarise their main features in Figure 2.4. More information about the numerical values of model parameters can be found in Table A.1 in Appendix A.

The Brunel model This *in silico* model is based on a randomly connected network exhibiting the balanced excitation-inhibition property that can be observed *in vivo* in the mammalian cortex (Brunel, 2000). While other connectivity strategies considered here also use randomness, the peculiarity of the Brunel model is that it is not constrained by experimental data nor locality considerations, such that a neuron has equal chance of forming a connection with any other neuron within the network. It uses the leaky integrate-and-fire model, a phenomenological point neuron model able to reproduce many types of electric neuronal behaviour,

coupled with static (and possibly plastic) current-based synapses. We consider here a very large-scale implementation that served as a proof of concept for the feasibility of human brain-scale simulations (Helias et al., 2012; Kunkel et al., 2014), although we exclude synaptic plasticity from their model. This large-scale *in silico* experiment contained 1.86×10^9 neurons and 11.1×10^{12} synapses, and was made possible by calculated software design decisions and optimisations, and a parallel cluster comprising 82944 nodes. It is characterised by a homogeneous set of parameters across neurons and synapses, large synaptic delays and a very large number of incoming connections per neuron.

The Spaun model The Spaun *in silico* model is able to replicate human-like cognitive functions using a spiking neural network (Eliasmith et al., 2012). It is based on the Semantic Pointer Architecture (MacNeil and Eliasmith, 2011) and the Neural Engineering Framework (Eliasmith and Anderson, 2004). A recent implementation on the neuromorphic substrate SpiNNaker was also demonstrated (Mundy, 2016; Mundy et al., 2015). The architecture of Spaun was inspired by the brain regions observed in the mammalian neocortex, but is ultimately engineered to achieve its cognitive goal. It uses 2.5×10^6 IAF neurons and 60×10^9 synapses and has been shown to learn tasks that require memory, classification and motor control. In terms of modelling choices, Spaun uses a large time integration step, very lightweight neuron and synapse models consisting of only a few parameters each and a few readout neurons with extremely large numbers of incoming connections.

The C2 model This *in silico* model has been developed as part of the private industry's effort to establish itself in the domain of cognitive computing. It achieved a very large scale simulation of 1.6×10^9 neurons and 8.87×10^{12} synapses over 147,456 CPUs in Lawrence Livermore's National Laboratory's Dawn Blue Gene/P in 2009 (Ananthanarayanan et al., 2009). It uses Izhikevic point neuron models (Izhikevich et al., 2004) and synapses with short-term plasticity, and its connectivity was obtained from the CoCoMac dataset (Bakker et al., 2012) by extracting connection probabilities. This model is characterised by the elimination of synchronous, blocking collective communications in favour of asynchronous sending and receiving of messages (Ananthanarayanan and Modha, 2007).

The Reconstructed model This *in silico* models is obtained by a detailed reconstruction of a neocortical microcircuit, i.e. a functional unit of cortical tissue (Markram et al., 2015). This model uses digitally reconstructed morphologically detailed neurons, with individual models for ion channels and synapses whose distribution is obtained via a parameter fitting process to reproduce electrophysiological data. Connectivity is implemented with G-based synapses with short-term plasticity following the Tsodyks and Markram (1997) formalism, while the connection probability is given by detecting touches between neurons and filtering them according to biologically inspired rules (Reimann et al., 2015). This model requires a small time integration step to properly account for the dynamics of fast ion channel currents. Moreover,

the distributions of synaptic delays and number of incoming connections are quite dispersed and exhibit low minimums and large maximums. A large-scale simulation of this model was achieved using 28,672 CPUs on the Blue Gene/Q system at Jülich in 2015 (Ovcharenko et al., 2015).

The Simplified model To tame the complexity of the Reconstructed model, a simplified version was developed with similar connectivity patterns but different neuron and synapse models (Rössert et al., 2016). In this *in silico* model, morphologically detailed neurons are replaced with point generalized integrate-and-fire (GIF) neurons (Pozzorini et al., 2015). While the G-based formalism for synapses is retained, the number of incoming connections per neuron is restricted to a fixed number, where different connections are lumped together according to their synaptic delay and distance from the soma. Thanks to the simplification process, the timestep of this model can be increased compared to the original Reconstructed one, and significantly less parameters are required to represent a neuron.

The Auditory model An extension of the Reconstructed model where some dendro-dendritic connections between cells of the same type – Layer 2/3 Basket Cells – of the neocortex have been instantiated using electrical synapses (gap junctions) (Amsalem et al., 2016). This study replicated experimental findings in the auditory cortex that gap junctions affect the selectivity of interneurons. This *in silico* model is based on complex morphologically detailed neurons, and even though an explicit scheme can be used for the propagation of gap junction voltages without introducing numerical instability, it requires a synchronization barrier at every timestep.

We identify three main modelling families which contain all the *in silico* models cited above. The Brunel, Spaun and C2 all fall under the category of I-based point neuron models. They distinguish themselves based on the strategy for determining connectivity, but in terms of overall algorithm structure they can be considered equivalent. For reference, the microcircuit proposed by Potjans and Diesmann (2012) also belongs to this category. The Simplified model is, to our knowledge, the only large-scale example of G-based point neuron *in silico* experiment. Finally, the third category comprising the Reconstructed and Auditory models is the family of G-based morphologically detailed abstractions. In this case, the difference between the two models is simply the presence of electrical synapses, which determines a slight modification in the interprocess communication patterns. These modelling families will constitute the backbone of our performance analysis, and we summarise in Table 2.1 the main features of each one.

Table 2.1 – Summary of main algorithmic features.

	point I-based	point G-based	detailed G-based
spike delivery	update neuron I_{syn}	update synapse g_{syn}	update synapse g_{syn}
current contributions	precomputed I_{syn}	update I_{syn}, I_{ion} using Ohm's law A.5	update I_{syn}, I_{ion} of relevant matrix element
voltage update	solve small system of ODE with at most four equations	solve small system of ODE with at most four equations	invert linear system matrix in $O(n)$ time using Hines algorithm
ion channel and synapse state update	exponential decay of I_{syn}	solve an ODE per synapse instance	solve an ODE per ion channel and synapse instance
spike exchange	delayed interprocess communication every $\delta_{\min}$	delayed interprocess communication every $\delta_{\min}$	delayed interprocess communication every $\delta_{\min}$

We summarise here the main algorithmic features of point I-based, point G-based and detailed G-based algorithms. While a detailed I-based modelling of neurons could in theory exist, it is not present in our list of *in silico* models and experiments and we are not aware of any publications using it. These three formalisms represent the backbone of all *in silico* models and experiments, but many variations exist, namely in the way the communication phase is implemented. In the case of gap junctions, for example, an additional communication phase is required at every time step. Another possibility that has been explored in the literature is to abandon the event-driven representation of synapses altogether, and model them as a continuously active process whose current depends on the voltage of the presynaptic neuron (Kozloski and Wagner, 2011).

2.2.2 Hardware-agnostic performance metrics

We now introduce a framework for characterising brain tissue simulations based on their performance properties. Our approach is based on a set of hardware-agnostic metrics that capture essential performance properties at the intersection between models and hardware. The technique of using metrics to characterise applications has been widely used in studying performance. A study focused on parallel applications identified a group of eleven metrics divided in three categories: computation, communication and synchronization (Van Amesfoort et al., 2010; van Amesfoort et al., 2012). Examples of such metrics include Arithmetic Intensity, Memory Footprint, Length of Communication Messages, Number of Global Synchronizations, and others. Other sets of partially overlapping metrics have been proposed (see e.g. Hoste and Eeckhout, 2007; Strohmaier and Shan, 2004; Treibig et al., 2012). Finally, metrics have also been applied to the assessment of high-performance computer systems and cloud infrastructure (see e.g. Furlani et al., 2013; Rodrigues et al., 2015). Our work distinguishes itself from previous research because we propose a set of hardware-agnostic metrics based on extensive use of domain knowledge. Our purpose is to provide a common tool that can be understood and utilized by both computational neuroscientists and HPC developers.

Our framework projects the computational costs on three orthogonal dimensions: Memory, Information Propagation and Sequential axis. In the first dimension, we consider aspects of the model that can affect the memory footprint such as the number of parameters and degrees of freedom required to represent a neuron. In the second dimension, we consider aspects tied to communication of information between neurons, which can be related to hardware features such as interconnect latency and bandwidth in a distributed network. Taking the point of view of a connection, we consider properties such as how many times it can carry information during one sequential iteration, how much information it carries and the topological structure of the network connectivity. Finally, in the third dimension of computational complexity, we consider aspects tied to sequential iterations which can be related to hardware features such as single node performance, processor frequency and memory bandwidth.

Characterisation of the memory requirements of the elementary unit A first source of complexity for neural networks arises from the memory requirements of the very large models being used. As a first metric, we report the number of variables required to represent a single unit of computation, namely one neuron and its connections, considering all the degrees of freedom such as the membrane potential as well as the read-only parameters such as time constants. We break down the unit size along three dimensions: number of variables to represent a disconnected neuron (*vars per neuron*), number of variables to represent a single connection (*vars per connection*), and number of connections per neuron (*connections per neuron*). Each of these dimensions can have heterogeneity within a model so we report mean, standard deviation and maximum values in Figure 2.5. Analysis of the neuron size in Figure 2.5 shows a clear trend: as more and more biological detail is included in the model, more variables are required to represent neurons' activation functions, but also more heterogeneity between

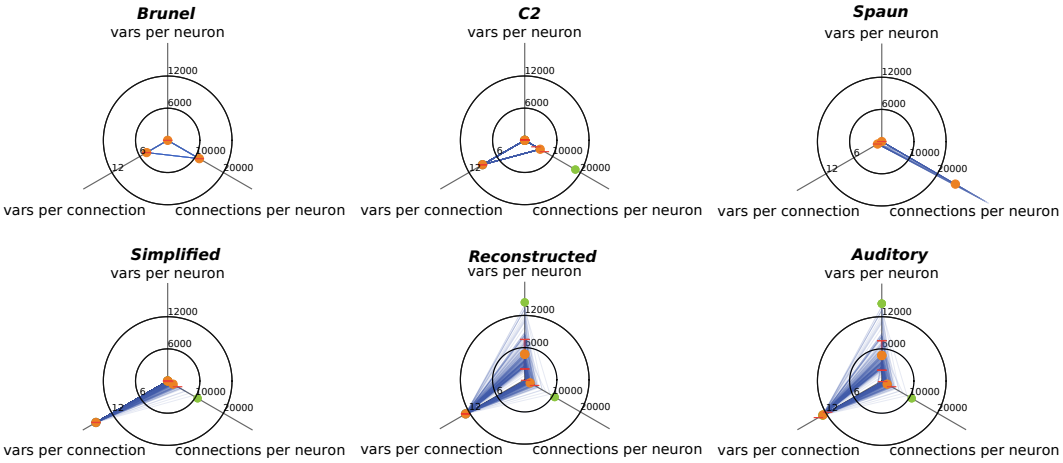

Figure 2.5 – Breakdown of the **unit size** metric. This metric captures the memory footprint of *in silico* models, broken down in three components: number of variables to represent a single neuron (excluding synaptic connections), number of variables to represent a connection, and number of connections per neuron. Orange dots represent mean values, red bars represent standard deviation and green dots represent maximal values. The blue lines represent actual samples from the model. Statistical distribution data was missing for the C2 model.

neuron models is observed. Moreover, we identify characteristics that are representative of classes of models. Point neuron models are characterised by simple neurons and slightly more complex synapses, and by a large number of connections per neuron but with a relatively low heterogeneity. The Spaun model represents an extremal example with simple neurons and very simple synapses and an extremely large number of connections per neuron. Finally, detailed models are characterised by complex neurons, complex synapses and a medium-large number of connections per neuron, with a high heterogeneity.

Characterization of parallel information propagation patterns We now consider the computational costs associated with weak scaling and distributed architectures. We conduct our analysis along three dimensions: how often, how much and to whom do neurons need to communicate information. With a few notable exceptions (see e.g. Kozloski and Wagner, 2011) almost all the *in silico* models exploit synaptic delays to decouple communication and computation (Plesser et al., 2007). To capture this property we define the *coupling ratio* in Figure 2.6 as the ratio of the minimum network delay to the simulation timestep, i.e. $\frac{\delta_{\min}}{\Delta t}$. Secondly, we define the *information transmitted by a connection* in Figure 2.6 as the number of variables being transmitted *on average* by a connection during a mindelay period, computed by the formula $f \times \delta_{\min} \times s$, where f is the average firing frequency of neurons and s is the number of variables to represent a spike. It is possible to estimate the number of variables communicated per iteration with the formula *coupling ratio* × *information transmitted by a connection*. As a third dimension of interest, we consider the structural topology of the analysed networks. We compute the *small-worldliness degree* as defined by Humphries and

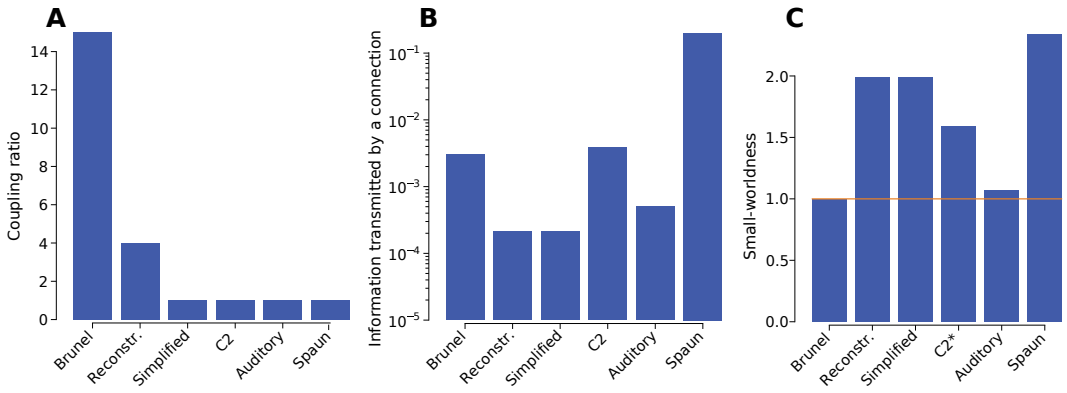

Figure 2.6 – Information sharing metrics. **A: Coupling ratio** denotes the number of simulation timesteps before a global synchronization point. **B: Information transmitted by a connection** the average number of variables transmitted via a connection during one minimum delay period. **C: Small-worldliness degree** a value of 1 indicates random networks, large values denote small-world networks. The asterisk in the C2 label indicates that the necessary data was not available to us, so instead we took the reported values for the macaque cortex from (Humphries and Gurney, 2008).

Gurney (2008): this value is equal to 1 for purely random networks, and becomes larger than 1 for networks exhibiting small-world characteristics (Watts and Strogatz, 1998).

Characterization of serial iterations We consider here sources of complexity that become relevant in a strong scaling scenario, by looking at the serial timestepping iterations. There are two ways in which the sequential complexity of a model can be reduced: by reducing the number of iterations or by reducing the cost of a single iteration. We start by defining the *sequential compressibility limit* in Figure 2.7 as the inverse of the timestep $\frac{1}{\Delta t}$, i.e. the number of iterations required to simulate one second of model time. Ideally we would continue by counting the number of operations required to fully complete an iteration on a single neuron, and use that as our second metric. Unfortunately, this value is ill-defined because the resolution of differential equations in the neuron and synapse models requires complex numerical methods often based on non-elementary mathematical operations such as exponentials and divisions. Instead, we define the *iteration compressibility limit* in Figure 2.7 as the number of variables updated per neuron during an iteration and distinguish between clock-driven and event-driven updates (Brette et al., 2007). It is possible to estimate the total sequential computational cost by multiplying the *sequential compressibility limit* times the *iteration compressibility limit*.

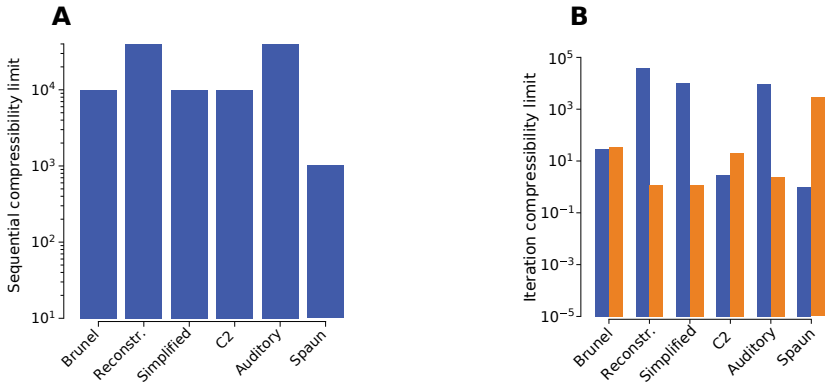

Figure 2.7 – Serial iterations metrics. **A: Sequential compressibility limit** denotes the number of time iterations required to simulate one second of biological time. **B: Iteration compressibility limit** denotes the number of degrees of freedom updated in a δ_{min} interval. Blue bars represent clock-driven updates, orange bars represent (average) event-driven updates.

2.2.3 Characterization of modelling approaches

Analysis of the hardware-agnostic metrics allows us to characterise *in silico* models and experiments based on their profile. For example, the Reconstructed model (Markram et al., 2015) has a low coupling ratio (see Figure 2.6) and has very computationally-intensive neuron models (see Figure 2.7). This makes it a good candidate for weak scaling on conventional distributed architectures as demonstrated by its good weak scaling efficiency on the BlueGene/Q supercomputer (Ovcharenko et al., 2015). On the other hand, point neuron models were shown to be communication bound (Ananthanarayanan and Modha, 2007), as can be inferred within our framework from comparing the low-complexity neuron models in Figure 2.7 to the high communication requirements in Figure 2.6, especially for the Brunel model. Our metrics capture the fact that the Spaun and C2 models are characterised by predominantly event-driven computations, high communication requirements and low heterogeneity between neuron models, making them a good candidate for neuromorphic architectures (Cassidy et al., 2014; Mundy et al., 2015). However, the implementation of these models on neuromorphic frameworks is not always trivial and special techniques such as matrix rank factorization may be required in order to overcome hardware limitations imposed on models with very high connectivity such as Spaun (Mundy et al., 2015). A non-trivial finding that arises from our analysis is the peculiarity of Spaun's connectivity profile: it shares characteristics with artificial neural networks (high connectivity readout neurons) and reconstructed biological neural networks (small world network with large clustering coefficient). This connectivity profile may explain the model's ability to reproduce cognitive behaviour using biologically inspired neuron and synapse models. We also observe that seemingly small changes to a model, for example the addition of gap junctions, can also have a great impact in the communication pattern by breaking the loosely-coupledness aspect as shown in Figure 2.6, and has lead researchers to investigate novel algorithmic solutions to recover the communication pattern (Hahne et al.,

2015).

From the point of view of computational costs, the explicit modelling of time sets a requirement on the number of iterations to complete one simulation because of the numerical discretisation (see Figure 2.7), a limit that cannot be improved via strong scaling of hardware resources, thus imposing a hard limit to real-time simulations (Zenke and Gerstner, 2014). An interesting finding that arises from our analysis is the fact that simple models exhibit a low sequential compressibility limit, a characteristic that could potentially confer them better strong-scaling properties than detailed, reconstructed networks, which on the other hand perform better in weak-scaling scenarios. Moreover, the spike formalism enables event-driven simulations (Brette et al., 2007), and models that exploit event-driven algorithms will generally have a computational cost that scales with the network activity (Ananthanarayanan and Modha, 2007), an effect that can be especially prominent on a SIMD architecture (Yavuz et al., 2016). We work on the assumption that due to its regularity a mainly clock-driven algorithm, on the other hand, does not require low-latency context switching, nor very powerful branch prediction, nor complex control structures in hardware, and is therefore amenable to an efficient implementation using throughput-optimized hardware. Furthermore, models dominated by event-driven computations may be a good candidate for highly parallel, low latency neuromorphic systems (Cassidy et al., 2014).

We have tried to include as many modelling abstractions and simulation strategies as possible in our collection of *in silico* models and experiments, but ultimately were forced to exclude some of them. Models of long-term synaptic plasticity would provide a complementary view of the modelling strategies and computational properties in brain tissue simulations. The unpredictable distribution of timesteps in adaptive timestepping methods (Lytton and Hines, 2005) makes them particularly difficult to model within our framework, although they have been shown to yield higher numerical accuracy and important performance benefits (Magalhaes et al., 2019c). Spatial decomposition methods (Kozloski and Wagner, 2011) discard the event-driven component entirely by explicitly modelling axonal transmission and synaptic boutons, thus making them unfit for our metrics. Finally, including artificial neural networks (LeCun et al., 2015; Schmidhuber, 2015) in our *in silico* models and experiments would give give an interesting comparison in terms of performance requirements.

Our approach combining the categorisation of *in silico* models and experiments based on their algorithmic properties with hardware-agnostic metrics that capture fundamental performance properties has enabled us to obtain a structured characterisation of the heterogeneous literature of brain tissue simulations. However, hardware-agnostic metrics lack a direct connection with hardware features. Building on the insight and characterisation gained from our hardware-agnostic analysis, we leverage analytic performance models as a bridge between biological abstractions and hardware to obtain quantitative insight and predictions on the performance of *in silico* models and experiments.

2.3 State of the art in analytic performance modelling

Performance modelling is an empirical and theoretical effort to abstract relevant hardware and software properties to obtain insight on the runtime and bottlenecks of a given application. Its main goal has been stated to be "to maximise the amount of completed science per cost and time unit" (Hoefler et al., 2011, p.2). Performance modelling can have different targets, from optimising an application's runtime, or tuning scheduling policies, to codesigning custom hardware. Moreover, it can be applied at different stages in the lifetime of a computer system, from its design, deployment, testing and maintenance. Other computational sciences have benefited from performance modelling, allowing to: assess and improve the resource consumption and scalability of user applications (Balsamo et al., 2004); assess and improve the scalability of third-party libraries (Shudler et al., 2015); establish a mapping of software and hardware architectures to maximise performance (Narayanan and Seznec, 2015); drive hardware purchase decisions (Calotoiu et al., 2013); drive hardware codesign (Navaridas et al., 2012).

Performance models are often categorised in two main classes: black-box and white-box approaches. *Black-box* models, based on the paradigm that the performance model should ideally be unaware of the code and the underlying platform, have been developed to obtain performance predictions even when detailed knowledge about the hardware or software is lacking. Catwalk (Calotoiu et al., 2013; Wolf et al., 2014) is a black-box model based on interpolating measured performance points with a set of model families and comparing the best fit to the scaling expectations defined by the user, with the ultimate goal of detecting performance bugs. Other researchers have proposed instead to use machine learning methods to predict the performance based on many collected data points (Ipek et al., 2005; Lee et al., 2007). Black-box models have a wide range of applicability, but wrong predictions may be rooted in inappropriate fitting procedures and do not necessarily lead directly to better insight through inference. *White-box* models, on the other hand, are based on known technical details of the hardware and some assumptions about how the software executes. A widely known example of a white-box approach is the Roofline model (Williams et al., 2009) but other approaches exist (Herodotou, 2011; Velez et al., 2019). White-box models offer *insight through failure*, by which a performance-aware developer can infer the presence of a performance bug by comparing the idealised expected performance of their program with the measurements. However, white-box models can be quite difficult to apply to real-world applications with complex behaviours, and it can also be difficult to distinguish situations in which the application has a performance bug to situations in which the model is too idealized.

2.3.1 Performance modelling of single-node shared-memory applications

Memory bandwidth represents a bottleneck in parallel scaling of shared-memory applications when the rate of data transfer from main memory is much lower than the rate of data consumption by the processor cores (Kagi et al., 1996). This represents a trade-off in terms

of optimisation: one must be aware of the data requirements of the application to avoid inefficient usage of hardware resources (Bailey et al., 2010; Hager and Wellein, 2010). A review of shared-memory performance modelling compared several approaches but did not include modern architectures and optimisations (Nechvatal, 1988) . To evaluate the cost of a shared memory bus an analytic approach based on mean value analysis combined with simulation of a processor with Instruction Level Parallelism (ILP) capabilities has been used (Sorin et al., 2003, 1998), while an approach based on Petri nets allowed to explicitly model the mismatch between processor and memory performance (Zuberek, 2018). Another analytic performance model based on mean value analysis showed that memory contention represents an important bottleneck in single-threaded applications with large memory requirements (Bardhan and Menascé, 2014). Finally, the tradeoff between maximising resource usage and meeting individual applications' performance goals has been studied (Chen et al., 2012) and an analytic performance model for caches based on queuing theory and application tracing has been proposed (Agarwal et al., 1989).

Microarchitectural features of the CPU determine the maximum instruction throughput that it can sustain. Optimizations such as pipelining, superscalar architecture, out-of-order scheduling, speculative execution, etc. have sustained a constant growth in the performance of CPUs (Hager and Wellein, 2010; Hennessy and Patterson, 2011). Given the complexity of execution paths in modern CPU architectures, few attempts at analytic modelling have been made. A very detailed model was developed to assist in the evaluation of design tradeoffs for superscalar processors (Dubey et al., 1991), while the tradeoffs of out-of-order and in-order execution and codesign tradeoffs in the power/performance design space have been investigated based on insights from analytic modelling (Breughe et al., 2015; Esmaeilzadeh et al., 2012). Focusing on the performance-critic exponential function, our work investigated the relationship between polynomial evaluation algorithms and microarchitectural features (Ewart et al., 2019). We were able to demonstrate that the latency and throughput of the exponential based on simple algorithms could be captured by an analytic formula, but in the general case it was impossible to establish a direct relationship between an algorithm's set of instructions and its performance characteristics. Due to the difficulty of capturing all the features of superscalar, out-of-order processors in an analytic formula, cycle-accurate simulators such as RSIM (Pai et al., 1997), SimpleScalar (Burger et al., 1996) and FastSim (Schnarr and Larus, 1998) have been developed.

The analytic performance models mentioned above require extensive parametrization and modelling efforts to deliver highly accurate performance predictions. While high accuracy is a desirable property, the complexity of these models often leads to loss of interpretability and generality. Other efforts have focused instead on clarity and explainability, for example using a simple model to understand the cost of memory latency and bandwidth in big data applications (Clapp et al., 2015), or a more complex analytical model to explore the power limitations in future multicore architectures (Esmaeilzadeh et al., 2012). The Roofline model (Williams et al., 2009) is perhaps the most prominent example of white-box approaches focused on simplicity. We give a brief introduction to this performance model in Appendix B.1.1. The

Roofline model was recently used to show that artificial neural networks are heavily compute-bound (Castelló et al., 2019). We applied the Roofline model to the Blue Gene/Q and IBM Power 8 architecture and found that G-based current kernels in a detailed morphology model were bounded by memory bandwidth, while the analysis for state kernels did not reach a satisfying conclusion (Ewart et al., 2015; Ovcharenko et al., 2015). The ECM model (Hofmann et al., 2017; Treibig and Hager, 2010) has a similar strategy of prioritizing inference over prediction capability, but introduces several optimisations that make it more suitable for streaming kernels. All our shared-memory performance models will be based on the ECM formalism, which we describe in detail in Section 3.1 and Appendix.

2.3.2 Performance modelling of distributed applications

A common way to enhance the performance of simulations of brain tissue is to implement a distributed algorithm. This simulation strategy allows one to distribute the dataset (i.e. the neurons) over multiple compute nodes without a shared physical memory. Distributed compute nodes, often referred to as ranks, can exchange information through message passing, a paradigm by which any compute node is allowed to send packets of information to any other compute node through a shared network connection. In this work we focus on implementations based on the Message Passing Interface (MPI) programming model, as it reflects the de facto standard in the HPC community. The process of sending a message across the network involves several steps, e.g. copying data to the network card, contacting the switches, routing the message through several hops in the network, ensuring the receiver is ready, and many others. All these steps have a performance cost associated with them, which represents a trade-off with the benefits gained from the parallelism enabled by distributed simulation approaches. Several approaches to modelling the performance of message-passing communication based on the classification of recurring patterns have been proposed (see e.g. Fortune and Wyllie, 1978; Karp and Ramachandran, 1990). While the characterisation of the type of communication pattern is an essential step in understanding the performance, is not sufficient to provide a quantitative prediction.

A fully black-box empirical approach enabled accurate performance predictions of MPI communication runtime at the cost of forfeiting an explicit, explainable connection with the underlying hardware (Calotoiu et al., 2013). This approach is based on automatically generating a scaling model by interpolating benchmark data with a set of candidate scaling functions, and considering that the best fit represents the scaling behaviour of that subroutine. While it has been demonstrated that this method possesses an excellent accuracy and generalizes well to large scales, it lacks in explainability, for example in relating the values of the fitted parameters to hardware characteristics. Other models based on interpolation have been proposed, with similar advantages and drawbacks (Kerbyson et al., 2001; Martinez et al., 2010).

At extremely large scales, subtle properties of the networking hardware and software can become bottlenecks, such as the maximum injection rate, the network contention, synchro-

nization delays and others. In these cases, a very detailed modelling of every phase in the communication process could be warranted. Approaches based on Discrete Event Simulation (DES) have been developed which take as input the communication pattern and several hardware parameters such as the network topology or the connection bandwidth, and output a runtime prediction. Typically such models are implemented within the framework of a simulator software, because it is often infeasible to perform the necessary calculations by hand. Examples of such performance simulators include Performance Prophet (Pllana and Fahringer, 2005), POSE (Wilmarth et al., 2005), the more recent CODES (Mubarak et al., 2016) and others. While such methods provide a powerful predictive tool and explicitly take into account hardware features, they do not produce as output an analytic expression, thus once again hindering the explainability and interpretability of the results.

The SLOWER model is an example of an analytic approach focused on extreme scale simulations (Sterling et al., 2014). In SLOWER, performance is modelled as the product of four factors: efficiency, scalability, serial performance and rate of failure of resources. Another example is the PAL model, based on both architecture specifications and synthetic benchmarks (Kerbyson et al., 2001). Analytic performance models based on queueing theory have been used to model client-server interactions (Petriu et al., 1994), to predict load imbalance in workloads with a random component (Haring and Lüthi, 1996), to account for failure of processors (Weerasinghe et al., 2002) and to assess the cost/performance tradeoff intrinsic to parallel simulations (Falsafi and Wood, 1997) Most of these models require excessive parametrization, do not target the level of granularity of interest to us or are too specific to the target application. To model the performance of interprocess communication in this thesis we use the LogGP formalism (Alexandrov et al., 1997), which we explain in detail in Section 3.2 and Appendix B.4.

2.4 State of the art in empirical performance analysis of brain simulations

Performance analysis of *in silico* neuroscience simulations is an important topic of research, and has the potential to affect the feasibility and our understanding of future brain simulations (Einevoll et al., 2019). The fact that brain tissue is composed of millions of physically and logically separate entities (neurons) that communicate through brief bursts of neurotransmitter release makes brain tissue simulations the ideal candidate for distributed memory computing. Distributed memory parallelism has been implemented in all brain tissue simulation software packages (see e.g. Akar et al., 2019a; Ananthanarayanan and Modha, 2007; Goddard and Hood, 1997; Kozloski and Wagner, 2011; Kumbhar et al., 2019b; Migliore et al., 2006; Plesser et al., 2007; Zenke and Gerstner, 2014). In scaling to very large networks of neurons, *in silico* neuroscientists soon realized that the interprocess communication was becoming an important bottleneck, so simulation strategies using non-blocking communication (Ananthanarayanan and Modha, 2007), explicit-implicit numerical schemes (Kozloski and Wagner, 2011) and selective sending (Hines et al., 2011) were introduced. The topic of scalability is still being actively discussed now (Fernandez Musoles et al., 2019; Simula

et al., 2019). A recent study on the theoretical limits of scalability concluded that brain-scale simulations could be impossible on processor-based simulators (Végh, 2019). Finally, more efficient representations of the connection infrastructure (Jordan et al., 2018; Kunkel et al., 2014) and techniques for assembling the network in parallel (Ippen et al., 2017) have proven to be essential in scaling simulation code to petascale and exascale regimes.

In the shared-memory framework, parallel threads execute the simulation algorithm concurrently while having access to the same memory locations. Hybrid-parallelism, i.e. both distributed and shared memory, is implemented into all state of the art simulators (Akar et al., 2019a; Kumbhar et al., 2019b; Migliore et al., 2006; Plesser et al., 2007; Zenke and Gerstner, 2014), while few projects only support shared-memory parallelism (Stimberg et al., 2014; Vitay et al., 2015). Shared-memory performance is of great interest in many use cases, ranging from embarrassingly parallel parameter fitting experiments in detailed neurons (Bhalla and Bower, 1993; Van Geit et al., 2016), simulations of medium-sized plastic networks (Zenke and Gerstner, 2014), large-scale simulations of point neurons (Kunkel and Schenck, 2017) and code-generation for detailed kernels (Kumbhar et al., 2019a). Since memory bandwidth is a shared resource, as the parallelism increases it has the potential of becoming a bottleneck. For point neuron simulations memory bandwidth was shown to be a major factor for performance and it was speculated to impose a severe limit to the strong scaling of plastic networks (Fox, 2013; Zenke and Gerstner, 2014). For compartmental neurons, their large memory traffic requirements entail that they are particularly affected by bandwidth saturation (see e.g. Kumbhar et al., 2019a,b), even in the case of sub-neuron level parallelism (Eichner et al., 2009).

As a preliminary empirical shared-memory performance study, we investigated the impact of the balance between memory bandwidth and computational requirements of detailed neuron simulations on IBM's Power8 architecture (Ewart et al., 2015). Our study found that over 80% of runtime was spent in computing the states and currents of ion channels and synapses, but that the performance profiles of these kernels was highly variable. We observed that peak Gflop/s performance could not be achieved on any kernel type, but while for current kernels we could impute this to a saturation of the memory bandwidth when using shared-memory parallelism, it remained unclear why the peak performance of state kernels could not be achieved.

Modern CPU architectures expose yet another level of parallelism at the core-level, in the form of Single Instruction Multiple Data (SIMD) vector units. Such units can perform multiples of the same instruction concurrently and within the domain of high-performance computing are often used to speed-up floating point operations. An analysis of the benefits of vectorisation has been carried out for multiple modern hardware architectures in the context of morphologically detailed models, showing that enforcing vectorisation with code generation can give great benefits in terms of performance even over auto-vectorized code (Kumbhar et al., 2019a). The kernels considered in their analysis, typical of the G-based formalism, represent a large factor of the overall performance of morphologically detailed neurons and, due to their intrinsic structure, are easily amenable to vectorisation. For point neuron simulations,

significant speedup of the vectorized implementation over the scalar version can be obtained although in this case significant programming effort was needed to conceive data structures that allowed efficient vectorisation, in particular for handling of synaptic events (Brette and Goodman, 2011). Strong scaling can be further improved in morphologically detailed models by splitting neuron chunks across different parallel threads or processes (Eichner et al., 2009; Hines et al., 2008). More recently a dramatic strong scaling speedup was obtained by exploiting asynchronous execution and a per-neuron minimum communication delay (Magalhães and Schürmann, 2019; Magalhaes et al., 2019a,b). Finally, modern CPUs are capable of improving performance using hardware level optimisations such as Out-of-Order (OoO) scheduling, Instruction Level Parallelism (ILP) and speculative execution (Hager and Wellein, 2010).

As hardware accelerators, specifically General Purpose Graphical Processing Units (GPGPUs), have become more widespread and easier to program, several simulators have tried to leverage the potentially large speedup promised by such devices, notably the NeMo library (Fidjeland et al., 2009), the code generation framework GeNN (Yavuz et al., 2016), aNNarchy (Vitay et al., 2015), and others (Scorcioni, 2010; Wang et al., 2011; Yudanov et al., 2010). We refer the interested reader to (Brette and Goodman, 2012) for a review of the state of the art. The performance analysis by Zenke and Gerstner (2014), despite being focused on a CPU implementation, concluded that accelerators or specialized hardware will be necessary to enable real-time simulations of medium-sized networks of plastic neurons. However, although GPGPUs posses very general computing capabilities, their efficient execution relies on specific programming and data access patterns, making them best suited to SIMD operations on coalesced data. Simulations on GPGPUs outperform their CPU equivalent for medium-sized networks, but not for small networks nor for very large ones (Yavuz et al., 2016). Moreover, the GPGPU speedup obtained on the more computationally intensive Hodgkin-Huxley model was larger than the speedup for the simpler Izhikevic model, and spiking activity also played a role, with quiet networks being most suitable for GPGPUs while irregularly firing networks yielding less performance improvement than expected (Yavuz et al., 2016). Other types of accelerators have been considered in simulation neuroscience, from co-processors such as Intel's Xeon Phi (Kumbhar et al., 2016) to Field Programmable Gate Arrays (FPGAs) (Cheung et al., 2012; Renaud-Le Masson et al., 2004). Such works are still in a preliminary exploratory phase, and although systematic reviews are being conducted (see e.g. Maguire et al., 2007) it remains unclear what are the best simulation strategies to fully exploit such heterogeneous architectures.

All the simulation examples presented so far have been executed on either general-purpose computers or hardware accelerators. In both cases, the underlying architecture has been inspired by the seminal work of Von Neumann (1993), who defined a fetch-decode-execute cycle where both the program and the data would be stored in a memory unit, conceptually and physically separated from the processing unit. This approach suffers from the so called *von Neumann bottleneck* (Backus, 1978), which consists of the fact that performance is dominated by the limited data transfer rate between the processing unit and the memory, compared to the amount of memory required for the execution. Inspired by the massively

parallel processing architecture of the brain researchers have developed neuromorphic architectures capable of solving several computing problems at high performance and low energy consumption (Indiveri et al., 2011; Nelson and Bower, 1990; Thakur et al., 2018; Zeki, 2015). Two approaches have demonstrated successful applications in the literature: the digital and the analogue approach. In the digital approach, a massively parallel architecture of relatively simple, low-energy digital cores is used. It should be noted that individual cores in this configuration typically still respect the von Neumann paradigm, while it is the massively parallel and asynchronous nature of the whole architecture that distinguished it from conventional approaches. One of the main proponents of the digital neuromorphic strategy is the SpiNNaker project (Sharp et al., 2011), which has been successfully applied to the simulation of a neural microcircuit composed of 80,000 neurons and 0.3 billion synapses (van Albada et al., 2018). However, a recent study concluded that careful GPGPUs implementations could compete with digital neuromorphic hardware in terms of energy consumption and performance, while HPC clusters remained the highest-performance but most energy demanding solution (Knight and Nowotny, 2018). Industrial research centres have also been interested in digital neuromorphic computing, such as IBM's TrueNorth chip, able to deliver a real-time simulation of 1 million neurons, 256 million synapses with an extremely low energy consumption (Cassidy et al., 2016, 2014) and Intel's Loihi chip (Denton et al., 2014), even though these efforts are more often aimed at solving machine learning tasks rather than simulating brain microcircuits. Analogue neuromorphic chips take the complementary approach of trying to emulate electrophysiological behaviours with electronic chips. The Spikey chip (Pfeil et al., 2013), based on an analogue representation of neural circuits, was able to achieve simulations faster than real time by a 10^3 factor (Wunderlich et al., 2018). Using a mixed analogue-digital approach, the Neurogrid (Benjamin et al., 2014) project achieved a real-time simulation of 1 million neurons, 1 billion synapses consuming just under 3 W of power.

3 Analytic performance modelling of neuron simulations

The contents of this chapter and related appendix are adapted from the following publications:

Francesco Cremonesi, Georg Hager, Gerhard Wellein, and Felix Schürmann. Analytic performance modeling and analysis of detailed neuron simulations. *The International Journal of High Performance Computing Applications*, 2019a. *In review*

Francesco Cremonesi and Felix Schürmann. Telling neuronal apples from oranges: analytical performance modeling of neural tissue simulations. *Neuroinformatics*, 2019. *In review*

and include figures from

Timothée Ewart, Stuart Yates, Francesco Cremonesi, Pramod Kumbhar, Felix Schürmann, and Fabien Delalondre. Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using morphologically detailed neurons. In *Proc. 6th Int. Workshop on Perfomance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015

Timothée Ewart, Francesco Cremonesi, Felix Schürmann, and Fabien Delalondre. Polynomial evaluation on superscalar architecture, applied to the elementary function e^x. *ACM Transactions on Mathematical Software (TOMS)*, 2019. *In review*

We have identified performance modelling as a tool to create a bridge between our descriptions of *in silico* models' algorithms and hardware properties. Our ultimate goal is to produce an analytic method able to predict the performance of a given simulation, but we believe that the challenges encountered in the process of building such a model represent an important driver to advance our understanding of the relationship between hardware, modelling abstractions, algorithms and performance. Therefore we present in this chapter the performance modelling methods that we have developed and the challenges we encountered in understanding the computational characteristics of brain tissue simulations. We divide our exposition in two

main sections: modelling of the shared-memory execution and modelling of the interprocess communication algorithm. As reference hardware architecture we consider the state-of-the-art HPC cluster available to the Blue Brain Project composed of Intel Skylake nodes connected via an Infiniband EDR fabric. An ECM model of the Skylake architecture had never been published before. Our main contribution is the first analytic performance model of brain tissue simulation algorithms able to cover multiple modelling abstractions as well as provide a direct explanation of the relationship with hardware properties.

3.1 Analytic performance modelling of shared-memory brain tissue simulation kernels

Among all the performance models we reviewed, we identify the Execution-Cache-Memory (ECM) as providing the level of granularity and accuracy that best suits our needs. However, several challenges prevent us from blindly applying the model as-is to our kernels. After a brief introduction to the core formalism of the ECM model and to the relevant hardware features of our reference architecture, we review these challenges and explain the process of extension and validation required to understand the shared-memory performance of *in silico* models and experiments. For a detailed explanation of the foundations of the ECM model, we invite the interested reader to consult Appendix B.1.

Core concepts of the ECM formalism The ECM model uses a *grey-box* mixed approach combining an analytic formulation with some phenomenological input, and outputs a runtime prediction at the granularity of individual clock cycles (Treibig and Hager, 2010). Since its introduction it has been refined and validated on modern Intel and AMD multicore architectures (Hofmann et al., 2017, 2018; Stengel et al., 2015). To compute the ECM performance model for serial execution one must first define several contributions to the runtime of a given loop, such as: the in-core execution time assuming data is already loaded in registers T_{OL}, the time needed to load data into registers from the L1 cache T_{nOL}, the data traffic time between caches T_{L1L2}, T_{L2L3} and the data traffic time from main memory T_{L3Mem}. Data traffic times are usually computed combining an estimation of the data traffic with the bandwidth of the relevant data link. T_{OL} and T_{nOL}, on the other hand, can be computed by hand but are typically extracted using code analysis tools such as Intel's IACA (Intel, 2017). See Appendix B.1 for details. These contributions must be combined to obtain two quantities: T_{core} and T_{data}, representing the time that the loop would spend in core execution if data were instantaneously available, and the time required to move the data across the memory hierarchy, respectively. One of the core assumptions of the ECM model is that these two quantities can overlap, therefore single-thread runtime predictions can be obtained using the formula

$$T = max(T_{core}, T_{data}). \tag{3.1}$$

Two shorthand notations are used to simplify the presentation of the model, one for the individual contributions

$$\{T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L3Mem}\}\,, \tag{3.2}$$

and one for the runtime prediction, assuming that the dataset fits in different levels of the cache hierarchy

$$\left\{T^{L1} \rceil\, T^{L2} \rceil\, T^{L3} \rceil\, T^{Mem}\right\}. \tag{3.3}$$

The ECM model is based on the full-throughput assumption, thus neglecting any latency effects in the execution. This assumption greatly simplifies the analysis by removing the need for an extremely detailed understanding of the execution flow while at the same time providing *insight through failure* for situations in which the program execution is the bottleneck. In this context, a particular kernel will be categorised a **core-bound** if $T_{core} > T_{data}$, and **data-bound** otherwise. Note that these definitions apply to the serial execution. To obtain a performance prediction for parallel execution, the ECM model assumes that performance scales linearly with the number of threads, until a bottleneck from a shared serial resource is used, typically the memory interface (Hofmann et al., 2015). The ECM also provides a formula for computing the **saturation point**, i.e. the number of shared memory threads at which saturation of the memory bandwidth occurs for a given kernel. Details for computing the ECM model for both the serial and parallel execution, as well as for computing several quantities of interest from the ECM model contributions, are provided in Appendix B.1.

Reference architecture: Skylake AVX512 Our reference architecture is the Intel(R) Xeon(R) Gold 6140 Skylake processor, whose most salient characteristics are presented in Table 3.1. The Intel Skylake (SKX) architecture presents a few peculiarities that had never been accounted for in an ECM model before our work. Aside from microarchitectural features, the two main novelties were the AVX512 vector registers and the L3 victim cache. We provide in Appendix B.2 a detailed explanation of the necessary steps to extend the ECM model to this new architecture. For validation and benchmarking we use the CoreNEURON implementation as reference (Kumbhar et al., 2019b). To measure relevant performance metrics such as data transfer through the memory hierarchy we used the likwid−perfctr tool from the well-established LIKWID framework (Gruber et al., 2018; Treibig et al., 2010). All the benchmarks in the rest of this work were executed multiple times under the same conditions (typically around 10 runs), and we define the error (or margin of error) as:

$$\text{error} = 100 \times \frac{\left|\text{predicted} - \text{median(measured)}\right|}{\text{median(measured)}}, \tag{3.4}$$

where the median is taken over the benchmark runs. While this definition of error is the most natural choice from a performance modelling perspective, we note that other definitions such as e.g. based on the maximum difference might be more suitable to time-critical applications.

	value	unit
CPU freq	2.3	GHz
Uncore freq	2.3	GHz
Peak DP performance (AVX512)	1324.8	Gflop/s
Mem BW	105	GB/s
LD/ST throughput	2 LD, 1 ST	1/cycle
L1-L2 BW per core	64	B/cycle
L2-L3 BW per core	2×16	B/cycle
AVX512 `exp()` throughput	1.5	cycle/scalar iter
AVX `exp()` throughput	3.5	cycle/scalar iter
SSE `exp()` throughput	6.7	cycle/scalar iter
scalar `exp()` throughput	15.1	cycle/scalar iter
scalar `exp()` latency	22.2	cycle/scalar iter

Table 3.1 – Hardware characteristics of reference architecture SKX.

Challenges in ECM modelling of brain tissue simulation kernels Computation of the ECM model is based on intimate knowledge of the underlying implementation and data structures, and must be validated thoroughly with a reference implementation. For this reason it was not possible to provide an ECM model for all the *in silico* models and experiments listed in Section 2.1. Instead, we decided to focus our analysis on three models chosen as representative of the three main families of algorithms summarised in Table 2.1: point neuron I-based Brunel, point neuron G-based Simplified, detailed morphology G-based Reconstructed. Within this restricted subset of *in silico* models we found that significant extensions and adaptations of the ECM were still required to obtain practical performance predictions as well as to analyse performance properties. For example many ion channel and synapse kernels in the G-based formalism are characterised by a large amount of indirect memory accesses, in particular for detailed models where different elements of the matrix can be modified by these kernels. In this case, we found that domain-specific knowledge about the data layout and modelling abstraction allowed us to introduce heuristics to estimate the impact of indirect memory accesses on performance (see Section 3.1.1 for a detailed description). Additionally, the unpredictable nature of synapse activation in the spike delivery kernels results in random accesses to memory. This data-access pattern falls squarely outside of the ECM's design space, nevertheless we show in Section 3.1.5 that using a combination of synthetic benchmarks and domain-specific knowledge about spiking patterns we can establish meaningful bounds and accurate predictions. Finally, knowledge about neuronal representations in detailed models allows us to establish the performance properties of the Hines algorithm for solving the linear system (see Section 3.1.2 for details).

Algorithm 1 Ion channel current kernel pseudocode.

```
for each i ∈ ion channel instances do
    v ← voltage from compartment                    ▷ indirect read
    e ← reversal potential of ionic species         ▷ indirect read
    m ← ion channel state
    g_ion ← conductance(m)
    I_ion ← g_ion(v − e)
    update ionic currents                           ▷ indirect write
    update diagonal matrix elements                 ▷ indirect write
end for
```

3.1.1 G-based kernels in a detailed neuron

To overcome challenges in the performance modelling of brain tissue simulation kernels we leverage domain-specific knowledge about the data layout and the relationship between modelled entities as a basis to define simplifying heuristics. Hereinafter, all our analysis will be based on the reference CoreNEURON implementation (Kumbhar et al., 2019b). As a first example we present in Figure 3.1 the salient features of data layout and algorithm for the simulations of morphologically detailed G-based neurons. General aspects of the modelling abstractions and simulation algorithm have already been described in Section 2.1.3 and Appendix A. Morphologically detailed neurons are modelled as a tree of unbranched sections, whose topology is represented by a vector of parent indices. Other relevant quantities such as the membrane potential and the tridiagonal sparse matrix are represented by double precision arrays with length equal to the number of compartments. More details about the matrix representation are given in Section 3.1.2. Additionally, ion channel-specific and synapse-specific quantities are held in separate data structures consisting of arrays of double precision values in Structure-of-Arrays layout (SoA), indices to the corresponding compartments and, if needed, pointers to other internal data structures.

Ion channels current kernels

Ion channel current kernels update the elements of the membrane equation matrix by computing contributions from the ionic current of different chemical species. All kernels of this type have a similar structure, which we present in Algorithm 1 as pseudocode. They are characterised by two main features: low arithmetic intensity and scattered loads/stores. The latter can present a modelling problem, but in practice we can obtain good accuracy using a few heuristics based on domain-specific knowledge. In particular, as a first approximation one can treat the indices in the voltage and ionic species arrays as perfectly contiguous (see _ni,_ion_idx in Figure 3.1 as a justification). To compute the ECM we use the code for the Im current kernel in Listing B.2 within Appendix B.3 as a representative example. In total, the kernel reads from four double and two integer arrays, and writes to six double arrays, leading to 136 B of overall data traffic per scalar iteration (this includes write-allocate transfers on store misses).

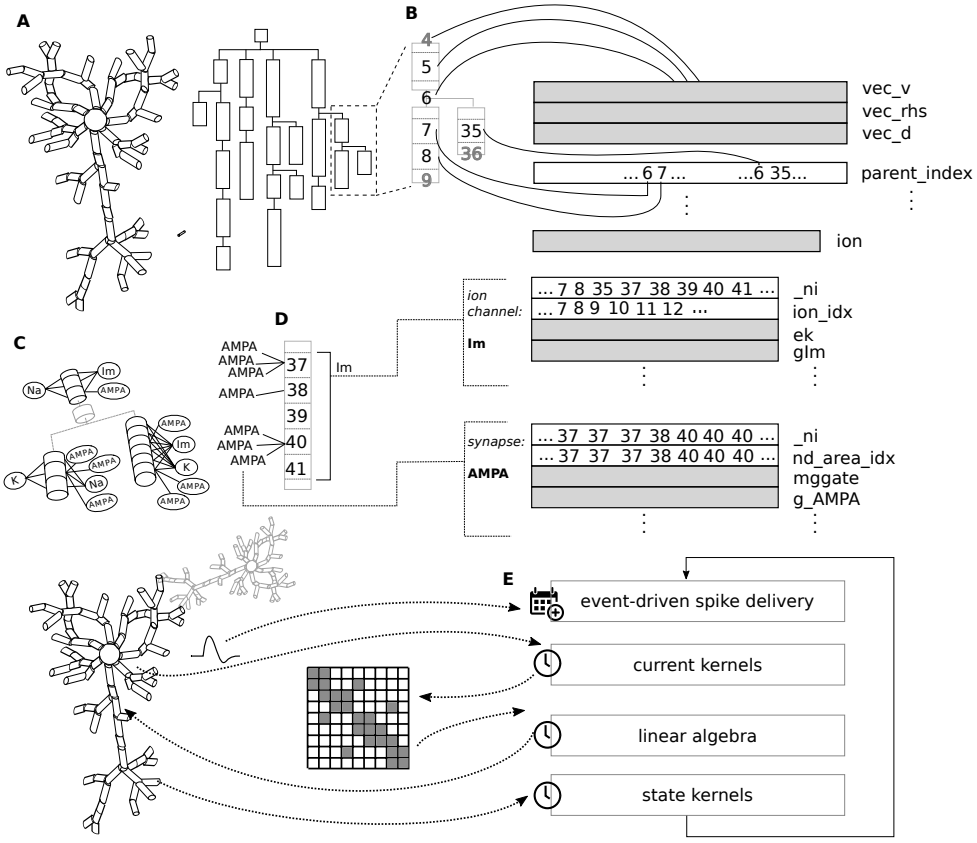

Figure 3.1 – Neuron representation and data layout in a morphologically detailed G-based model. **A**: Neurons are represented as a tree of unbranched sections, where each section can be further split into compartments for numerical discretisation **B**: Each compartment is numbered according to the schema in (Hines, 1984), and the tree structure is represented in memory by an array of parent_index. Additional arrays are used to represent the neuron's state (e.g. vec_v holds the membrane potential of each compartment), and three arrays are used for a sparse representation of the time integration matrix. Arrays of double precision values are coloured in grey, while arrays of integer indices are white and contain some elements to give an idea of their structure. **C**: Additionally, every compartment can be endowed with zero or more ion channels or synapses, which require additional arrays to be represented. Branching points (in grey) are treated as any other compartments for the purposes of linear algebra, but cannot have any instances of ion channels or synapses. **D**: Ion channels (e.g., Im) either have a single instance in all the compartments of a section, or do not have any instances at all in that section. Synapses (e.g. AMPA) can have multiple instances per compartment and do not need to be represented in all the compartments belonging to the same section. **E**: The application's workflow, excluding bookkeeping and parallel communication. First, the spike delivery kernel is called only for all the events that have been generated by other neurons and that have an effect on synapses of this neuron; then, at every timestep, the current, linear algebra and state kernels are executed. Current kernels read information from the state of the neuron and update the linear system's matrix. Linear algebra solves the linear system using a custom method and updates the state of the membrane potential. State kernels read the membrane potential and update the state of all the ion channels and synapses.

	contributions	predictions	measurements
SSE	{7.8 ‖ 5.5 | 2.1 | 5.5 | 3.0}	{7.8 ⌉ 7.8 ⌉ 13.1 ⌉ 16.1}	(n/a ⌉ 9.1 ± 0.1 ⌉ 11.0 ± 1.0 ⌉ 15.3 ± 1.0)
AVX	{7.3 ‖ 4.8 | 2.1 | 5.5 | 3.0}	{7.3 ⌉ 7.3 ⌉ 12.4 ⌉ 15.4}	(n/a ⌉ 8.7 ± 0.1 ⌉ 11.4 ± 0.0 ⌉ 15.0 ± 1.2)
AVX512	{5.3 ‖ 3.0 | 2.1 | 5.5 | 3.0}	{5.3 ⌉ 5.3 ⌉ 10.6 ⌉ 13.6}	(n/a ⌉ 7.6 ± 0.0 ⌉ 10.6 ± 0.8 ⌉ 15.6 ± 1.5)

Table 3.2 – ECM model and serial measurements per scalar iteration [cy/it] for the Im current kernel. Measurements are reported using the notation median ± IQR.

Combining the data volume estimates with in-core predictions from IACA (using the full throughput assumption) we generate the ECM model predictions for all levels of vectorisation in Table 3.2. For all predictions, we use cycles per scalar iteration as unit of measure. Analysing the assembly code we remark that the compiler is able to employ scatter/gather instructions for this kernel on SKX. As expected, the model predicts that the performance of this strongly data-bound kernel will degrade as the data resides farther from the core. Vectorization is not beneficial at all except for AVX512 with data in L1, which can be attributed to the required scalar load instructions when gather/scatter instructions are missing.

To validate the predictions we designed a serial benchmark that allowed fine-grained control over the dataset size by removing all ion channels and synapses except Im from our dataset, but still executing the complete application loop. The resulting dataset size was roughly 200 KB for the L2 benchmark and 7 MB for the L3 benchmark. Due to overheads, it was impossible to construct a benchmark for the L1 cache. We find that the ECM model delivers a satisfying degree of accuracy for all cache and vectorisation levels.

We conclude that the Im current kernel, and ion channel current kernels in general, are data-bound and limited solely by data transfer capabilities of the system across the memory hierarchy. Even for an in-memory dataset, wider data paths between the caches would thus improve the performance of the kernel. The clock frequency will have a significant but weaker than linear impact on the performance because memory transfer rates are only weakly dependent on it.

Synaptic current kernels

Synaptic current kernels are particularly important for performance, and pose a modelling challenge because of their complex chain of intra-loop dependencies, memory accesses and presence of transcendental functions. As with their ion channel counterpart, they are characterised by indirect accesses and high data-traffic requirements. Additionally, the excitatory AMPA/NMDA synapses considered here include in their conductance function a model for the dynamics of the Mg^{2+} block for the NMDA receptor which significantly increases the arithmetic complexity. We present the pseudocode of synaptic current kernels in Algorithm 2. To demonstrate the ECM model we use as an example the source code for the excitatory

AMPA/NMDA synapse current in Listing B.3 within Appendix B.3. The expensive exponential and divides in this code are balanced by large data requirements. The kernel reads one element each from eight double and two integer arrays, and writes one element each to nine double arrays, which would amount to a traffic of 216 B per iteration. However, as shown in Figure 3.1 the typical structure of the voltage and surface are index arrays is different from the ion channel kernels. In particular, as a direct consequence of multiple synapse instances being able to coexist within the same compartment, the voltage and surface area index arrays often exhibit sequences of repeated elements. This means that subsequent iterations of the kernel can exploit some temporal locality. To account for this we reduce the expected traffic from these arrays by a weighting factor equal to the average length of a sequence of repeated elements, which corresponds to the average number of synapses per compartment (3 in the case of our benchmarks). Thus the updated data traffic estimate is 205 B through the complete memory hierarchy.

To compute T_{OL} the inverse throughput of the vectorized exponential operation from Table 3.1 must be added to the kernel runtime reported by IACA, and T_{nOL} is derived from the retired load instructions as usual. We then obtain the ECM predictions per scalar iteration in Table 3.3. The analysis reveals a complex situation. The SSE code on SKX is predicted to be core-bound as long as the data fits into any cache, and data-bound when data fits in memory. The AVX and AVX512 code on SKX, however, become data-bound already in the L3 cache.

Again we used a benchmark dataset containing only synapses to validate the model, with a size of roughly 500 KB for the L2 benchmark and 11 MB for the L3 benchmark. The model predictions are optimistic compared to measurements by a 10–50% margin. Even though the predictions are not all within a small accuracy window, the model still allows us to correctly categorise the relevant bottlenecks, and is especially effective in capturing the fact that on SKX with AVX512 the kernel is rather strongly data-bound. Given the complex inter-dependencies between operations in the kernel, we speculate that a critical path might be invalidating the full-throughput assumption of the ECM model. As a result from the analysis we conclude that, for an in-memory dataset, the performance of the serial excitatory synapse current kernel would improve significantly only if in-core execution and data transfers were enhanced at the same time.

Algorithm 2 Synapse current kernel pseudocode.

for each $s \in$ synapse instances **do**
  $v \leftarrow$ voltage from compartment ▷ indirect read
  $m \leftarrow$ synapse state
  $g_{\text{syn}} \leftarrow$ conductance(m)
  $I_{\text{syn}} \leftarrow g_{\text{syn}}(v - e)$
  $a \leftarrow$ surface area of compartment ▷ indirect read
  $I_{\text{syn}} \leftarrow I_{\text{syn}} / a$
  update shadow vectors
end for

	contributions	predictions	measurements
SSE	{21.6 ‖ 9.9 | 3.2 | 8.3 | 4.5}	{21.6 ⌉ 21.6 ⌉ 21.6 ⌉ 25.9}	(n/a ⌉ 31.3 ± 0.1 ⌉ 31.4 ± 0.1 ⌉ 32.2 ± 0.0)
AVX	{13.5 ‖ 7.0 | 3.2 | 8.3 | 4.5}	{13.5 ⌉ 13.5 ⌉ 18.5 ⌉ 23.0}	(n/a ⌉ 16.9 ± 0.1 ⌉ 17.0 ± 0.5 ⌉ 23.9 ± 3.5)
AVX512	{7.2 ‖ 3.5 | 3.2 | 8.3 | 4.5}	{7.2 ⌉ 7.2 ⌉ 15.0 ⌉ 19.5}	(n/a ⌉ 10.9 ± 0.1 ⌉ 13.5 ± 0.8 ⌉ 25.1 ± 1.9)

Table 3.3 – ECM model and serial measurements per scalar iteration [cy] for the excitatory synapse current kernel. Measurements are reported using the notation median ± IQR.

Algorithm 3 Ion channel state kernel pseudocode.

```
for each i ∈ ion channel instances do
    v ← voltage from compartment                ▷ indirect read
    compute voltage-gated rates
    solve ion state ODE based on rates
end for
```

Ion channels state kernels

During the execution of a state kernel, the state variables of an ion channel or a synapse are integrated in time and advanced to the next timestep. Ion channel state kernels follow a simple structure reported in Algorithm 3. Note that in this case the step of solving an ODE typically requires the evaluation of one or more exp and several div functions.

Ion channel state kernels are characterised by a very large T_{OL} contribution due to exponential functions and division operations, combined with low data requirements. This gives reason to expect a clearly core-bound situation. As an example, we compute the ECM model for the Im state kernel in Listing B.4. In analogy with the previous ion channel example, we treat the indices in the voltage array as contiguous. Therefore this kernel requires reading one element each from one double and one integer array, and writing one element each to three double arrays, amounting to a traffic of 60 B per iteration. On the other hand, the kernel needs three exponential function evaluations and eight divides, of which some might be eliminated by compiler optimisations (common subexpression elimination and substitution of multiple divides by the same denominator for a reciprocal and several multiplications).

Again combining the IACA prediction with measured throughput data for exp() (see Table 3.1) and the data delay we arrive at the ECM predictions per scalar iteration in Table 3.4. State kernels can be considered as the polar opposite of current kernels in terms of their computational profile, and the model predicts that their performance will be independent of the location of the working set in the memory hierarchy. According to the performance model these kernels are dominated by the throughput of the exp function and the eight divides, by comparable amounts; for instance, the SKX-AVX version spends 16 cycles in divides and another 10.4 cycles in exp(). No optimisations concerning the divides are done by the compiler, although the number of divides may be reduced to three by the methods mentioned above.

We validated our predictions with dataset sizes of 500 KB for the L2 benchmark and 5 MB for the L3 benchmark. Except for the AVX kernels, for which the accuracy is more than satisfying, the predictions are optimistic by between 15% and 35%. It must be stressed that when a loop is strongly core bound and has a long critical path, the automatic out-of-order execution engine in the hardware may have a hard time overlapping successive loop iterations. Since the ECM model has no concept of this issue, predictions' accuracy may be affected.

Despite all inaccuracies, the conclusion from the analysis is clear: faster exponential functions, wider SIMD execution for divide instructions and a higher clock frequency would immediately (and proportionally) boost the performance of the serial Im state kernel. In shared-memory scaling the AVX and AVX512 versions will be able to eventually hit the memory bandwidth limit, albeit at a larger number of cores than with the more data-bound kernels.

	contributions	predictions	measurements
SSE	{36.1 ‖ 6.0 | 1.4 | 3.2 | 2.0}	{36.1 ⌉ 36.1 ⌉ 36.1 ⌉ 36.1}	(n/a ⌉ 53.4 ± 0.2 ⌉ 53.4 ± 0.1 ⌉ 52.3 ± 0.0)
AVX	{26.4 ‖ 3.4 | 1.4 | 3.2 | 2.0}	{26.4 ⌉ 26.4 ⌉ 26.4 ⌉ 26.4}	(n/a ⌉ 29.9 ± 0.1 ⌉ 29.9 ± 0.1 ⌉ 28.8 ± 0.0)
AVX512	{12.1 ‖ 1.9 | 1.4 | 3.2 | 2.0}	{12.1 ⌉ 12.1 ⌉ 12.1 ⌉ 12.1}	(n/a ⌉ 18.6 ± 0.1 ⌉ 18.3 ± 0.1 ⌉ 19.0 ± 0.1)

Table 3.4 – ECM model and serial measurements per scalar iteration [cy] for Im state kernel. Measurements are reported using the notation median ± IQR.

Synaptic state kernels

Synapse state kernels have computational properties similar to ion channel state kernels, i.e., a dominating in-core overlapping contribution due to exponentials and divides, coupled with low data requirements. Their general structure is also similar, with one important difference: the state update of synapses does not depend on the membrane potential of the compartment. Algorithm 4 presents the corresponding pseudocode. Note that the step of solving an ODE typically requires the evaluation of one or more exp and several div functions.

As an example, we compute the ECM model for the excitatory AMPA/NMDA synapse in Listing B.10. This kernel reads one element each from four double arrays and updates one element each from four other double arrays, thus totalling 96 B of data volume per iteration. The ECM predictions per scalar iteration are listed in Table 3.5. An important observation to be made here is that using the AVX2 instruction set was crucial to obtaining good performance on Skylake. Indeed the exp function invoked by the AVX instruction set has a much worse throughput (despite having the same vector width) and thus would significantly degrade

Algorithm 4 Synapse state kernel pseudocode.

```
for each s ∈ synapse instances do
    solve synapse state ODE
end for
```

Algorithm 5 Linear algebra pseudocode.

```
for each c ∈ reverse(compartments) do                ▷ triangularization
    update diagonal element at parent location       ▷ indirect write
    update rhs at parent location                    ▷ indirect write
end for
for each c ∈ compartments do                         ▷ backward substitution
    update rhs of c using rhs of parent              ▷ indirect read
end for
```

the performance of this kernel. As expected, all other observations and conclusions are the same as for the ion channel state kernels in the previous section. For the same reasons of out-of-order scheduling, all predictions are optimistic by 20–30%.

	contributions	predictions	measurements
SSE	{34.8 ‖ 6.5 | 1.5 | 4.0 | 2.1}	{34.8 ⌉ 34.8 ⌉ 34.8 ⌉ 34.8}	(n/a ⌉ 45.7 ± 0.0 ⌉ 45.7 ± 0.0 ⌉ 44.9 ± 0.0)
AVX	{22.0 ‖ 3.8 | 1.5 | 4.0 | 2.1}	{22.0 ⌉ 22.0 ⌉ 22.0 ⌉ 22.0}	(n/a ⌉ 25.5 ± 0.1 ⌉ 25.5 ± 0.1 ⌉ 25.7 ± 0.2)
AVX512	{9.7 ‖ 1.7 | 1.5 | 4.0 | 2.1}	{9.7 ⌉ 9.7 ⌉ 9.7 ⌉ 9.7}	(n/a ⌉ 13.1 ± 0.1 ⌉ 13.4 ± 0.2 ⌉ 13.7 ± 0.2)

Table 3.5 – ECM model and serial measurements per scalar iteration [cy] for the excitatory synapse state kernel. Measurements are reported using the notation median ± IQR.

3.1.2 Hines solver

The most common approach for time integration of morphologically detailed neurons is to use an implicit method (typically backward-Euler or Crank-Nicolson) in order to take advantage of its stability properties for stiff problems. A linear-complexity algorithm was introduced by Hines (1984) to solve the quasi-tridiagonal system arising from the branched morphologies of neurons, by reducing it to a quasi-tridiagonal problem (Thomas, 1949). This algorithm is based on a sparse representation of the matrix using three arrays of values (vec_a,vec_b,vec_d representing the upper, lower and diagonal of the matrix, respectively) and one array of indices (parent_index). Throughout the simulation, the off-diagonal terms of the matrix remain constant, while the right-hand side (rhs) and the diagonal elements are updated at every timestep. The algorithm is structured in two main phases: triangularization and a backward substitution. We report the pseudocode in Algorithm 5.

We use the code shown in Listing B.6 as a basis for the ECM model. We must tackle a few challenges: indirect accesses make it difficult to estimate the data traffic, and dependencies between loop iterations could break the full-throughput hypothesis. Moreover, an optimised variant of the algorithm that exploits a permutation of node indices to maximise data locality is executed by default by the simulation engine (Kumbhar et al., 2019b).[1] For reasons of brevity

[1] See the open-source code at https://github.com/BlueBrain/CoreNEURON. This permutation of node indices can be disabled with the command line argument `-cell-permute 0`.

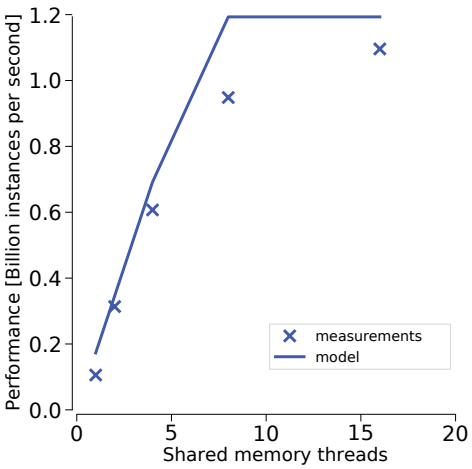

Figure 3.2 – Validation of ECM model for linear algebra kernel. Measured performance (markers) and predictions (lines) for the linear algebra kernel in Giga-compartments per second. Dashed lines represent the model predictions in the optimistic full-throughput scenario.

of exposition we restrict our analysis to this optimised variant of the solver.

In order to give a runtime estimate we examine two corner-case scenarios. The first, optimistic scenario assumes that indirect accesses can exploit spatial data locality in caches and thus do not generate any additional memory traffic. The combined data traffic requirements of triangularization and back-substitution then amount to reading one element each from four double arrays and two integer arrays, and writing one element each to three double arrays, i.e., 88 B per iteration. Considering the opposite extreme, it might happen that at every branching point the value of parent_index[i] is so much smaller than i that this generates an additional cache line of data traffic through the full memory hierarchy. We call this the worst-case branching hypothesis, in which we adjust the memory traffic predictions by assuming that every section boundary, i.e., the location of a potential discontinuity in the parent_index array, requires a full cache line transfer of which only one variable will constitute useful data. Considering that compartments are internally ordered to maximise data locality, we take the optimistic scenario as a basis for our predictions.

Even though the dependencies between loop iterations could potentially break the full-throughput hypothesis, considering that compartment indices are by default internally rearranged to optimise data locality we still use the full throughput as a basis for our predictions. It should be noted that indirect addressing and potential loop dependencies hinder vectorisation. IACA reports that the combined inverse throughput of triangularization and back substitution amounts to 8.12 cycle/scalar iter and $T_{nOL} = 6$ cycle/scalar iter. This leads to the runtime predictions in Table 3.6.

We measured the performance of the linear algebra kernel on a specially designed dataset with a very large number of cells and neither ion channels nor synapses, thus ensuring that the only

data locality effects are intrinsic to the algorithm and not a consequence of a small dataset. Our predictions based on the full-throughput hypothesis are validated by measurements of both the performance (see Figure 3.2) and the memory traffic (see Table 3.8). This kernel highlights very strongly an important feature of the SKX architecture: compared to previous generations SKX has a much better divide unit able to deliver one result every 4 cycles. This is reflected in the T_{OL} prediction, although the triangularization kernel on SKX is actually load bound by a small margin. The single-core median measurements are a little higher than predicted but also prone to some statistical variation; the best measured value is very close to the model. The only way to boost performance would be to enhance the performance of the memory hierarchy (in serial mode) or the memory bandwidth (in parallel). Having more than ten cores per chip would be a waste of transistors.

We remark that it remains unclear whether the node permutation optimisation is applicable in all cases or suffers from some constraints, and that our full-throughput predictions heavily rely on it. Therefore it may happen that, in some cases where it is impossible to reorder the nodes effectively, our predictions would only provide an optimistic upper bound on performance.

3.1.3 Clock-driven point neuron kernels

We now demonstrate how to extend the ECM model to clock-driven kernels from the Simplified and Brunel *in silico* models. For simplicity of exposition, from now on we only present the ECM model for AVX512 vectorisation, i.e. the maximum level allowed by our reference architecture. As with morphologically detailed neurons, domain-specific knowledge and an intimate understanding of the data layout are required to obtain an accurate performance description. We thus present in Figure 3.3 the characteristics of the G-based Simplified and I-based Brunel necessary to achieve a satisfactory performance model.

The G-based Simplified model is implemented in the following way. We create single compartment neurons endowed with a GIF process and 36 synapses as described in Section 2.1.3. We still make use of the vec_v,vec_rhs,... arrays to represent the state of each neuron but now every entry in this array corresponds to a distinct neuron. In the current kernels step, contributions from the GIF and synapse instances update the vec_d array which can be thought of representing the terms of a diagonal matrix in which every row corresponds to a neuron. In the update vec_v step, a simple backward Euler step consisting of a single division operation updates the

contributions	T^{Mem}_{ECM}	measured T^{Mem}
$\{8.1 \parallel 6.0 \mid 1.4 \mid 4.0 \mid 1.9\}$	13.3	18.8 ± 5.3

Table 3.6 – ECM model and serial measurements [cycle/scalar iter] for the linear algebra kernel. Vectorization levels are not considered because indirect write accesses prevent vectorisation. Measurements are reported using the notation median ± IQR.

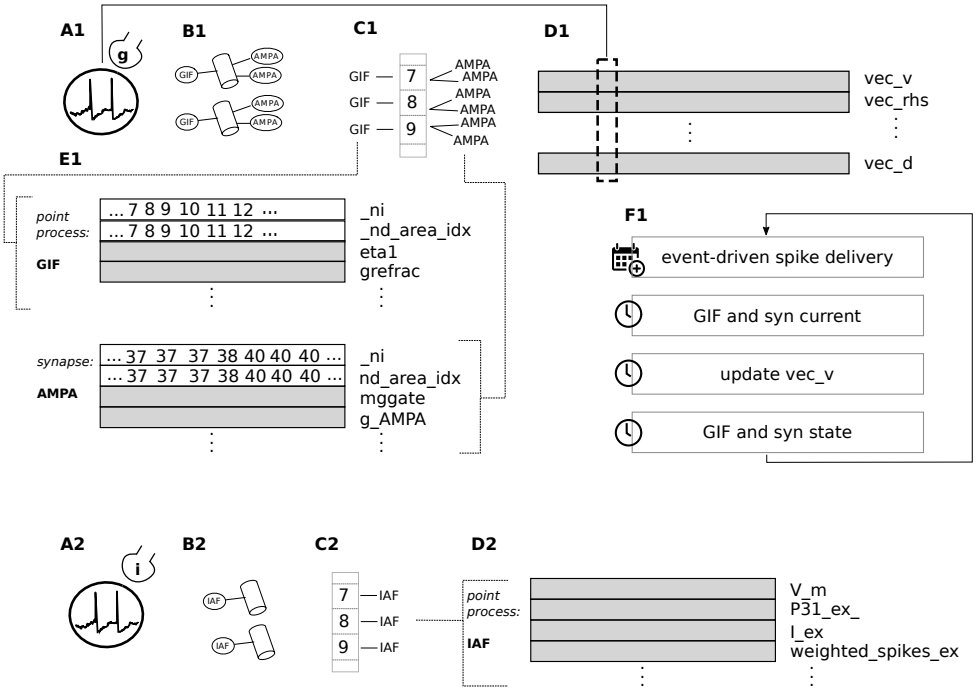

Figure 3.3 – Neuron representation and data layout in point neuron models. **A1** G-based Simplified model. **B1** single compartments are endowed with the GIF model and the appropriate number of synapse instances. **C1** each compartment contains exactly one instance of the GIF model, and 36 synapse instances. **D1** the vec_v (and similar) arrays are still used, such that the voltage of the neuron is not contained within the GIF data structure, bur rather in the vec_v array. **E1** GIF and synapse instances are represented by index and double precision arrays. **F1** The G-based simulation workflow, which, as opposed to the detailed model, does not require the resolution of a linear system.
A2 I-based Brunel model. **B2** compartments are endowed with a single instance of the IAF model. The vec_v (and similar) arrays are not used. **C2** each compartment contains exactly a single instance of the IAF model, and nothing else. **D2** the state of the neuron is wholly contained in the IAF data structure. No index arrays are required. The implementation details of the I-based simulation workflow are not represented as they are indistinguishable from Figure 2.2.

voltage of the neuron, and in the state kernels step the auxiliary GIF state variables and the synaptic state variables are integrated in time. We summarise the clock-driven portions of the Simplified model in Algorithm 6. For the sake of brevity we do not analyse in detail the performance of the modified AMPA/NMDA synapse presented in (Rössert et al., 2016) as it bears a high degree of similarity with the original AMPA/NMDA synapse analysed in Section 3.1.1, but we still report its ECM model in Table 3.7 and the reference code in Listing B.9.

To compute the ECM model of GIF kernels we use as reference the code provided in Listings B.7 and B.8. The GIF current kernel requires reading from 12 double precision and 2 integer arrays and writing to 8 double precision arrays, for a total traffic of 232B per scalar iteration, while IACA reports a $T_{OL} = 6.9\text{cycles}$ and $T_{nOL} = 3.4\text{cycles}$ per scalar iteration. The GIF state kernel requires reading from and writing to 6 double precision arrays for a total traffic of 144B per scalar iteration, while IACA reports $T_{OL} = 19.5\text{cycles}$ to which we must add the cost of 6 exponentials and $T_{nOL} = 6.3\text{cycles}$. The steps for computing the ECM model for the modified AMPA/NMDA model are similar to those reported before, although we note that in this case we know the exact length of repeated indices, which is equal to the fixed value of 36, i.e. the number of synapses per neuron.

We base our implementation of the I-based Brunel model on the iaf_psc_alpha model of the NEST simulator (Peyser and Schenck, 2015). For consistency reasons we still create single-compartment neurons endowed with an IAF process, but we do not rely anymore on the vec_v,vec_rhs,... arrays to describe the membrane potential. Instead the whole state of the IAF neuron is contained in the IAF data structures. In the neuron update step and PSC contributions step (see Figure 2.2) we simply update the relevant variables. An exact time-integration method is implemented exploiting the time-invariance of the coefficients of the ODEs that describe the neuron's evolution, such that several compute-intensive functions can be precomputed (Diesmann et al., 2001; Rotter and Diesmann, 1999). We neglect refractory periods that follow a spike emission in this work, due to their infrequent occurrence under physiological conditions and relatively low impact on performance. The pseudocode summarising the main algorithmic steps is reported in Algorithm 7.

To compute the ECM model we use as reference the code provided in Listings B.13 and B.13. The neuron update kernel requires reading from 11 and writing to 5 double precision arrays, for a total traffic of 168B. IACA reports $T_{OL} = 2.4\text{cycles}$ and $T_{nOL} = 1.75\text{cycles}$. The PSC kernel, on the other hand, requires 80B of traffic and has $T_{OL} = 0.6\text{cycles}$, $T_{nOL} = 0.4\text{cycles}$.

Table 3.7 presents a summary of all ECM model contributions and predictions for all the clock-driven kernels in the three *in silico* models considered here. For simplicity we present here an aggregate of the ion channel and synapse kernels in the Reconstructed model, and refer to Chapter 5 for a detailed validation of individual types.

Algorithm 6 Pseudocode for clock-driven portions of Simplified model.

```
for each n ∈ neurons do                                   ▷ GIF current
    v ← voltage from compartment                          ▷ indirect read
    η, γ ← GIF state
    I_η ← current(η)
    I_GIF ← g_GIF v + I_η
    a ← surface area of compartment                       ▷ indirect read
    I_syn ← I_syn / a
    update vec_d
end for
for each n ∈ neurons do                                   ▷ update vec_v
    vec_v[n] /= vec_d[n]
end for
for each n ∈ neurons do                                   ▷ GIF state
    η ← solve ODE
    γ ← solve ODE
end for
```

Algorithm 7 Pseudocode for clock-driven portions of Brunel model.

```
for each n ∈ neurons do                                   ▷ IAF neuron update
    V_m ← precomputed update step
    I_syn ← exponential decay
end for
for each n ∈ neurons do                                   ▷ IAF PSC contributions
    I_syn ← PSC × spike counter
    clear spike counter
end for
```

	kernel name	T_{OL}	T_{nOL}	T_{L1L2}	T_{L2L3}	T_{Mem}	T_{ECM}^{L1}	T_{ECM}^{L2}	T_{ECM}^{L3}	T_{ECM}^{Mem}
B	iaf update	2.41	1.75	2.62	8.00	3.68	2.41	4.38	12.38	16.05
	iaf psc	0.62	0.38	1.25	3.00	1.75	0.62	1.62	4.62	6.38
S	synapse current	7.44	3.50	3.51	9.03	4.92	7.44	7.44	16.03	20.95
	gif current	9.88	3.38	3.75	11.00	5.26	9.88	9.88	18.12	23.38
	synapse state	13.31	2.62	2.00	5.50	2.80	13.31	13.31	13.31	13.31
	gif state	28.50	6.25	2.38	6.50	3.33	28.50	28.50	28.50	28.50
R	synapse current	7.20	3.50	3.21	8.33	4.50	7.20	7.20	15.04	19.54
	ion channel current	4.68	2.56	1.92	5.07	2.70	4.68	4.68	9.55	12.25
	linear algebra	8.10	6.00	1.40	4.00	1.90	8.10	8.10	11.40	13.30
	synapse state	9.70	1.70	1.50	4.00	2.10	9.70	9.70	9.70	9.70
	ion channel state	15.82	2.16	1.59	3.56	2.24	15.82	15.82	15.82	15.82

Table 3.7 – ECM model of clock-driven kernels of *in silico* models and experiments. For simplicity, we only report the model based on the AVX512 vectorisation. The ECM contributions and predictions are reported in cycles per scalar iteration. Horizontal lines distinguish the three *in silico* models: Brunel (B), Simplified (S), Reconstructed (R).

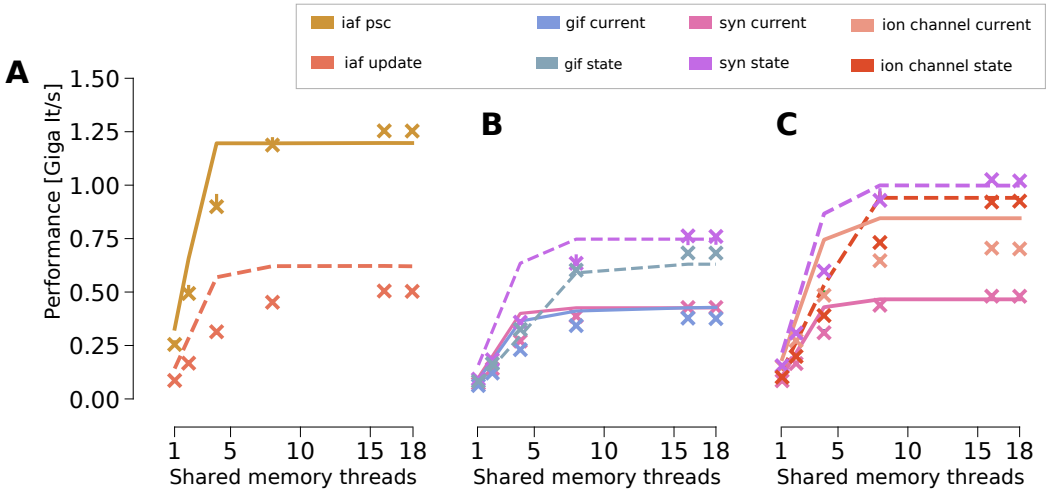

Figure 3.4 – Validation of performance model applied to clock-driven kernels of *in silico* models. Performance is measured as Giga iterations per wallclock second for a single instance of the kernel. The reference architecture is SKX with AVX512 vectorisation. Lines represent our predictions using the ECM model, while markers represent median benchmark measurements. Error bars represent the 25%-75% percentiles, but variability is so low that they are often hidden. To improve readability we used dashed lines for state update kernels and solid lines for current kernels. All benchmarks were designed with big enough datasets to ensure data was always coming from DRAM. **A,B,C** Clock-driven kernels of the Brunel, Simplified and Reconstructed model respectively.

3.1.4 Validation of clock-driven kernels

To validate our predictions we benchmarked and measured the serial and parallel runtime of the individual kernels in a simulation that is representative of a typical workload. Due to overheads it was impossible to design benchmarks for the L2 and L3 caches for the point neuron kernels described above. Therefore all the benchmarks presented here have a sufficiently large dataset to only fit in DRAM. For all *in silico* models we validated our predictions for the serial execution as well as for shared memory scaling. Validation results for the Brunel, Simplified and Reconstructed model are presented in Figure 3.4 and Table 3.8. An important aspect to take into account when reporting the benchmarked performance is dynamic frequency scaling, by which the CPU can dynamically throttle the frequency to reduce power consumption and prevent overheating. Even though we manually set the frequency using the LIKWID tool[2] the CPU is still allowed to throttle it in case of overheating, especially when the AVX512 units are being used. Therefore we scale all our performance measurements by the reported average frequency during the execution of that kernel.

In the serial case, the prediction errors are all within 20-30% of the measured runtime. Obtaining more accurate predictions is challenging, and the reasons for this can vary across kernels.

[2] By running the command likwid−set_Frequencies −f 2.3 −−umin 2.3 −−umax 2.3 before our benchmarks

		serial		parallel		memory volume	
	kernel name	pred [cycles]	meas [cycles]	pred [cycles]	meas [cycles]	pred [B]	meas [B]
B	iaf update	16.1	26.4±0.9	3.7	4.5±0.0	168	193.6 ± 0.9
	iaf psc	6.4	8.2±0.6	1.8	1.7±0.0	80	76.2 ± 0.8
S	synapse current	21.0	28.9±1.8	4.9	4.9±0.1	224	225 ± 1.7
	gif current	23.4	33.7±1.7	5.3	6.0±0.2	232	237.1 ± 12.7
	synapse state	13.3	21.1±0.4	2.8	2.8±0.1	128	127.7 ± 0.1
	gif state	28.5	25.4±0.0	3.3	3.1±0.0	144	123 ± 0.6
R	syn current	19.5	24.6±1.5	4.5	4.4±0.1	205	207.1 ± 2.1
	ion channel current	12.2	15.2±0.3	2.7	3.3±0.1	123	120.3 ± 11.0
	linear algebra	13.3	18.8±5.3	1.9	2.2±0.2	88	90.7 ± 4.2
	syn state	9.7	13.8±0.2	2.1	2.0±0.0	96	94.3 ± 1.3
	ion channel state	15.8	20.6±0.1	2.2	2.3±0.0	100	99.9 ± 2.0

Table 3.8 – Validation of ECM performance model for clock-driven kernels from all *in silico* models and experiments. Validation conducted using the SKX-AVX512 reference architecture. The parallel column refers to full-chip shared memory parallelism (18 threads). Measurements are shown as median values ± interquartile range from a dataset of 10 independent benchmark executions. Runtime measurements and predictions are reported in cycles per scalar iteration. Horizontal lines distinguish the three *in silico* models: Brunel (B), Simplified (S), Reconstructed (R).

For G-based state kernels a long critical path in the loop kernel code could be weakening the accuracy of our predictions due to a failure of the full throughput assumption, while errors in the ion channel current predictions could be imputable to memory traffic overhead due to indirect memory addressing. While it is still possible to obtain reasonably accurate predictions for the state kernels, it must be noted that ultimately it is extremely hard to predict the dynamic behaviour of the out-of-order engine in a complex, modern architecture. Finally, given that most of the predictions in this case are optimistic, it is reasonable to assume that performance limiting factors such as dynamic CPU throttling, as well as intrinsic factors such as critical paths and instruction latencies, could be impacting the performance negatively.

Digging deeper into the complexity of the computer architecture to obtain more accurate predictions is outside the scope of this thesis, whose purpose is to employ performance modelling as a means of explaining bottlenecks and predicting future trends, not to implement optimisations at the granularity of individual cycles. However, in a practical effort to optimise performance-critical portions of the code we investigated different strategies for computing the exp function based on polynomial evaluation (Ewart et al., 2019). We introduced the factorization pattern as a general way to decompose a polynomial in a product of subfactors, which under certain conditions could be computed independently if the architecture exposes ILP. Figure 3.5 shows the results of our benchmarks based on different factorization patterns and different polynomial evaluation algorithms. When we tried to predict the throughput and latency of each combination, we found it to be extremely challenging. We concluded that

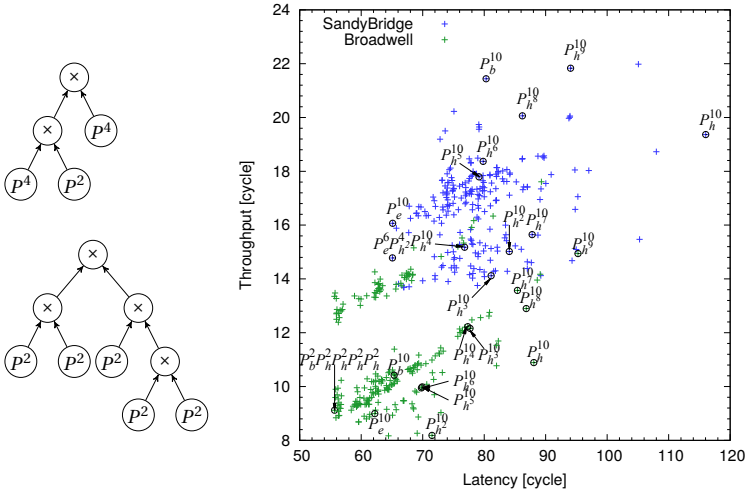

Figure 3.5 – Empirical assessment of factorization patterns for the evaluation of the exp function. Taken from (Ewart et al., 2019). We show two possible factorization patterns to compute a polynomial of degree 10 *(left)*. We benchmarked all possible patterns, evaluating each subpolynomial with different algorithms from the state of the art, on two Intel architectures *(right)*. Predicting the throughput and latency of each combination proved to be an extremely challenging task.

architectures have become so complex that an approach based uniquely on comparing the asymptotic complexity of the algorithm is not enough. Given the difficulty in predicting a processor's allocation of microarchitectural resources, benchmarks or code analysis tools are mandatory to determine the best performing solution with high accuracy.

3.1.5 Spike delivery kernels

The spike delivery kernel is arguably the most distinguishing aspect of brain tissue simulations, when compared to models of other biological or physical phenomena. In terms of algorithm design, all state-of-the-art software use some sort of priority queue or ring buffer to store synaptic events to be delivered within a timestep. For modelling and benchmarking, we separate the operations related to the bookkeeping of events inside the queue from the actual kernel execution, and we only consider the latter, because the scope of our analysis is restricted to computational and communication kernels.

When a spike is received, the postsynaptic process must integrate its effects in the state of the neuron or synapse. In I-based synapses this amounts to increasing a spike counter by the relative weight of the connection, as shown in Algorithm 8. For G-based synapses an equivalent quantal update of the synaptic states must be computed, a computationally expensive procedure since the *in silico* models considered in this work all contain a short-term plasticity model. As we did previously, we base our performance predictions on an actual implementation of these algorithms shown in Listings B.11 and B.14 in Appendix B.3 for the

G-based and I-based models respectively.

The spike delivery kernel is characterised by erratic memory accesses, because the order of activation of synapses is unpredictable. We always consider the worst possible case in which every spike to be delivered could not be cached and thus must come from main memory. This approximation has a strong impact on our estimates of the memory traffic and the scalability of the spike delivery kernel, notably in the strong scaling scenario, but we believe it represents a valid heuristic because of the very low activation of individual synapses. Indeed, given that physiological values of the firing frequency lie around 1 Hz, this means that each synapse would receive an event roughly once every 40000 time iterations, such that caches implementing a LRU policy would most likely have gotten rid of the corresponding cache line by then. Due to the erratic access, one is tempted to speculate that memory latency will be a dominant factor in the performance of this kernel. Upon deeper analysis, we find that spike delivery kernels are indeed affected by the latency of the memory system, albeit not dominated by it. The reason is that while the CPU is handling the delivery of one spike, it has potentially access to the information about the index of the next spikes to be handled, since it already read several values from the spike events array. Thus the CPU is in principle able to schedule as many memory accesses in advance as its queue of outstanding memory requests allows it to, partially hiding the latency of processing individual spikes. This is different from the classic purely latency bound kernels in which the CPU is only allowed to begin a loop iteration after the previous one is fully completed. On the other hand, all the requests for data are non-contiguous and therefore we expect that none of the pipelining and prefetching techniques are very effective in hiding the latency of fetching the data. In terms of memory traffic we consider two scenarios: a best-case one in which all synapses are activated in memory-contiguous order and a worst-case scenario in which synapses are activated in random order.

Best-case scenario: an optimistic upper-bound on performance In the best-case scenario we assume that the execution engine will be able to fully pipeline the execution and hide all

Algorithm 8 Spike delivery kernels: I-based spike delivery *(top)* and G-based spike delivery *(bottom)*.

```
for each e ∈ spike events do                          ▷ I-based spike delivery
    target, index ← e.info
    w ← weights[index]
    spike counter + = w
end for
for each e ∈ spike events do                          ▷ G-based spike delivery
    target, index ← e.info
    w ← weights[index]
    compute quantal update
    apply quantal update to state variables
end for
```

	contributions	T_{ECM}^{Mem}	CP
G-based	{57.8 ‖ 19.5 | 3.2 | 8.8 | 4.5}	57.8	123.4
I-based	{2 ‖ 1 | 0.7 | 2.3 | 1.0}	5	15

Table 3.9 – Best-case ECM model of the spike delivery kernel. Vectorization levels are not considered because indirect accesses prevent vectorisation. The CP prediction is actually for the Haswell architecture (see text for details).

latencies. Note that this scenario is highly unlikely to appear in practice, and serves more as a proof of concept and to establish a very optimistic upper bound on performance. Thus we base our performance predictions for this scenario on either the full-throughput hypothesis or a critical path. Given that the G-based delivery kernel requires a read-only transfer on seven double arrays, three integer arrays and one pointer array, and an update or write/write-allocate transfer on eight double arrays, we estimate a (best-case) memory traffic of 204 B per iteration. On the other hand, the I-based kernel requires a read-only transfer on one double array, three integer arrays and one pointer array, and a write transfer on one double array, thus we estimate its (best-case) memory traffic to be 44 B per iteration. From IACA we learn that the inverse throughput of the G-based kernel is 57.8 cycles/it, and T_{nOL} is 19.5 cycles/it, while for the I-based kernel we have $T_{OL} = 2$ cycles/it, $T_{nOL} = 1$ cycles/it. Similarly to the linear algebra kernel, indirect accesses prevent vectorisation. Under the full-throughput assumption, this leads to the single-thread predictions per iteration shown in Table 3.9. Given the complex chain of interdependencies in the kernel, we suspect that a critical path (CP) effect could also be present. A critical path here is defined as a sequence of instructions that, due to latencies induced by either their data dependencies or their usage of microarchitectural resources, invalidates the full-throughput hypothesis. This is an example of how insight can be gained through the failure of the standard ECM model to provide accurate performance predictions. Thus the CP values are also reported in Table 3.9.

In spite of the fact that this represents a particularly unlikely scenario, we designed a validation benchmark based on the G-based kernel. We placed several thousands of synapses on the same dendrite and activated them in order of instantiation, to ensure memory contiguity of data. For the parallel execution, we duplicated our dataset 16 times and used 16 threads. Assuming a CP-bound execution, this amounts to a parallel prediction of 7.7 cycle per scalar iteration, while the core-bound parallel prediction would be 4.5 cycle , since saturation would have occurred. We measured 122.1 ± 0.5 cycle for the serial execution, and 7.9 ± 0.1 cycle in the parallel benchmark. These results point to the fact that the G-based spike delivery kernel is CP-bound in the artificial best-case scenario.

Worst-case scenario: a realistic lower-bound In the worst-case memory access scenario we assume that a full cache line of data needs to be brought in from memory *for every data*

Algorithm 9 Synthetic benchmark mimicking the spike delivery access pattern.

```
spike event indices ← range(1,N)
random_shuffle(spike event indices)
initialize A,B arrays of size N
for each e ∈ spike events do
    i ← e.index
    A[i] = B[i]
end for
```

access. Note that because of write-allocate, every write operation counts as two accesses. To build a performance model, we are now tasked with figuring out which memory accesses are non-contiguous and thus represent a potential issue for performance. A first approach would be to consider that every memory access, except those directly indexed by the loop counter, could be non-contiguous. For example, in Listing B.11 this amounts to considering every memory access except `events[e],spike_event.target,spike_event.weight_index` as non-contiguous. For non-contiguous memory accesses, the predicted memory traffic is quite large because we assume that a full cache line (64 B) must pulled from the memory even though only a single double-precision variable (i.e. 8 B) would be required. Following this first approach, the G-based kernel would required 25 non-contiguous data accesses, amounting to a predicted worst-case memory traffic of $64 \times 25 = 1472$ B per iteration, while the I-based kernel, would require 5 non-contiguous data accesses, leading to a total traffic of 320 B per iteration. While this situation might be possible in theory, in practice we found that the measured memory traffic is consistently lower than the above estimates. One reason for this could be that memory accesses that are not tied to the synaptic instance, and thus are not directly performed in random order in our benchmark, are better handled by the CPU. Thus we consider an alternative hypothesis in which only accesses to synapse-specific data are non-contiguous. This amounts to only counting accesses indexed by `i` in Listing B.11 and Listing B.14. In this second hypothesis, the G-based kernel requires 22 non-contiguous data accesses for a total of 1408 B per iteration, and the I-based kernel a meagre 2 accesses for a total of 128 B per iteration. In benchmarks, we measure a memory traffic of 1396.9 ± 2.2 B per iteration for the G-based kernel, and 149.3 ± 11.9 B for the I-based kernel, thus lending high credibility to the second hypothesis that only variables tied to the synaptic instance represent a memory latency issue. Figure 3.6B shows a validation of the estimated memory traffic against realistic benchmarks.

Estimating the runtime proves to be a very challenging task. It is clear that the best-case scenario is very unlikely to happen in practice, and while the CP could still be a bottleneck, we suspect that memory latency may play an important role. One option would be to simply multiply the memory latency by the number of accesses, but as we explained above the spike delivery kernel is not fully bounded by memory latency. Measuring the memory latency of our SKX machine with Intel's MLC tool v3.6 (Viswanathan et al., 2018) yields a result of 93 ns, i.e. 214 cycles, which multiplied by the number of accesses would give a prediction of 5350 cycles

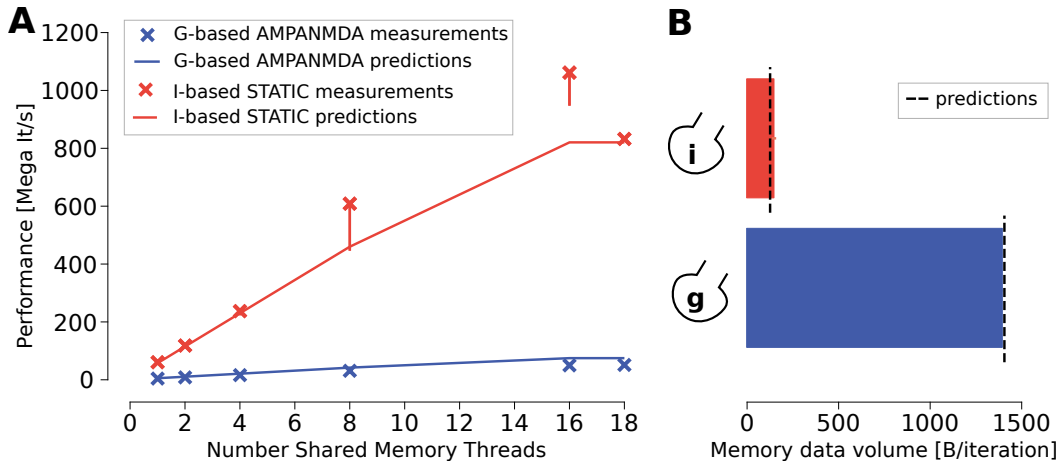

Figure 3.6 – Validation of the spike delivery kernel. **A** Performance predictions (dashed lines) and benchmark measurements (X markers). Performance is measured in Mega-delivered spikes per wallclock second. Error bars represent the 25% and 75% percentiles. **B** Validation of memory traffic per delivered spike. Bars represent measurements, dashed lines represent our predictions based on the worst-case scenario. Memory traffic is measured in B per delivered spike.

for the G-based kernel and 642 cycles for the I-based kernel. These predictions are far too pessimistic compared to the benchmark measurements, as will be shown below. Inspired by the STREAM approach (McCalpin, 1995), we designed a synthetic benchmark that mimics the memory access of the spike delivery kernel, without any computation, as shown in Algorithm 9. For completeness, we tested variations of the number of independent write streams, i.e. the code in Algorithm 9 has a single write stream to the A array. Our results indicate that there is extremely high variability in the memory traffic and runtime measurements until a certain critical size threshold is reached. Then the variability drops significantly, and the average latency appears to stabilize around a value of roughly 20 cycle per memory access, regardless of the number of write streams in the benchmark. We report the benchmark values in Table B.1 in Appendix B.3.

In the case of the G-based model, following the hypothesis that only the parameters tied to the synaptic instance represent a memory latency problem, the procedure above leads us to a single-thread prediction of $22 \times 20 = 440$ cycle per delivered spike, to be compared to the benchmark measurement of 571.6 ± 14.9 cycle, which represents roughly a 23% error. For multiple threads, we assume that the performance scales linearly with the number of threads until the bottleneck of memory bandwidth is exhausted. At the maximum number of threads, this amounts to a predicted runtime of 30.8 cycle per delivered spike, against benchmark measurements of 45.0 ± 0.1 cycles, i.e. a 31% error. For the I-based model, we predict a serial runtime of 40 cycle and measure 38.0 ± 1.8 cycles, giving a small error margin of 5%, while at maximum thread we predict a runtime of 2.8 cycle and measure 2.8 ± 0.04. The memory

traffic estimates and the runtime estimates are all within an acceptable margin of error for both models. Given the complexity introduced by the out-of-order execution and memory access scheduling, we deem these predictions quite satisfactory. In Figure 3.6 we present the predicted and measured performance and memory traffic for the spike delivery kernel.

3.1.6 Discussion

Within its design space, the ECM model yielded accurate predictions for the runtime of all the simulation kernels in *in silico* brain tissue models. It must be stressed that some of these kernels are rather intricate, with hundreds of machine instructions and many parallel data streams. This confirms that analytic modelling is good for more than simple, educational benchmark cases. We have also, for the first time, set up the ECM model for the Intel Skylake architecture, whose cache hierarchy differs considerably from earlier Intel server CPUs. Our analysis shows that the assumption of non-overlapping cache data transfers applies there as well, including all data paths between main memory, the L2 cache and the victim L3.

As expected, the modelling error was larger in situations where the bottleneck was neither streaming data access nor in-core instruction throughput. By making a few simplifying assumptions we were still able to predict with good accuracy the performance of a kernel with a complex memory access pattern and dependencies between loop iterations such as the tridiagonal Hines solver (Hines, 1984). In cases where the bottleneck is suspected to be the memory latency – such as the spike delivery kernels – the ECM model could only provide upper and lower bounds, but by combining its formalism with a synthetic benchmark that mimics a similar access pattern we were able to recover accurate serial and parallel runtime predictions. Investigating the details of how microarchitectural features handle the peculiar access pattern of the spike delivery kernel could be done via very detailed analytic modelling (see e.g. Tsuei and Yamamoto, 2002), but is outside the scope of this work. Overall, the ECM model was able to correctly identify the computational characteristics and thus the bottlenecks of all the kernels under analysis.

We found that G-based ion channel current kernels from the detailed model are data-bound while all state kernels are core-bound for all cache levels and all SIMD levels. The case of the excitatory synapse current kernel was special in that the kernel was core-bound as long as the dataset fits in the caches, but switched to data-bound when the data comes from memory. This effect was most prominent when using AVX512. Moreover, we observed that wider SIMD units were indeed capable of providing benefits in terms of reduced runtime but we also noticed diminishing returns as the SIMD units grew wider. The importance of high-throughput exp and div functions cannot be overrated, as was shown by a comparison with the Ivy Bridge architecture (Cremonesi et al., 2019a).

There are a few shortcomings that hinder the comprehensive applicability of the ECM model for all the kernels in brain tissue simulations. Long critical paths in the loop body, for example, reduce the validity of the full throughput hypothesis. In this case, a more detailed analysis

of the out-of-order execution, instruction-level parallelism and memory parallelism (see e.g. Levinthal, 2014) could significantly improve the accuracy and performance insight of the model. Automated code analysis tools such as the Open Source Architecture Code Analyzer (OSACA) are planned to become a versatile substitute for IACA, which does not provide CP predictions for modern Intel CPUs (Laukemann et al., 2018). Memory latency effects, which have been extensively discussed above, reduce the applicability of the ECM model and require special benchmarking to recover accurate predictions. Finally, runtime hardware modifications such as CPU throttling must be properly accounted for in benchmarking, and may hinder the practical relevance of our analysis for real-world simulations.

3.2 Performance modelling of interprocess communication

Among all the performance models for interprocess communication that we reviewed, the LogP model family meets our criteria of targeting small to medium clusters and providing a direct link with hardware features (Culler et al., 1993). The original LogP model was based on single-byte messages and point-to-point communication. The LogGP model is an extension that includes long messages and targets medium scale clusters up to thousands of parallel processes (Alexandrov et al., 1997). Several other extensions have been proposed, such as e.g. the LogGPO model in which the overhead term has a component that increases linearly with the message size (Chen et al., 2009), or the LogGPS model which takes into account performance penalizations due to synchronization of very large messages (Ino et al., 2001). Eventually the LogGOPSim simulator was also developed allowing to automatically obtain a performance predictions from a machine description file and a trace from an MPI execution as input (Hoefler et al., 2010). In this thesis we focus on the LogGP model and consider that all the other extensions to LogP fall outside of the scope of this work, although for generalizations to extreme scales such as whole brain simulations, the all-encompassing LogGOPS model might be a good candidate to overcome the fact that the LogGP's design space is limited to scales where synchronization is not an issue.

3.2.1 Modelling the interprocess communication in brain tissue models

Core concepts of the LogGP model In the LogGP model, the cost of sending a single message of size m B is given by two contributions: a latency contribution corresponding to the time it takes for the first byte of the message to reach its destination, and a bandwidth contribution corresponding to the throughput at which messages can be communicated through the interprocess network. The parameters in the LogGP model are:

- L is the network latency;
- o is the overhead from non-network operations;
- g is the inverse of the injection rate;

parameter	value	unit
L	1.54	μs
o_i	0.133	μs
o_s	4.59×10^{-5}	μs/B
g	0.526	μs
G	1.42×10^{-4}	μs/B
p_L	0.593	μs
p_G	1.875×10^{-4}	μs/B

Table 3.10 – LogGP parameters for Infiniband EDR with HPE-MPI. These values were obtained using the Netgauge tool (Hoefler et al., 2007a).

- G is the inverse of the network bandwidth;
- P is the number of processes involved in the communication.

One of the main insights in the LogGP model is that, under certain circumstances, CPU-side operations such as copying of data can overlap with network-side operation such as data sending. A detailed account of how to apply the LogGP model in a simple example is provided in Appendix B.4.

Reference architecture: Infiniband EDR with HPE-MPI We focus our validation and analysis to a representative example of an HPC network architecture: a vendor (HPE) optimised MPI implementation over an Infiniband EDR 100 GB/s fabric. We consider cluster sizes of up to $\sim 10^2$ distributed ranks to remain within the design space of the LogGP model. As discussed at the end of this chapter, using more complex models would allow us to generalise to even larger clusters, however we believe that our scaling conclusions would not be qualitatively changed by considering larger distributed clusters. We use the Netgauge tool (Hoefler et al., 2007b) to obtain the LogGP parameters reported in Table 3.10, although we found that a latency and bandwidth penalty – denoted p_L, p_G – were necessary to account for a degradation in performance for large messages. The reason for this degradation is unknown and was not automatically detected by the Netgauge tool, but since details of the underlying communication protocol are not published, one can make at best an educated guess about possible communication protocol switches. The reference architecture allows for both shared memory and distributed parallelism. Throughout this work, we maintain the nomenclature of shared memory threads and distributed ranks. When we use the generic term of parallel processes, we make the assumption that shared memory parallelism capabilities are always exhausted before distributed memory parallelism.

Spike exchange algorithm All the *in silico* models and experiments considered here are based on the Bulk Synchronous Parallel (BSP) model (Valiant, 1990), which prescribes a clear

Algorithm 10 Simulation algorithm with a focus on the spike exchange routine. This pseudo-code contains the full simulation algorithm, but abstracts away the computation steps in the call to a general function advance(neuron, t). The focus is instead on the parts relevant to spike exchange.

```
for t_0 = 0 ; t_0 ≤ T_stop ; t_0 += δ_min do
    clear spike_buf
    for t = t_0 ; t ≤ t_0 + δ_min ; t += Δt do
        for each neuron do
            advance(neuron, t + Δt)
            if neuron membrane potential crosses threshold then
                spike_buf ← (t + Δt, neuron ID);
            end if
        end for
    end for
                                                    ▷ Beginning of spike exchange
    n ← len(spike_buf)
    num_spikes ← MPI_Allgather(n)                   ▷ implicit barrier
    recv_buf ← MPI_Allgatherv(spike_buf,num_spikes)
                                                    ▷ End of spike exchange
end for
```

distinction between an on-node computation phase (happening in a distributed parallel fashion) and an inter-node communication phase. For brain tissue simulations, the inter-node communication phase corresponds to the spike exchange step in Figure 2.2. In all the widely-used state-of-the-art simulators, the spike exchange step is implemented by a blocking collective call, typically the MPI_Allgatherv operation. This entails that all the parallel ranks have, at the end of the communication step, knowledge of all the spikes produced by the simulation during the last min-delay period. Recent work has shown that at extremely large scales, this implementation can become prohibitively expensive in terms of memory requirements, and proposed to use instead the Alltoall operation to deliver spikes only to the ranks where they are required (Jordan et al., 2018). Other alternative implementations have been suggested, using non-blocking point-to-point communication (Ananthanarayanan and Modha, 2007) or spatial decomposition (Kozloski and Wagner, 2011). All these fall outside of the scope of our performance models, which are focused on medium-to-large cluster sizes and widely used software solutions.

We now describe the spike exchange algorithm using the CoreNEURON implementation (Kumbhar et al., 2019b) as reference, although we believe the main characteristics of this algorithm are quite general across different implementations. During the neuron computation loop, the state of neurons is advanced by a $\delta_{\min}$ interval, making small steps of size Δt. At each step, the membrane potential of neurons is checked, and if it crosses a threshold a spike event is issued. Each spike event is represented by a tuple (t, neuron ID), where t is the time at which the spike occurred and neuron ID is the identifier of the source neuron. These events are buffered in an array until the $\delta_{\min}$ boundary is reached, then all parallel processes reach a barrier where

they stop to synchronise. After all parallel processes have reached the barrier, the interprocess spike exchange begins. First all distributed ranks exchange the number of spikes in their local buffer via an MPI_Allgather operation. To account for the fact that different ranks might have different numbers of spikes, the actual spike information is then communicated using an MPI_Allgatherv operation. The receive buffer for this operation can be prepared just based on the knowledge of how many spikes will be sent from each rank, already obtained from the first step of the algorithm. We present in Algorithm 10 the simulation algorithm with a focus on the spike exchange routine.

Spikes are represented by the (t, neuron ID) tuple in widely used simulators such as NEURON (Carnevale and Hines, 2006), CoreNEURON (Kumbhar et al., 2019b), NEST (Gewaltig and Diesmann, 2007), and others. A common implementation is thus based on MPI compound data types to define a spike type as an aggregate of a double-precision variable representing the time of spike and an integer variable representing the ID of the source neuron. This implementation, however, can turn out to be unsatisfactory in terms of performance, in particular when one tries to predict the runtime of a collective communication, because hidden data shuffling and copy operations can take place (Carpen-Amarie et al., 2017). To address the performance issues deriving from custom data types, we reimplemented the spike exchange operation to perform two consecutive MPI_Allgatherv operations: the first on the array of timings and the second on the array of source neuron IDs. In our experience, this reimplementation provides both better performance and lower variability in benchmarks.

LogGP model of collective spike exchange The LogGP framework allows us to directly model the latency of point-to-point communication using the formula $(L+2o_i)+(G+2o_s)m$, where m is the message size. However, additional work is required to model collective communication because different algorithms can be used to disseminate the messages across the network during the Allgatherv operation. There are several implementations described in the literature and which one is used can be changed at runtime according to a complex function of the static network characteristics such as the topology as well as dynamic specifications such as the message size and number of ranks involved (see Thakur et al., 2005). The most widely-used implementations are: the *ring* algorithm in which the data from each rank is sent around in a virtual ring of processors, the *recursive doubling* algorithm in which a binary tree of pairwise exchanges is built, and the *Bruck* algorithm which generalizes the recursive doubling to situations where the number of ranks is not exactly a power of 2. Unfortunately the details of which algorithm is used and in which situation are proprietary information of the providers of the MPI implementation in our reference architecture, therefore we do not have access to such information. A review mentions however that the ring algorithm is the most commonly used, especially for large messages, and thus we base our predictions on this paradigm (Thakur et al., 2005). The formula for the LogGP model of the Allgatherv operation in the case of recursive doubling has been published and can easily be extended to the ring algorithm by noting that the total number of communicated bytes remains the same, while the latency term scales linearly instead of logarithmically (Mamadou et al., 2006) . Therefore,

we model the total latency to perform an Allgatherv operation among P parallel ranks with a total message size of m as:

$$T_{Allgather} = (P-1)(L+2o_i) + \frac{P-1}{P}(G+2o_s)(m-1), \tag{3.5}$$

where we make the simplifying assumption that each rank contributes an equal portion of data to the total message size. Therefore in our modelling and validation we always consider that the number of spikes communicated by each rank is roughly homogeneous, such that the above formula applies. In cases where there is significant variability in the number of spike communicated by each rank, formula 3.5 provides an optimistic lower bound on the latency, while a pessimistic upper bound could be given by:

$$T_{Allgather} = (P-1)(L+2o_i) + (P-1)(G+2o_s) \max_{r \in ranks} (m_r - 1), \tag{3.6}$$

where m_r represents the size of the portion of message sent by rank r.

Let us now consider a simulation of N neurons, each firing with average firing rate f, distributed over P parallel ranks. The total number of spikes generated by the whole network in a minimum delay period will thus be $Nf\delta_{\min}$. Following our observation that much better performance is obtained by splitting the communication of neuron IDs and the communication of spike times, we will thus model the spike exchange as two separate contributions: in the communication of neuron IDs $T_{comm,ID}$, each spike contributes $m_{ID} = 4\,\mathrm{B}$ to the total message, while in the communication of spike times $T_{comm,t}$ each spike contributes $m_t = 8\,\mathrm{B}$. Using formula 3.5 as the average case, this leads us to the performance modelling prediction of the spike exchange algorithm:

$$\begin{aligned} T_{comm} &= T_{comm,ID} + T_{comm,t} \\ &= 2(P-1)(L+2o_i) + \frac{P-1}{P}(G+2o_s)\left[Nf\delta_{\min}(m_{ID}+m_t) - 1\right]. \end{aligned} \tag{3.7}$$

Validation For validation, we first implemented a synthetic benchmark testing only the Allgatherv operation for different message sizes and different cluster sizes. In this case the underlying communicated type is not relevant, since only the total message size matters. Figure 3.7 presents the results of this synthetic validation. The distinction between smaller message sizes – where the performance penalties are not observed – and larger message sizes is quite clear from the data. The model has a good overall agreement with the measured data, rarely exceeding 10% relative error and often within the intrinsic variability of the data itself. Moreover, our assumption that only the *ring* algorithm is used is justified because the latency of small messages increases linearly, and not logarithmically, with the number of parallel ranks. This grants an acceptable degree of confidence in our predictions, at least for small to medium cluster sizes.

After this first validation, we tested our model in the actual simulation environment, by

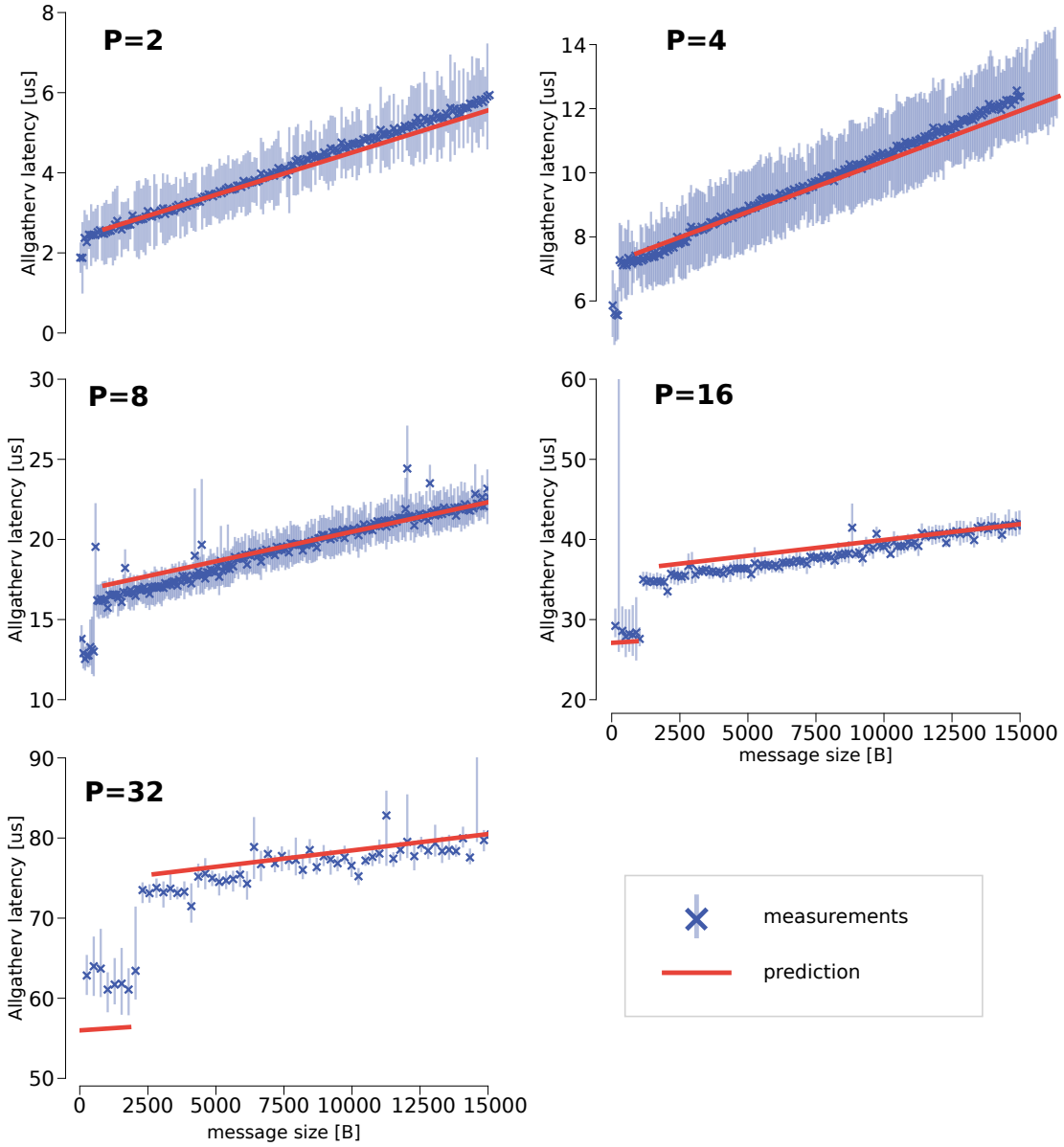

Figure 3.7 – Validation of LogGP model for interprocess communication using a synthetic benchmark. Measurements and LogGP predictions for a synthetic benchmark involving just a call to the MPI_Allgatherv communication routine with increasing message sizes. Note the different limits on the y axis. The distinction between small and large messages is clear, especially at larger numbers of parallel ranks, where there is a notable discontinuity in the measured latency according to B.26. Our model, based on the assumption that the ring algorithm is used, predicts the latency for both small and large messages with a reasonable degree of accuracy.

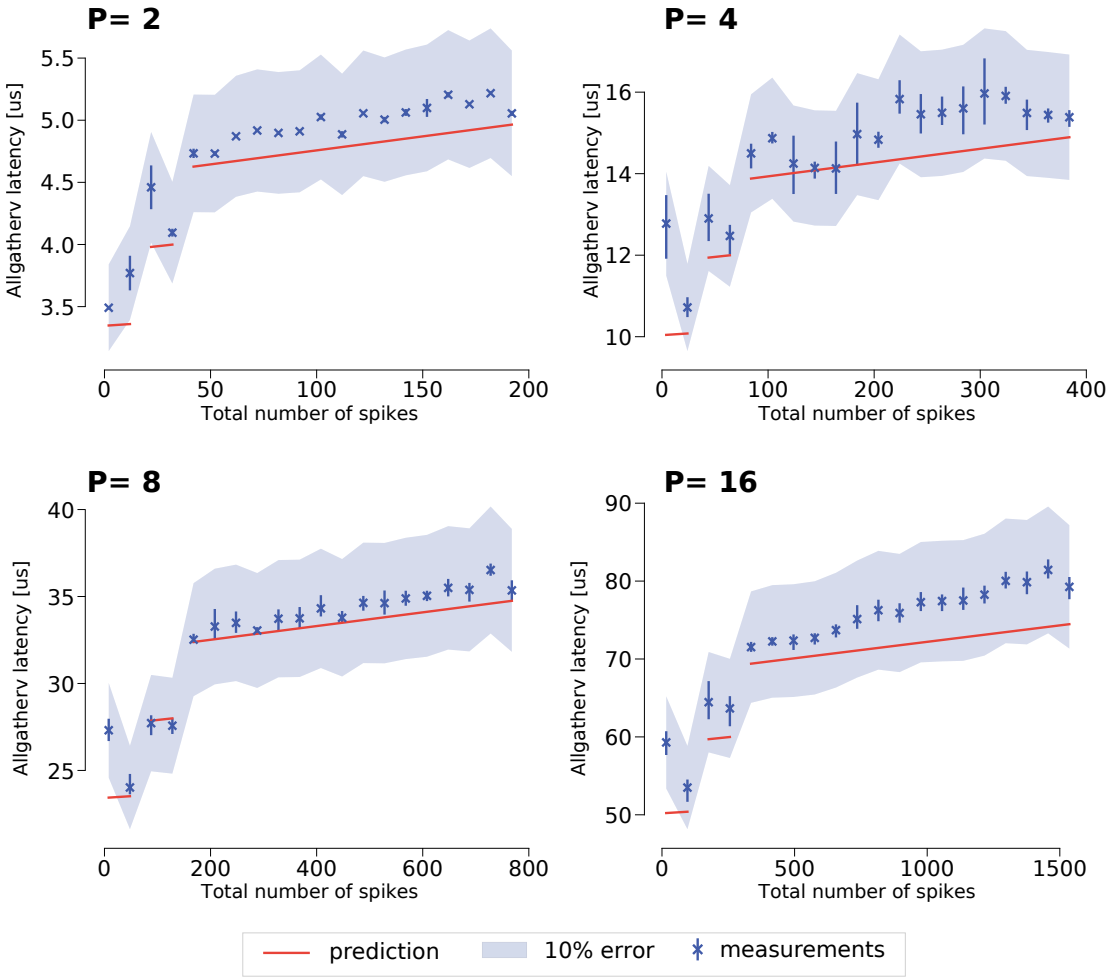

Figure 3.8 – Validation of LogGP model for interprocess spike exchange in the simulation environment. Weak scaling of a CoreNEURON simulation where individual neurons are contrived to emit a predetermined number of spikes per timestep. All neurons emit the same number of spikes. Measurements are obtained by summing the contributions from communicating the IDs of spiking neurons and the time of spiking $T_{comm,ID} + T_{comm,t}$, while predictions are split in three regions according to message size as described in the text. Red lines represent the model's predictions, while blue markers represent the measured latency averaged across parallel ranks. Error bars represent the minimum and maximum latency across parallel ranks. The shaded area represents a 10% error w.r.t measurements.

artificially contriving neurons to fire a predetermined amount of spikes at each time interval. Results are presented in Figure 3.8. Computation of the LogGP predictions in this case requires careful treatment of the message size. We identify three regions of interest: the first one where the total message size is so small that the condition ($m < 65P$) is satisfied for communication of both the neuron IDs (integers) and the spike times (doubles); a second region where the performance penalty is valid for the communication of the spike times, but not the neuron IDs; a third region where the performance penalty applies to both communication phases. Our model seems to have small optimistic bias, but overall it always falls within a 10% error region. Interestingly, the data seems to have much less variability within the simulation environment, compared to the synthetic benchmark in Figure 3.7. This could be due to the fact that small load imbalances caused by randomness would be mitigated in the process of calling Allgatherv twice.

Discussion The accuracy of the model's predictions have been validated both for a synthetic benchmark and within the simulation environment. This guarantees a high degree of confidence in our performance analysis contained in the coming chapters. However, a few considerations hinder the generality of our method. At large cluster sizes, synchronization and messaging protocol changes may impact negatively our predictions. We consider our validation sufficient to ensure that our quantitative and qualitative predictions hold a high degree of generality. However, it should be stated that extreme variations in hardware specifications or cluster size would require a complete reparametrisation or even an extension of the LogGP model. For example, if one wanted to extend our performance predictions to brain-scale simulations distributed over large-scale clusters, more complex models such as LogGOPS and possibly even an approach based on simulation (such as LogGOPSim) would be required (Hoefler et al., 2010; Ino et al., 2001). Moreover, we have only considered the implementation of spike exchange based on blocking collectives, but other strategies more suited to larger scales are being considered in the literature, based namely on non-blocking point-to-point or all-to-all collective communication (Ananthanarayanan and Modha, 2007; Jordan et al., 2018). An extension of our model to cover these strategies would provide insight on next-generation brain-scale distributed simulations.

For simplicity we have made the assumption that every rank must communicate the same amount of spikes at every minimum delay interval. While this considerably simplifies our analysis and provides an optimistic upper bound on performance, it is not realistic in practice. Issues of load imbalance and synchronization may considerably worsen the performance of the collective communication. However, without intimate knowledge of the underlying algorithm it is impossible to model the effect of differences in the communicated message size across ranks.

Finally, while the LogGP model is based on interpretable parameters that can be associated with hardware features, the link is less clear than in the ECM model for shared-memory execution. For example, the G parameter is associated with communication bandwidth,

such that its nominal vendor value for EDR 100 should be $8 \times 10^{-5} \mu s/B$. We measure $G = 1.42 \times 10^{-4} \mu s/B$, which is reasonably close, although it is not clear why there should be almost a factor 2x difference. On the other hand, for the o parameter the link with hardware is not so clear. While one might be tempted to associate it with memory bandwidth, the measured value is a factor 4.8x smaller than the nominal DRAM bandwidth, indicating that other hardware features might be more relevant.

Despite some of its shortcomings and limitations, we have presented a performance model able to capture the main hardware bottlenecks and provide accurate performance predictions in the context of brain tissue simulations distributed over clusters consisting of up to a few hundred parallel nodes. We based our predictions on the state-of-the-art implementation using blocking collective calls, at the core of most of the simulators available in the literature. Combined with our performance model of shared-memory single-node execution, we now possess a valuable tool to explain and predict the performance of brain tissue simulations over a wide range of modelling abstractions and cluster configurations.

4 Performance landscape of brain tissue simulations

The contents of this chapter are adapted from the following publication:

Francesco Cremonesi and Felix Schürmann. Telling neuronal apples from oranges: analytical performance modeling of neural tissue simulations. *Neuroinformatics*, 2019. *In review*

The large amount of research dedicated to understanding the performance of brain tissue simulations demonstrates the relevance of this topic to the community. We provide in this chapter a quantitative appraisal of the performance landscape of brain tissue simulations based on the modelling methods described in Chapter 3. Due to the high level of detail required by our performance model, we restrict our analysis to three *in silico* models and experiments representing the three broad categories of modelling abstractions: I-based point neurons, G-based point neurons and G-based detailed neurons. We thus pick the Brunel, Simplified and Reconstructed model from the list introduced in Chapter 2. While the Brunel and Simplified models are composed of neurons that share the same computational characteristics, to cope with the heterogeneity in the Reconstructed model we pick a Layer 5 thick-tufted pyramidal cell as representative of the computational complexity of a detailed G-based neurons. In Chapter 5 we will investigate further the variability of neuron models in the Reconstructed microcircuit. We deliver a detailed analysis of the relationship between an *in silico* experiment, the underlying neuron and connectivity model, the simulation algorithm and the hardware platform being used. The performance modelling framework developed in Chapter 3 enables us to expose current hardware bottlenecks and make projections for future brain tissue simulations in different scaling regimes corresponding to different real world scenarios. Our analysis allows us to not only verify common knowledge about performance bottlenecks, but also uncover potential future challenges and complications in scaling to the next generation of brain tissue simulations.

	flop	Traffic [MB]	Capacity [KB]	Communication [B]
Brunel	2.3×10^5	1.0	2.7×10^2	12
Simplified	1.7×10^7	1.3×10^2	10	12
Reconstructed	8.5×10^9	3.1×10^4	2.9×10^3	12

Table 4.1 – Classical performance metrics for *in silico* models. We report an estimate of the average number of flop per neuron to advance its state by 1 second of simulated time, including event-driven computations and the ion channel and synapse kernels in the G-based formalism. The memory Traffic is also defined as the data volume required to update the state of a neuron, and all of its synapses and ion channels, for an interval of 1 second of simulated time, while the memory Capacity is defined as the static memory footprint. The requirements in this Table are computed considering only the data structures strictly relevant to computation, thus neglecting overhead from implementation details such as MPI buffers, data structure representation, memory padding, etc. The Communication metric is defined as the size of the outgoing message emitted by a neuron firing at an average rate of $f = 1Hz$ during one second.

4.1 Analysis of the performance landscape

In contrast with their artificial counterparts, models of biological neurons can be characterised by a high level of complexity. We counted the average number of flop and data reads/writes required per neuron update, and used this to estimate the memory traffic. Memory capacity can also be roughly estimated by multiplying the number of state variables and synapses in a neuron by 8 B, the size of a double-precision variable. A detailed analysis of the memory footprint for the Brunel model has already been conducted (Kunkel et al., 2014). After applying memory-saving optimisations, they report a memory footprint of 1 KB per neuron object and 16 B per static synapse object, amounting to a total of 181 KB per neuron, reasonably close to the 270 KB we report with our rough estimate. We present in Table 4.1 these classical performance metrics that complement the hardware-agnostic approach introduced in Chapter 2. As expected, the requirements in terms of number of operations and memory traffic increase as the level of detail in the biological model increases, from I-based point neurons to G-based detailed neurons. Interestingly, the memory capacity requirements are smaller in the Simplified than in the Brunel model, which can be attributed directly to the Brunel's extremely high fan-in compared to the Simplified's optimised value of 36 synapses per neuron. Despite the interesting characterisation supported by these metrics, this approach is not sufficient to deliver a satisfactory characterisation of performance. In the remainder of this chapter, we dig deeper into these *in silico* models to reveal their fundamental performance properties using the methods described in Chapter 3.

Based on the characterization in Singh et al. (1993) we consider five different simulation regimes, arising from the combination of two axis. The first axis is defined by the strategy for parallelisation: serial, shared-memory and distributed-memory. The second axis is defined

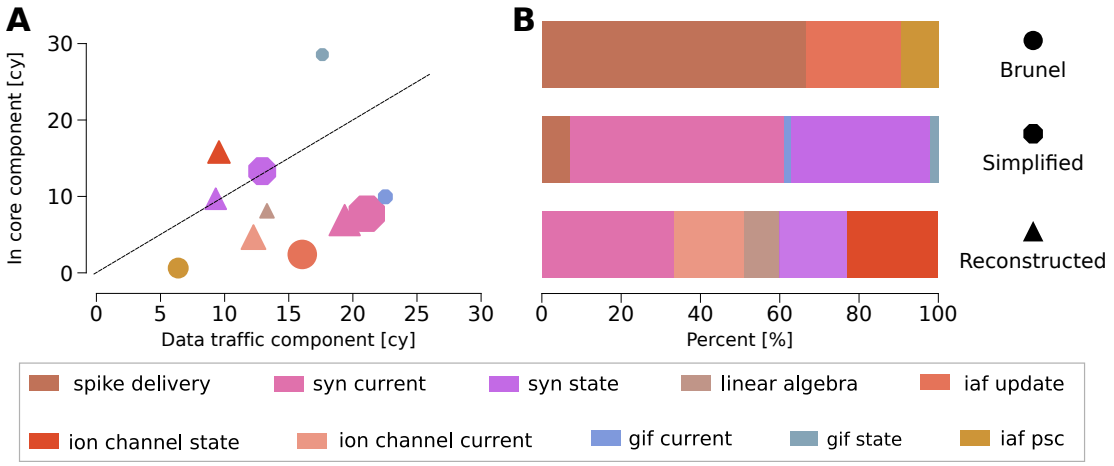

Figure 4.1 – predicted serial performance characteristics of computational kernels in brain tissue simulations. We predict the serial runtime of *in silico* models as a sum of their individual kernels on the reference SKX-AVX512 architecture. **A**: T_{core} and T_{data} components of the clock-driven kernels from brain tissue simulations. The dashed black line delineates the boundary between core-bound kernels (over the line) and data-bound kernels (under the line). Marker type denotes the *in silico* model whence the kernel is taken, while marker size is proportional to the relative importance of the kernel in the total runtime. **B**: breakdown of serial runtime in different kernels.

by the strategy for scaling the problem size with the available hardware: maximum-filling or constant problem size, as defined in (Singh et al., 1993) where maximum-filling corresponds to the memory-constrained approach. In the rest of this work, we will also refer to memory-constrained scaling as weak-scaling and constant problem-size scaling as strong-scaling. We make an additional distinction between the regimes: in max-filling we ignore the loop ordering optimisation and thus consider that simulation data must be fetched from main memory at every time iteration; in constant problem size we introduce instead this optimisation, thereby assuming that simulation data must be fetched from main memory only for the first iteration within each minimum delay period, and from the L3 cache for all other iterations. In this chapter the reference single-node architecture is Intel's Skylake processor with AVX512 vectorisation, considered to be a prototypical example of state-of-the-art HPC microarchitectures. For distributed simulations, the reference network architecture is a vendor-optimized HPE implementation of the MPI standard over an Infiniband EDR 100 GB/s fabric.

4.1.1 Serial regime

Analysis of the serial regime can reveal important insights into a kernel's computational properties. At first we look at the core-bound or data-bound profiles by defining the two

quantities:

$$\begin{aligned} T_{core} &= T_{OL}, \\ T_{data} &= T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem}. \end{aligned} \tag{4.1}$$

From the definition of the ECM runtime prediction (3.1) we know that a kernel will be core-bound if $T_{core} > T_{data}$, data-bound otherwise. Figure 4.1 shows a scatter plot of these two dimensions for the clock-driven kernels of the *in silico* models and experiments. Most kernels lie in the data-bound regime, with the only exceptions being the core-bound ion channel state kernels and the G-based synapse state kernels that lie on the boundary. To draw conclusions about the overall models, we need to intersect this information with the relative importance of the individual kernels on the overall runtime. The serial performance of G-based models is dominated by the state and current kernels, in roughly equal parts. Thus we conclude that G-based models are mainly data-bound. In the I-based model the most time consuming kernel is the event-driven spike delivery. This implies that, while the clock-driven portion of the I-based model is definitely data-bound, the serial performance of the whole neuron model is severely affected by memory latency.

4.1.2 Shared memory max-filling

One of the most common simulation configurations involves scaling the number of neurons until the memory capacity limit is reached. This configuration has been used as proof-of-concept for brain tissue simulations to the scale of brain regions and even the full brain and constitutes a fundamental tool for neuroscientists to simulate networks whose sizes are representative of the neural systems they are studying (Ananthanarayanan et al., 2009; Izhikevich and Edelman, 2008; Jordan et al., 2018).

Memory bandwidth limits shared-memory parallelism Modern architectures are typically designed with memory bandwidth as the most relevant bottleneck for shared-memory parallelism (McCalpin, 1995). This means that if all the shared memory parallel threads are used, it is very likely that performance will be bounded by the memory bandwidth. Indeed, this has been demonstrated to be the case for simulations of detailed neurons (Cremonesi et al., 2019a) and strongly suspected in the case of point neurons (Zenke and Gerstner, 2014). To verify the hypothesis that memory bandwidth could indeed be the main bottleneck in the maximum filling regime we compute the memory bandwidth utilization for the three *in silico* models considered in this work. Details for computing the memory bandwidth utilisation from the ECM model are explained in Appendix B.1.4. The results are shown in Figure 4.2A. We find that all models pass the threshold of 90% utilization well before all available parallel threads are utilized, meaning that memory bandwidth is indeed a bottleneck in the maximum filling scenario, under the assumption that data must be pulled from main memory *at every time iteration*. Moreover, this also implies that the parallelism exposed by the architecture cannot be fully exploited in the maximum filling regime. However, we surprisingly also find that,

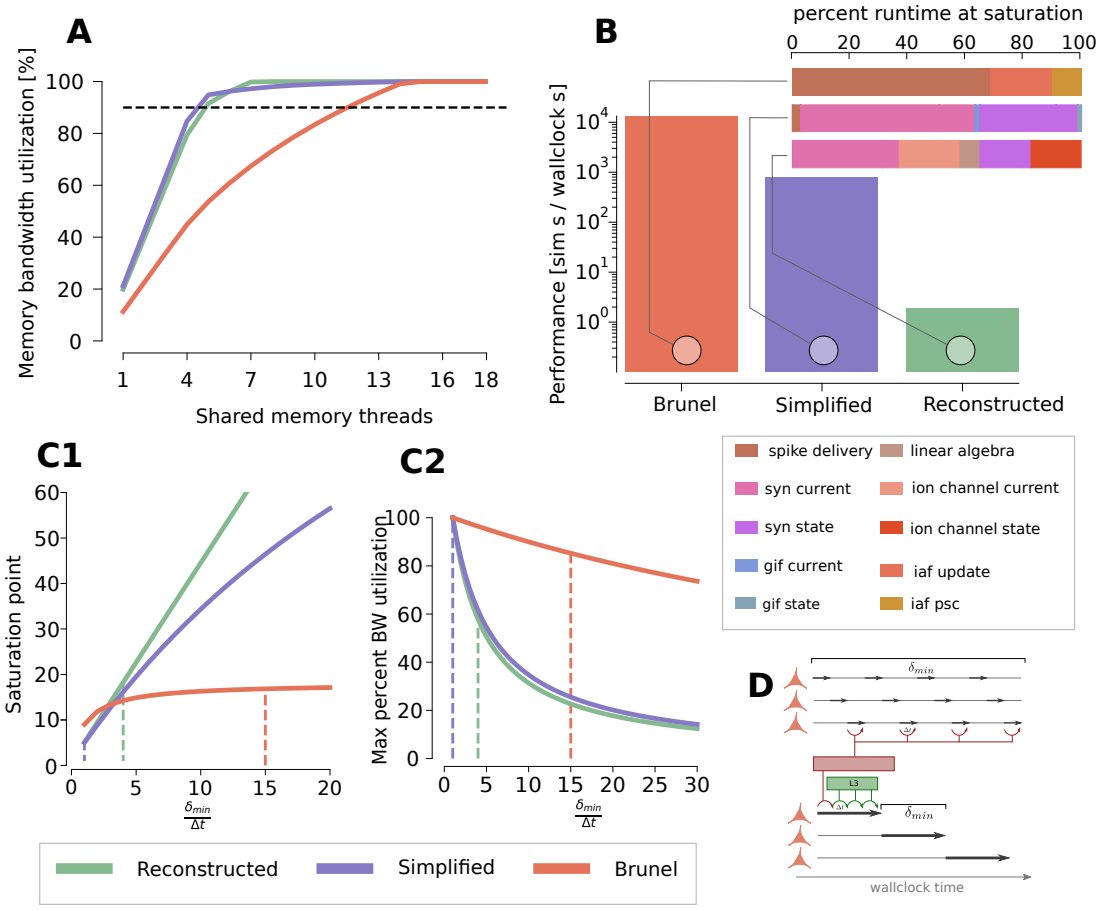

Figure 4.2 – Predicted shared-memory performance characteristics in the max-filling regime. We predict the shared-memory runtime of *in silico* models as a sum of their individual kernels on the reference SKX-AVX512 architecture. **A** Percentage of memory bandwidth utilization defined in (B.14) as a function of the number of shared memory threads. The dashed black line denotes the threshold of 90% utilization. **B** Predicted simulation performance measured in biological seconds per wallclock second per neuron. We assume complete *saturation of the memory bandwidth*. A performance prediction for a whole network simulation may be obtained by dividing the predicted performance per neuron reported here by the number of neurons in the network. Note the logarithmic scale on the y axis. The *inset* shows the percent of the total runtime required by each kernel within a model's simulation loop. **C1** To mitigate the effect of memory bandwidth saturation, a smart ordering of time and neuron loops is implemented by state-of-the-art simulators, as shown in the diagram on the right. We plot the number of threads required to reach saturation of memory bandwidth as a function of the coupling ratio. **C2** Bandwidth utilization when using the maximum parallelism allowed by the reference architecture, as a percent of the theoretical peak bandwidth, as a function of the coupling ratio. **D):** summarises the loop ordering optimisation to improve cache reuse described in Section 2.1.3. The top shows the naïve implementation where each neuron is advanced by a single timestep, while the bottom shows the optimised version in which a neuron is advanced by several timesteps until it reaches a $\delta_{\min}$ boundary, thus enabling cache reuse.

regardless of the level of morphological detail, G-based models share a similar pattern of early saturation while the I-based IAF model requires slightly more parallelism to achieve memory bandwidth saturation. In G-based models this is explained by the dominance of synaptic and ion channel current kernels which determine the early saturation pattern, whereas in the I-based model the saturation is driven by the memory latency effect on the spike delivery kernel.

State-of-the-art HPC memory chips can sustain fast simulations of the Brunel and Simplified models at full saturation From a practical point of view, in addition to analysing the scaling behaviour of simulations, computational neuroscientists are also interested in predicting the actual runtime for a given model. Therefore, we predict the simulation performance for the three *in silico* models under the assumption that *memory bandwidth is fully saturated*. The results are plotted in Figure 4.2B, and Table 4.1 reports the memory traffic requirements to simulate one second of activity, alongside the predicted memory capacity. As unit of measure for performance we chose simulated seconds per wallclock second per neuron, in order to present our results in a way that is independent from the network size and the duration of the simulation. Our results indicate that the modern, fast memory chips on the reference architecture could be able to sustain faster-than-constant problem size simulations of up to roughly 10^4 neurons in the Brunel model, and 10^3 neurons in the Simplified model, while faster-than-constant problem size simulations of the Reconstructed model are predicted to be theoretically possible only by a narrow margin, and in practice probably impossible. As an important remark, these predictions only consider the *computational and communication kernels* of an *in silico* model, and notably neglect event bookkeeping efforts, random numbers generation and computation of stimuli. In a real world scenario, the empirically measured performance of a model could be much smaller if the model's execution is not dominated by computational aspects.

Event-driven synaptic integration dominates I-based performance, while clock-driven kernels dominate G-based performance. We also predict the breakdown of relative importance of different kernels that constitute the simulation algorithm of a model, shown in the inset of Figure 4.2B, allowing us to highlight some interesting differences between models. In the maximum filling scenario the Reconstructed model is not dominated by a single kernel. Instead, synaptic and ionic current kernels constitute almost 60% of the execution time, while state update kernels, which are commonly regarded as more costly, are a few points short of taking up the remaining 40%. This has been extensively validated in our analysis and can be explained by the fact that current kernels have stronger data requirements and lower computational requirements, and thus poorer performance when the bottleneck is constituted by the memory bandwidth (Cremonesi et al., 2019a). The Simplified model is similarly dominated by the computation of synaptic current kernels. Since the Simplified and the Reconstructed model share the same G-based synaptic formalism, this can explain why some of their performance properties are very similar, in spite of the fact that their representation of neurons'

morphologies is extremely different. Conversely, the performance of the Brunel model is determined for more that 60% by a single kernel: the event-driven integration of synaptic events. In the following sections we explore in detail how this characteristic has an impact on determining which hardware feature is most relevant for the performance of *in silico* models.

Ordering of loops to avoid memory bandwidth saturation. State-of-the-art simulators employ a specific ordering of the loops over neurons, timesteps (Δt) and minimum network delay steps (δ_{min}) to minimise the impact of memory bandwidth by maximising data locality. This optimisation was explained in Section 2.1.3 and is summarised in Figure 4.2D. Throughout this work we make the conservative assumption that, when using the loop ordering optimisation, data must be fetched from main memory on the first timestep and from the L3 cache on consecutive timesteps. In special cases where the memory traffic requirements of a single neuron are very low and the number of synaptic events integrated in a timestep is sufficiently small, data could potentially come instead from higher levels in the hierarchy such as the L2 cache; however for the sake of simplicity we make the assumption that reused data must always be fetched from L3. The number of timesteps within a minimum delay period has of course a great influence on the effectiveness of this strategy in terms of reducing pressure on the memory bandwidth. To quantify this, we compute the number of threads to reach saturation – n_{satur} – as defined in (B.12) and plot the results in Figure 4.2C1 as a function of the coupling ratio defined by $\frac{\delta_{\text{min}}}{\Delta t}$. In G-based models, there is an almost linear relationship between the coupling ratio and n_{satur}, indicating that investigating ways to increase the coupling ratio could be highly beneficial for parallelism. Note that, in this regard, increasing the coupling ratio by decreasing Δt presents a performance tradeoff: it allows more parallelism but increases the computational requirements (number of iterations) of the model. Conversely, while the δ_{min} is obviously a fixed parameter of the network that cannot be arbitrarily changed, methods that experimented with a per-neuron delay, instead of a network-wide minimum delay, demonstrated significant speedup (Magalhães and Schürmann, 2019). The relationship between coupling ratio and n_{satur} for I-based models is bounded by a relatively small limit of roughly $n_{satur} \leq 17$, above which no additional parallelisation is predicted to provide any benefit. This is explained by the fact the spike delivery kernel, in virtue of its event-driven nature, is unaffected by the benefits of the coupling ratio. Since our assumption is that data for this kernel must always come from main memory, as soon as it becomes the dominating performance factor and it reaches saturation, it inhibits any benefit from parallelism. To provide an example more concretely tied to hardware, we also predict the maximum memory bandwidth utilization as a function of the coupling ratio. For G-based models this follows a steep decline for small values of the coupling ratio, indicating that even just a few timesteps exploiting data locality can provide great benefits in terms of shared memory scaling, while in the case of the I-based model this decline is much slower, which could explain why the published Brunel model (Kunkel et al., 2014) benefits from such a large value of the coupling ratio while the Reconstructed and Simplified models have relatively lower values (see Figure 2.6).

4.1.3 Distributed max-filling

To overcome the limit on network size imposed by the memory capacity of individual compute nodes, computational neuroscientists have begun executing parallel distributed simulations on a cluster of compute nodes. In the distributed maximum filling regime we still consider that the memory capacity limit of each individual compute node will be maxed out, but we introduce the possibility of increasing the number of interconnected compute nodes as a means of increasing the computational power and capacity of the architecture. In terms of scaling the problem size proportionally to the number of compute nodes, the fact that we assume the memory capacity limit to be always maxed out translates to the concept of *weak scaling*, on which we will base our analysis in this section. In this section we ignore the loop-ordering optimisation described above and assume instead that the memory bandwidth of individual compute nodes will always be saturated. In some sense, this represents an upper bound on the achievable performance using shared-memory parallelism. We will lift the memory saturation assumption in later sections.

Weak scaling properties and performance predictions of *in silico* models We predict the performance of distributed simulations in a weak scaling, maximum filling scenario for different numbers of neurons per rank and all *in silico* models, using our performance model. Results are presented in Figure 4.3A, where the solid lines correspond to 10^5 neurons per rank while the dashed lines correspond to a small number of neurons per rank, computed such that its memory footprint barely exceeds the 25MB of the reference architecture's L3 cache; smaller values would not be possible because they would break the memory bandwidth saturation assumption. As expected, for a fixed configuration and a small cluster size the Brunel model has the best predicted performance, beating by roughly a factor 10 the performance of the Simplified model and roughly a factor 10^4 the performance of the Reconstructed model. Interestingly, these differences are much less pronounced for large cluster sizes, where the difference between the Brunel and Reconstructed model can be reduced to less than a factor 10. Networks with the largest numbers of neurons considered here possess excellent weak scaling properties regardless of the underlying modelling abstraction, at least up to the cluster sizes considered here. Conversely, only small networks of the Reconstructed model retain the same quasi-optimal scaling behaviour, while both the Brunel and Simplified model suffer from performance degradation at large cluster sizes, with the average rate of degradation being significantly larger for the Brunel model.

Explaining weak scaling performance properties through bottleneck analysis Figure 4.3B explains the degradation of performance by identifying the most relevant bottlenecks for different configurations. Here a hardware bottleneck is defined as the most relevant hardware feature as computed by our performance model. The methodology for computing hardware bottlenecks is explained in detail in Appendix B.1.4. The first striking property revealed by this analysis is that network bandwidth is never a bottleneck for the *in silico* models

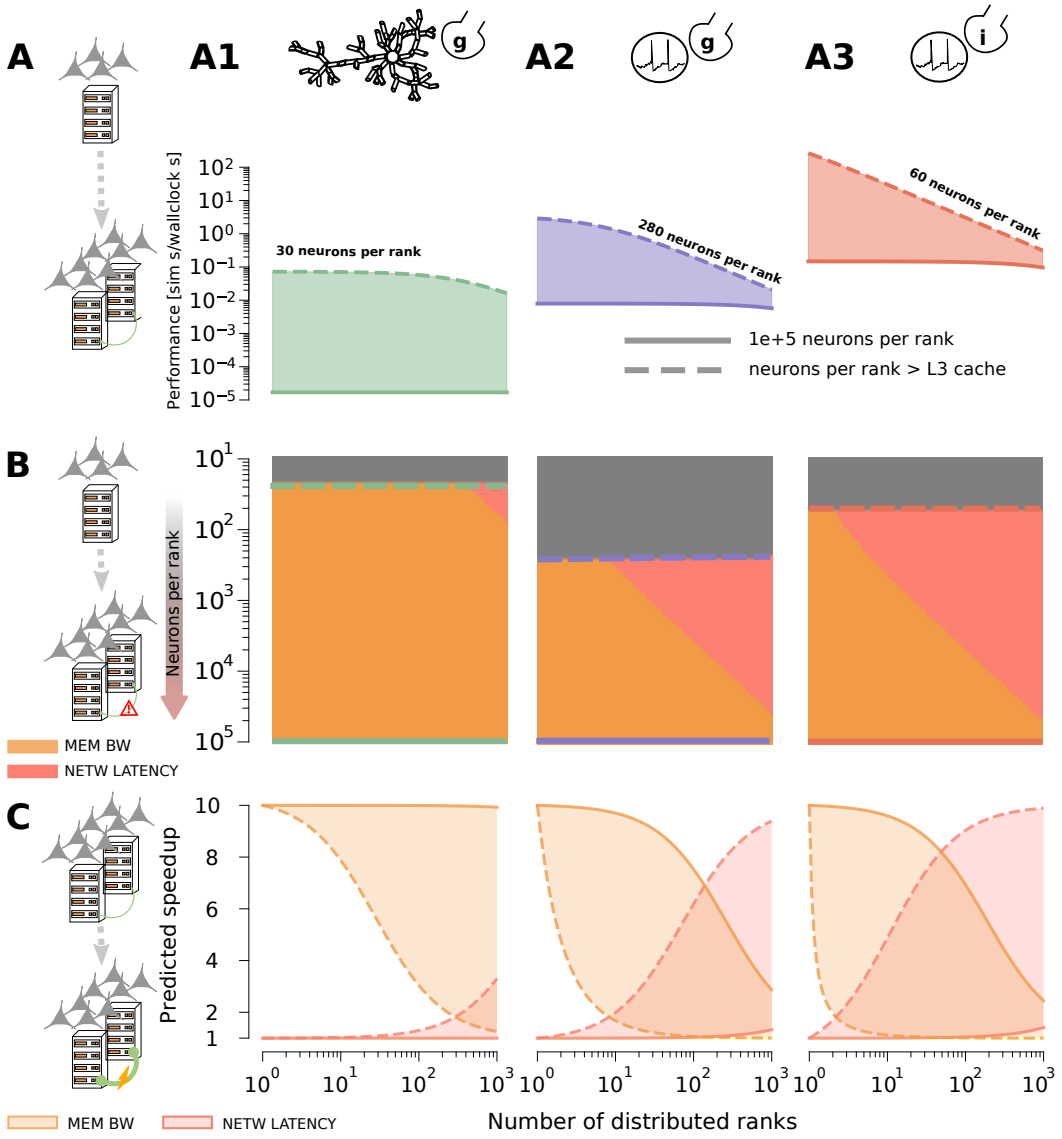

Figure 4.3 – Predicted performance characteristics of the distributed max-filling regime. The SKX AVX512 architecture with HPE Infiniband EDR is used as reference. **A1,A2,A3** Reconstructed G-based, Simplified G-based and Brunel I-based model, respectively. **A** Predicted performance of the three *in silico models* in a weak-scaling, max-filling scenario. We consider several values of the number of neurons per rank, ranging from the smallest number that would exceed a typical L3 size of 25MB to 10^5 neurons per rank. The unit of performance is simulated seconds per wallclock second to simulate the whole network. We make the assumption of complete *saturation of the memory bandwidth*. **B** Hardware bottlenecks defined in (B.17) as a function of the number of neurons per rank (inverted y axis) and the number of distributed ranks (x axis). The grey regions denote an area where the number of neurons per rank would require less memory than a typical L3 cache size, thus invalidating the saturation assumption. **C** Speedup predicted by the model when a single hardware feature is improved by a factor of 10x w.r.t the reference architecture. Shaded areas represent the speedup for different values of the number of neurons per rank, in the range described in A.

considered here. Instead, large-scale simulations are dominated by the latency of the collective communication. This points to the fact that investigating spike communication strategies such as neighbourhood collectives (Jordan et al., 2018), non-blocking point-to-point schemes (Ananthanarayanan and Modha, 2007) or asynchronous execution (Magalhaes et al., 2019b) is essential to reach brain-scale simulations.

Distributed simulations of large networks can be improved by increasing memory bandwidth, small networks by decreasing network latency The aforementioned bottleneck analysis is useful to understand which is the most important hardware feature in a given simulation configuration, but it does not provide any information about the relative importance of the other features. Therefore we predict the speedup corresponding to a 10× improvement of a single hardware feature, for different numbers of neurons per distributed rank, as a function of the number of ranks. The results are plotted in Figure 4.3C, which shows that at small cluster sizes all models would benefit from an improvement in the nodes' memory bandwidth, but not from an improvement in network latency, while the situation is reversed at large cluster sizes. The predicted speedup obtained by improving the memory bandwidth degrades much faster for simulations with a small number of neurons per rank versus a large number of neurons per rank, while the opposite is true for the speedup coming from an improved network latency.

4.1.4 Shared memory constant problem size

Another widespread simulation regime is not focused on simulating the largest possible network, but in simulating a fixed size network as fast as possible. We call this the constant problem size regime, because that represents an ideal target for the performance of a simulation. Currently, on the one hand it is unclear whether constant problem size is realistically achievable on modern hardware (Zenke and Gerstner, 2014), and on the other hand special hardware that breaks this limit by design has already been conceived and tested for small networks (Aamir et al., 2018).

Memory bandwidth dominates the shared-memory strong scaling of Brunel and Simplified models, while a mix of hardware features influences the performance of the Reconstructed model Given that it is possible to observe superlinear speedup as the dataset is made increasingly small by virtue of it fitting into faster cache memory, we predict the performance per neuron of all *in silico* models assuming the dataset could be fully contained in different levels of the memory hierarchy. For simplicity, we neglect the fact that some of these model and cache combinations are infeasible in practice, e.g., due to the memory footprint of a single neuron in the Reconstructed model exceeding the L1 cache size. For DRAM, we assume that the ordering of loops technique is used to avoid saturation of the memory bandwidth, thus extending our analysis by discarding the saturation hypothesis made in previous sections. For all performance predictions we assume that all available threads in the reference

model	performance (DRAM)	speedup in L3	speedup in L2	speedup in L1
Brunel	3.9×10^4	2.5	8.7	33.9
Simplified	7.9×10^2	4.7	6.5	6.6
Reconstructed	3.9	1.8	2.4	2.4

Table 4.2 – Full-chip predicted performance, relaxing the saturation assumption. Predicted performance is measured in simulated seconds per wallclock second per neuron. We do not make the assumption of memory bandwidth saturation, and consider instead that all available parallelism (18 threads) is used in the reference SKX AVX512 architecture.

architecture (18 in total) are being utilized. Results are reported in Table 4.2, where we give the raw performance value when data is in DRAM, and the corresponding (superlinear) speedup factor as the dataset becomes small enough to fit in higher levels of the cache hierarchy.

Figure 4.4 shows the predicted performance breakdown into simulation kernels as well as hardware features for all *in silico* models, assuming that the dataset fits in different levels of the memory hierarchy. When data is in the highest level of the cache hierarchy (L1), the most important kernels for all models are state update kernels, and the most relevant hardware feature is the CPU throughput. Additionally, in the G-based models the computation of the exponential (for updating the synaptic states) constitutes a significant portion of the overall execution time. As the dataset increases in size and is only able to fit in lower levels of the cache (L2 or L3) the predicted performance of the G-based models remains quite stable while that of the Brunel model degrades rapidly, although admittedly our model for the spike delivery kernel in caches might be highly optimistic. In practice, this could be an indication that the Brunel model is bounded by the data path while the G-based models are bounded by the maximum achievable flop rate. Our breakdown analysis confirms this, although for the reference architecture G-based models are best represented by a mix of core-bound and data-bound kernels, especially when the dataset fits only in the L3 cache. Complementarily, in G-based models the relative importance of the core-bound state update kernels gradually loses weight in favour of data-bound current kernels, while in the Brunel model the weight of the spike delivery kernel gradually increases, eventually becoming the most relevant kernel in the execution, as data moves further away from the CPU. In spite of this technique, both point neuron models are clearly dominated by the saturation of the memory bandwidth. In particular, the fact that memory bandwidth is the only factor in determining the performance of the Simplified model can be directly related to the fact that its coupling ratio has a value of 1, as shown in Figure 2.6. The performance profile of the Reconstructed model is more diverse, and while 60% of the execution time is still dominated by memory bandwidth, the data transfers between the caches, arithmetic instructions, and throughput of exponential function evaluations also take up a significant portion of the runtime. Again we stress that this phenomenon is tightly linked to the hardware architecture used as reference, specifically to its balance of compute power, memory bandwidth and number of threads. However, we

can assume that the general application of these conclusions still holds as most modern architectures are designed with a similar balance point (McCalpin, 1995).

4.1.5 Distributed constant problem size

An effective strategy for improving simulation performance or to handle larger networks is to dedicate more hardware to the task, distributing the simulated neural network across multiple compute nodes. For a fixed problem size, this translates to the concept of strong scaling. The limits to strong scaling of medium sized plasticity networks have been empirically explored in (Zenke and Gerstner, 2014). Here we generalize their analysis to other *in silico* models as well as provide clear explanations for the causality of bottlenecks, backed by our performance model.

Performance predictions for strong scaling of arbitrarily sized networks We predict the performance and scaling bottlenecks of *in silico* models in a strong scaling scenario, using our performance model, and present the results in Figure 4.5. As in the distributed maximum filling scenario, the problem size has a major impact on performance, so we include several possible sizes in our analysis. Some researchers have explored the possibility of splitting a neuron across more than a single parallel process, such as the multisplit method (Hines et al., 2008), branch-parallelism (Magalhaes et al., 2019a) and domain decomposition (Kozloski and Wagner, 2011). We do not include such techniques in our analysis because their granularity falls outside the scope of this work. In practice, this imposes a limit on how many distributed ranks a given neural network could be scaled on because the limitation of one neuron per distributed rank cannot be overcome. In Figure 4.5A we present raw performance predictions for different network sizes in a strong scaling scenario. The dashed lines correspond to the minimum network size such that strong scaling can be carried out until occupancy of the full cluster, set here at 10^3 distributed ranks. For all *in silico* models, as long as the network size is sufficiently large, the performance initially improves as we distribute the problem over increasingly more ranks. However, for all *in silico* models there exists a threshold number of ranks after which the benefits from adding hardware become less prominent. Scaling to larger cluster sizes after this threshold can be counter-productive, and even result in performance degradation. The threshold value itself is a function of the hardware architecture, *in silico* model and problem size. Interestingly the striking differences in performance between *in silico* models at small cluster sizes can be evened out quite significantly at large cluster sizes in this scenario. For example, simulating a large Brunel network on 10 distributed ranks can be roughly four orders of magnitude faster than a Reconstructed network on the same hardware, but the difference between models goes down to two orders of magnitude at large cluster sizes.

Network latency and memory bandwidth are the main bottlenecks in strong scaling Following the same procedure of the maximum filling scenario we investigate the reasons for performance degradation by plotting the most significant hardware bottlenecks for all com-

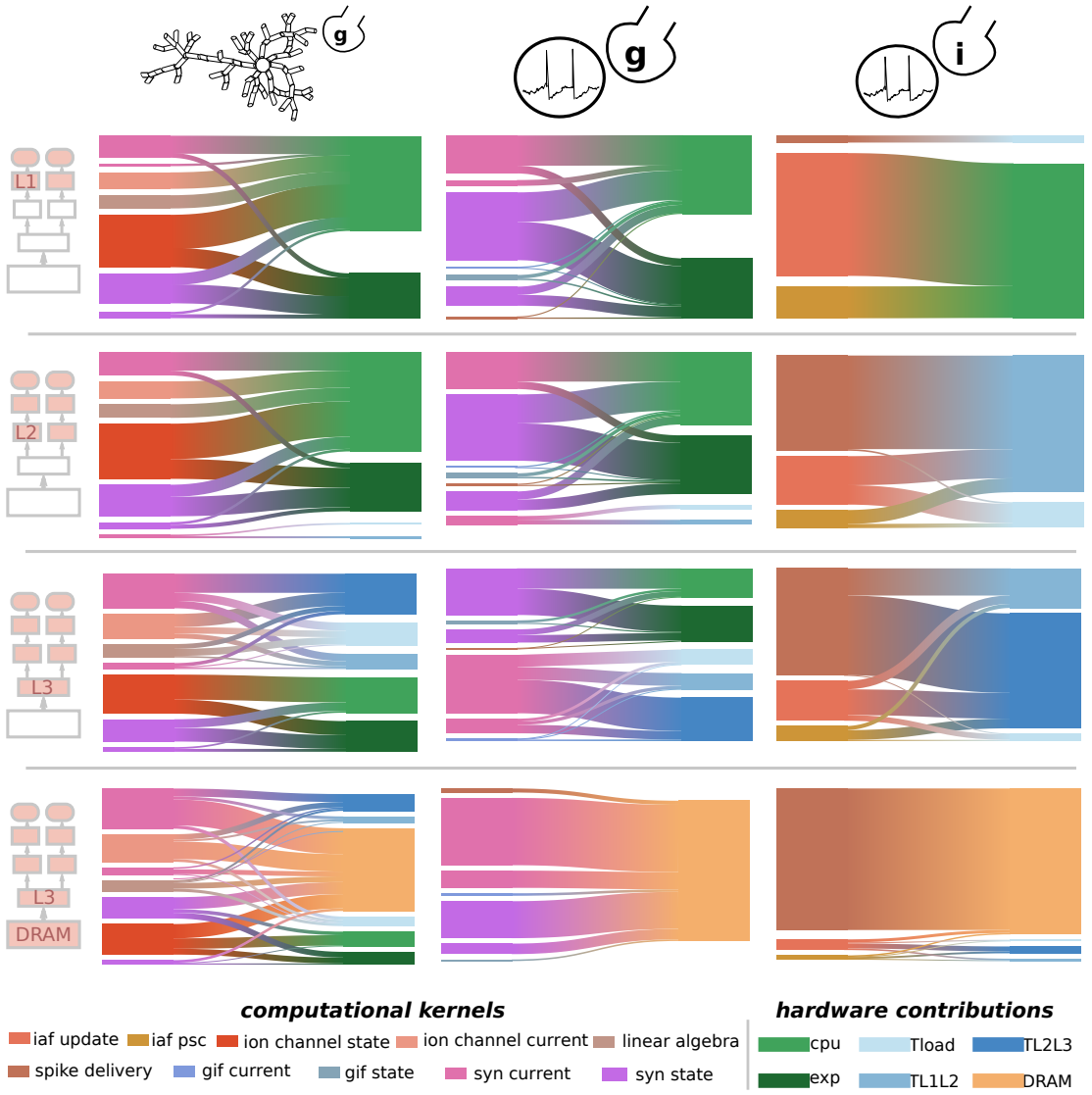

Figure 4.4 – Predicted shared-memory runtime contributions from computational kernels and hardware features. We assume a single node of the SKX architecture with AVX512 vectorisation and using the maximum number of threads (18 threads). We do not make the assumption of memory bandwidth saturation, but we assume that the loop ordering optimisation is used. For each level of the cache hierarchy, we show the breakdown of the total runtime into computational kernels on the left of each box. Furthermore, we show the breakdown of the runtime, as well as the breakdown of individual computational kernels, into hardware contributions on the right of each box. Hardware contributions labels have the following meaning: CPU stands for the execution of non-memory access instructions in the core (excluding the exponential function), exp for the computation of exponential function, T_{load} for the execution of memory access instructions in the core, and the rest for the data traffic time of the relevant datapath.

binations of network size and cluster size in Figure 4.5B. We assume that the loop ordering optimisation is being used. Even though we do not make the explicit assumption of memory bandwidth saturation, this hardware feature is still among the most relevant for all *in silico* models (as was also shown in Figure 4.4). Moreover, network bandwidth is never the dominating bottleneck for all *in silico* models and all configurations, while network latency always becomes the most important bottleneck at large cluster sizes. At very small cluster sizes, we recover the results from Figure 4.4.

Network latency gives significant performance improvements for point neuron models, but no improvement in a single factor would be sufficient to increase performance in the Reconstructed model To further investigate the relevant bottlenecks, we predict the expected speedup corresponding to a tenfold increase in a single hardware feature and present the results in Figure 4.5C. Both point neuron models show a similar structure: an improvement in the features of a single node such as CPU frequency or cache throughput yields an improvement in performance only for very small networks, while a tenfold improvement in network latency would guarantee a significant improvement in performance for networks of all sizes and sufficiently large cluster sizes. The only difference between the two point neuron models in this analysis are the benefits to be gained from improving the memory bandwidth: while there would be almost no benefit for the Brunel I-based model, it has a moderate influence in the performance of the Simplified G-based model, especially for large networks. The situation for the Reconstructed model differs, and there is no single factor that would result in a significant performance improvement at any network size, except for strong scaling of very small networks where the CPU throughput is the dominating hardware feature. This can be explained by the diversity of relevant hardware factors identified in Figure 4.4: when a single hardware feature is improved, another bottleneck is quickly reached and the total resulting performance improvement is suboptimal. For large cluster sizes the situation normalizes to that of the other *in silico* models, and network latency becomes the dominant factor, such that a tenfold increase results in a nearly-equivalent performance boost, especially for small networks.

4.1.6 Dependence of performance on model parameters

Parameters of the *in silico* models have an important, yet often difficult to explain, impact on performance. We test the impact of the firing frequency, minimum network delay and fan-in using our performance model, and present the results in this section. We neglect the timestep because it has a generally straightforward relationship with performance, and was omitted for brevity.

Firing frequency's differential effect on communication and computation Firing frequency is commonly cited as one of the most impactful parameters on simulation performance (Yavuz et al., 2016). In this work we consider it a parameter even though it usually cannot be explicitly

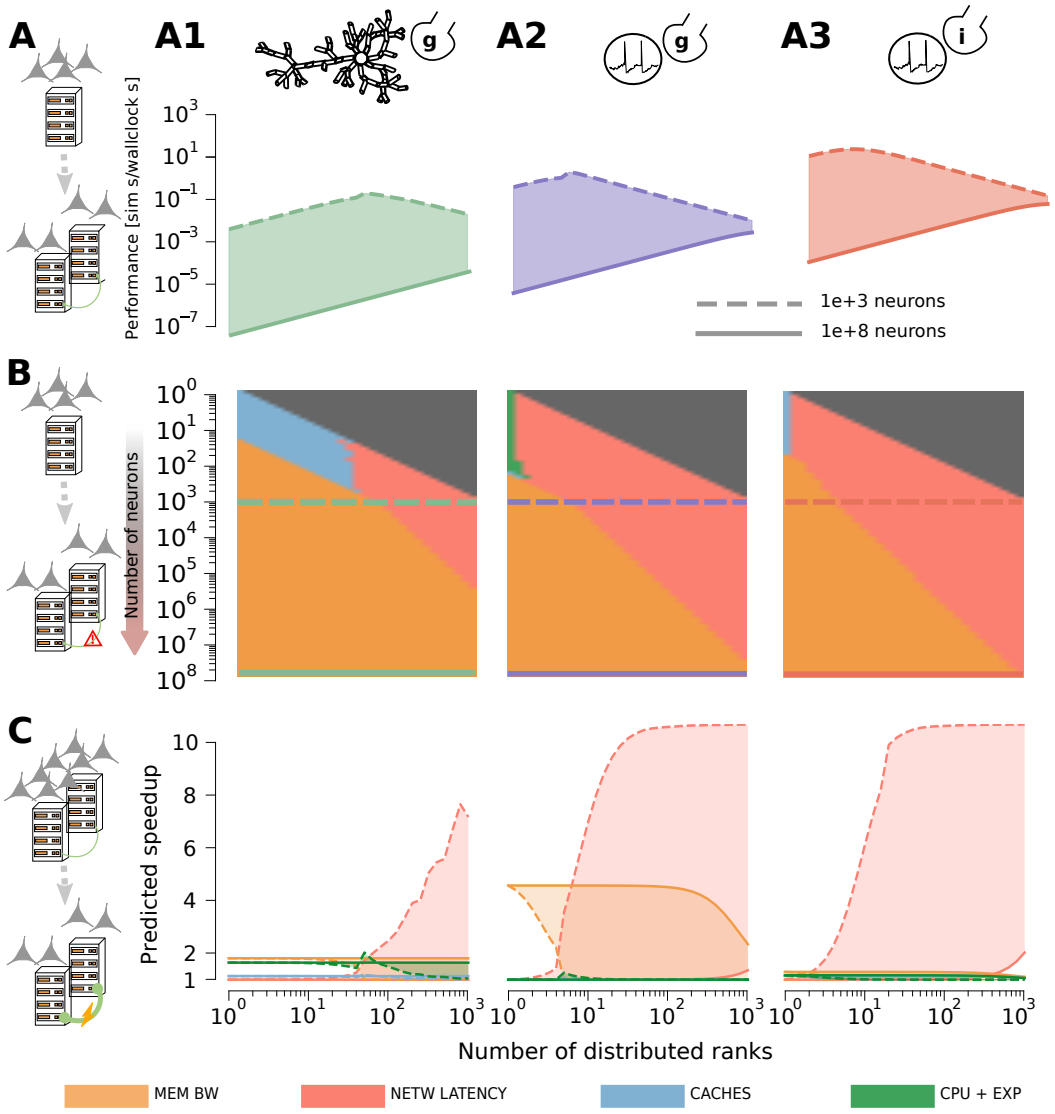

Figure 4.5 – Performance characteristics of the distributed constant problem size regime. The SKX AVX512 architecture with HPE Infiniband EDR is used as reference. **A1,A2,A3** Reconstructed G-based, Simplified G-based and Brunel I-based model, respectively. For all models, we do not assume that memory bandwidth is saturated but we assume that the loop ordering optimisation is used. **A** Predicted performance of the three *in silico models* in a strong-scaling scenario. We consider different total network sizes. The dashed and solid lines represent simulations with networks of 10^3 and 10^8 neurons respectively. The unit of performance is simulated seconds per wallclock second to simulate the whole network. **B** Hardware bottlenecks defined in (B.17) as a function of the total number of neurons (inverted y axis) and the number of distributed ranks (x axis). The grey areas denote a configuration that would require splitting of individual neurons. **C** Speedup predicted by the model when a single hardware feature is improved by a factor of 10x w.r.t the reference architecture. Shaded areas represent the speedup for different values of the total number of neurons, in the range described in A.

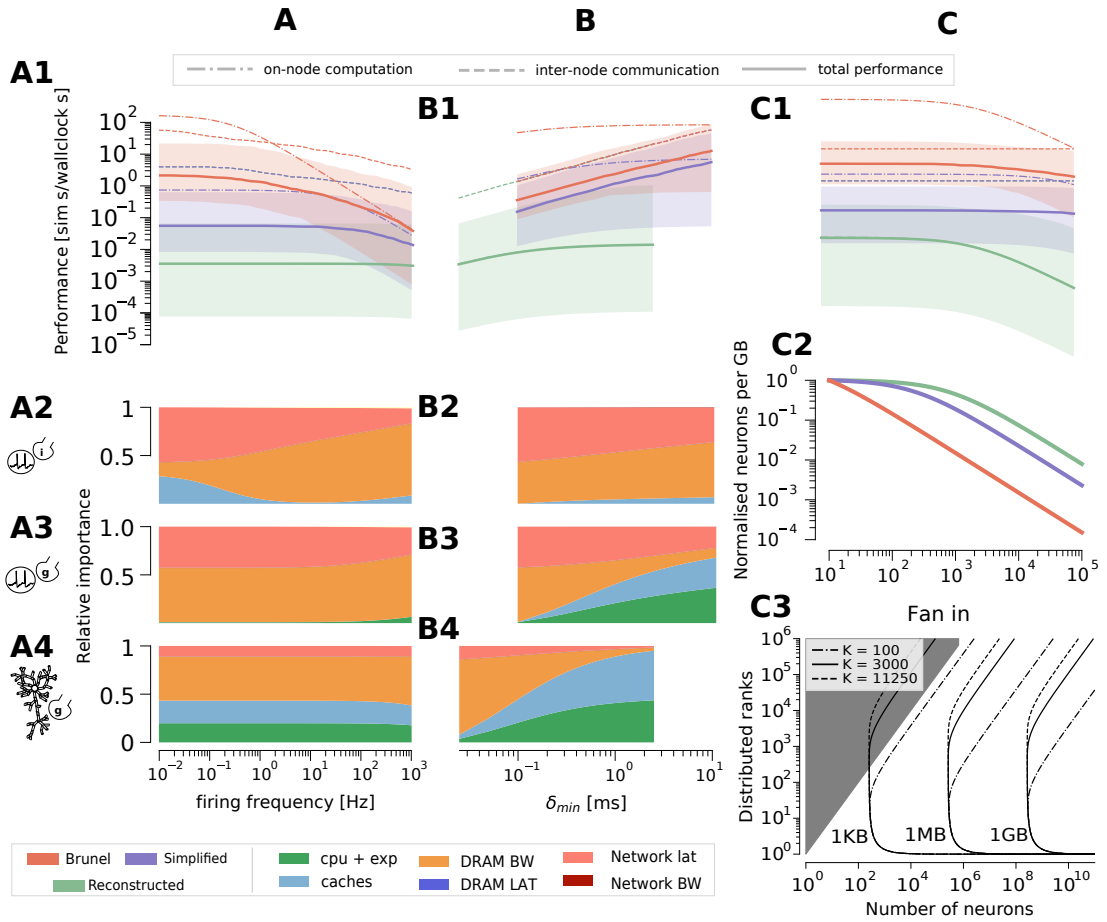

Figure 4.6 – Effect of model parameters on performance. **A** Effect of firing frequency on the performance of distributed simulations. **A1** Median performance over 1000 randomly generated simulation samples defined by number of neurons and number of distributed ranks. The shaded area surrounding the total performance represents the 25th and 75th percentiles. **A2,A3,A4** Stacked plot of the mean relative contributions from hardware features, respectively for the Brunel, Simplified and Reconstructed model. **B** Effect of δ_{min} on the performance of distributed simulations. **B1** Median performance over 1000 randomly generated simulation samples defined by number of neurons and number of distributed ranks. The shaded area surrounding the total performance represents the 25th and 75th percentiles. **B2,B3,B4** Stacked plot of the mean relative contributions from hardware features. The range of acceptable values for δ_{min} changes across different *in silico* models because they were computed as multiples of the model's timestep. **C** Effect of fan-in K on the performance of distributed simulations. **C1** Median performance over 1000 randomly generated simulation samples defined by number of neurons and number of distributed ranks. The shaded area surrounding the total performance represents the 25th and 75th percentiles. We assumed that the reference architecture had infinite memory capacity. **C2** Number of neurons able to fit in one GB, normalized by the memory requirements of a model with 10 incoming synapses. **C3** Contour plot of predicted memory requirements of the connections table, as a function of the total number of neurons (x axis) and the number of distributed ranks (y axis). The contour levels corresponding to 1KB, 1MB and 1GB are shown for different values of the fan-in.

set by the user, and is instead an emerging property of the simulation. Figure 4.6A shows the predictions of our performance model for the three *in silico* models, for values of the firing frequency in a physiological range. To take into account the obvious fact that the total number of neurons and distributed ranks can introduce a large variability in our predictions, we randomly generate 1000 value pairs of [number of neurons, number of distributed ranks] and plot the median predicted performance, additionally broken down into its two components of inter-node communication and on-node computation. For the number of neurons we consider values in the range $[1, 10^8]$ while for the number of ranks we consider values in the range $[1, 10^3]$; furthermore, we discard the few configurations for which the number of neurons was randomly chosen to be smaller than the number of ranks, as that would imply the splitting of neurons. We generate random numbers of neurons and ranks following a log-uniform distribution, with the effect that all orders of magnitude are equally likely, thus introducing a very large variability. Firing frequency has an effect on communication by changing the size of the spike message as well as on computation by changing the amount of events that must be integrated by neurons. In Figure 4.6A1 it is noticeable that there exists a threshold frequency below which f does not affect performance, but once this threshold is passed firing frequency becomes a primary factor, inducing a linear, almost unit-slope degradation in performance. This effect is clearly visible in the Simplified model, and even more so in the Brunel models. For the Reconstructed model this threshold value exists, but is so large that we can safely assume that, in the median case, firing frequency has no effect on performance. To investigate the reasons for performance degradation we look at the breakdown of relative importance of different hardware features as a function of the firing frequency, plotted in Figure 4.6A1,A2,A3. Similarly to before, we randomly generate 1000 couples of [number of neurons, number of distributed ranks] but we plot the mean relative importance instead of the median to keep the total constantly equal to 100%. Our analysis shows an interesting behaviour: as the firing frequency becomes larger, the relative pressure on the memory bandwidth (and eventually the network bandwidth) becomes larger, while the relative pressure on the network latency becomes smaller. So not only there is more computation to be done as firing frequency gets larger, but also the mix of hardware bottlenecks changes. This behaviour was observed empirically not only on general-purpose CPUs (Zenke and Gerstner, 2014) but also on GPUs which are additionally more susceptible to dynamic load balancing because of the extremely large number of parallel cores (Yavuz et al., 2016) We remark that the large variability in the performance predictions in Figure 4.6A1,B1,C1 can be at least partially explained by our choice of sampling strategy as mentioned above.

Minimum network delay affects the relative importance of hardware features Another parameter of interest is the minimum network delay, denoted $\delta_{\min}$. In terms of communication, $\delta_{\min}$ affects the number of times that global communication must happen to simulate one second of biological time, although it does not affect the total number of spikes communicated. In terms of computation, assuming that the loop ordering strategy to minimise pressure on the memory bandwidth is employed then $\delta_{\min}$ affects the number of time iterations in

which data locality can be exploited. Figure 4.6B1 shows the predictions of our performance model for the three *in silico* models, for different values of the minimum network delay. Since this delay can only be an integer multiple of the timestep, we exploit the concept of coupling ratio to define a range of plausible minimum delay values by setting a range of values for the coupling ratio and obtaining the corresponding $\delta_{\min}$ by multiplication with Δt. By looking at the breakdown of performance, we see that larger $\delta_{\min}$ improves the performance of inter-node communication but also, somewhat surprisingly, on-node computation. However, while the communication performance seems to improve indefinitely, the improvement of on-node computation saturates quite quickly. For the point neuron models, within the range of $\delta_{\min}$ values considered here, there is a transition from a regime dominated by communication to one dominated by computation, while the Reconstructed model is dominated by computation for all $\delta_{\min}$ values. To investigate the reasons for changes in performance, we plot the breakdown of relative importance of different hardware features in Figure 4.6B1,B2,B3. In the case of G-based models, larger $\delta_{\min}$ correspond to decreased pressure on the memory bandwidth and network latency, and larger pressure on more scalable hardware features such as CPU instruction throughput and caches throughput. This points to the fact that simulations based on G-based models with a large $\delta_{\min}$ could strongly benefit from shared-memory parallelism. On the other hand, a larger $\delta_{\min}$ in the I-based model results in decreased pressure on the network latency, but a higher pressure on memory bandwidth.

Large fan-in can be advantageous for performance of point neuron models, but has almost no effect on Reconstructed model Finally, we examine the effect of fan-in, defined as the average number of incoming connections per neuron and denoted by K. This parameter has subtle effects on performance that are difficult to analyse. For G-based models, a larger fan-in technically means more synapses to simulate thus an expected degradation of performance. Additionally, for all models a larger K determines an increase in event-driven computation, thus once again an expected degradation of performance. These hypothesis are confirmed in Figure 4.6C1, which shows that K does not affect communication but has a very strong effect on the Reconstructed model. Unexpectedly, fan-in seems to only marginally affect the performance of the Simplified model, in spite of it being a G-based model too. This can be easily explained by the fixed number of synaptic instances in this model (28 excitatory and 8 inhibitory (Rössert et al., 2016)), such that much like the I-based Brunel model, ultimately the fan-in affects only the average number of events a neuron must integrate within a certain time period. Another important point should be made about the effect of fan-in, because changing the number of connections of a neuron has an impact on the *in silico* model's memory requirements. Figure 4.6C2 shows the ratio of neurons that can fit in a Gigabyte of memory, according to our performance model, as a function of fan-in. In a strong scaling scenario, this information sheds new light on the conclusions above, because if only $\frac{1}{x}$ neurons fit in a GB, this can result potentially in a x-fold increase in performance from parallelism (disregarding potential communication bottlenecks). Therefore for the Brunel and Simplified model it can be advantageous to have a large number of incoming connections per neuron,

because the performance price paid is more than compensated by the required increase in parallelism. Conversely, in the Reconstructed model, these two effects appear to balance out almost evenly.

For very large scale simulations, another subtle effect of fan-in is represented by the size of the connection table, an issue that was raised and investigated in (Kunkel et al., 2014). The connections table of a given rank contains all the GIDs of presynaptic neurons that are relevant for at least one neuron in that rank. The size of the connection table depends, among other things, on K as well as the total number of neurons and number of distributed ranks. In Figure 4.6C3 we plot the values of the expected total size of the connection table on the (total number of neurons, number of distributed ranks)-plane as a contour plot, highlighting the contours corresponding to a total size of 1KB, 1MB and 1GB. We report the formula for completeness, even though it has already been published (see Kunkel et al., 2014):

$$N\left(1-\left(1-\frac{1}{N}\right)^{\frac{NK}{P}}\right), \tag{4.2}$$

where N is the total number of neurons, K the number of incoming connections per neuron and P the number of parallel processes. Although in some *in silico* models connectivity may be determined by complex rules influenced by cell type and spatial locality, for simplicity we compute here the expected size of the connections table assuming uniform connection probability and random distribution of neurons across ranks. In a strong scaling scenario the size of the connections table steadily decreases as the number of ranks increases, starting from the minimum of two ranks required by a distributed simulation. This can be explained by the fact that fixing the network size and increasing the number of ranks entails that there will be less incoming connections to a given rank. In a weak scaling scenario, such as the maximum filling regime, the size of the connections table transiently increases when the number of distributed ranks is low, but at large scale reaches a constant value steady state determined only by K.

4.2 Discussion of Performance Landscape and Future Projections

In this chapter we have delivered a quantitative characterisation of the performance properties of different published *in silico* models at the core of state-of-the-art brain tissue simulations. Using a grey-box model that combines biological and algorithmic properties with hardware specifications we have identified performance bottlenecks under different simulation regimes, corresponding to a variety of prototypical scientific questions that can be answered by simulations of biological neural networks.

General purpose computing has sustained a diverse performance landscape up to now Our results show that there exists a large diversity of performance profiles and bottlenecks that shape the landscape of brain tissue simulations, corresponding to the diversity of sizes

and scales at which research questions in simulation neuroscience can be asked. Thus, our research highlights that the computational neuroscience community is currently greatly benefiting from the adaptability of general purpose computing, exploiting the ease of development and high performance capability to explore different areas of the modelling landscape.

Complex out-of-order cores enable high performance and efficiency in brain tissue neuron models In the serial execution, we have shown that G-based models are dominated by state and current kernels. Our analysis has highlighted that high-throughput exponential and division functions and wide SIMD units enable high-performance simulation of G-based state and current kernels (see also Section 3.1.6 and Cremonesi et al., 2019a). The fact that most kernels have a data-bound profile on our AVX512 reference architecture is also an indication that the performance gains in the core deriving from an out-of-order execution have been pushed to the limit at this level of vectorisation, as is shown e.g. by the fact that G-based synapse current kernel switches from a core-bound to a data-bound profile when data is in memory. I-based neurons, on the other hand, are dominated by the spike delivery kernel, and thus their performance is affected by memory latency. As we have observed, however, the full cost of the memory is not paid by this kernel because of its peculiar data access pattern. Therefore the ability of complex, out-of-order cores to exhibit instruction-level and memory-level parallelism enables hiding the cost of memory latency and delivering high performance serial simulations of I-based neurons.

Memory bandwidth and network latency severely limit maximum filling and constant problem size strong scaling Using a state-of-the-art HPC server CPU and cluster as a reference, our analysis revealed that all the *in silico* models saturate the memory bandwidth using quite a small number of shared memory threads. The high load on memory bandwidth imposed by brain tissue simulations is also reflected in custom codesigned chips where low-resolution synapses are used to minimise data traffic (Merolla et al., 2014; Pfeil et al., 2012). Even when algorithmic improvements are put into place to mitigate this effect, we have identified that the coupling ratio, a dimensionless number that counts the number of timesteps in a minimum network delay period, strongly regulates the saturation of memory bandwidth and, in the extreme case of the Simplified model analysed here, effectively prevents any benefit to be gained from the effort of developing a more efficient algorithm. Additionally, we discovered that it is not the level of morphological detail, but rather the formalism used to represent synapses, that is the most important factor in explaining the memory bandwidth saturation profile, with G-based models saturating much faster than the I-based model. In distributed simulations we identified the network latency, and not the network bandwidth, as the major bottleneck for scaling to very large networks or very large cluster sizes. This provides a new motivation and justification for the extensive efforts in designing a specific communication infrastructure for the SpiNNaker neuromorphic system (Navaridas et al., 2012).

Loop ordering is essential to guarantee high parallel scalability in G-based neurons The loop ordering optimisation is essential to enable cache reuse, which in turn greatly decreases the memory bandwidth pressure and increases the saturation point of simulating neurons. In G-based neurons where the memory saturation happens at the level of clock-driven kernels this represents a paramount optimisation to achieve high performance. In cases where the model specifications do not allow room for the loop-ordering optimisation, such as the Simplified model having a coupling ratio equal to 1, it might still be worth investigating whether the benefits of reducing the timestep – i.e. higher numerical accuracy and better shared-memory scalability – could outweigh the loss in performance due to the larger amount of work necessary to simulate one second of activity. While the loop ordering optimisation pushes towards processing a single neuron at a time, it should be noted that a tradeoff with vectorisation has been recently highlighted because sufficiently many neurons must be grouped together to enable efficient use of the vectorized computation units (Magalhaes et al., 2019a). Ultimately, this leads us to the remark that the L3 cache size is a crucial hardware feature, as it must be large enough to hold enough neurons for both vectorisation and maximum parallelism. Finally, we expect the loop ordering optimisation to be less beneficial in the simulation of I-based neuron, because the main driver for memory saturation is the event-driven spike delivery kernel whose performance is independent from the relationship between timestep and minimum network delay.

Model-specific features have a significant impact on performance Inspection of our performance model allowed us to pinpoint which kernels, hardware specifications and model parameters have the largest impact on performance. The Brunel model based on the I-based formalism and IAF neurons is mainly bounded by the spike delivery kernel, which exhibits a good shared-memory scaling behaviour and, in the case of extreme strong scaling, a strong dependence on the inter-cache data paths for good performance. The two G-based models we analysed, i.e. Simplified and Reconstructed, have a similar shared-memory scaling behaviour, mainly driven by the *current* kernels required to compute the contributions of individual synapses (and ion channels) to the membrane potential equation. However, while the Simplified model is 100% dominated by memory bandwidth, the morphologically detailed Reconstructed model is dominated partially (around 40%) by other hardware components such as caches and CPU throughput. Interestingly, differences between models are significantly reduced when comparing the performance of distributed simulations of large networks. This can be attributed to the fact that in this case the network latency is the dominant factor, thus in the extreme situation where performance is only determined by communication, the only differences between *in silico* models would be in terms of their minimum delay. It becomes clear that a performance model and a detailed performance analysis are fundamental tools to disentangle the complex web of relationships between *in silico* models, their software implementation and hardware concretisation.

Static and dynamic model parameters affect performance in significant but subtle ways Finally, we examined the impact of model parameters on the performance profiles described above. We found that firing frequency, but surprisingly also minimum network delay, can have a large impact on determining which hardware features may constitute a performance bottleneck. For firing frequency it is obvious that larger values correspond to more operations required by the simulation algorithm, and thus a lower performance, but our analysis shows that different values of the firing frequency also change the relative importance of hardware features. Interestingly we found that the minimum network delay, in spite of it not affecting the total number of operations per simulated second, can have an effect on performance simply by shifting the importance of the hardware bottlenecks. We also found that the average number of incoming connections per neuron plays a subtle role in influencing performance. Trivially, a larger fan-in increases the computational requirements of a single neuron. However, it also increases the memory capacity requirements, thus requiring a larger degree of parallelism to handle the same network size. This creates a tradeoff between performance degradation arising from larger computational requirements and performance improvement from parallelism requirements.

Relevance of computational and communication kernels to whole-simulation performance The scope of our work is limited to the computational and communication kernels in brain tissue simulations, thus excluding from our analysis auxiliary algorithm phases such as generation of stimuli, random number generation and managing the queue of synaptic events. While these kernels constitute necessary steps in the simulation of brain tissue, the goal of our investigation is to study the performance properties related to the mathematical modelling of neurons, and not implementation and hardware details such as the most efficient random number generation strategy. Despite excluding some parts of the simulation algorithm, our analysis still maintains a lot of relevance with regards to the overall simulation performance. For G-based models several studies have shown that the combined runtime of ion channels, synapses and spike delivery amounts to over 90% of the total simulation runtime, with linear algebra contributing another 5% in the case of detailed models (Akar et al., 2019a; Ewart et al., 2015; Kumbhar et al., 2019a; Rössert et al., 2016). In I-based models the relative impact of handling queue events and spike delivery increases with parallelism and eventually dominates the runtime (Schenck et al., 2014). Since their analysis does not distinguish between these two operations, one cannot tell whether the degradation in performance comes from the spike delivery kernel – which we consider in our analysis – or the management of the queue. A similar study (Peyser and Schenck, 2015) found that spike delivery of plastic synapses dominates the runtime of I-based point neuron simulations, even at large scale. They also demonstrated high performance gains by switching off the buffering of random events, however this may be considered more of an implementation detail than an intrinsic property of the model. Finally a GPGPU implementation (Yavuz et al., 2016) found that spike delivery dominated the runtime in synchronous and asynchronous firing regimes, while the quiet regime was dominated by random number generation. In distributed simulations at very

large scales, communication has been found to dominate the runtime regardless of modelling abstraction (Ananthanarayanan et al., 2009; Jordan et al., 2018; Ovcharenko et al., 2015). These studies demonstrate that computational and communication kernels make up a significant portion of the runtime of brain tissue simulations, and although our performance model could be complemented with auxiliary algorithm phases, their exclusion from the analysis does not hinder the validity and generality of our results.

5 A case-study in the heterogeneous performance of a cortical microcircuit

The contents of this chapter are adapted from the following publication:

Francesco Cremonesi, Pramod Kumbhar, and Felix Schürmann. Heterogeneity of performance properties in the simulation of a cortical microcircuit. *To be submitted*, 2019b

At all physical scales, the brain is a fundamentally heterogeneous structure. Studies have identified over ten thousand different protein types in the human brain (Ping et al., 2018), over 300 ion channel types solely in the inner ear of humans and mice (Gabashvili et al., 2007), tens of different cell types in rat's neocortex alone (Kanari et al., 2019; Narayanan et al., 2017) and in the order of a hundred different cell types in the whole mouse brain (Hodge et al., 2018). This diversity is reflected in the computational properties of *in silico* models that try to capture the corresponding biological detail. Our analysis of hardware-agnostic metrics in Section 2.2.2 identified that the Reconstructed model is susceptible to a high degree of heterogeneity, and we speculated that this would have an effect on performance. Moreover, our analysis of the Power8 architecture and of different strategies for computing the exponential function demonstrated that heterogeneity across hardware platforms influences performance in ways that, without a systematic understanding such as the one offered by analytic performance modelling, is difficult to explain (Ewart et al., 2015, 2019). Thus we believe that an in-depth analysis of the heterogeneous performance profile of *in silico* models, viewed through the performance modelling lens, is warranted.

The goal of this chapter is understanding how the intrinsic heterogeneity of biological systems interacts with other sources of heterogeneity such as modelling choices and hardware platforms, and ultimately determines the efficiency of a given simulation setup. We choose to focus on the Reconstructed microcircuit, because it offers multiple axis of heterogeneity, as was captured by our hardware-agnostic metrics in Figure 2.5. In terms of performance, we distinguish between static sources of heterogeneity, whose impact and nature is known a priori, and dynamic sources of heterogeneity, tied to the temporal evolution of a specific simulation. Examples of static heterogeneity include different neuron representations arising from biological variability, such as different morphologies or connectivity patterns; differ-

ences in the computational properties of modelling abstractions, such as data-bound or compute-bound kernels; use of heterogeneous hardware such as CPU/GPU architectures. On the other hand, dynamic sources of performance heterogeneity in brain tissue simulations include differences in the workload of parallel processors due to irregular spiking behaviour or event-driven simulation of neurons; differences in workload due to adaptive time-stepping; unexpected workloads due to concurrent consumption of simulation data such as interactive visualization.

In what follows, we use the performance modelling methods described in Chapter 3 to explore to the best of the model's capabilities the effect of heterogeneity on the performance of the Reconstructed *in silico* model. The complexity and number of distinct simulation kernels within the Reconstructed model makes pen-and-paper performance modelling a daunting task. Therefore we develop a novel system that exploits the synergy of code generation and automatic performance modelling to obtain runtime predictions for the clock-driven ion channel and synaptic kernels in a cortical microcircuit model. To put this workflow in place we extend the NMODL source-to-source compiler (Kumbhar et al., 2019a) by adding a new code generation backend to output the performance-relevant loops within the simulation kernels for analysis by the automatic performance modelling tool Kerncraft (Hammer et al., 2017). With the addition of a post-processing step that accounts for parts of the loop that were discarded in the code-generation process, we are able to automatically obtain the ECM model for all the kernels in the microcircuit, which account for roughly 80% of the overall execution time (Cremonesi et al., 2019a; Kumbhar et al., 2019a). This enables us to analyse the heterogeneous distribution of performance properties at different levels of the modelling abstraction, in a bottom-up order: from individual simulation kernels, to individual neurons and finally the whole network. Based on this analysis we are able to evaluate possible algorithmic optimisations such as overlapping of kernels as well as identify future directions for hardware codesign. At the network level, we are able to evaluate the impact of static and dynamic load imbalance on simulations of networks of neurons on a cluster. An in-depth analysis of the different sources of heterogeneity in a microcircuit model and their impact on simulation performance has, to our knowledge, never been published.

5.1 Automatic performance modelling of ion channel and synapse simulation kernels

Simulation neuroscientists curate repositories of ion channel and synapse models, such as e.g. ModelDB (Hines et al., 2004) or the NeuroML database (Birgiolas et al., 2019), to allow sharing of resources and replicability of *in silico* experiments. The Blue Brain repository (Ramaswamy et al., 2015) contains roughly 100 models, of which around 20 are used currently in the cortical microcircuit. These models are constantly being updated as more and more experimental data is integrated in the digital reconstruction process. Because of the large number of models, as well as their continuous development, it would be infeasible to manually construct a performance model for each kernel. Instead, building on our validated approach by hand,

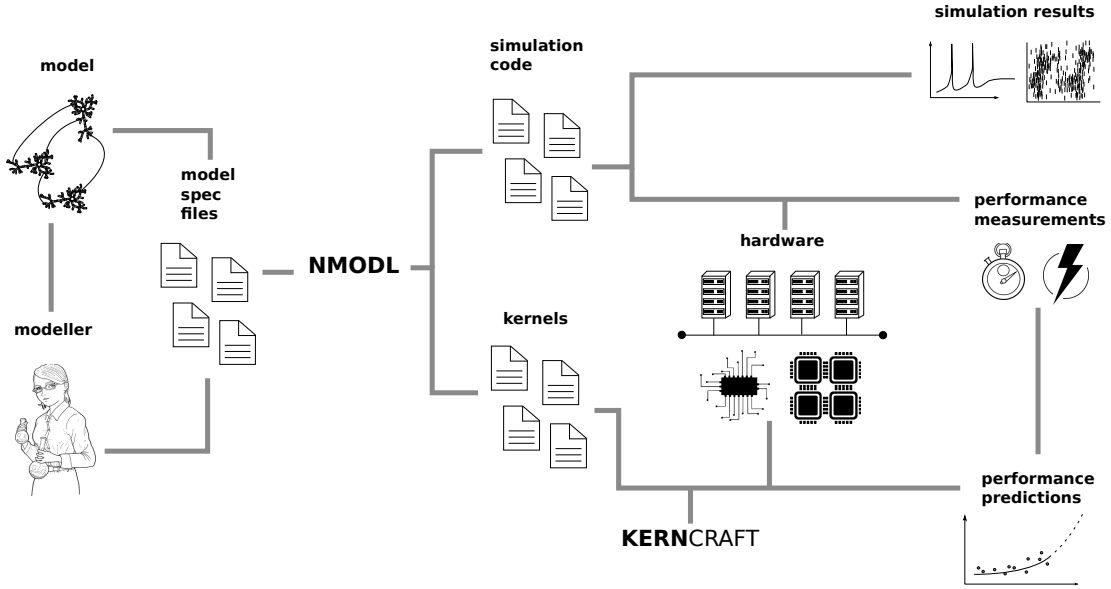

Figure 5.1 – Automatic performance modelling workflow. Computational neuroscientists develop models and specification for *in silico* experiments. To be able to simulate their models, they write model files respecting precise grammar and syntax rules. In the typical simulation workflow (*top*), the NMODL source-to-source compiler then translates these files into compilable code, than can be assembled into a library and linked by the simulation engine. Ultimately, scientists are able to simulate their models on a given hardware, obtain the corresponding results and if needed a performance profile of the execution. In the automatic performance modelling workflow (*bottom*), the NMODL source-to-source compiler produces kernel files that can be parsed by the Kerncraft tool. Such files can be used to automatically obtain a performance prediction of the kernel based on the hardware representation. Ultimately, runtime predictions can be compared against measurements for validation.

we leverage automatic tools as a way to reduce the performance modelling effort and enable greater flexibility. We use the Kerncraft (Hammer et al., 2017) tool that is able to automatically provide the ECM model for a particular kernel, based on code parsing and user-provided hardware specifications. Even though Kerncraft can significantly simplify the performance modelling endeavour, some effort is still required to transform the kernel code in a format that is digestible by the tool. For this reason we demonstrate an extension of the NMODL source-to-source compiler (Kumbhar et al., 2019a) to output kernel source code that can be directly analysed by Kerncraft. This approach, summarised in Figure 5.1, allows for a seamless integration approach from the computational scientist's description of an ion channel or synapse to the corresponding performance model and runtime predictions. Additionally, we are able to easily close the validation loop from performance model to simulation, by using the same NMODL tool to generate both the simulation code and the Kerncraft-transformed equivalent.

5.1.1 NMODL

Computational neuroscientists make extensive use of Domain Specific Languages to describe various aspects of their modelling efforts. Prominent examples include NeuroML (Gleeson et al., 2010), NineML (Raikov et al., 2011), NESTML (Plotnikov et al., 2016) and NMODL (Hines and Carnevale, 2000; Kumbhar et al., 2019a). The Blue Brain Project maintains an implementation of the cortical microcircuit based on the NEURON/CoreNEURON simulator (Carnevale and Hines, 2006), based on the NMODL language for the definition of ion channel and synapse models. For this reason, we focus in this work on the NMODL language and the homonymous source-to-source translator (Kumbhar et al., 2019a).

In a typical simulation workflow, scientists define new *in silico* models in files that respect grammatical and syntactic rules defined by the NMODL language constructs. A source-to-source compiler is then used to translate the user defined models into C/C++ code, which is ultimately compiled into an executable library that is dynamically linked to the NEURON/-CoreNEURON simulator. Historically the software responsible for source-to-source translation was embedded in the NEURON releases. The new NMODL compiler is based on an internal Abstract Syntax Tree (AST) representation (Kumbhar et al., 2019a). The internal representation can be manipulated to perform static analysis on the model, and obtain valuable information such as the number of variables and parameters, the data access patterns, the number of operations per kernel, and so on. In the following, we demonstrate an extension of the code generation backend to support analysis by the Kerncraft tool described below.

5.1.2 Kerncraft

Kerncraft (Hammer et al., 2015, 2017) is a performance modelling tool that allows one to automatically construct Roofline and ECM models for simulation loops. Kerncraft performs the required code and data transfer analysis from a well-formed loop code respecting a set of constraints. In this work we find that it is possible to sidestep some of the syntactic and data structure constraints imposed by the tool by implementing a post-processing step that complements the output of Kerncraft, thus making it feasible to obtain reasonably accurate performance predictions with low manual effort.

In brief, Kerncraft takes as input a `.c` file containing a definition of the constants and the loop to be analysed, in a simple format such as the one shown in Listing 5.1 for the STREAM triad kernel. Additionally, a machine specification file must be provided with information about the peak bandwidth and flop performance, as well as detailed information about the cache hierarchy. We base our analysis on the SKX-AVX512 machine described in the file `SkylakeSP_Gold-6148.yml` distributed with Kerncraft, since it mirrors the specifications of the SKX-AVX512 hardware used in the analysis presented in the previous chapters. Internally Kerncraft produces an abstract syntax tree representing the simulation loop from which it derived data access patterns. Additionally, a small binary executable is compiled to be used with IACA in order to obtain accurate predictions for in-core execution. In our configuration,

Listing 5.1 Example of Kerncraft input: STREAM triad.

```
#define N 10000
for(i=0;i<N;++i){
    a[i] = b[i] + k*c[i];
}
```

Kerncraft outputs a file containing information about the kernel's memory traffic, ECM model contributions and other useful details.

5.1.3 Extensions of NMODL's code generation backend for Kerncraft automatic performance analysis

The types of operations allowed in a loop to enable automatic analysis by Kerncraft are limited by a few constraints. We find that the constraints that mostly limit the out-of-the-box application of the tool, in our case, are:

- only the code of the loop and the definition of constant variables may be contained in the file;
- only primitive types and C-style array accesses are allowed;
- only single and double precision arrays are allowed;
- function calls are not allowed;
- indirect addressing is not allowed.

As the kernels generated by the NMODL compiler for simulations often contain C++ objects, integer index arrays for indirect accessing and function calls to user and system libraries, the limitations above make it impossible to directly use the simulation code for automatic performance analysis. Therefore it becomes necessary to implement a code-generation backend able to output Kerncraft-readable code, whilst remaining as faithful as possible to the corresponding simulation code.

While satisfying the first two requirements is a simple matter of discarding unneeded lines of code and potentially redefining some variables as C-style arrays, the other two constraints require a significant amount of effort. To avoid function calls in the loops we employ two complementary tactics. At first we invoke the source-to-source compiler using the **––inline** argument, which ensures that any calls to functions internally defined within the mod file will be inlined. However, this proves not to be sufficient, because in many cases functions from external libraries, namely exp,log from math libraries, are required for the resolution of ODEs within the mod file. In these cases we must distinguish between single-argument functions – e.g. exp – and multiple-argument functions such as pow. In the case of functions

Listing 5.2 Elision of mathematical function calls and indirect addressing in code for Kerncraft analysis. Original code *(left)* compared to modified code to enable parsing by Kerncraft *(right)*.

```
// Single-argument function
y = a + exp( -x/tau );

//Multiple-argument function
y = 10.*pow(4*x, b);

//Indirect addressing
ena[i] = ion_ena[ion_indices[i]];
```

```
// Single-argument function
y = a + (-x/tau); // increase exp counter
  by 1

//Multiple-argument function
arg_1 = 4*x;
arg_2 = b;
y = 10.*arg_1; // increase pow counter by 1

//Indirect addressing
ena[i] = ion_ena[i] // elided 1 indirect
  access
```

taking a single argument, we simply discard the function's name, but keep the evaluation of the argument expression, and increment a counter that stores the information of how many function calls have been elided in this way. For functions taking multiple arguments, we define new local variables for each argument and assign them to the corresponding expression, then we arbitrarily substitute the function call simply with one of its arguments. While both of the approaches described here completely hinder the results from evaluating the expressions containing the function calls, they have the property of maximally preserving the performance properties, by making sure that argument expressions are still evaluated. Examples of dealing with both single- and multiple-argument functions are provided in Listing 5.2. In a subsequent post-processing step, the function counters are read and the ECM T_{OL} estimate is updated with the relevant throughput values, which can be obtained by benchmarking or tables such as (Fog, 2017).

To avoid indirect addressing, we replace them by direct accesses as shown in Listing 5.2 and again increase a counter to keep track of our transformations. In this case the post-processing step is more complicated. Indeed, not only must we account for the additional memory traffic due to reading the **int** index variable, we must integrate additional information about the biological entity – i.e. ion channel or synapse – and the type of kernel – current or state – whence the loop was extracted to inform the heuristics we developed in Section 3.1. Using these heuristics we can adjust the memory traffic estimated by the Kerncraft tool, which in turn allows us to update the $T_{L1L2}, T_{L2L3}, T_{L3Mem}$ ECM contributions in the performance model. While it is true that such transformations lead to an incorrect estimate of T_{nOL} that cannot be corrected in a post-processing step, we find that the overall impact on the model's accuracy is low.

kernel name	type	T_{OL}	T_{nOL}	T_{L1L2}	T_{L2L3}	T_{L3Mem}
Ca_HVA2	current	1.06	0.75	3.38	9.00	4.96
Ca_LVAst	current	1.06	0.75	3.38	9.00	4.96
Ih	current	0.62	0.31	1.56	4.25	2.30
KdShu2007	current	0.97	0.62	2.69	7.25	3.95
K_Pst	current	1.06	0.75	3.38	9.00	4.96
K_Tst	current	2.54	0.62	3.38	9.00	4.96
Nap_Et2	current	3.10	2.02	4.50	12.50	6.61
NaTg	current	3.11	2.02	4.50	12.50	6.61
ProbGABAAB_EMS	current	1.69	1.12	3.12	8.50	4.59
ProbAMPANMDA_EMS	current	15.86	1.01	3.25	8.50	4.78
SK_E2	current	2.36	1.40	3.81	10.25	5.60
SKv3_1	current	0.96	0.51	3.25	8.50	4.78
Ca_HVA2	state	19.44	1.12	3.38	8.00	5.24
Ca_LVAst	state	18.88	1.00	2.38	6.00	3.69
Ih	state	10.28	0.51	1.56	3.75	2.43
KdShu2007	state	9.85	0.44	1.81	4.25	2.82
K_Pst	state	18.05	1.01	2.38	6.00	3.69
K_Tst	state	19.84	1.01	2.38	6.00	3.69
Nap_Et2	state	27.30	2.01	4.50	11.50	6.99
NaTg	state	25.12	2.39	5.00	13.50	7.35
ProbGABAAB_EMS	state	1.61	1.12	1.69	4.75	2.48
ProbAMPANMDA_EMS	state	1.61	1.12	1.69	4.75	2.48
SK_E2	state	6.65	0.69	2.06	6.25	3.03
SKv3_1	state	8.84	0.56	1.62	4.50	2.39

Table 5.1 – Automatically obtained ECM models for all cortical microcircuit kernels. All models are based on the SKX AVX512 reference architecture. Quantities are expressed in cycles per scalar iteration.

5.1.4 Automatically obtained ECM models of ion channel and synapse kernels

We apply the automatic performance modelling and post-processing workflow to the set of ion channel and synapse kernels within the Reconstructed *in silico* model. In total, we consider 24 kernels equally divided among current and state types. The obtained ECM models are presented in Table 5.1, while the corresponding predictions for different levels of the cache hierarchy are provided in the appendix Table C.1. For ion channels we confirm the trends observed in Chapter 3: current kernels have low arithmetic intensity and scattered loads/stores while state kernels have larger in-core components even at the maximum vector register size. Excitatory synaptic current kernels are also characterised by the same boundary behaviour observed in Chapter 3, causing a switch from a core-bound profile when data is in the caches to a data-bound profile when data is in main memory, while inhibitory synaptic currents are clearly data-bound. On the other hand, synaptic state kernels are characterised

by a very low arithmetic intensity, in stark contrast with what we observed previously. This can be traced back to a manual optimisation that is embedded in the description of the model's ODEs based on the observation that the dynamics are independent of external variables, and thus it is possible to precompute the update value in the case of fixed timestep integration. While in Chapter 3 our purpose was to remain as general as possible and therefore we did not consider this optimisation, in this chapter we include it to give a concrete analysis of the specific cortical model under consideration.

5.1.5 Validation

In order to validate the predictions, we developed a synthetic benchmark in which a single dendrite is endowed with as many instances of a given channel are required to safely exceed the L3 capacity of a single chip, thus ensuring that data resides in main memory. Then a simulation is executed using the CoreNEURON software, adequately instrumented to be able to measure the execution time of individual kernels at the granularity of cycles. The benchmarks were run on the SKX-AVX512 machine described in Section B.2. We present the main validation results in Table 5.2, and a detailed validation of the shared-memory scaling predictions is provided in Figure C.2 in Appendix C.

Confirming the patterns we already saw in Section 3.1, serial and parallel predictions for the data-traffic bound current kernels are satisfactorily correct, with errors always beneath the 30% threshold. For the compute-bound state kernels, both serial and parallel predictions are always pessimistic by a factor between 20%-50%, with only three error peaks of almost a factor 2x between measurements and prediction. The higher error margin in the state kernels has already been observed and can be explained by the difficulty in modelling the intricacies of the out-of-order engine in our Skylake reference architecture (Cremonesi et al., 2019a), but in the case of automatic modelling it could also be attributed to a mismatch between the automatically generated object file analysed by IACA under the hood of the Kerncraft tool and the compiled code for the benchmark. While we would have expected an higher accuracy in the parallel predictions, especially compared to our pen-and-paper model from Chapter 3, we speculate that the automatic modelling might be introducing some error in estimating the memory traffic.

Overall we computed predictions of state and current kernels for 10 ion channel types and 2 synapse types, for 1,2,4,8,16,18 threads, for a total of 144 predictions, out of which over 80% had an error margin below the 30% threshold. The error for the state kernels was in general larger as shown in Figure 5.2 for the reasons of out-of-order scheduling discussed above but none of the predictions exceeded an error margin of 50%. In total these predictions can be considered satisfactory and while they could be improved by a deeper investigation in the details of the Kerncraft tool, we are confident that for our purposes of capturing relevant bottlenecks this level of accuracy is suitable.

		serial			parallel		
kernel name	type	pred	meas	error	pred	meas	error
Ca_HVA2	cur	18.08	24.87±0.11	27.29	4.95	3.99±0.01	24.09
Ca_LVAst	cur	18.08	24.08±0.14	24.90	4.95	3.99±0.01	24.03
Ih	cur	8.42	11.77±0.07	28.49	2.29	2.16±0.01	6.12
KdShu2007	cur	14.51	20.20±0.14	28.16	3.94	3.13±0.01	26.12
K_Pst	cur	18.08	25.11±0.11	27.98	4.95	4.00±0.01	23.92
K_Tst	cur	17.95	23.51±0.19	23.62	4.95	4.00±0.01	23.74
Nap_Et2	cur	25.63	31.23±0.15	17.91	6.61	5.37±0.01	23.13
NaTg	cur	25.63	30.85±0.18	16.91	6.61	5.37±0.03	23.11
ProbAMPANMDA_EMS	cur	17.53	30.38±0.22	42.28	4.77	4.71±0.07	1.37
ProbGABAAB_EMS	cur	17.34	30.12±0.15	42.42	4.59	4.48±0.01	2.60
SK_E2	cur	21.06	25.83±0.08	18.45	5.60	4.49±0.01	24.63
SKv3_1	cur	17.03	22.44±0.10	24.07	4.77	3.82±0.01	24.73
Ca_HVA2	state	19.43	26.96±1.42	27.92	5.24	4.27±0.08	22.66
Ca_LVAst	state	18.87	25.70±1.34	26.58	3.69	2.90±0.07	26.86
Ih	state	10.27	15.97±0.01	*35.67*	2.42	1.94±0.04	24.80
KdShu2007	state	9.85	11.37±0.01	13.43	2.81	2.27±0.04	23.79
K_Pst	state	18.05	26.15±0.99	30.99	3.69	2.92±0.06	26.32
K_Tst	state	19.83	22.77±0.92	12.91	3.69	2.90±0.06	27.02
Nap_Et2	state	27.30	49.12±0.12	*44.42*	6.99	5.59±0.09	24.95
NaTg	state	28.23	47.01±0.16	*39.94*	7.34	6.32±0.05	16.16
ProbAMPANMDA_EMS	state	10.04	10.76±0.09	6.71	2.48	2.01±0.01	23.19
ProbGABAAB_EMS	state	10.04	10.70±0.07	6.18	2.48	2.03±0.03	21.67
SK_E2	state	12.03	17.62±0.03	31.73	3.03	2.06±0.03	*46.96*
SKv3_1	state	9.076	15.15±0.26	*40.12*	2.38	1.87±0.05	27.17

Table 5.2 – Validation of automatic ECM models (single-thread, data in memory). The data was obtained by benchmarking individual channels as described in the text, using the SKX AVX512 reference architecture. The dataset was always large enough to fit in main memory. The unit of measure for predictions and measurements is cycles per scalar iteration and measurement data is shown as median ± IQR over 10 repetitions.

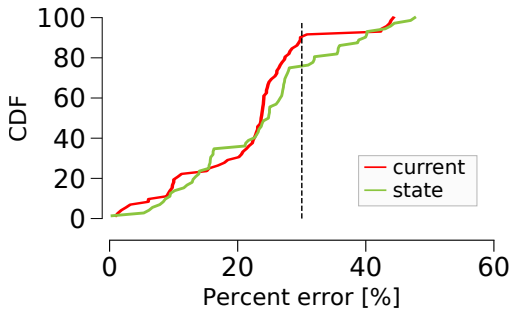

Figure 5.2 – Cumulative distribution function of the automatic prediction error margin for state and current kernels. All predictions for 1,2,4,8,16 and 18 threads were considered.

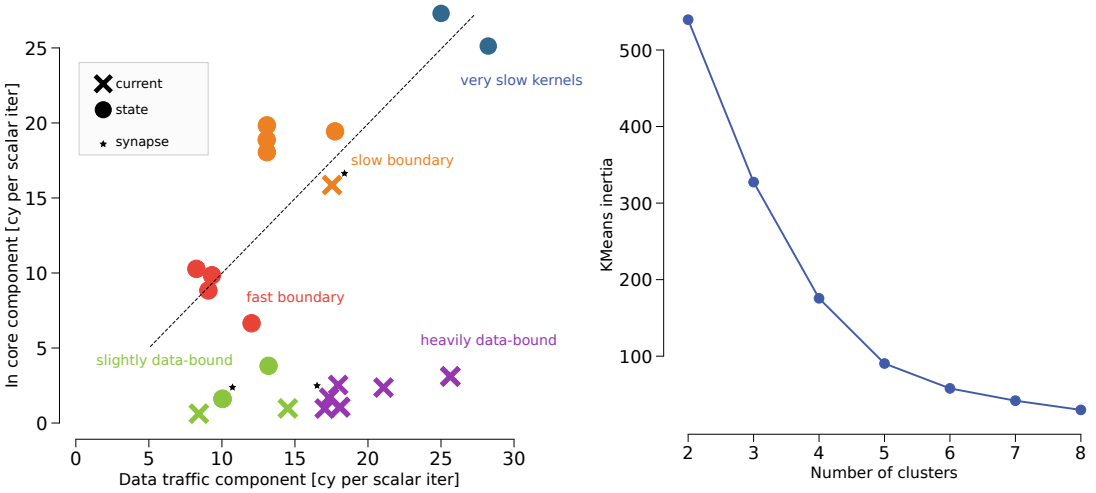

Figure 5.3 – Serial performance profile of individual instances of ion channel and synapse kernels. *Left*: result of K-means clustering of state and current kernels. Different colours denote different clusters, while markers denote current and state kernels. The dashed line separates compute-bound (over the line) from data-bound (under the line) kernels. Markers corresponding to synaptic kernels are annotated with a black star (the two synaptic state kernels are overlapping). *Right*: Inertia of the K-means algorithm as a function of the number of clusters. Used to determine the optimal number of clusters (K=5) using the elbow method.

5.2 Heterogeneity in ion channel and synapse computational properties

We analyse the intrinsic performance properties of the computational kernels of individual ion channels and synapses within the Reconstructed microcircuit model. Our analysis is aimed at exposing the hardware bottlenecks and defining the performance characteristics of each computational loop. This information is relevant not only to quantitatively describe the performance of simulations of biological neurons, but it also allows us to explore algorithmic optimisations – such as overlapping of compute and memory bound kernels – as well as provide codesign directions – such as quantifying the ideal number of parallel cores – for future architectures.

Using the ECM dimensions as a representation of a kernel's computational properties, we explore the variability in ion channel and synaptic models. We observe a large variability in the T_{OL} dimension (i.e. the in-core execution) and, as expected, a strong correlation between T_{nOL}, T_{L1L2}, T_{L2L3}, T_{L3Mem} is observed (see Figure C.1 in Appendix C). These two observations lead us to a dimensionality reduction strategy, and we only consider the two dimensions T_{core}, T_{data} defined in (4.1) to describe the computational properties of a kernel.

We plot the ion channel and synaptic kernels' data on the two axis defined in (4.1) in Figure 5.3, where the black bisector line distinguishes compute-bound from data-bound kernels. Marker

types distinguish kernel types: circles for state and crosses for current kernels. All current kernels lie well below the bisector line indicating that they should be considered data-bound. The excitatory synapse current kernel (ProbAMPANMDA_EMS current) lies very close to the boundary because its heavy data-traffic requirements are offset by the presence of several compute-heavy functions such as div,exp. State kernels, on the other hand, all lie very closely to the boundary or slightly over it, thus should be considered core-bound by a narrow margin. For ion channel kernels we thus confirm our findings from Chapter 4 that current kernels are mainly data-bound, while state kernels lie on the boundary. As we found in our analysis on different levels of vectorisation in Chapter 3, this is a direct consequence of using the maximum level of vectorisation – i.e. AVX512 – while smaller vector registers would result in larger T_{core} components. This could also be interpreted as an indication that, keeping all other parameters fixed, AVX512 is the maximum level of vectorisation that can still provide significant runtime benefits. For synapse kernels we observe a different performance profile in both the state and current kernel compared to the model examined in Chapter 4. In the case of the current kernel, this is due to the fact that our previous model was an average of the excitatory synapse and inhibitory synapse models, while in this chapter we keep them separated. In the case of the state kernel, a manual optimisation has been built-into the original model based on the observation that the state update does not depend on voltage, and therefore can be pre-computed, effectively turning a core-bound kernel with several evaluations of the exp function into a (faster) data-bound kernel. This example showcases the tight relationship between biological models, their software implementation and their performance profile, and reminds us that our analysis is conditioned on all of these aspects.

To gain better insight into the classes of kernels in our dataset, we also colour the kernels in Figure 5.3 according to the result of applying the K-means clustering techniques. On the right of Figure 5.3 we plot the K-means inertia as a function of the number of clusters. Using the elbow method, we define 5 to be a satisfying amount of clusters (Ketchen and Shook, 1996). We interpret the clusters as follows: heavily data-bound kernels with long runtime, slightly data-bound kernels with shorter runtime, and three boundary profiles distinguished by their runtime (fast, slow and very slow). At a first approximation we observe that all heavily data-bound kernels are of the current type, and all boundary kernels are of the state type (with the only exception being the core-bound excitatory synapse current). The only group having a mix of types is the slightly data-bound cluster.

5.2.1 Overlap of kernels having complementary performance properties

The analysis above presented a successful characterisation of ion channel and synapse kernels, but does not include important details about the execution algorithm such as the order of kernels and the potential for overlapping or concurrent execution. The reference implementation in CoreNEURON is based on a loop over ion channel and synapses such that different types are execute serially, but within a type parallelism at the instance level is exploited. An alternative approach was investigated by analysing the dependency graph to exploit potential parallelism

in the execution of independent kernels (Magalhaes et al., 2019b). Their analysis pointed to an inefficiency in the state-of-the-art implementation: whenever the memory bandwidth is not fully saturated, it could be used to fetch data for subsequent data-intensive kernels, and conversely if a kernel's execution is dominated by waiting for data from memory, the stalled core could be used to perform operations for the compute-intensive kernels. We investigate a static solution based on fusion of kernels with complementary performance properties. The automatic code-generation tool leads potentially to a straightforward implementation by fusing the code and data-structures for different kernels in the same file.

We now describe how to use the performance modelling method to estimate the potential benefits from overlapping. Given that almost no data structures are shared between kernels, we can safely estimate that the data traffic components of the fused kernel will be given by the sum of the individual components. For the in-core contribution, the out-of-order engine should be able to sustain a better throughput than the sum of the two sources, but since this is hard to predict and impossible to guarantee, we take the conservative assumption that the T_{OL} of the fused kernel is given by the sum of the source components. These two considerations enable the modelling of a fused kernel based on the ECM models of the two sources.

We define the benefit from overlapping as the ratio of the sum of the source runtime over the runtime of the fused kernel:

$$S = \frac{T_1 + T_2}{T_{fused}} \tag{5.1}$$

Figure 5.4 shows all possible fusions of two kernels, and the corresponding overlap benefit. Our analysis shows that overlapping two current kernels, regardless of the underlying biological entity, would provide virtually no benefit. Moreover, the maximum observed benefit happens with the (Ih current, K_Tst state) combination with a 1.31x speedup. As a general rule, the only combinations that provide a significant speedup are overlapping ion channel state with ion channel current kernels, or overlapping ion channel state with synapse state kernels. The benefits from overlapping ion channel state kernels are unclear, with a few select combinations being able to deliver a speedup just under a factor 1.2x, but most combinations having a runtime that is no better than the sum of the two source kernels. Finally, the performance model predicts that overlapping synapse current kernels with ion channel state kernels only provides benefits in the case of inhibitory synapses. This can be directly attributed to the fact that the excitatory synapse current kernel is compute bound due to the presence of exp function. We present a summary of the overlap benefits in Table 5.3. Given the potential risk of numerical error when fusing kernels, we point out that the only completely safe option would be the overlapping of synapse state kernels with select ion channel state kernels. In this case, the largest benefit can be obtained with the (ProbAMPANMDA_EMS state, K_Tst state) combination, yielding a 1.29x improvement factor.

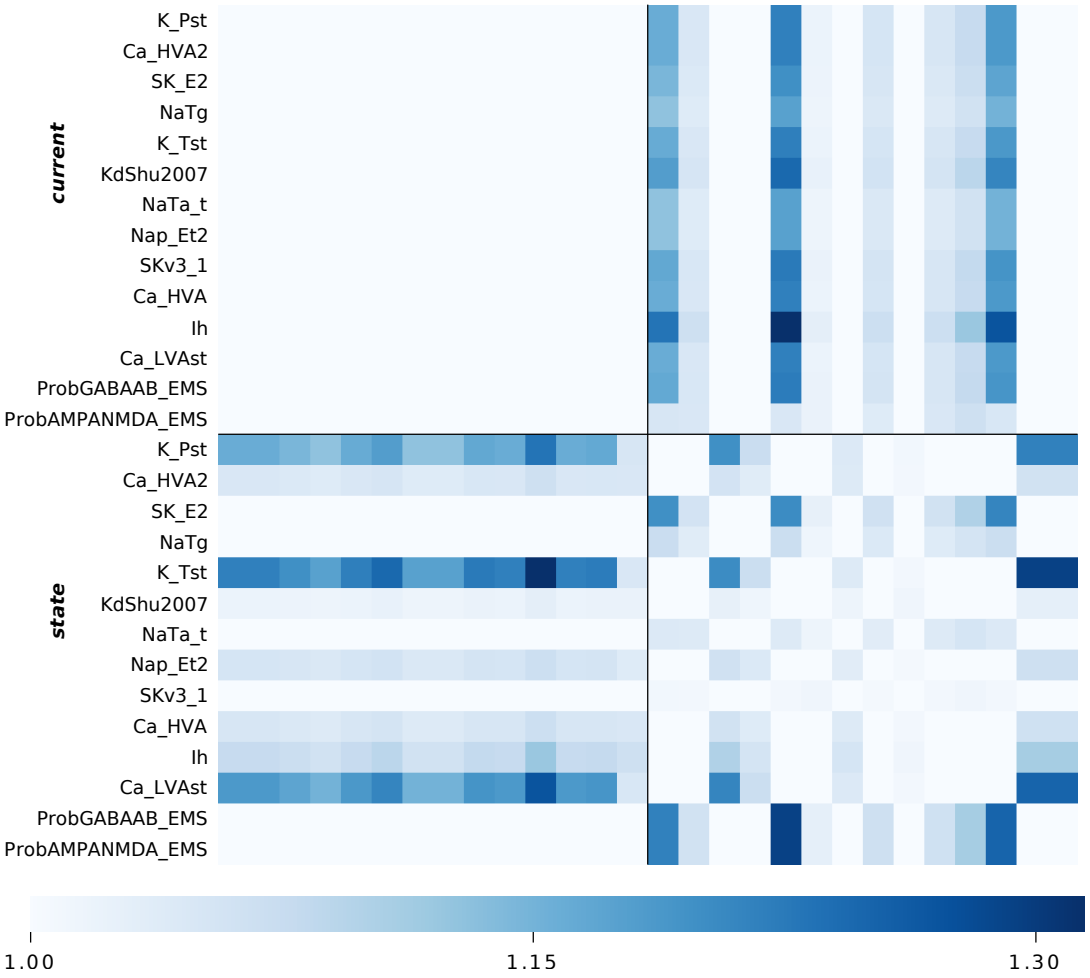

Figure 5.4 – Potential serial performance improvement from overlapping of different kernels. Heatmap of the maximum theoretical benefit from overlapping kernels defined in 5.1. The dark continuous lines separate current and state kernels, while the dashed lines separate ion channel and synapse kernels.

		current		state	
		ion channel	synapse	ion channel	synapse
current	ion channel	bad	bad	**good**	bad
	synapse		bad	unclear	bad
state	ion channel			unclear	**good**
	synapse				bad

Table 5.3 – Summary of predicted performance improvement from kernel overlap.

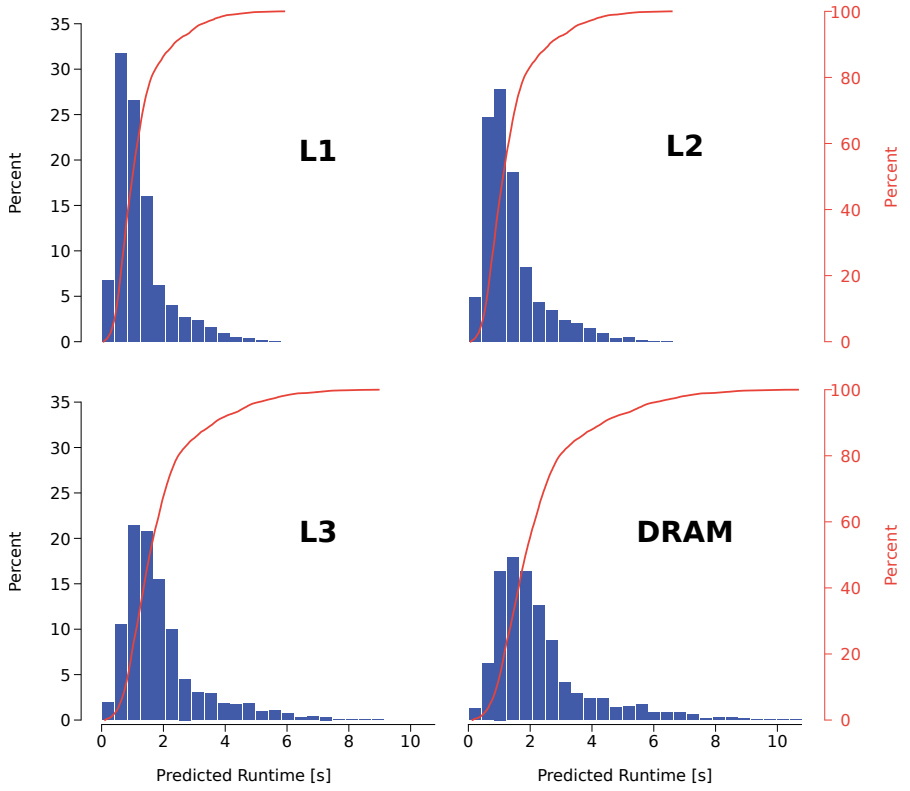

Figure 5.5 – Distribution of predicted serial runtime of whole neurons as a function of the memory hierarchy level where data resides. Runtime is measured in wallclock seconds to simulate one second of biological time. The DRAM predictions are obtained neglecting the optimised loop ordering, i.e. assuming that data must come from main memory at each timestep iteration. Each histogram bar represents a 0.4s bin and refers to the axis on the left. Red lines show the cumulative distribution function and refer to the axis on the right.

5.3 Heterogeneity in the performance properties of neurons

We complement our previous analysis on the performance properties of individual ion channel and synapse kernels by considering additional information from the *in silico* model, i.e. by considering the mix of ion channel and synapse instances in individual neurons. We intersect the performance model of individual kernels with the information of the distribution of ion channels and synapses in different neurons within the microcircuit. The goal of this section is pointing out fundamental and intrinsic performance properties of neurons within the cortical microcircuit, to extract general guidelines for software optimisation, hardware codesign and efficient utilization of resources.

	L1	L2	L3	DRAM
min	0.067	0.077	0.111	0.133
median	0.999	1.123	1.553	1.849
max	5.909	6.580	8.965	10.667

Table 5.4 – Summary of distribution of predicted serial runtime. Runtime is measured in wallclock seconds to simulate one second of biological time.

5.3.1 Serial performance

We examine the distribution of predicted serial runtime over neurons. In this context, the predicted runtime of a neuron is defined as the sum of contributions from individual kernels (i.e. from state and current kernels of ion channels and synapses), where each contribution is given by the ECM prediction of that kernel multiplied by the number of instances of that ion channel or synapse in the neuron. Thus for a neuron defined by a set of kernels $\{k\}_i$ and corresponding number of instances a_i, we have that:

$$T^{Mem}(neuron) = \sum_i a_i T_i^{Mem}. \tag{5.2}$$

Throughout this chapter we ignore the linear algebra and the spike delivery kernels because our goal is to analyse the effect of the heterogeneity of ion channels and synapses on neurons' performance. Thus in the rest of this chapter we will take the word neuron to define the sum of its ion channel and synapse kernels. While this approach considerably simplifies the analysis, we do not expect it to have a big impact on the outcome because, as we have shown, the runtime is generally dominated by these kernels anyway.

At first we look at the distribution of predicted runtime for different levels of the cache hierarchy, as shown in Figure 5.5. As expected the overall runtime increases as data resides in lower levels of the cache. For example when data is in L1 only 10% of neurons have a runtime larger than 2.3s wallclock seconds to simulate one biological second, while this percentage rises to 34% when data is in DRAM. Table 5.4 reports minimum, median and maximum predicted runtime. Comparing L1 to DRAM there is almost a factor 2x increase in all runtimes, indicating that cache locality affects all neurons in roughly the same way. However, we would expect speedups of between 2.4x–3.4x just by comparing L1 bandwidth with DRAM bandwidth. Finally, we observe an important tail of "slow" neurons when data resides in DRAM.

This type of performance profile in which performance improves with cache locality is typical of data-bound kernels, however we do not observe the ideal speedup. We therefore examine the T_{core} and T_{data} contributions for neurons by computing the sum over all ion channel and synapse kernels belonging to each cell. Note that while the runtime of a single kernel is given by the maximum of its T_{core} and T_{data} components, the runtime of the whole neuron is given by the sum of the T_{core} or T_{data} components. This can be explained by the process

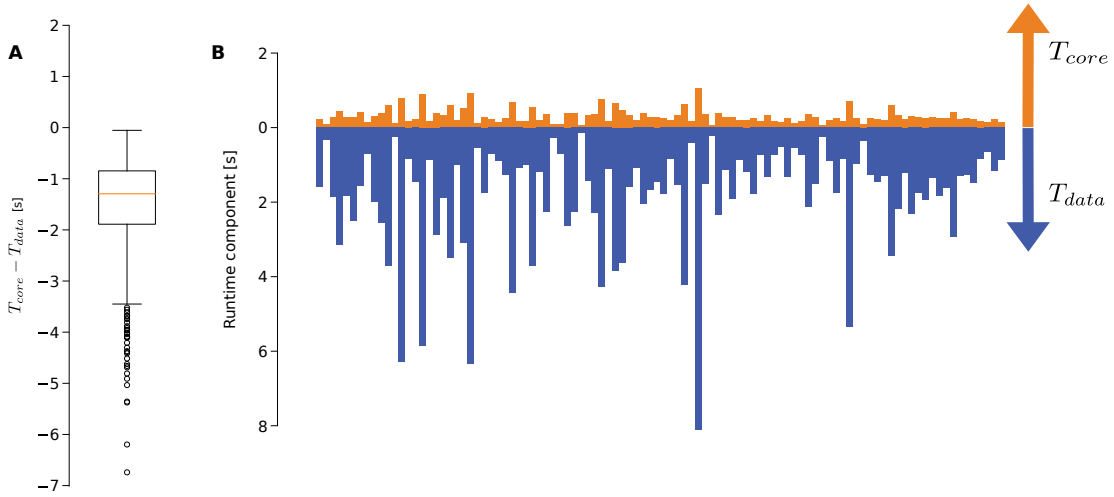

Figure 5.6 – Single-thread T_{core} and T_{data} contributions by neuron. We computed the serial T_{core} and T_{data} per neuron to simulate one second of biological time, assuming data comes from main memory at each timestep iteration. We report their values in wallclock seconds. **A** boxplot of the difference $T_{core} - T_{data}$ in a serial execution. Negative values correspond to neurons for which the time spent in data-bound kernels is larger than the time spent in core-bound kernel. The whiskers in the boxplot are located at 1.5x the interquartile range, and points outside of this interval are considered outliers. **B** a sample of the T_{core} and T_{data} components of 100 randomly selected neurons.

that leads to the computation of these two quantities. For a neuron defined by a set of kernels $\{k\}_i$ and corresponding number of instances a_i, let $T_{core,i}$ be either equal to 0 if the kernel was data-bound, or equal to the T_{core} value of the i^th kernel in the neuron, and respectively for T_{data}, such that for each individual kernel only one of T_{core}, T_{data} will be different than 0. Then we have

$$\begin{aligned} T_{core}(neuron) &= \sum_i a_i T_{core,i}, \\ T_{data}(neuron) &= \sum_i a_i T_{data,i}. \end{aligned} \tag{5.3}$$

Figure 5.6A shows a boxplot of the difference $T_{core} - T_{data}$, with positive values denoting neurons in which the sum of T_{core} components from kernels was larger than the sum of T_{data} components. As we explained before, a positive difference does not imply that a neuron is core-bound, rather it means that a larger portion of the neuron's runtime is spent in core-bound kernels rather than data-bound kernels. The distribution is quite strongly skewed towards the T_{data} component, with all the neurons having $T_{data} \geq T_{core}$, and almost half of the neurons spending on average 1.3s of wallclock time longer in data traffic than in computation.

We are also interested in understanding the relationship between T_{core} and T_{data}. We randomly select a subset of neurons and plot the two components side-by-side in Figure 5.6B.

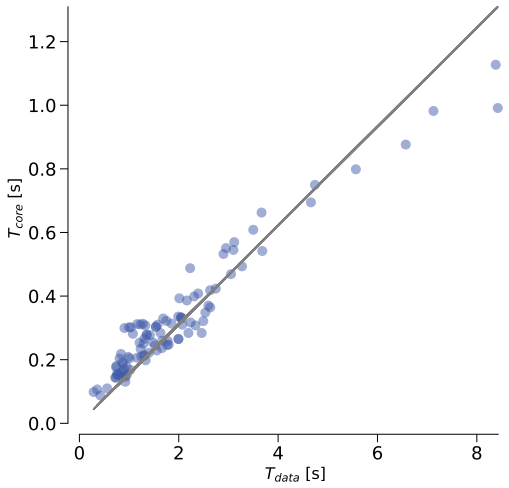

Figure 5.7 – Linear relationship between serial T_{core} and T_{data} per neuron. We computed the serial T_{core} and T_{data} to simulate one second of biological time, assuming data comes from main memory at each timestep iteration, for each neuron. We report their values in wallclock seconds. We computed the linear fit $T_{core} = 0.156 T_{data}$, with score $r^2 = 0.97$.

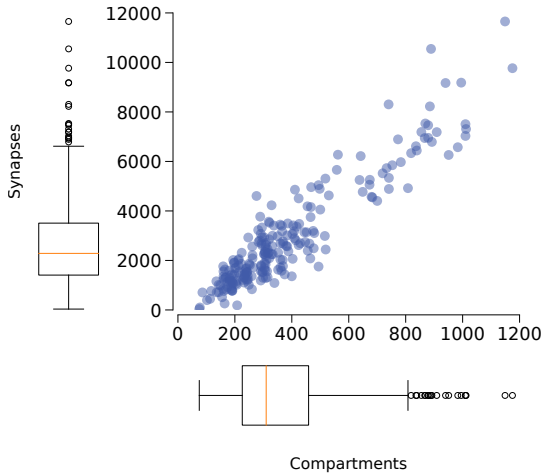

Figure 5.8 – Number of compartments and synapses co-vary. We plot the number of compartments and synapses per neuron. The boxplot present the distribution of the individual axis. The whiskers in the boxplot are located at 1.5x the interquartile range, and points outside of this interval are considered outliers.

This plot highlights a possible correlation: neurons with a larger T_{core} seem to also have a large T_{data}. Computation of Pearson's correlation coefficient yields $\rho = 0.97$, lending high credibility to this hypothesis. We therefore fit a linear regression model to quantify the relationship between the core and data components. We find that within each neuron T_{data} is roughly six times larger than T_{core}, with a high score for the linear fit ($r^2 = 0.89$, see Figure 5.7).

This correlation suggests the existence of an underlying performance factor that determines whether a neuron is a *fast* neuron (i.e. small runtime) or a *slow* one. We investigate the relationship between the number of compartments, number of synapses and the predicted runtime, motivated by the fact that these quantities are typically reported as a proxy for complexity. We fit a linear model $\mathbf{y} = \mathbf{AX} + \mathbf{b}$, with input $\mathbf{X} = [\mathbf{c}|\mathbf{s}]$, where $\mathbf{c}, \mathbf{s}$ represent the number of compartments and synapses respectively for each neuron, and output the predicted serial runtime computed as the sum of all current and state kernels of that neuron when data is in memory $\mathbf{y}$. We compute the 20-fold cross-validation results, yielding high score and accuracy, as reported in the first row of Table 5.5.

The number of compartments and synapses in a neuron strongly co-vary in the Reconstructed model, as shown in Figure 5.8(Pearson's ρ=0.94, p-value < 1e-10) . This can be easily explained by the fact that, even though compartments can have different sizes, it is generally true that

		median r^2	IQR r^2	median MSE	IQR MSE
aggregate	both	0.99	0.01	0.00	0.00
	only synapses	0.95	0.05	0.03	0.03
	only compartments	0.88	0.16	0.08	0.06
current	both	0.99	0.01	0.00	0.00
	only synapses	0.97	0.03	0.01	0.01
	only compartments	0.86	0.20	0.03	0.03
state	both	0.98	0.01	0.00	0.00
	only synapses	0.91	0.10	0.01	0.01
	only compartments	0.91	0.10	0.01	0.01

Table 5.5 – Summary of test-set metrics for submodels and full model. We considered state and current kernels both as an aggregate, or separated. For each group, we fitted the predicted wallclock time using either only the synapses, only the compartments, or both as a predictor. For each combination we computed a 20-fold cross-validation, and report the r^2 score and the Mean Square Error (MSE).

a large number of compartments is indicative of a *large* cell, in the sense of total dendritic length. Since in the Reconstructed microcircuit synapses are formed where a presynaptic axon touches a postsynaptic dendrite, this leads to the fact that in this *in silico* model *larger* neurons have more synapses. Building on this information, we investigate whether only one of the two factors is sufficient to explain the runtime. We therefore fit a linear model using only the compartments (or synapses), but the resulting score and Mean Squared Error (MSE) were both significantly reduced, and their variability increased (see second and third line of the "aggregate" row in Table 5.5). We conclude that neglecting either the number of compartments or the number of synapses leads to a model with larger bias and larger variance, thus greatly reducing the explainability. In short, our results strongly indicate that both the number of compartments and the number of synapses are required to explain the variability in the runtime of morphologically detailed neurons.

We refine our analysis by looking at state kernels' and current kernels' runtimes separately. We hypothesize that the runtime of state kernels is correlated with the number of compartments, while the runtime of current kernels is correlated with the number of synapses. We therefore fit separate linear models for the state and current kernels. In both cases, using the full model including compartments and synapses leads to excellent scores and low prediction error, confirming our results from before. Using only compartments to predict current kernels' performance leads to a degraded model with larger bias and variance, while using synapses preserves the goodness of fit of the model. This confirms our hypothesis that the variability in the runtime of current kernels is best explained by the number of synapses. In state kernels, on the other hand, the single-feature models display a similar loss in accuracy and score, indicating that no single factor is dominant in determining the variability in the performance

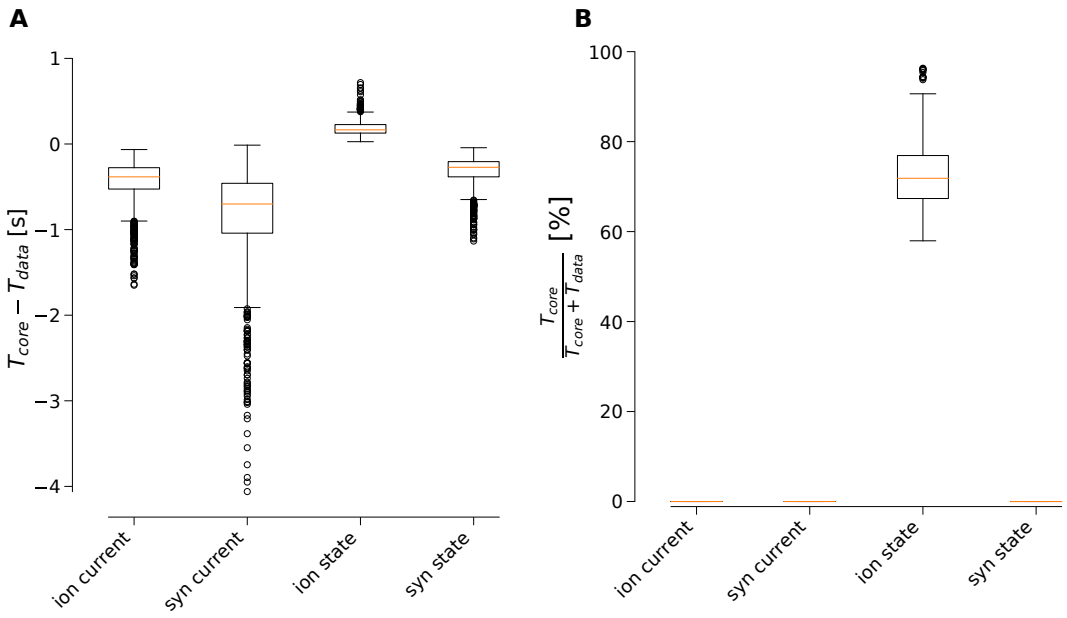

Figure 5.9 – Distribution of serial runtime properties of kernel families per neuron. We computed the serial T_{core} and T_{data} to simulate one second of biological time, assuming data comes from main memory at each timestep iteration, for each neuron. We report their values in wallclock seconds. **A** boxplot of the raw difference $T_{core} - T_{data}$, separated by kernel type. The whiskers in the boxplot are located at 1.5x the interquartile range, and points outside of this interval are considered outliers. **B** boxplot of the relative impact of T_{core} over the total runtime: $\frac{T_{core}}{T_{core}+T_{data}}$, separated by kernel type. The whiskers in the boxplot are located at 1.5x the interquartile range, and points outside of this interval are considered outliers.

of state kernels. The results of 20-fold cross-validation of the fitted models are provided in the "state" and "current" rows of Table 5.5.

Our analysis so far has established a direct relationship between the serial runtime of kernels and variability in model parameters such as the number of compartments and synapses. However, we lack an understanding of the computational properties – i.e. core or data boundedness – of state and current kernels in neurons. Figure 5.9A shows the distribution of $T_{core} - T_{data}$ separated by biological entity (ion channel or synapse) and kernel type (current or state). Each data point represents and aggregate of kernels of a given type and biological entity for a single neuron. For example the first boxplot in Figure 5.9A demonstrates that for 50% of the neurons, when computing ion channels' currents, the runtime spent in traffic-bound kernels is larger by roughly 0.4 s than in core-bound kernels. We draw the following conclusion: when computing ion channel currents and synaptic currents and states, more runtime is spent in traffic-bound kernels than in compute-bound kernels, while computing ion channel states leads to a balanced profile. Although looking at the difference of $T_{core} - T_{data}$ highlights important information about the absolute time spent in the core-bound and data-bound regimes, it remains unclear how this relates to the total runtime. In Figure 5.9B we plot the percent

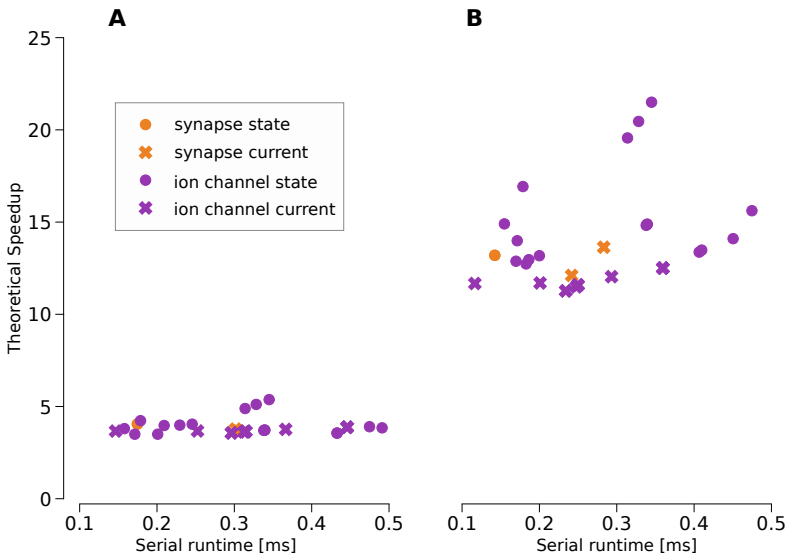

Figure 5.10 – Parallel speedup of individual ion channel and synapse instances. **A** Maximum theoretically achievable speedup and predicted serial runtime for a single instance and a single iteration of the kernel. Serial runtime is reported in ms to simulate one instance for one second of biological time. In this version of the plot we assume that data must be fetched from main memory at each iteration. **B** Maximum theoretically achievable speedup and predicted serial runtime for a single instance of the kernel, but simulated for a whole second of biological time, assuming the loop ordering technique minimising memory pressure is used.

of time spent in the core-bound regime over the total runtime for that specific combination. While computing ion channel currents and synaptic currents and states, 100% of the time is spent in the data-bound regime. On the other hand, half of the neurons spend more than 70% of runtime in the core-bound regime when computing ionic states.

5.3.2 Shared memory parallel performance

We have identified the salient performance features of the serial simulation of neurons in the Reconstructed microcircuit, but our performance analysis must be completed by accounting for parallelism. As previously explained in Section B.1 the presence of a serial memory bottleneck leads to a saturation effect that imposes a bound on the maximum achievable speedup using parallelism. We therefore compute the theoretical speedup (i.e. the saturation point) as defined in (B.12) for all the kernels in the Reconstructed model. Figure 5.10A shows the serial runtime and the corresponding maximum achievable speedup from parallelism for the simulation of 1s of biological time for a single instance of state and current kernels. None of the current kernels can achieve a larger speedup than roughly 4x and in general the maximum achievable speedup does not exceed 6x.

Compared to the amount of parallelism exposed by modern architectures and future many-

core architectures, which could reach hundreds of parallel threads, this represents a serious drawback in performance. The loop ordering optimisation described in 2.1.3, which allows us to increase the saturation point by exploiting cache reuse, has been conceived to overcome the saturation problem. For the purposes of performance modelling we can reuse formula (B.12) considering as an iteration not a single timestep but a whole mindelay, composed of one timestep in which data comes from memory, and a sequence of timesteps in which data comes from the L3 cache. For example, supposing $\Delta t = 0.025$ ms and $\delta_{\min} = 0.1$ ms, which are the values for the Reconstructed *in silico* model, after the first iteration with data from DRAM there are three more iterations with data from L3. This leads to the following modified ECM model:

iteration definition	ECM model
Δt	$\{T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L3Mem}\}$
$\delta_{\min}$	$\{4T_{OL} \parallel 4T_{nOL} \mid 4T_{L1L2} \mid 4T_{L2L3} \mid T_{L3Mem}\}$

Figure 5.10B shows the wallclock time to simulate a single instance for one second of biological time in a serial execution, and the corresponding speedup, when using the loop ordering optimisation. The serial runtimes for some kernels are slightly improved by the reuse but the largest effect is demonstrated in the speedup from parallelism. Despite the loop ordering optimisation, current kernels still benefit significantly less than state kernels from parallelism, with the average ideal speedup being around 12x for current kernels and around 15x for state kernels. The largest speedup predicted by the model are around 22x for some ion channel state kernels, which barely exceeds the number of parallel threads of our reference architecture. Overall, a distinction becomes clear between highly parallelisable kernels (a subset of state kernels) and moderately parallel kernels whose maximum speedup does not exceed 15x.

Analysing single kernels does not provide a complete understanding of the benefits from parallelism, since during execution the global speedup will always be limited by the slowest components in the simulation. We therefore look at the average memory bandwidth utilization, defined for individual kernels in (B.16). We rework the definition of bandwidth utilization to account for the fact that the unit of interest is a whole neuron. Therefore we define the average bandwidth utilization of a neuron by the ratio of sums over kernels k of that neuron:

$$BW_{util} = \frac{\sum_k T_{L3Mem,k}}{\sum_k T_k^{Mem}}, \tag{5.4}$$

which is equivalent to computing the average BW_{util} of each kernel in the neuron, weighted by the kernel's execution time over the total runtime.

Figure 5.11 shows the distribution of the average bandwidth utilization per neuron. In total, the memory bandwidth is saturated around 12 parallel threads, which can be considered a low number compared to modern HPC architectures exposing tens of parallel cores. Moreover, consistently with our earlier analysis we observe a large difference between early-saturating

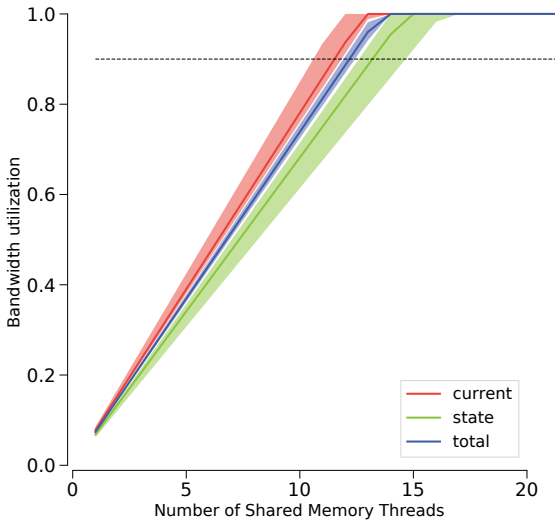

Figure 5.11 – Average bandwidth utilization per neuron. Utilization is measured as the ratio to peak memory bandwidth, and is computed on a per-neuron basis using (5.4). The solid lines represent the median across neurons, while the shaded areas represent the minimum and maximum across neurons.

current kernels and late-saturating state kernels. Finally, the variability is quite low, with the minimum and maximum across neurons being quite close to the median. None of the neurons exhibit saturation (on average) until about 10 kernels, pointing to the fact that low amounts of shared memory parallelism indeed provide significant benefits in the simulation of a detailed cortical microcircuit.

From the point of view of hardware design, another analysis of parallel performance can be conducted by considering the maximum achievable speedup. We thus compute the average maximum achievable speedup per neuron by inverting (5.4), recovering a formula similar to (B.13). This allows us to present essentially the same information from a different perspective, investigating the issue of memory saturation from the point of view of maximum theoretical achievable speedup, rather than utilization. Figure 5.12 plots the theoretical maximum speedup versus the serial runtime to simulate one second of activity for each neuron. There is a remarkably low variability in the potential speedup from parallelism, with all neurons in the cortical microcircuit saturating the memory bandwidth at 13–14 shared memory threads. While this means that in theory every neuron could be simulated in real time, in practice the slowest neurons may not be accelerated much faster than that.

We summarise in Table 5.6 some relevant statistics of the predicted runtime of neurons at different levels of parallelism. As expected, increasing the number of threads decreases the overall average runtime and standard deviation. However, the relative variability – as measured by the coefficient of variation $\frac{\sigma}{\mu}$ – remains constant. This is an indication that, in our case, multithreading does not really seem to affect the performance variability, but rather only

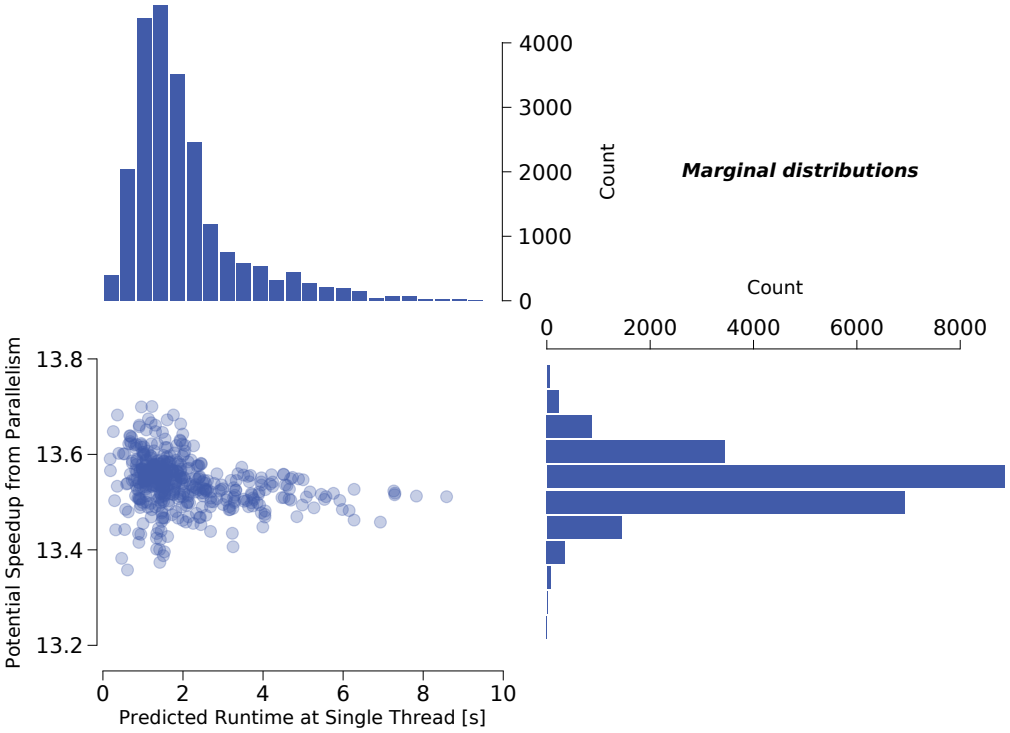

Figure 5.12 – Distribution over neurons of predicted serial runtime and theoretical maximum speedup from shared memory parallelism. We predicted the serial runtime for each neuron, assuming the loop ordering optimisation is being used. The runtime is measured in wallclock seconds required to simulate one second of biological time. Then, for each neuron, we computed the maximum theoretical speedup inverting (5.4), as described in Section B.1.3. The scatter plot on the bottom left presents the joint distribution, while the two histograms represent the marginal distributions.

the absolute performance values. Moreover, we can interpret this as a confirmation of our findings that neurons' scaling properties, in terms of saturation of memory bandwidth, are quite homogeneous. At full saturation the average neuron displays a 13.5x speedup, while the reported maximum represents a hard limit to the real-time factor achievable with this hardware: short of changing the algorithm it would be impossible to run simulations faster than 0.7x real-time. Considering that our analysis neglects the spike delivery and linear algebra kernels, this could represent a significant limitation on the achievability of real-time simulations for G-based detailed models on our reference Skylake hardware.

5.4 Load imbalance at the network level

In a parallel simulation it is important to ensure that each worker consumes similar amounts of work, to avoid idle waiting time due to parallel processes waiting idly for others that still have not finished. The inefficiency due to a inhomogeneous distribution of workloads is called

	mean	std dev	coeff var	min	25-%	median	75-%	max
serial	1.998	1.342	0.672	0.117	1.119	1.628	2.349	9.391
4 threads	0.499	0.335	0.672	0.029	0.280	0.407	0.587	2.348
18 threads	0.149	0.100	0.672	0.009	0.083	0.121	0.175	0.698
saturation	0.148	0.099	0.673	0.009	0.083	0.120	0.174	0.695

Table 5.6 – Statistics on the predicted runtime of individual neurons. Data is always assumed to come from DRAM. Units are wallclock seconds to simulate one second of biological time.

load balancing, and is known to be a NP-hard problem to solve (Garey and Johnson, 1978). Load imbalance can be especially important in the simulation of detailed neurons for two reasons: the dispersion of the distribution of neurons' runtime and the comparatively large relative weight of individual neurons' runtime over the total simulation. The high degree of heterogeneity identified first in the metrics in Figure 2.5 and secondly in the distribution of runtime in Figure 5.5 can be considered the culprit of why the load balancing problem is difficult to solve in a microcircuit.

The problem of ensuring good load balance in brain tissue simulations has been studied empirically. In strong scaling of morphologically detailed neurons, Hines et al. (2008) suggested a method to split larger neurons across multiple processes, to allow branch-based balancing with smaller neurons. A more aggressive splitting of neurons at the granularity of branches and the exploitation of an asynchronous runtime to dynamically dispatch execution to idle workers was also investigated (Magalhaes et al., 2019a). Important load balancing issues in the simulation of small networks of detailed neurons on many-core processors have been highlighted (Kumbhar et al., 2018), and a review found that load balancing plays a key role in obtaining good performance in the simulation of point neurons on a distributed cluster of GPUs (Nair et al., 2015). Additionally, irregular spiking behaviour was found to have a detrimental effect to performance on GPUs (Yavuz et al., 2016).

5.4.1 Static load balance

Static load imbalance is determined purely by non-dynamic properties, and remains constant throughout the execution. It can be mitigated through a load balancing algorithm that prescribes a strategy for distributing neurons across parallel workers. We base our analysis on the Least Processing Time (LPT) algorithm (Graham, 1969), which is the underlying strategy used by our reference simulator CoreNEURON (Kumbhar et al., 2019b). In short, the LPT algorithm is based on a preconditioned greedy approach, as detailed in Algorithm 11.

When simulating a full microcircuit, we expect the importance of static load imbalance to increase as the level of parallelism grows. We consider in this section only strong-scaling experiments, since in weak scaling all workers have by definition the same amount of work.

Algorithm 11 Pseudo-code for the LPT algorithm applied to load balancing of neurons.

```
NC ← list of neuron complexities
W ← list of workers
sort(NC) in descending order
for n ∈ NC do
    w ← argmin W
    assign(n → w)
end for
```

Moreover, expanding our previous analysis, we consider here distributed parallelism in addition to shared-memory parallelism. Thus we introduce the concept of Parallel Process, which can be either a distributed rank or a shared-memory thread. Hereinafter, when we only report the number of parallel processes, we assume that all distributed machines are filled at full shared-memory capacity, such that two parallel processes on the same node communicate via shared memory parallelism, while parallel processes on different nodes communicate via message passing.. Taking the example of our reference architecture SKX-AVX512, which exposes 18 parallel threads, when we report 36 parallel processes it should be understood as 2 distributed ranks, 18 shared-memory threads per rank.

Figure 5.13 reports the impact of load imbalance on simulation runtime by considering two quantities: the relative load imbalance and the absolute runtime. Load imbalance can become a significant portion of the total simulation runtime when strong scaling a relatively small network of 1024 neurons. When the number of neurons per parallel process is smaller than a threshold value of roughly four, the relative load imbalance becomes larger than 10%, eventually reaching almost 40% when there is only one neuron per parallel process. Moreover, static load imbalance always reaches almost the same weight as the computation runtime in the extreme case of a single neuron per parallel process. This can be easily explained by the fact that, in this configuration, the performance is simply given by the slowest parallel process, while the load imbalance is defined as the difference between the slowest and the fastest. This phenomenon is present also for strong scaling of larger networks (note the different y scales in Figure 5.13). An important difference with regards to the small network use case is evident here: relative load imbalance in this case never exceeds a few percentage points of overall runtime. This can be readily explained by the fact that in the extreme strong-scaling configuration consisting of only a few neurons per parallel process, communication takes over and becomes the dominant performance factor. Given that load imbalance cannot exceed the computation time, if communication is the dominant factor than load imbalance must a fortiori have a relatively low impact. In the largest network considered here, consisting of 10^5 neurons, load imbalance never exceeds 1% of total runtime.

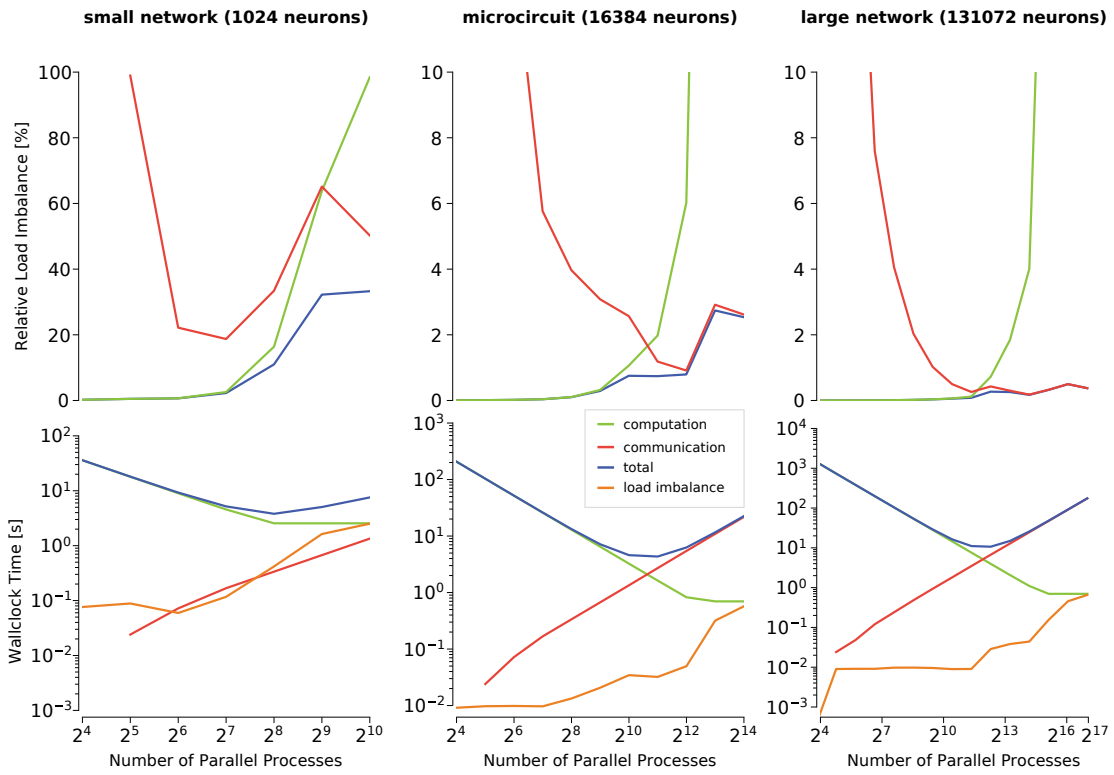

Figure 5.13 – Static load imbalance in a distributed simulation of a detailed microcircuit. Relative load imbalance (*top*) as a percentage of computation time (brown line), communication time (green line) and total runtime (light blue line); Absolute wallclock time (*bottom*) for computation (brown line), communication (green line), total (light blue line) and load imbalance (purple line) measured in wallclock seconds to simulate one biological second. We plot relative and absolute load imbalance for strong scaling a small network of $\sim 10^3$ neurons *(left)*, a microcircuit of $\sim 10^4$ neurons *(centre)* and a large network of $\sim 10^5$ neurons *(right)*. Note the different scales on all y and x axis.

5.4.2 Dynamic load balance

In the course of a simulation, neurons produce spikes through a stochastic process that can hardly be anticipated a priori. This can have a load imbalance effect if, for example, by chance the number of spikes to be processed by a parallel worker is significantly larger than the number of spikes processed by the others. We present here a method based on probability theory to estimate the load imbalance – i.e. the difference in number of spikes to be processed by parallel workers – as a function of the number of workers, the number of neurons, their connectivity and their firing frequency.

The spike load imbalance distribution Let us consider a number P of workers, where each worker's load W is defined as the number of spikes it must process in a given time period. For tractability, we make the customary assumption that spikes are generated by indepen-

	$P=2$		$P=8$		$P=64$		$P=512$	
Nf	$E[\Lambda]$	$Var[\Lambda]$	$E[\Lambda]$	$Var[\Lambda]$	$E[\Lambda]$	$Var[\Lambda]$	$E[\Lambda]$	$Var[\Lambda]$
100	4.35	11.06	4.54	1.21	2.63	0.45	1.59	0.28
1000	13.81	109.17	15.22	10.75	9.86	2.09	3.98	0.43
10000	43.70	1090.30	48.34	105.65	32.01	19.08	14.24	2.79
100000	138.20	10901.57	152.93	1054.63	101.46	188.88	46.47	16.22

Table 5.7 – Distribution of load imbalance under several configurations. We report the average load imbalance $E[\Lambda]$ and the variance $Var[\Lambda]$ for the distribution defined in (5.5), under different simulation configurations. $E[\Lambda]$ is measured in numbers of events generated during a minimum delay interval. N denotes the total number of neurons in the network, P the number of parallel processes, f the average firing frequency of the neurons in Hz. The fan-in K and minimum network delay $\delta_{\min}$ were considered constant $K = 3000$, $\delta_{\min} = 0.1ms$.

dent Poisson processes. Setting N as the number of neurons in the whole network, K the average number of incoming connections per neuron, and neglecting the fact that neurons can form multiple connections, a good first approximation is to assume that the spikes to be processed by the $\frac{N}{P}$ neurons belonging to the same parallel process are given by W i.i.d, $W \sim Poisson(\frac{NK}{P} f\delta_{\min})$, where f is the average firing frequency and $\delta_{\min}$ the minimum network delay.

We define the dynamic load imbalance as

$$\Lambda = \max_P W - \min_P W. \tag{5.5}$$

It can be shown that the load imbalance distribution can be expressed in terms of the joint distributions of the maximum and the minimum by:

$$Pr(\Lambda = k) = \sum_{x=k}^{\infty} Pr(\max_P W = x, \min_P W = x-k), \tag{5.6}$$

with $k \geq 0$ and the joint distribution itself given by

$$Pr(\max_P W = x, \min_P W = x-k) = \begin{cases} [F_W(x) - F_W(x-1)]^P & k = 0, \\ [F_W(x) - F_W(x-k-1)]^P + [F_W(x-1) - F_W(x-k)]^P \ldots & \\ \cdots - [F_W(x) - F_W(x-k)]^P - [F_W(x-1) - F_W(x-k-1)]^P & k > 0, \end{cases} \tag{5.7}$$

where F_W is the cumulative distribution function of W. We report the necessary mathematical steps to reach this formulation in Appendix D. We report the expected value and variance of the load imbalance Λ in Table 5.7 under various configurations. For example, in a simulation of a network of 1000 neurons firing at 1Hz using over 64 parallel processes (either distributed

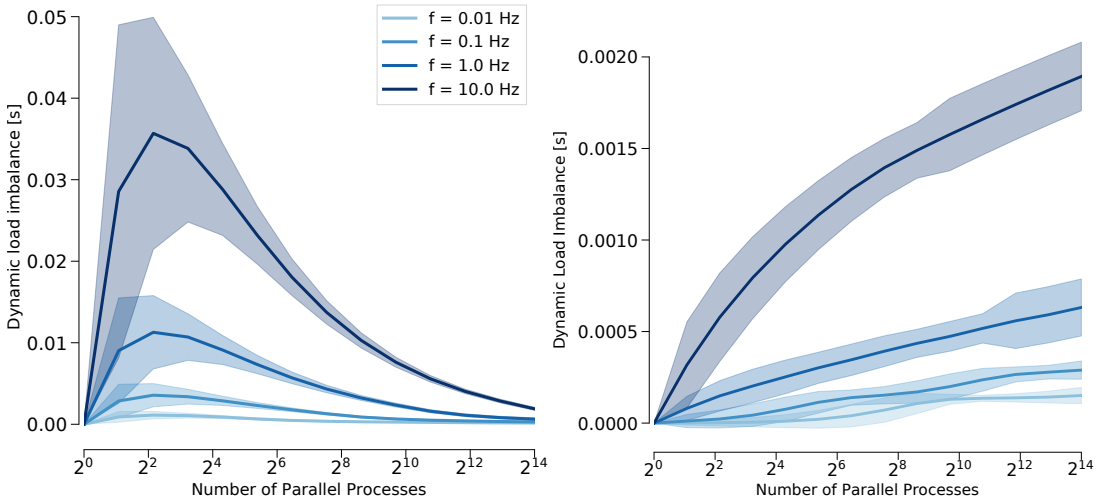

Figure 5.14 – Dynamic load imbalance arising from spike delivery. We plot the dynamic load imbalance defined in (5.5), multiplied by the wallclock time required to integrate a synaptic event. Thus the unit of measure is wallclock seconds of load imbalance over a simulation of one second of biological time. The solid lines represent the average values (over time or multiple iterations), while the shaded areas represent one standard deviation from the mean. For strong scaling *(left)* we considered a microcircuit simulation of $\sim 10^4$ neurons, with fan-in $K = 3000$ and minimum network delay $\delta_{\min} = 0.1$ ms, at various firing frequencies (darker colours denote higher frequencies). For weak scaling *(right)* we considered a scenario with 2 neurons per parallel process, fan-in $K = 3000$ and minimum network delay $\delta_{\min} = 0.1$ ms, at various firing frequencies (darker colours denote higher frequencies).

ranks or shared memory threads) the average load imbalance, in terms of difference of number of events to be processed, will be about 10. From (5.5) we can obtain the average distribution of the load imbalance in the simulation of one biological second by multiplying the load imbalance (measured as a the difference in number of events), times $\frac{1}{\delta_{\min}}$ (the number of mindelay intervals in a second), times the cost of processing one spike, which in our case was taken to be 30.8 cycle from Section 3.1.5.

Dynamic load imbalance in a microcircuit simulation We present the results of our method for estimating dynamic load imbalance in Figure 5.14. In the strong scaling case we observe an inverted-U shape: as the number of parallel processes increases, the load imbalance initially increases, reaches a maximum point and then slowly decays to almost zero. This can be intuitively explained by comparing sources of variability. At small parallel processes counts, our model for load imbalance is equivalent to drawing a small number of samples from Poisson variables with large parameters. Thus the variability is large, and the work per process is large. As the number of parallel processes increases, two effects are competing: we are drawing more samples, thus making it more likely to see extremal values – i.e. larger maximums and lower minimums – but at the same time the parameter of the Poisson process

is decreasing, thus decreasing the variability. Our computations show that the latter effect rapidly becomes dominant, and the load imbalance decays towards zero. In terms of firing frequency, as expected larger frequencies lead to more imbalance, but the effect does not seem to be linear, as for example a tenfold increase in average firing frequency only results in a roughly 4x increase in imbalance. The runtime values should be compared to the simulation runtime in Figure 5.13: for example, when using 64 parallel processes the simulation runtime is roughly 50s to simulate one second of biological time, while the dynamic load imbalance is around 0.02s, i.e. less than 0.1%. Even in the most extreme scenarios, static load imbalance far outweighs dynamic load imbalance in the simulation of a microcircuit. Note that the relative weight of dynamic load imbalance to simulation runtime is highly affected by the network size. However, we found that even for small networks of roughly 1K neurons dynamic load imbalance never exceeded 1% of the total runtime, whereas we demonstrated before that static load imbalance can reach up to 40% of total runtime.

In the weak scaling case the dynamic load imbalance grows indefinitely, as shown in Figure 5.14. Also in this case firing frequency has a direct, but sublinear, relationship. While we do not show a direct comparison to runtime, once again dynamic load imbalance never exceeds a few percentage points of the runtime to simulate a neuron. Thus we have shown that in both the weak and strong scaling case, dynamic load imbalance is highly unlikely to play an important role in determining the performance of the Reconstructed model. This is in stark contrast with other neuron abstractions, and in particular point neurons with plasticity, where it has been speculated that dynamic load imbalance plays a role in the degradation of strong scaling parallel performance (Zenke and Gerstner, 2014).

5.5 Discussion

In this chapter we demonstrated the use of our methods to analyse the performance properties of the simulation of a morphologically detailed cortical column within the Reconstructed *in silico* model. To cope with the wide variety of ion channel and synapse models, we leveraged Kerncraft (Hammer et al., 2017) – an automatic performance modelling tool – by extending the NMODL source-to-source compiler (Kumbhar et al., 2019a) with a code generation backend compatible with Kerncraft requirements. We showed that by using a post-processing step that complements the automatic performance model generated by Kerncraft, it is possible to obtain an acceptable level of accuracy for almost all performance predictions. We make extensive use of the inference properties of the ECM model to analyse the intrinsic heterogeneity of performance properties of the Reconstructed model. A performance analysis with this level of detail had never been conducted before in the context of simulations of G-based morphologically detailed neurons. Our analytical assessment has confirmed several findings from our previous analysis in Chapter 4. In particular, we find once again that G-based neurons are dominated by data-bound kernels and saturate the memory bandwidth with a low number of threads in a shared-memory scenario. Moreover, we reconfirm the importance of cache reuse enabled by the loop ordering optimisation for scalability of the shared-memory

execution.

Classification of current and state kernels G-based kernels in the cortical microcircuit model can be classified in five groups according to their serial performance profile: a heavily data-bound group containing exclusively current kernels, a mixed group of slightly data-bound kernels and three groups on the compute-data boundary containing almost exclusively state kernels. Using this information we explore fusing of kernels with different performance profiles as a possible way to maximise resource efficiency. However, we find only few combinations of kernels that would be able to provide a performance benefit. Moreover our analysis shows that, taking into account practical limitations to this technique, the estimated speedup from fusion for the best possible combination would be around 1.3x. We conclude that while this technique could improve efficiency, the tradeoff in performance gains versus development effort would probably not be favourable.

Characterization of serial performance of G-based detailed neurons The performance profile of G-based detailed neurons is essentially dominated by data-bound kernels, with the T_{data} component of neurons being on average six times larger than T_{core}. Despite all neurons sharing a similar bottleneck, the runtimes of individual neurons have significant variability, especially when data is in main memory. This variability in serial runtime can be explained by considering both the number of synapses and number of compartments in a neuron, but not by considering them individually. Digging deeper, we find that the number of synapses accounts for a large portion of the variability in the execution of current kernels, while state kernels are affected in equal measure by compartments and synapses.

Characterization of parallel performance of G-based detailed neurons The analysis of individual kernels reveals that, unless the loop ordering optimisation is implemented, only very small performance gains can be obtained via parallelism. When this optimisation is implemented, we observe a large gap between highly and moderately parallelisable kernels. Ion channel and synaptic kernels are not all present in equal quantities, and looking at their relative importance by analysing their specific mix within neurons is a more realistic approach than looking at individual kernels. In this context we find a strikingly small variability in the parallel speedup achievable by neurons, between 13–14x for all kinds of neurons regardless of morphology, electrical type or circuit function. In conclusion, our results point to the fact that while morphologically detailed neurons can indeed benefit from shared memory parallelism, the saturation effect becomes prominent at a comparatively small number of shared memory threads, thus hindering the full exploitation of future many-core architectures.

Heterogeneous neurons have homogeneous serial performance profiles The distributions of the number of compartments and the number of synapses per neuron are extremely dispersed and have many outliers, demonstrating a significant degree of heterogeneity in the

distribution of neuron models. However, our analysis shows that neurons within the reconstructed microcircuit are all data-bound in the serial execution, and saturate the memory bandwidth at roughly the same number of shared memory threads, indicating that the biological variability is not reflected in heterogeneous performance profiles. While detailed G-based neuron models have a rather homogeneous performance profile, their runtimes are quite heterogeneous and reflect the variability in the number of channels and synapses.

Load balancing has differential effects according to network size Motivated by the fact that a long tail of complex neurons in the distribution of runtime could pose a significant load balancing problem, we examine the possible static and dynamic load imbalance that could arise due to variability of neurons and of spikes. We show that using the LPT load balancing algorithm in a BSP framework leads to negligible static load imbalance, except in the case of strong scaling of small networks of neurons. We also conduct an analysis of heterogeneity due to irregular spiking activity. We find that dynamic load imbalance due to irregular spiking activity is 1-3 orders of magnitude smaller than the runtime to simulate even a single detailed neuron, such that its relative impact on simulation runtime is hardly noticeable on our reference HPC CPU architecture. Regardless of this result, dynamic load imbalance represents an interesting and challenging topic to investigate and can lead to non-negligible effects in the simulation of simpler neurons or on highly parallel architectures. Our methodology can in principle be extended to cover both cases.

Throughput-optimized SIMD architectures may suffer from static load balance issues Given that we have identified a high degree of similarity in the performance profiles of neurons, one might consider whether high-throughput SIMD architectures such as e.g. GPGPUs would constitute a good candidate for accelerating simulations of detailed neurons. In this case, however, heterogeneity in the morphologies as well as in the distribution of ion channels and synapses could constitute a major hindrance to efficient utilization of such architectures (see (Kumbhar and Hines, 2016; Kumbhar et al., 2018), even though we do not exclude it might increase performance overall. Moreover, GPGPUs typically have limited memory capacity, and our analysis shows that strong scaling small networks to large degrees of parallelism can incur in significant static load imbalance.

Neuromorphic hardware may suffer from dynamic load imbalance On the other hand, experimental architectures such as neuromorphic chips offer a high degree of parallelism and a fast, efficient communication network. One of the principles of neuromorphic chips is to minimise overprovisioning of single cores by assigning them small amounts of work, and exploiting an extremely distributed architecture to maximise performance. Since neuromorphic chips are typically designed with scalability in mind, they do not suffer from the limited memory capacity of GPUs. However, we argue that slower individual cores could lead to a larger static and dynamic load imbalance problem. The reason is that simpler cores with

smaller memory capacity are less capable of amortizing the static load imbalance costs, and the lack of a pipelined architecture able to expose ILP would suffer greatly from dynamic load imbalance. In conclusion, our analysis points to the fact that HPC distributed architectures composed of a low-latency communication network and high-frequency out-of-order chips exposing moderate amounts of shared-memory parallelism represent the ideal architecture for high-performance simulations of detailed G-based models.

5.5.1 Limitations and future work

We have shown the impact on performance of heterogeneity originating from several sources. However, it was impossible to include all the sources listed at the beginning of this chapter in our analysis. From the perspective of modelling in computational neuroscience, important load imbalance effects can arise from using an adaptive timestep method for the time integration of the membrane equation (Magalhaes et al., 2019c). Given the high unpredictability of timestep sizes and their variations depending on the evolution of the whole simulation, as of right now it was impossible to include such methods in our analysis. Additionally, although we have tried to infer the properties of performance on hardware accelerators and heterogeneous platforms, our model does not yet include hardware architectures other than CPUs. As such we are unable to deliver a full characterisation of the performance profile and load imbalance risks on such architectures. An extension of our performance model to other hardware architectures would provide great benefits in terms of completeness and scope of our analysis.

6 Discussion

This work was motivated by the realization that the simulation neuroscience community lacked a systematic, quantitative understanding of the computational characteristics of brain tissue simulations, and their performance implications for modern hardware. With this goal in mind, we compiled a list of simulation use cases representative of trends in the literature, which we dubbed the *in silico* models and experiments. We used hardware-agnostic metrics to characterise the main properties of these use cases, finding differences and unexpected similarities among them. However, this description lacked a direct link with hardware, and thus predictive and inference power, so we decided to use performance modelling techniques allowing us to convolve our high-level descriptions with hardware properties to obtain a quantitative characterisation of performance. We selected a subset of three highly representative *in silico* models and we built a fully fledged performance model based on modern HPC hardware, able to account for several factors such as CPU characteristics, memory bandwidth and capacity, interconnect network, and more. The model's ability to deliver accurate predictions allowed us to validate it in a set of benchmarks, but the real value of our approach was in the model's explainability and inference properties. By dissecting and analysing in detail the model's parameters, we were able to reach the characterisation of brain tissue simulations described below, allowing us to pinpoint similarities and differences across *in silico* models' algorithms and their implications for performance.

The main contributions of this thesis are as follows. First we conducted an extensive literature review of the state of the art in brain tissue modelling and performance-related aspects. To introduce structure in the complex landscape of brain tissue simulations we identified a novel set of hardware-agnostic metrics allowing us to capture the key characteristics of an *in silico* model. We used these metrics to dissect the performance properties of a collection of *in silico* models and experiments representative of the different modelling and simulation approaches in the literature. We then intersected *in silico* model descriptions with hardware on two axes. For the single-node performance axis we introduced an abstraction of the Skylake architecture based on the ECM model that had never been published before. We extended the ECM formalism and overcome certain of its limitations by making extensive use of domain-specific

knowledge to obtain the first analytic performance model of brain tissue simulation kernels on a single node with shared-memory parallelism. For the communication axis we leveraged the LogGP model to obtain the first description of the performance of interprocess spike exchange providing a direct link with interconnect hardware characteristics. Using the performance modelling methods developed above we complemented our analysis of the state of the art with a quantitative assessment of the salient performance properties and hardware bottlenecks of *in silico*models and experiments. We then implemented a workflow based on code generation and machine analysis able to automatically build a performance model for arbitrary neuron abstractions. To our knowledge, an analysis of the performance of G-based detailed models was never conducted at this level of granularity and detail. Our approach allowed us to focus our investigation both on the wide breadth of modelling abstractions and on the heterogeneity intrinsic to a single cortical microcircuit. By inspecting the parameters and outputs of the performance models models we were able to validate and confirm common knowledge – such as the importance of fast memory bandwidth and interconnect hardware in simulations of large neural networks – but also shed new light on the properties of the simulation algorithms and identify future codesign directions. More importantly, our methods enabled us to gain a deep understanding of the relationship between intrinsic features of different *in silico* models and their performance on various hardware architectures.

6.1 Computational characteristics of brain tissue simulations

During the course of our investigation, we identified three main features of brain tissue simulation algorithms, namely: clock-driven update kernels, event-driven spike delivery and loose temporal coupling. The presence of these three characteristics and the way in which they are combined is specific to brain tissue simulations and generates their unique computational fingerprint. To our knowledge, an in-depth analysis of how these characteristics affect performance and efficient use of hardware has never been conducted, nor is it possible to find the same combination of algorithmic features outside of the field of simulation neuroscience.

Event-driven spike delivery Believed to be the neural substrate of thought, event-driven spike delivery is undoubtedly one of the main identifying features of brain tissue models. In spite of its undisputed functional purpose, theoretical analysis and empirical measurements show that this phase of brain tissue simulations has an impact on performance only for certain types of neuronal abstractions, namely current-based (I-based) models, while conductance-based (G-based) models are usually dominated by clock-driven kernels even over a wide range of parameter values and cluster configurations. More importantly, the spike delivery kernel possesses a unique performance profile that has not yet been fully understood. Its main feature is the dynamic indirect access pattern which introduces a latency cost, large memory traffic overhead and prevents vectorisation on modern architectures. Our analysis has been able to establish performance bounds for this kernel, and while benchmarks show that its serial runtime is slower than the kernel's theoretical throughput, and even slower than its

critical path predictions, it is definitely not fully bounded by the memory latency. We have also confirmed the hypothesis that this kernel's performance scales linearly with the number of shared memory threads until the memory saturation bottleneck is hit. In the saturated bandwidth regime, this kernel's large memory traffic requirements are responsible for almost 70% of an I-based model's performance.

Clock-driven kernels Clock-driven kernels represent the main driver of performance in G-based models and still constitute a significant portion of I-based models' runtime. We have shown that the serial performance of these kernels is characterised by a high degree of heterogeneity both across and within neuronal abstractions. All the clock-driven kernels of I-based models that we considered turned out to be data-bound in the serial case, while the G-based point neuron model was characterised by data-bound current kernels and state kernels with a boundary profile. In the morphologically detailed G-based model, we observed a large variability across ion channel and synapse kernels although the profiles of each family of kernels were largely preserved. In spite of this variability, when we take into account the specific mix of ion channels per neuron as well as the runtime of each kernel, we find that most neurons spend almost all the runtime in data-bound kernels during a serial execution. When shared-memory parallelism is introduced, we uncover among the G-based kernels a distinction between fast-saturating current kernels and slowly-saturating state kernels. The memory saturation properties of clock-driven kernels are directly responsible for the shared-memory parallel performance of the G-based models in our analysis, while clock-driven kernels of the I-based IAF model account for less than a third of the overall runtime.

Loose-coupling The property of loose-coupling represents one of the main distinguishing features in terms of structure of the simulation algorithm as compared to state-of-the-art simulation workflows in other computational sciences. One of the most visible effects of loose coupling is that it allows a significant reduction of the relative cost of the interprocess communication phase by delaying synchronization and communication steps as much as possible. However, our analysis shows that for point neuron models this effect is not sufficient to mitigate the costs of communication and for medium to large scale simulations the latency of the interprocess network hardware remains the dominating factor. Our performance model has allowed us to identify and study in detail another, less widely-known, beneficial effect of loose coupling: enabling cache reuse through a loop ordering optimisation. Thanks to this algorithmic enhancement, the G-based models in our analysis enjoyed important shared-memory performance gains by exploiting improved data locality. On the other hand, I-based models whose shared memory scalability is bounded by the spike delivery kernel only observe a very limited benefit from the loop ordering optimisation.

6.2 Hardware implications

One of the main goals of this work was to identify the most relevant hardware features and explain their effect on brain tissue simulations. Among the complex and ever-changing state of the art in modern hardware, we identified two key characteristics that drive the performance of biological neural networks more than any other: the memory bandwidth and the network latency. However, several other hardware characteristics that we discuss here play an important role for performance and, especially in extreme cases, could become new dominant factors.

Memory bandwidth On almost all modern architectures memory bandwidth represents an important bottleneck, especially in the case of shared memory parallelism. The *in silico* models and experiments that we analysed are all heavily affected by this hardware design pattern, albeit for different reasons. G-based models contain data-bound kernels characterised by poor scaling properties, while I-based models are dominated by the spike delivery kernel with a large memory traffic overhead. Investigating how much models would benefit from a larger memory bandwidth, we found a significant difference between the max-filling regime and the real-time regime. In the former, where bandwidth is assumed to be saturated, improvements in the memory bandwidth would be directly reflected in better performance, at least in sufficiently small clusters such that communication is not a dominating factor. On the other hand, in the more realistic real-time scenario that drops the saturation hypothesis, we find that strong-scaling benefits from faster memory would quickly be diminished by other bottlenecks such as CPU throughput for detailed G-based models. Thus, according to our analysis, memory bandwidth severely limits the shared-memory parallel scaling properties of brain tissue simulations, but improving it alone would not automatically result in an equivalent improvement of performance. This leads us to the conclusion that future chips with hundreds or more cores run the concrete risk of being inefficient for simulations of biological neural networks, regardless of modelling abstraction, as a consequence of being starved for data. While of course the data-computation tradeoff should be analysed in detail for each new architecture, our analysis points to the general conclusion that, except for a few use cases, large degrees of shared-memory parallelism cannot be fully exploited by *in silico* models and experiments unless they are sustained by a dramatic increase in memory bandwidth.

Fast cores with complex microarchitectural features The reference architecture used throughout this work is characterised by a relatively fast core exposing important optimisations at the hardware level, such as a superscalar architecture, speculative execution, out-of-order scheduling, operand forwarding, handling of multiple outstanding memory requests, etc. An important question to ask is whether these features play an important role in the performance of *in silico* neuroscience simulations. While our performance model does not explicitly account for these in-hardware optimisations, we can still infer their importance from modelling. For example, the fact that many of the G-based state kernels were found to be on the boundary

between core-bound and data-bound profiles despite their very high arithmetic intensity indicates that vectorisation and superscalar hardware have already provided extensive benefits. In addition, we can speculate that while the full-throughput hypothesis obviously does not apply to the spike delivery kernel, the fact that it is able to reach high performance in spite of the high memory latency on the reference architecture is directly linked to the degree of exposed memory-level parallelism. Thus we conclude that any architecture with low amounts of shared memory parallelism is highly dependent on complex microarchitectural features in order to reach high-performance simulations, by hiding the latency of core-bound kernels in G-based models, and of the spike delivery kernel in I-based models.

Distributed parallelism The small-world nature of biological neural networks makes them highly suitable for distributed programming. This computing technique has been used mainly to enable simulations of larger and larger networks (weak scaling), and less often to accelerate networks with a fixed size (strong scaling). Our analysis, based on a state of the art high-throughput and low-latency interconnect network, revealed that when communication is the bottleneck the network latency is the main driver of performance, a property that we impute to the use of blocking collective communications. Concerning load imbalance, in the weak scaling scenario we identified that static load imbalance is by definition impossible, but dynamic load imbalance grows as the level of parallelism increases. In the strong scaling scenario, static load imbalance due to heterogeneity between neurons can grow to represent a significant portion of the computation time, while dynamic load imbalance has an interesting inverse-U shape, with a peak at a relatively small number of distributed ranks. In all cases, we are able to show that in the context of morphologically detailed G-based models, the impact of load imbalance is usually neglectable if compared to the whole simulation runtime (computation and communication), except when strong scaling small networks of neurons.

SIMD and MIMD parallelism Single Instruction Multiple Data (SIMD) architectures have been gaining a lot of traction in modern computer design, from superscalar multiprocessors that expose wider and wider vector units (e.g. Intel's AVX512), to GPUs and vector processors. In terms of pure performance, SIMD parallelism is proven to be extremely beneficial for simulations of the clock-driven kernels of all *in silico* models and experiments. Throughput-optimized SIMD architectures that typically offer very high peak performance and memory bandwidth can support the high-performance simulation of both core-bound and data-bound clock-driven kernels. However, SIMD architectures would quickly suffer from both static and dynamic load imbalance. While the former can be partially mitigated by time-multiplexing multiple neurons on a processor, the latter is an intrinsic limitation to the high performance of the simulation algorithm, and we have shown that it is more suited to latency-optimized architectures. On the other hand, extreme levels of Multiple Instruction Multiple Data (MIMD) parallelism would more similarly reflect the natural architecture of the brain. Even though general-purpose processing has until now sustained high-performance simulations of neural tissue, custom designed architectures such as neuromorphic chips could overcome the mem-

ory bandwidth and network latency issues that are currently limiting the scaling to very large networks. In this case, the intrinsic heterogeneity of the *in silico* model plays a fundamental role in determining the efficient utilization of the hardware. Homogeneous abstractions that have similar mathematical model, similar connectivity and similar, low firing rates across all neurons would be the perfect candidate for extreme MIMD parallelism. On the other hand *in silico* models with a large degree of heterogeneity, even in just one of the aspects cited above, would inevitably lead to inefficient use of hardware. Moreover, due to the random nature of spike events, even homogeneous models exhibiting very large firing rates could be affected by dynamic load imbalance.

6.3 Limitations and future work

In this work we have concentrated solely on the aspect of maximising performance and resource utilization, without considering limitations such as cost or energy. However, it must be stated that energy efficiency is a central issue in the computational neuroscience community, and one of the main selling points of neuromorphic hardware (Cassidy et al., 2014; Stromatias et al., 2013). Therefore, a meaningful extension to this work would be to incorporate a model for power consumption alongside performance prediction, as a way to constrain the feasibility and efficiency of certain simulation configurations. To achieve this, one could exploit already established power consumption models that are easily integrated with the ECM and have been shown to provide valuable insight into the power and performance properties of simulation kernels (Hager et al., 2016; Hofmann et al., 2018).

From the modelling point of view, an important aspect that we have neglected in this analysis is synaptic plasticity. A large portion of research questions that require brain tissue simulations involve learning and synaptic plasticity, which was shown to represent a significant portion of the total runtime (Zenke and Gerstner, 2014), so this represents an important extension to our analysis. However, in this work we decided to concentrate on the inference part of brain tissue simulations because the diversity and complexity of plasticity models warrants a separate analysis. Although the modelling of the spike delivery kernel has proven to be technically difficult, we do not think that adding a plasticity kernel would prove to be technically challenging, but it would considerably complexify the resulting analysis.

Different *in silico* experiments may require measurements and reporting on different values, even when the underlying modelling abstraction is the same. For example, an experiment simulating local field potentials in the Reconstructed model requires storing the membrane currents of each compartment at every timestep (Reimann et al., 2013), while visualization or analysis constraint may require specific output data layouts. In addition to the added complexity, differential storing of state variables may introduce load imbalance in a simulation. To include this aspect in our analysis would require an extension of the performance model to I/O operations, which we deemed outside of the scope of this work. However, simulation I/O and data analysis can represent an important bottleneck in brain tissue simulations and

empirical optimisation studies are being actively investigated (Eilemann et al., 2016; Ewart et al., 2017; Planas et al., 2018; Schürmann et al., 2014).

Our analysis was focused on the computational requirements of brain tissue simulations, but an interesting question that may be asked is how much paying the price of more costly simulations actually buys in terms of computational capability of the network. In the field of artificial neural networks a review introduced metrics such as the *information density* to represent the accuracy per parameter in a classification task (Canziani et al., 2016). Similar metrics do not exist for biological neural networks, but preliminary studies attempting to quantify the complexity of a neuron's behaviour are being conducted, for example by establishing a one-to-one mapping between a biological neuron's I/O relationship and a deep neural network (David et al., 2019). This makes it possible, at least in theory, to relate the computational cost of a neuron with its computational capability, measured by proxy as the information density of the equivalent deep neural network. Future investigations in this regard could answer important questions about the cost-benefit tradeoffs of *in silico* models.

From an hardware modelling perspective, we focused on HPC CPU clusters and chose a medium-granularity model able to capture salient hardware features at the cost of a less detailed perspective. The extension of our performance model to different hardware architectures, especially GPUs and neuromorphic, would broaden the scope and saliency of our conclusions. While cycle-accurate hardware simulators such as RSIM would yield high accuracy (Pai et al., 1997), we fear that the loss in explainability would be too high to justify their use. On the other hand, more complex analytical models would permit the investigation of the relative importance of individual microarchitectural features (Levinthal, 2014) as well as interesting future directions for hardware codesign such as dark silicon (Esmaeilzadeh et al., 2012).

6.4 Closing remarks

Our analysis lead to the characterisation of brain tissue simulations from an algorithm and hardware perspective. We showed that, if future iterations of general-purpose hardware architectures maintain the same balance as the current state-of-the-art, it will be very difficult to achieve fast, large scale simulations of brain tissue. Even if hardware peak performance were to improve significantly over the next years, the required speedup could only be achieved via specifically targeted advancements and under very restrictive simulation and model configurations. To support the next generation of brain tissue simulations, the community must therefore focus on the design of dedicated hardware. In order to achieve this goal, computational neuroscience modellers must cooperate with software developers and hardware designers. Ultimately, we stress that performance modelling represents a powerful tool to enable communication between these communities This work embodies a concrete effort to define and understand key performance properties of a wide variety of *in silico* models, a necessary step to enable the next generation of brain tissue simulations.

A Simulation algorithms and fundamental concepts

Minimum network delay All cellular-level models of brain tissue must define a strategy to advance the integration of neurons' and synapses' state variables in time. The work of Morrison et al. (2005) presented a new paradigm that, barring very few exceptions (Kozloski and Wagner, 2011), has been implemented by all simulators. Starting from the observation that axons are very reliable in transmitting an AP with the same speed (Cox et al., 2000), they propose to model synapses as a simple delay, based on their distance from the axon initial segment, where APs are typically initiated. This means that coupling between neurons is mediated only through such synaptic delays. In turn, this implies that there is a time interval during which a neuron can be considered as completely independent from any other. This time interval can be computed conservatively by taking the minimum across all synaptic delays in the network, thus we denote it δ_{min} and often refer to it simply as minimum delay or mindelay. Within a minimum delay interval, neurons can be considered independent from one another. This is a fundamental property of *in silico* models and experiments that determines many of its performance characteristics.

Spike delivery: integration of synaptic inputs in the I-based and G-based formalisms The membrane potential of neurons is monitored at every instant in time, and whenever it reaches a specified threshold, a spike is initiated. In all the *in silico* models and experiments considered here, spikes are treated algorithmically as events that populate a priority queue based on their delay to delivery. This enables a lean representation of spikes: all the required information can be summarised in the identifier number of the source neuron and the time at which the spike occurred. In terms of integrating the effects of spikes, at the beginning of every timestep the priority queue is popped of all the events to be delivered in that instant, and for each event the relevant spike delivery coroutine is called. In I-based models, each neuron has a state variable that represents the total synaptic current I_{syn}, and when a spike is delivered this quantity is updated by the connection weight w:

$$I_{\text{syn}} \leftarrow I_{\text{syn}} + w. \tag{A.1}$$

Note that I_{syn} is a neuron-level state variable, meaning that when two synapses connected to the same postsynaptic neuron are activated in the same timestep, a data race is possible and must be prevented. Typically this is not an issue because synapses belonging to the same postsynaptic neuron are processed serially.

In the G-based model, the synaptic conductance g_{syn} is updated by the connection weight, thus indirectly determining an increase in the synaptic current via the membrane voltage v and the synaptic receptors' reversal potential e_{syn}.

$$\begin{aligned} g_{\text{syn}} &\leftarrow g_{\text{syn}} + w, \\ I_{\text{syn}} &= g_{\text{syn}} \left(v - e_{\text{syn}} \right). \end{aligned} \tag{A.2}$$

In this case, g_{syn} is a per-synapse variable, and while I_{syn} is instantiated on a per-neuron basis, the actual computation of I_{syn} does not happen in the spike delivery phase, but rather in the synaptic current phase (see Figure 2.2).

Update of voltage in the point neuron and morphologically detailed representations The evolution of the membrane potential is described by a differential equation, typically arising from the equivalence with a Resistance-Capacitor (RC) circuit. In the simplest model, the lipidic structure of the membrane acts as a capacitor, while the presence of leak ion channels that allow free flow of ions acts as a parallel resistance, as shown in Figure A.1. To give an example for the point neuron formalism, in the Integrate-and-Fire (IAF) neuron the resistance term is linear leading to the equation

$$\tau_m \frac{dv}{dt} = -(v(t) - e_{rest}) + R_{memb} I(t), \tag{A.3}$$

where $v(t)$ is the neuron's membrane potential, $\tau_m = R_{memb} C_{memb}$ is the membrane time constant and $I(t)$ includes synaptic currents such as A.1 as well as input currents from external stimuli. Thus updating the neuron voltage in the IAF point neuron formalism amounts to solving A.3 at every timestep. For the Brunel model in our analysis, an exact time-integration method based on precomputing the matrix exponential is used (Rotter and Diesmann, 1999). In the case of the IAF, whenever the AP threshold is reached, a spike event is elicited and then, after a refractory period where A.3 is not integrated and synaptic inputs are ignored, the membrane potential is simply reset to e_{rest} and time integration restarts. More complex versions of A.3 exist, that introduce nonlinearities to account for the whole complexity of AP dynamics, such as the Izhikevic (Izhikevich, 2003), GIF (Pozzorini et al., 2015) and the Adaptive Exponential (Brette and Gerstner, 2005) models, but while the complexity of solving nonlinear versions of A.3 may require careful numerical treatment, the underlying principle remains the same.

In compartmental models, dendrites and axons are considered part of the modelling abstraction. To account for their morphological properties, they are split into compartments that are

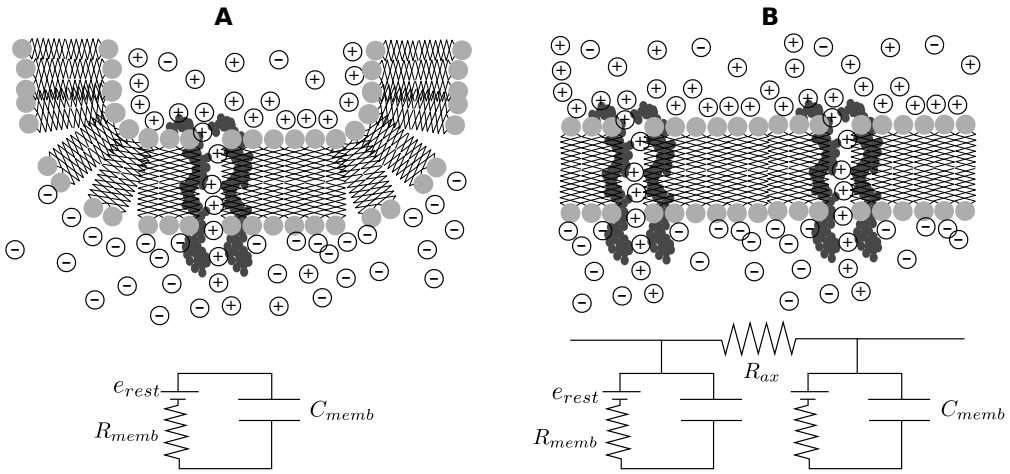

Figure A.1 – Equivalent RC circuit for neuron representations. **A** presents a point-neuron model with the equivalent RC circuit. Ion channels that allow the free flow of ions are lumped into a membrane resistance R_{memb}, while the impermeability of the membrane leads to a capacitance C_{memb}. A battery e_{rest} denotes the resting potential of the neuron. **B** presents a compartmental model. In this case, an additional resistance term R_{ax} is used to connect several RC circuits together, thus allowing modellers to include morphological detail in their representation.

individually modelled as a single RC circuit, and connected through an axial resistance term R_{ax}. For a single section of unbranched dendritic compartments, cable theory is used (Rall, 1962) leading to the partial differential equation:

$$C_{memb}\frac{\partial v}{\partial t} = -\frac{1}{R_{ax}}\frac{\partial^2 v}{\partial x^2} + I(t), \tag{A.4}$$

where x denotes the axial spatial dimension along the dendrite. Branching points require special treatment (Carnevale and Hines, 2006). In the spatial dimension a second order finite-difference scheme is the most common numerical method, while in the temporal dimension implicit time-integration methods are almost always used to solve A.4 for stability reasons (Carnevale and Hines, 2006). The spatial coupling through a second-order derivative and the use of implicit time-integration schemes leads to a linear system to be solved at every time step. While in the general case this can lead to a very costly inversion of a matrix, Hines (1984) introduced the Hines algorithm: a linear-complexity direct method using the specific tree-like structure of neurons to reduce the problem to a quasi-tridiagonal structure (Thomas, 1949).

PSC contributions in the I-based formalism In the I-based formalism, each neuron holds at least one I_{syn} variable representing the total synaptic input. This variable follows its own dynamics, and while we have shown how its value can be increased by incoming spikes, we

have not yet explained how its values decays over time in the absence of spikes. A classical model is to assume single-exponential decay (Roth and van Rossum, 2009), which states that the synaptic current decays exponentially over time with a fixed time constant. Note that this enables an optimisation: instead of keeping track of each individual synaptic current, all the synapses with the same time constant can directly contribute to the same I_{syn} variable, and the exponential decay needs only to be computed for the "lumped" I_{syn}.

Ion channel and synapse models in the G-based formalism In the G-based formalism, we keep track of synapses' and ion channels' conductance g, instead of directly computing their current, which can a posteriori be computed via Ohm's law given the membrane potential v and a resting potential e:

$$I = g(v - e). \tag{A.5}$$

In this formalism, synapses and ion channels are themselves described by a set of state variables and evolution equations. The most well-known model of this type is the Hodgkin-Huxley (HH) (Hodgkin and Huxley, 1952) biophysical model. Given that the evolution of the membrane potential and the evolution of ion channels' and synapses' states is very tightly coupled in this formalism, this poses difficult numerical challenges in solving the corresponding equations. However, Hines (1984) introduced an efficient time-staggering technique, whereby the time integration of one timestep can be divided into three distinct steps in the simulation workflow: before updating the voltage the contributed currents to the membrane equation must be computed; then the voltage can be updated by solving the membrane equation; after the voltage has been updated, the state variables of ion channels and synapses must be advanced in time via numerical integration. Note that in the G-based formalism the current contributions and the state updates must be computed for each ion channel and synapse instance. Moreover, in the case of morphologically detailed models computation of synaptic and ion channel currents amounts to updating relevant terms in the corresponding matrix. On the other hand, even in the morphologically detailed case, integration of synaptic and ion channels' states can be performed independently across instances, even those of the same type or belonging to the same compartment.

Spike exchange Whenever a neuron's membrane potential surpasses the spike threshold, it must communicate this event to other neurons. In the case of distributed simulations, neurons from one rank may need to communicate their spikes to neurons located on a different node. The property of minimum network delay enables an optimisation: instead of communicating spikes at every instant, they can be buffered and communicated every mindelay interval. A common approach for implementing the spike exchange routine is via blocking collective communication functions. While this choice may seem not fitting with the local connectivity structure of neural networks, studies have found that in practice the collective calls do not impose a performance burden (Hines et al., 2011), although it is speculated that this may

change for extremely large clusters (Fernandez Musoles et al., 2019; Jordan et al., 2018). In the *in silico* models and experiments considered here, the C2 model represents an exception to this characterisation: instead of blocking collective calls, it uses nonblocking point-to-point communication.

In the presence of gap junctions, even though technically they do not exchange spikes, an additional communication step is required to exchange membrane potential values at gap junction locations. Since eletrical synapses represent a tight coupling of the membrane potential of two neurons, this presents an algorithmic challenge when neurons are distributed across different parallel processors. Despite the potential for instability, in practice it has been shown that an explicit scheme where the voltage at the gap junction is considered constant for the whole timestep represents a satisfying approximation that considerably simplifies the implementation (Hines et al., 2008), and while other options have been explored (Hahne et al., 2015), the explicit algorithm remains the most commonly found in literature. Therefore in the case of gap junctions an additional communication step is introduced, in which at every timestep the gap junction voltage is exchanged across the interconnect network. This communication step is also usually implemented via a blocking collective function call (Hines et al., 2008). Note that, as direct consequence, gap junctions break the loosely-coupled property arising from the minimum network delay.

A.1 Supplementary material for Chapter 2

Table A.1 reports the numerical values for relevant parameters of the *in silico* models and experiments. These values were used to compute the hardware-agnostic metrics described in Section 2.2.2.

	Brunel	Spaun	C2	Reconstructed	Simplified	Auditory
N_{neu}	10^9	10^6	10^9	10^4	10^4	10^4
ODEs per neuron	2	2	2	10^3	3	10^3
param per neuron	5	0	4	10^3	7	10^3
K	11250	10^4	5500	10^3	36	10^3 (1% GJ)
ODEs per syn	0	1	0	4	5	4 (0 GJ)
param per syn	1	1	1	10	12	10 (1 GJ)
Δt [ms]	0.1	1	0.1	0.025	0.1	0.1
δ_{min} [ms]	1.5	1	0.1	0.1	0.1	0.1
f [Hz]	7.6	100	1	1	1	1

Table A.1 – Parameters of *in silico* models and experiments. GJ denotes Gap Junctions. We only count parameters that can vary across neuron or synapse instances, while constant, homogeneous parameters are not counted.

B Details of the ECM and LogGP model

B.1 The ECM model

We present here a detailed account of the fundamental concepts and necessary steps in the computation of the ECM model. We begin by describing the roofline model which in a sense constitutes the basis for the ECM and introduces some key concepts such as arithmetic intensity and memory bandwidth (Williams et al., 2009).

B.1.1 The roofline model

The roofline model is a cornerstone of performance modelling, representing one of the most widely-known white-box approaches (Williams et al., 2009). It is based on the following key quantities:

- the work w, defined as the number of useful operations performed by a given kernel;
- the arithmetic intensity AI, defined as the ratio $\frac{w}{b}$, where b is the memory traffic needed to perform the work;
- the peak performance P_{peak}, defined as the maximum achievable rate of work per unit of time;
- the peak memory bandwidth BW, defines as the maximum rate of data transfer.

In its original presentation the roofline model was conceived as a visual technique able to identify the main bottlenecks of a kernel. In that case, computing the roofline model involved two phases: defining the machine's peak performance and bandwidth, and measuring the kernel's performance and memory traffic. Note that defining the work w is not a trivial choice, as it must satisfy the requirement of being a meaningful definition of work for the kernel at hand, and at the same time it must allow computing a theoretical peak performance, ideally without benchmarking. The most typical definition of work is in terms of flops, leading to

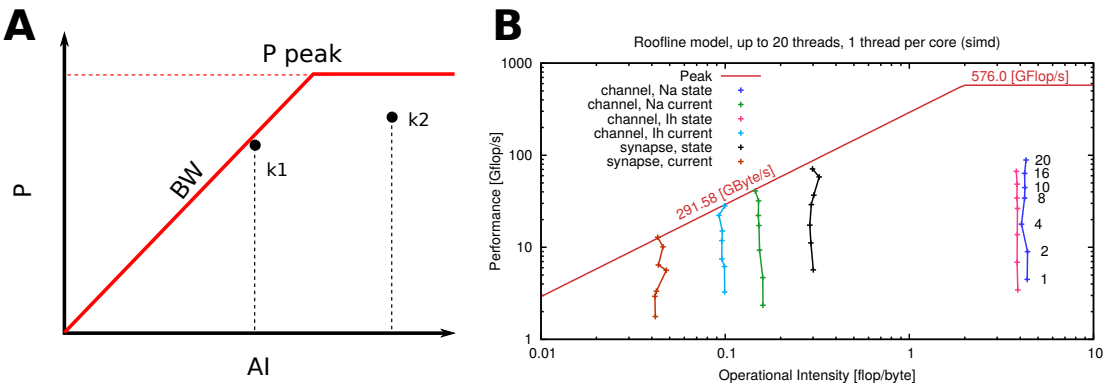

Figure B.1 – Visual representation of the roofline model. **A** $k1$ is a memory-bound kernel that has reached peak performance; $k2$ is a compute-bound kernel that has not reached peak performance. **B** application of the roofline model to G-based state and current kernels on a Power 8 machine from (Ewart et al., 2015). Current kernels are memory-bound, and reach saturation of the memory bandwidth at maximum parallelism. State kernels are compute-bound, but are unable to reach peak performance, although additional investigation is warranted to explain the cause.

a definition of performance in Gflop/s, however for kernels that do not perform arithmetic operations – e.g. sorting – finding a meaningful definition of w can be quite difficult. A second challenge in computing the roofline model is understanding the concept of arithmetic intensity. It mathematical definition is quite simple, but it hides a few subtleties:

$$AI = \frac{w}{b}. \tag{B.1}$$

In addition to the difficulty of defining a meaningful work metric w, one must also decide what to include in the data traffic b. The standard definition is to consider only traffic originating from main memory, such that for a kernel with cache reuse, the "reused" traffic would not be counted. This definition is consistent with using the memory bandwidth as the maximum rate of data transfer. However, it remains unclear whether the "reused" traffic should actually be included in b, or in cases where unnecessary data is transferred – e.g. for noncontiguous data access – it is unclear whether only the useful data should be counted, or all the data.

Despite a slight lack of clarity in some of its definitions, the roofline model is an extremely powerful tool for performance diagnosis. Once all the relevant quantities have been defined and measured, it is possible to draw a diagram such as the one in Figure B.1, which presents both the idealized machine model and the empirical measurements of kernels. At a first glance, it is easy to see that the closer a kernel is to the idealized machine performance, the smaller optimisation efforts are required. Moreover, the roofline model allows to distinguish between two performance profiles: core-bound kernels – i.e. those that lie to the right of the red line elbow in Figure B.1 – and memory bound kernels. For example, $k1$ in Figure B.1 is a memory-bound kernel that has achieved maximum performance. Short of changing its data

access pattern to decrease the pressure on the memory subsystem, no optimisation could improve its performance. On the other hand, $k2$ is a core-bound kernel that has not achieved maximum performance. Optimizations such as increasing the Instruction Level Parallelism (ILP) or SIMD vectorisation could have beneficial effects on performance.

In addition to its diagnostic power, the roofline model can be endowed with a predictive power by assuming that the performance of a kernel will only be bounded by its relevant bottleneck. Thus, given a kernel's memory traffic b and amount of work w, the roofline estimate of performance is given by

$$P_{roofline} = min\big(P_{peak}\,,\; AI \times BW\big). \tag{B.2}$$

While this assumption may seem simplistic, it offers a powerful yet simple tool to reason about performance. The roofline model has been widely-used as a fundamental preliminary tool in software performance analysis, and has been extended to include non-contiguous memory accesses (Nugteren and Corporaal, 2012), to better account for the presence of caches (Ilic et al., 2013) and even to FPGAs (Da Silva et al., 2013). However, it suffers from a few shortcomings especially for serial execution and for memory-bound kernels.

Application of the roofline model to G-based ion channel and synapse kernels on an IBM Power 8 architecture In our performance analysis of the IBM Power 8 architecture (Ewart et al., 2015) we used the roofline model as a tool to understand the performance properties of ion channel and synapse kernels in a G-based detailed neuron model. We found that performance increased linearly with the number of threads until peak memory bandwidth was reached, and that vectorisation improved the performance of all compute-bound and some memory-bound kernels. However, most state kernels did not reach peak performance on this architecture as shown in Figure B.1, thus methods to reduce instruction dependencies and latency should be investigated. Finally, we found that the powerful simultaneous multi-threading (SMT) capabilities of the architecture did not provide significant performance benefits.

B.1.2 Fundamentals of The ECM model

The Execution-Cache-Memory performance model is a *grey-box* model developed by the HPC group at the university of Erlangen. It uses a mixed approach combining an analytic formulation with some phenomenological input, and outputs a runtime prediction at the granularity of individual clock cycles. The ECM model was first introduced by Treibig and Hager (2010) and successively refined and validated on modern Intel and AMD multicore architectures (Hofmann et al., 2017, 2018; Stengel et al., 2015). A recent review provides a clean and detailed description by abstracting, formalizing and recasting it as a universal modelling approach based on a strict differentiation between application and machine models (Hofmann et al., 2019).

In the ECM model, one must first define several contributions to the runtime of a given loop, such as: the in-core execution time assuming data is already loaded in registers T_{OL}, the time needed to load data into registers from the L1 cache T_{nOL}, the data traffic time between caches T_{L1L2}, T_{L2L3} and the data traffic time from main memory T_{L3Mem}. These contributions must be combined to obtain two quantities: T_{core} and T_{data}, representing the time that the loop would spend in core execution if data were instantaneously available, and the time required to move the data across the memory hierarchy, respectively. One of the core assumptions of the ECM model is that these two quantities can overlap, therefore single-thread runtime predictions can be obtained using the formula

$$T = max(T_{core}, T_{data}). \tag{B.3}$$

Estimating the core component To estimate T_{core} one typically assumes maximum throughput of all instructions, even though in this work we are sometimes forced to extend the model by discarding this assumption to obtain more accurate results. Code analysis tools such as the Intel Architecture Code Analyzer (IACA) (Intel, 2017) or the experimental Open Source Code Analyzer (OSACA) (Laukemann et al., 2018) can provide some non-analytical input to the model, allowing to greatly increase its prediction accuracy at the cost of losing some interpretability.

Estimating the data component In this case we need to provide two kinds of information: the runtime contributions from the individual caches and the formula to combine them. For individual caches we must find the memory traffic, a difficult task that can be affected by the following aspects:

- total memory requirements of the kernel;
- cache reuse or blocking;
- cache associativity;
- victim caching;
- cache replacement policy.

For the relatively simple case of a kernel with no reuse, requiring a data traffic of b, denoting the L1L2 bandwidth with BW_{L1L2} we compute the contribution:

$$T_{L1L2} = \frac{b}{BW_{L1L2}}. \tag{B.4}$$

To combine the individual caches' contributions into T_{data}, on all recent Intel server microarchitectures, the best accuracy in predictions is obtained assuming that there is no temporal overlap between any cache transfers, as shown in Figure B.2 from (Hager and Wellein, 2016).

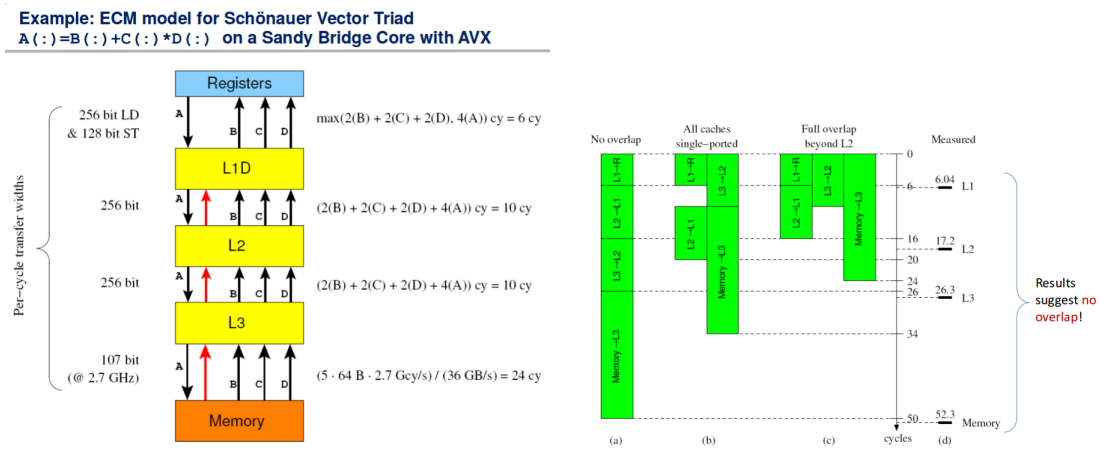

Figure B.2 – Schematic view of estimating the data component in the ECM model. Taken from (Hager and Wellein, 2016). *Left:* runtime contributions for traffic between caches and with main memory are computed independently based on bandwidth and data requirements of the kernel. *Right:* different assumptions for overlap of cache transfers are tested. On Intel architectures, the non-overlapping assumption yields the most accurate predictions.

Thus the formula to compute T_{data}, assuming the dataset must be fetched from main memory, is

$$T_{data}^{Mem} = T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem}. \tag{B.5}$$

In ECM, two shorthand notations are used to simplify the presentation of the model, one for the individual contributions

$$\{T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L3Mem}\}, \tag{B.6}$$

and one for the runtime prediction, assuming that the dataset fits in different levels of the cache hierarchy

$$\left\{T^{L1} \rceil T^{L2} \rceil T^{L3} \rceil T^{Mem}\right\}. \tag{B.7}$$

Assuming there is no overlap between caches, the predictions for data originating in different levels of the cache hierarchy are defined using eq. (3.1) as:

$$\begin{aligned} T^{L1} &= max(T_{OL}, T_{nOL}), \\ T^{L2} &= max(T_{OL}, T_{nOL} + T_{L1L2}), \\ T^{L3} &= max(T_{OL}, T_{nOL} + T_{L1L2} + T_{L2L3}), \\ T^{Mem} &= max(T_{OL}, T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem}). \end{aligned} \tag{B.8}$$

B.1.3 Runtime ECM predictions for shared memory parallelism

The procedure described above prescribes a way to define the fundamental computational characteristics of a kernel under the ECM model and to make runtime predictions for serial execution. To make predictions on the parallel runtime (in a shared memory configuration), an additional assumption is required. The standard ECM model assumes that performance scales linearly with the number of threads, until a bottleneck from a shared serial resource is used, typically the memory interface (Hofmann et al., 2015). Thus starting from the definition of an amount W of *work* done – e.g. one fully processed Cache Line (CL) worth of data – we can define the single thread performance as

$$P(1) = \frac{W}{T}, \tag{B.9}$$

where T is the runtime required to complete the amount of work W and can be obtained from eq. (3.1). From the assumption that performance scales linearly until a bottleneck is reached, e.g. P_{BW} determined by the memory bandwidth, we can easily obtain the formula for shared memory parallelism using n_t threads

$$P(n_t) = min(n_t P(1), P_{BW}). \tag{B.10}$$

Although the standard approach has been extended to account for an additional latency term picked up by threads as the level of parallelism (and thus the contention of the memory interface) increases (Hofmann et al., 2018), for our intents and purposes eq. (B.10) provides sufficient accuracy and leads to better interpretability. Since the proposed extension requires the fitting of an additional hardware-specific and kernel-specific parameter, and we wish to maintain our model as general as possible, we decide instead to use the standard approach.

The ECM model allows to compute the *saturation point*, defined as the number of threads at which the serial memory bottleneck is saturated and dominates performance. Once this point is reached, the kernel's performance cannot be improved by increasing the number of threads. Figure B.3 shows a data bound kernel on the left, and a core bound kernel on the right. For both kernels, using up to three threads does not saturate the memory bandwidth (see Figure B.3B) and a small portion of the execution time could still be shaved off by adding more parallelism, represented by the purple slots that do not overlap with the yellow memory slots in the equivalent serial timelines under the parallel threads. At four threads, as shown in Figure B.3C, saturation effects dominate the performance. It is possible to write the runtime in (3.1) assuming that data resides in different levels of the cache, and also including parallelism,

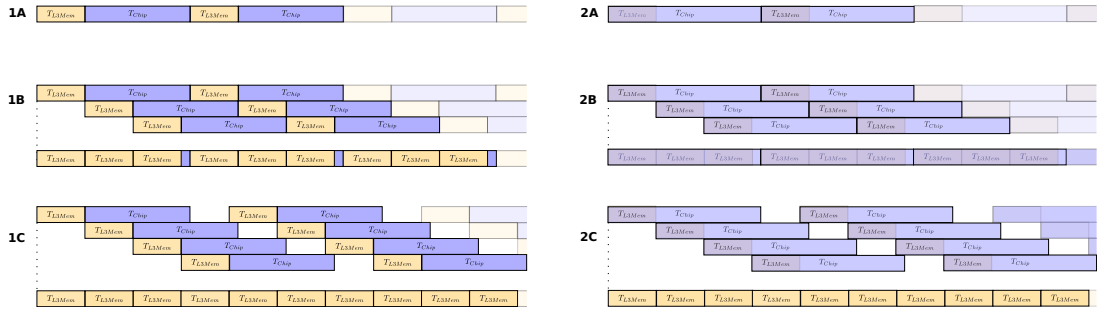

Figure B.3 – Diagram of modelled parallel execution in the ECM model. The layout of this image was inspired by (Hager and Wellein, 2016). **1A,1B,1C** represent a kernel bounded by data traffic, executed using 1,3 and 4 parallel threads respectively. At 4 parallel threads we have saturation of the memory bottleneck. **2A,2B,2C** represent a kernel bounded by in-core arithmetic operations, executed using 1,3 and 4 parallel threads respectively. **2A** In the serial execution, the time for memory transfers is completely hidden by the time spent in the computation. **2B** Using 3 shared memory threads saturation still has not happened, and the parallel component to the runtime still dominates. **2C** At 4 parallel threads we have saturation of the memory bottleneck.

using (B.10) as follows:

$$\begin{aligned} T^{L1}(n_t) &= \frac{T^{L1}(1)}{n_t}, \\ T^{L2}(n_t) &= \frac{T^{L2}(1)}{n_t}, \\ T^{L3}(n_t) &= \frac{T^{L3}(1)}{n_t}, \\ T^{Mem}(n_t) &= max\left(\frac{T^{Mem}(1)}{n_t}, T_{L3Mem}\right). \end{aligned} \tag{B.11}$$

For simplicity of notation, we will often omit the (n_t) notation, in which case it shall be considered that $n_t = 1$. The presence of the max operator in $T^{Mem}(n_t)$ is a consequence of the saturation phenomenon, reflected by the fact that once $T_{L3Mem} \geq \frac{T^{Mem}(1)}{n_t}$, then adding parallelism does not decrease the runtime. It is possible to explicitly compute the number of threads at which saturation occurs, using the formula:

$$n_{satur} = \left\lceil \frac{T^{Mem}}{T_{L3Mem}} \right\rceil. \tag{B.12}$$

The intuition behind this formula can be understood looking at Figure B.3, as the value of n_{satur} represents none other than the number of times that the yellow boxes (T_{L3Mem}) fit into the total runtime for a serial iteration. Finally it should be noted that ignoring the rounding-up

to the nearest integer in (B.12) yields the formula

$$S = \frac{T^{Mem}}{T_{L3Mem}}, \tag{B.13}$$

which could be interpreted as the maximum potential speedup that could be gained from shared memory parallelism for a given kernel.

B.1.4 Inference based on the ECM model

We explain here how we make extensive use of the ECM model's interpretability to make inference on various performance properties. The first notion we introduce is the difference between **core-bound** and **data-bound** kernels. This distinction stems directly from the definition of the predicted runtime (3.1): core-bound kernels are such that $T_{core} \geq T_{data}$, and conversely for data-bound kernels. Note that this is purely an intrinsic property of the kernel that has a direct effect only on its serial runtime. While it is often true that data-bound kernels have worse shared-memory scalability than core-bound kernels, this is not necessarily the case, as it is possible to conceive a kernel with $T_{OL}, T_{L3Mem} << T_{nOL}, T_{L1L2}, T_{L2L3}$ that would be data-bound as well as highly scalable over multiple shared memory threads. Thus memory bandwidth saturation is a property that is not necessarily tied to data-boundedness.

Another quantity that we are often interested in computing is bandwidth utilization, defined based on the kernels's memory requirements b as:

$$\begin{aligned} BW_{util} &= \frac{BW_{expressed}}{BW_{Mem}}, \\ BW_{expressed} &= \frac{b}{T^{Mem}} \text{ [GB/s]}. \end{aligned} \tag{B.14}$$

In practice, BW_{util} represents the percent of saturation of the memory bandwidth by a given kernel, compared to the peak available memory bandwidth of the system. The ECM model provides a way to express the utilization only in terms of ECM time contributions. Recall that the T_{L3Mem} is defined following (B.4) by

$$T_{L3Mem} = \frac{b}{BW_{Mem}}. \tag{B.15}$$

Thus we can rewrite

$$BW_{util} = \frac{BW_{expressed}}{BW_{Mem}} = \frac{b}{BW_{Mem}} \times \frac{1}{T^{Mem}} = \frac{T_{L3Mem}}{T^{Mem}}. \tag{B.16}$$

Note that bandwidth utilization, saturation point and maximum achievable parallel speedup are all related concepts, as can be easily seen by their definitions (B.12), (B.13), (B.16). This is a direct consequence of the idealized full-throughput assumption, and in practice these quantities need not be so tightly linked by mathematical formulas.

To put the emphasis on the hardware bottlenecks identified by the ECM model, we can "invert" the runtime prediction formulas to obtain a description in terms of hardware contributions. In the serial case, hardware contributions are simply defined based on the regular ECM dimensions (3.2). We define the contributions as follows:

$$
\begin{aligned}
T_{core} &= \begin{cases} T_{OL} & T_{OL} \geq T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem} \\ 0 & \text{otherwise} \end{cases} \\
T_{caches} &= \begin{cases} 0 & T_{OL} \geq T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem} \\ T_{nOL} + T_{L1L2} + T_{L2L3} & \text{otherwise} \end{cases} \\
T_{DRAM} &= \begin{cases} 0 & T_{OL} \geq T_{nOL} + T_{L1L2} + T_{L2L3} + T_{L3Mem} \\ T_{L3Mem} & \text{otherwise} \end{cases}
\end{aligned}
\tag{B.17}
$$

In the parallel case, however, special care must be taken to distinguish scalable and non-scalable contributions. Scalable contributions are logically improved by adding more threads, and usually correspond to physical hardware that is replicated for each thread. The scalable contributions are T_{core} and T_{caches}, while the non-scalable contribution is T_{DRAM}. In a parallel execution using n_t threads, we define the hardware contributions as follows: non-scalable contributions remain unchanged from the serial execution; scalable contributions are scaled by $\frac{1}{n_t}$ until saturation of the memory bandwidth, then set to 0. Thus the definition of parallel hardware contributions is given by:

$$
\begin{aligned}
T_{core}(n_t) &= \begin{cases} 0 & \text{data-bound} \\ \frac{T_{core}(1)}{n_t} & \text{core-bound, unsaturated} \\ 0 & \text{core-bound, saturated} \end{cases} \\
T_{caches}(n_t) &= \begin{cases} \frac{T_{caches}(1)}{n_t} - \frac{n_t-1}{n_t} T_{DRAM}(1) & \text{data-bound, unsaturated} \\ 0 & \text{data-bound, saturated} \\ 0 & \text{core-bound} \end{cases} \\
T_{DRAM}(n_t) &= T_{DRAM}(1)
\end{aligned}
\tag{B.18}
$$

The correction factor $\frac{n_t-1}{n_t} T_{DRAM}(1)$ is necessary to maintain the property that the runtime of a data-bound kernel can be obtained via the sum $T_{caches} + T_{DRAM}$. It can be explained by looking at Figure B.3 1B and computing the impact of the small portion of purple – i.e. T_{cache} – line that remains visible.

In the course of our application of the ECM model, we have sometimes been forced to relax the full-throughput hypothesis to account for significant deviations of measured runtime from the model's predictions. An example of this has been in kernels with a strong Critical Path (CP) component. For inference on **CP-bound** kernels it is sufficient to replace T_{OL} by T_{CP} and treat them as regular core-bound kernels. Another example is kernels where memory latency effects are not negligible. While our understanding of such kernels is still a bit limited, we are still

interested in performing inference on them as they can represent a significant portion of brain tissue simulations' runtime. First, we must distinguish between fully latency-bound kernels and partially latency-bound kernels. While we have not encountered any kernels of the first type during our investigation, we speculate that for inference it is simply possible to replace the various data-traffic ECM contributions with their corresponding latencies and treat them as regular data-bound kernels. For **partially latency-bound** kernels, as will be shown to be the case of the spike delivery kernel, we make the assumption that memory latency dominates over all other latencies, and thus attribute all of the kernel's runtime to the DRAM contribution. In terms of computing their parallel performance, we assume as for other kernels that the serial performance scales linearly with the number of threads, until the memory bandwidth bottleneck is reached.

B.2 ECM on Intel Skylake

The Intel Skylake (SKX) architecture presents a few peculiarities that had never been accounted for in an ECM model before our work. Aside from a few microarchitectural features, the two main novelties were the AVX512 vector registers and the L3 victim cache (Cremonesi et al., 2019a; Hager et al., 2018). We present here a detailed account of the strategy that we developed to compute the individual components of the ECM model applied to the SKX architecture.

Hardware characteristics Our reference architecture is the Intel(R) Xeon(R) Gold 6140 Skylake processor with AVX512 vectorisation. We consider that Sub-NUMA clustering is always turned off, while the L3 prefetcher should be enabled (it was disabled by default on our machines). The most relevant hardware characteristics required by the performance model are summarised in Table 3.1. We obtained these values either directly from the vendor's spec sheets, by custom-designed benchmarks or from reference tables (Fog, 2017). In particular, the memory bandwidth was obtained by running the STREAM (McCalpin, 1995) benchmark at maximum thread capacity, and taking the best reported value. The peak performance, on the other hand, was computed using the fact that each core can retire two vectorized FMA instructions per cycle, and that an AVX512 register holds 8 double-precision floating point values, leading to the formula:

$$18\ P_{peak}(1\text{ core}) = 18 \times 2.3 \times 8 \times 2 \times 2. \tag{B.19}$$

For validation and benchmarking we use the CoreNEURON implementation as reference (Kumbhar et al., 2019b). We compiled all benchmarks using the Intel 18.0.1 compiler with options −xCORE−AVX512 −qopt−zmm−usage=high \lstinline-qopt-streaming-stores never!!, and inserted # **pragma** ivdep, **#pragma** vector aligned and **#pragma** omp simd simdlen(N) directives where appropriate to ensure vectorisation and to disable the generation of nontemporal stores. The frequency of all CPU cores was set to 2.3GHz with the tool likwid−setFrequencies −f 2.3 −−umin 2.3 −−umax 2.3.

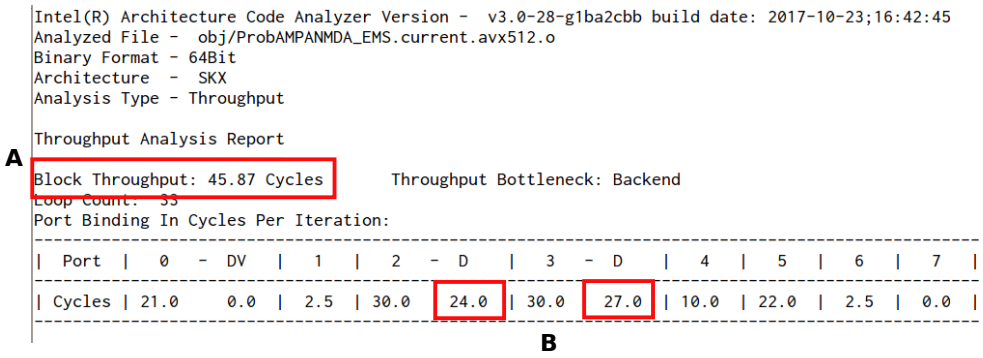

Figure B.4 – Example of IACA output. **A** the block throughput is used to estimate T_{OL}. IACA provides the value for each block iteration of the code in the assembly loop. Therefore vectorisation and loop unrolling must be taken into account to obtain predictions per scalar iteration. **B** IACA also reports the cycles spent on the data ports (port 2-D and 3-D in the SKX reference architecture). The maximum of these two values is used to compute T_{nOL}. As before, vectorisation and loop unrolling must be taken into account.

Estimating T_{OL}, T_{nOL} As we already mentioned, there are two ways to approach this task. One is to use code analyser tools such as IACA (Intel, 2017), the other to do it by hand. We almost exclusively use the former because in our experience it allows to obtain predictions that are much more accurate by analysing directly the code generated by the compiler. Moreover our analysis has shown that it is extremely difficult to predict the throughput of complex sequences of operations on modern hardware (Ewart et al., 2019). There are two quantities of interest in IACA's output. To obtain T_{OL} we look at the reported block throughput (see Figure B.4A), taking care to divide by the ratio of scalar to vector iterations (e.g. divide by 8 for double precision AVX512 vectorisation) because IACA's output is reported in loop iterations. To obtain T_{nOL}, we take the maximum of the cycles reported on the two data ports (see Figure B.4B), and once again divide by the ratio of scalar to vector iterations to obtain T_{nOL} per scalar iteration.

When a Critical Path (CP) prediction is required, IACA can also provide a more accurate estimate than trying to figure out the latency of all the operations. While on older architectures the –analysis LATENCY option allowed to directly obtain a CP estimate, this is no longer possible in IACA v3.0, which is the minimum version that supports the SKX architecture. For this reason we resort to using the estimate for the Haswell architecture (HSW) from IACA v2.1.

To instead compute T_{OL} by hand, a straightforward strategy would be to compute the sum of the throughputs of all relevant operations, and divide by the number of ports in the microarchitecture able to carry out such operations, similarly to what was done to compute the peak performance in (B.19). By not analysing the assembly code, however, one may overlook important details such as additional operations or optimisations introduced by the compiler. Moreover, by neglecting port utilization, one might end up over-optimistically estimating the core performance of the kernel.

To estimate T_{nOL} by hand one can directly use the knowledge from Table 3.1, by estimating the number of load and store operations via the number of variable reads and writes in the kernel. Once again, without looking at the assembly code, the modeller cannot know with certainty how many load/store operations were issued, and in the presence of register reuse or register spilling the estimates by hand could be off by several cycles.

Estimating T_{L1L2}, T_{L2L3} This phase requires intimate knowledge of the cache hierarchy and the memory requirements of the kernel. For streaming kernels, i.e. kernels with little to no data reuse, a typical approximation that works quite well is to count all *unique* reads and writes of data, and assume that after the first time a variable is touched it remains in the L1 cache for the execution of the kernel.

Once the number of reads and writes has been established, it must be converted into load traffic b_{load} and store traffic b_{store}. Note that, in certain cases, the value of b_{load} and b_{store} may be different according to the level in the cache hierarchy. For the L1-L2 traffic on the SKX architecture, we make the following assumptions:

- write-allocate is always used, thus a write always generates the same amount of both store *and* load traffic;
- dirty evicted cache lines from L1 are moved to L2, but not clean ones;
- the data transfer bandwidth is 64 B/cycle.

Thus we have the formula for the L1L2 runtime contribution:

$$T_{L1L2} = \frac{b_{load} + b_{store}}{64}. \tag{B.20}$$

The SKX architecture introduced a victim L3 cache, in contrast with Intel's recent architectures. For this cache, we make the following assumptions:

- write-allocate is always used, thus a write always generates the same amount of both store *and* load traffic;
- load traffic from DRAM goes straight to L2;
- all evicted cache lines from L2, both clean and dirty, are moved to L3;
- the data path between the L2 and the L3 cache can be assumed to provide a bandwidth of 16 B/cycle in both directions (i.e., full duplex).

In this case, the formula for the L2L3 runtime contribution is given by:

$$T_{L2L3} = max\left(\frac{b_{load}}{16}, \frac{b_{store}}{16}\right). \tag{B.21}$$

Computing the cache-level data traffic comes with an additional complexity of predicting reuse. This occurs often for example in multidimensional stencil kernels, where because of row-wise (or column-wise) representation, elements from the non-leading dimension may still be in cache after a few iterations along the leading dimension. In these cases, the ECM allows to compute a *layer condition*, i.e. the maximum number of leading dimension iterations that still allows cache reuse. Due to the peculiar streaming nature of our kernels, we never used this type of analysis in our work, so we point the interested reader to (Hammer et al., 2017) for a detailed explanation on how to compute layer conditions within the ECM model framework.

Estimating T_{L3Mem}, T_{L2Mem} Due to the L3 victim cache, it seems that special care must be taken in estimating the traffic from memory. This leads us to distinguish the two quantities: T_{L2Mem} representing the load traffic from memory to L2, and T_{L3Mem} representing the store traffic from L3 to back to memory. In any case, both are bounded by the memory bandwidth, thus we compute them with the formula:

$$\begin{aligned} T_{L2Mem} &= \frac{b_{load}}{BW}, \\ T_{L3Mem} &= \frac{b_{store}}{BW}, \end{aligned} \tag{B.22}$$

where b_{load} accounts for both read-only and write-allocate generated traffic. Give that maintaining this distinction can be cumbersome, we often aggregate them in a total traffic from DRAM. For consistency with other architectures and simplicity of notation, we typically abuse the T_{L3Mem} symbol to be defined as $T_{L3Mem} + T_{L2Mem}$, taken to simply mean the total traffic to and from DRAM.

Listing B.1 Illustrative example using the STREAM triad kernel.

```
for(i=0;i<N;++i)
{
    A[i] = B[i] + k*C[i]
}
```

An example on the STREAM triad kernel We illustrate the application of the ECM model to SKX with the STREAM triad kernel developed by McCalpin (1995) shown in Listing B.1. Considering only AVX vectorisation as an example, this kernel has the following properties *per scalar iteration*:

- inverse throughput prediction of $T_{OL} = 0.375$ cycle/scalar iter, estimated by pen-and-paper calculations;

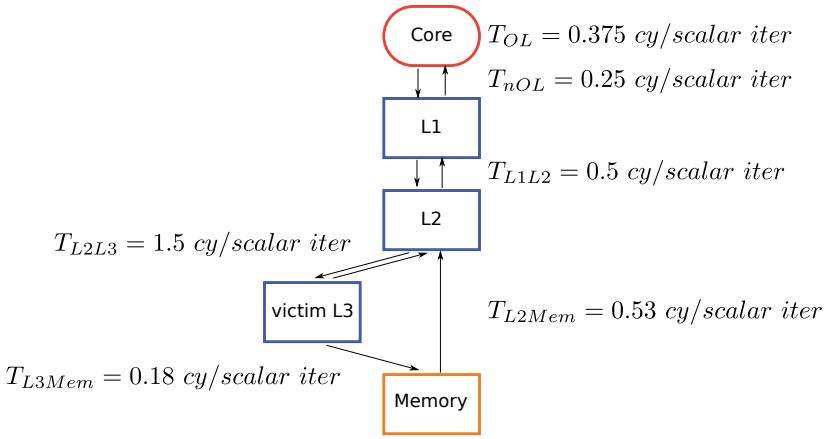

Figure B.5 – Diagram of ECM contributions, respecting the cache hierarchy of the reference SKX architecture. Individual contributions are computed based on the data traffic requirements and the bandwidth of the relevant data path, or by using the IACA tool. The non-overlapping assumptions prescribes that no contribution from data traffic overlaps with any other. This assumption has been shown to provide accurate predictions for Intel architecture, and was validated also on our reference architecture.

- Two loads and one store, so T_{nOL} = 0.25 cycle/scalar iter, estimated by pen-and-paper calculations;
- V_{L1L2} = 32 B/scalar iter (including write-allocate);
- $T_{L1L2} = \frac{32}{64} \frac{\text{B/scalar iter}}{\text{B/cycle}}$ = 0.5 cycle/scalar iter;
- Due to the victim L3 cache, we have to distinguish in-memory and in-L3 datasets:
  - L3: V_{L2L3}^{L3} = 48 B/scalar iter (read + write);
  - Memory: V_{L2L3}^{Mem} = 24 B/scalar iter (write-only);
- The transfer times between L2 and L3 are the same in this particular case because reads and writes to L3 can overlap:
  - L3: $T_{L2L3}^{L3} = \max\left(\frac{24\,\text{B/it}}{16\,\text{B/cy}}, \frac{24\,\text{B/it}}{16\,\text{B/cy}}\right)$ = 1.5 cycle/scalar iter;
  - Memory: $T_{L2L3}^{Mem} = \frac{24\,\text{B/it}}{16\,\text{B/cy}}$ = 1.5 cycle/scalar iter (write-only);
- V_{L2Mem} = 24 B/scalar iter (read-only traffic);
- $T_{L2Mem} = \frac{24\,\text{B/it}}{105\,\text{GB/s}} \times 2.3\,\text{Gcy/s}$ = 0.53 cycle/scalar iter;
- V_{L3Mem} = 8 B/scalar iter (write-only traffic);
- $T_{L3Mem} = \frac{8\,\text{B/it}}{105\,\text{GB/s}} \times 2.3\,\text{Gcy/s}$ = 0.18 cycle/scalar iter.

Figure B.5 summarises the individual contributions in a diagram representing the relevant ECM components. The ECM model contributions for the STREAM triad kernel in Listing B.1 on SKX-AVX therefore are:

$$\{T_{OL} \parallel T_{nOL} \mid T_{L1L2} \mid T_{L2L3} \mid T_{L2Mem} + T_{L3Mem}\} = \{0.38 \parallel 0.25 \mid 0.5 \mid 1.5 \mid 0.71\} \text{ cycle/scalar iter}, \tag{B.23}$$

with corresponding predictions according to the non-overlapping machine model of

$$\{0.38 \rceil 0.75 \rceil 2.25 \rceil 2.96\} \text{ cycle/scalar iter.} \tag{B.24}$$

For validation we compared these predictions to benchmark measurements and obtained (0.39⌉0.73⌉2.37⌉4.3) cy/it, which is in reasonable agreement with the model. The deviation in memory could be fixed by introducing a latency penalty (see (Hofmann et al., 2017)), but since the memory contribution is rather small for most of the kernels studied here we opted for a simpler model.

B.3 Kernel code listings

This section contains several pages of code listings for the most relevant simulation kernels analysed in this thesis.

Listing B.2 Example of ion channel current kernel: Im current.

```
for(int i=0; i<cntml; ++i) {
  int nd_idx = _ni[i];
  double v = vec_v[nd_idx];
  ek[i] = ion_data_ek[ion_idx[i]];
  gIm[i] = gImbar[i] * m[i] ;
  ik[i] = gIm[i] * ( v - ek[i] ) ;
  ion_data_ik[ion_idx[i]] += ik[i] ;
  vec_rhs[nd_idx] -= ik[i];
  vec_d[nd_idx] += gIm[i];
}
```

Listing B.3 Example of Excitatory synapse current kernel: ProbAMPANMDA_EMS current.

```
for(int i=0; i<cntml; ++i) {
  double v = vec_v[_ni[i]];
  mggate[i] = 1.0 + exp (-0.062*v)*(mg[i]/3.57);
  mggate[i] = 1.0/mggate[i];
  g_AMPA[i] = gmax * ( B_AMPA[i] - A_AMPA[i] ) ;
  g_NMDA[i] = gmax * ( B_NMDA[i] - A_NMDA[i] ) ;
  g_NMDA[i] *= mggate[i] ;
  g[i] = g_AMPA[i] + g_NMDA[i] ;
  i_AMPA[i] = g_AMPA[i] * ( v - e[i] ) ;
  i_NMDA[i] = g_NMDA[i] * ( v - e[i] ) ;
  i_tot[i] = i_AMPA[i] + i_NMDA[i] ;
  double rhs = i_tot[i];
  double _mfact =  1.e2/(_nd_area[nd_area_idx[i]]);
  double loc_g =  g_AMPA[i] + g_NMDA[i] ;
  loc_g *= _mfact;
  rhs *= _mfact;
  vec_shadow_rhs[i] = rhs;
  vec_shadow_d[i] = loc_g;
}
```

Listing B.4 Example of ion channel state kernel: Im state.

```
for(int i=0; i<cntml; ++i) {
  double v = vec_v[_ni[i]];
  mAlpha[i] = 6.43e-3*(v + 154.9);
  mAlpha[i] /= exp((v + 154.9)/11.9)-1.;
  mBeta[i] = 0.193*exp(v/33.1);
  mInf[i] = mAlpha[i]/(mAlpha[i]+mBeta[i]);
  mTau[i] = 1./(mAlpha[i]+mBeta[i]) ;
  double incr = (1-exp(-dt/mTau[i]));
  incr *= (mInf[i]/mTau[i])/(1./mTau[i]) - m[i];
 m[i] += incr;
}
```

Listing B.5 Example of synapse state kernel: ProbAMPANMDA_EMS state.

```
for(int i=0; i<cntml; ++i) {
    A_AMPA[i]+=(1.-exp(dt*(-1./tau_r_AMPA[i])))*(-A_AMPA[i]);
    B_AMPA[i]+=(1.-exp(dt*(-1./tau_d_AMPA[i])))*(-B_AMPA[i]);
    A_NMDA[i]+=(1.-exp(dt*(-1./tau_r_NMDA[i])))*(-A_NMDA[i]);
    B_NMDA[i]+=(1.-exp(dt*(-1./tau_d_NMDA[i])))*(-B_NMDA[i]);
}
```

Listing B.6 Linear algebra kernel.

```
//triangularization
for (i = ncompartments - 1; i >= ncells; --i) {
  p = vec_a[i]/vec_d[i];
  vec_d[parent_index[i]] -= p*vec_b[i];
  vec_rhs[parent_index[i]] -= p*vec_rhs[i];
}
//solve boundaries (ignored)
for (i = 0; i < ncells; ++i) {
  vec_rhs[i] /= vec_d[i];
}
//backward substitution
for (i = ncells; i < ncompartments; ++i) {
  vec_rhs[i] -= vec_b[i]*vec_rhs[parent_index[i]];
  vec_rhs[i] /= vec_d[i];
}
```

Listing B.7 GIF current kernel.

```
for(int i=0; i<cntml; ++i) {
  double v = vec_v[ _ni[i]];
  i_eta[i] = eta1[i] + eta2[i] + eta3[i] ;
  gamma_sum[i] = gamma1[i] + gamma2[i] + gamma3[i] ;
  lambda[i] = lambda0[i] * exp ( ( v - Vt_star[i] - gamma_sum[i] ) / DV[i] ) ;
  p_dontspike[i] = exp ( - lambda[i] * ( dt * ( 1e-3 ) ) ) ;
  irefrac[i] = grefrac[i] * ( v - 0.0 ) ;
  i_tot[i] = irefrac[i] + i_eta[i] ;
  double rhs = i_tot[i];
  double loc_g = grefrac;
  double _mfact = 1.e2/(_nd_area[nd_area_idx[i]]);
  loc_g *=  _mfact;
  rhs *= _mfact;
  vec_shadow_rhs[i] = rhs;
  vec_shadow_d[i] = loc_g;
}
```

Listing B.8 GIF state kernel.

```
for(int i=0; i<cntml; ++i) {
  eta1[i]   += (1.-exp(dt*(-1./tau_eta1[i])))*(-eta1[i]) ;
  eta2[i]   += (1.-exp(dt*(-1./tau_eta2[i])))*(-eta2[i]) ;
  eta3[i]   += (1.-exp(dt*(-1./tau_eta3[i])))*(-eta3[i]) ;
  gamma1[i] += (1.-exp(dt*(-1./tau_gamma1[i])))*(-gamma1[i]) ;
  gamma2[i] += (1.-exp(dt*(-1./tau_gamma2[i])))*(-gamma2[i]) ;
  gamma3[i] += (1.-exp(dt*(-1./tau_gamma3[i])))*(-gamma3[i]) ;
}
```

Listing B.9 Example of excitatory synapse current kernel from Simplified model: ProbFiltAMPANMDA_EMS current.

```
for(int i=0; i<cntml; ++i) {
  double v = vec_v[_ni[i]];
  mggate[i] = 1.0 + exp (-0.062*v)*(mg[i]/3.57);
  mggate[i] = 1.0/mggate[i];
  g_AMPA[i] = gmax * ( B_AMPA[i] - A_AMPA[i] ) ;
  g_NMDA[i] = gmax * ( B_NMDA[i] - A_NMDA[i] ) ;
  g_NMDA[i] *= mggate[i] ;
  g[i] = g_AMPA[i] + g_NMDA[i] ;
  i_AMPA[i] = g_AMPA[i] * ( v - e[i] ) ;
  i_NMDA[i] = g_NMDA[i] * ( v - e[i] ) ;
  i_dend[i] = i_AMPA[i] + i_NMDA[i] ;
    i_tot[i] = w_corr[i] * idend[i] ;
  double rhs = i_tot[i];
  double _mfact =  1.e2/(_nd_area[nd_area_idx[i]]);
  double loc_g =  g_AMPA[i] + g_NMDA[i] ;
  loc_g *= _mfact;
  rhs *= _mfact;
  vec_shadow_rhs[i] = rhs;
  vec_shadow_d[i] = loc_g;
}
```

Listing B.10 Example of synapse state kernel from Simplified model: ProbFiltAMPANMDA_EMS state.

```
for(int i=0; i<cntml; ++i) {
    A_AMPA[i]+=(1.-exp(dt*(-1./tau_r_AMPA[i])))*(-A_AMPA[i]);
    B_AMPA[i]+=(1.-exp(dt*(-1./tau_d_AMPA[i])))*(-B_AMPA[i]);
    A_NMDA[i]+=(1.-exp(dt*(-1./tau_r_NMDA[i])))*(-A_NMDA[i]);
    B_NMDA[i]+=(1.-exp(dt*(-1./tau_d_NMDA[i])))*(-B_NMDA[i]);
    i_soma[i]+=(1.-exp(dt*(-1./tau_corr[i])))*(- (idend[i] - idend[i]/tau_corr[i])
/(1./tau_corr[i]) - isoma[i]);
}
```

Listing B.11 Spike delivery kernel of the G-based AMPA/NMDA synapse.

```
Event events[];
// loop over n spike_events
for(int e=0; e<n; ++e)
{
  Event spike_event = events[e];
  Target * target = spike_event.target;
  int weight_index = spike_event.weight_index;
  int type = target.type;
  int i = target.index;
  double _lweight_AMPA = _weights[weight_index];
  double _lweight_NMDA = _lweight_AMPA;
  _lweight_NMDA *= NMDA_ratio[i];
  u[i] = u[i] * exp(-(t-tsyn[i])/Fac[i] );
  u[i] += Use[i]*(1.-u[i]);
  R[i] = 1.-(1.-R[i])*exp(-(t-tsyn[i])/Dep[i]);
  Pr[i] = u[i]*R[i];
  R[i] = R[i] - u[i]*R[i] ;
  tsyn[i] = t ;
  A_AMPA[i] += Pr[i]*_lweight_AMPA*factor_AMPA[i];
  B_AMPA[i] += Pr[i]*_lweight_AMPA*factor_AMPA[i];
  A_NMDA[i] += Pr[i]*_lweight_NMDA*factor_NMDA[i];
  B_NMDA[i] += Pr[i]*_lweight_NMDA*factor_NMDA[i];
}
```

Listing B.12 IAF update kernel.

```
  for(int i=0; i<cntml; ++i) {
    V_m[i] = P31_ex_[i] * dI_ex[i] + P32_ex_[i] * I_ex[i] + P31_in_[i] * dI_in[i] +
  P32_in_[i] * I_in[i] + expm1_tau_m_[i] * V_m[i] + V_m[i] ;
    I_ex[i] = P21_ex_[i] * dI_ex[i] + P22_ex_[i] * I_ex[i] ;
    dI_ex[i] = dI_ex[i] * P11_ex_[i] ;
    I_in[i] = P21_in_[i] * dI_in[i] + P22_in_[i] * I_in[i] ;
    dI_in[i] = dI_in[i] * P11_in_[i] ;
  }
```

Listing B.13 IAF PSC contributions kernel.

```
  for(int i=0; i<cntml; ++i) {
    dI_ex[i] = dI_ex[i] + EPSCInitialValue_[i] * weighted_spikes_ex_[i] ;
    weighted_spikes_ex_[i] = 0. ;
    dI_in[i] = dI_in[i] + IPSCInitialValue_[i] * weighted_spikes_in_[i] ;
    weighted_spikes_in_[i] = 0. ;
  }
```

Listing B.14 Spike delivery kernel of the I-based excitatory synapse.

```
Event events[];
// loop over n spike_events
for(int e=0; e<n; ++e)
{
  Event spike_event = events[e];
  Target * target = spike_event.target;
  int weight_index = spike_event.weight_index;
  int type = target.type;
  int i = target.index;
  exc_postsyn_cur[i] += _weights[weight_index];
}
```

write streams	size [MB]	pred traffic [B/it]	meas traffic [B/it]	latency [cy/access]
1	80.0	196	143.8 ± 3905.8	15.0 ± 368.5
1	800.0	196	154.8 ± 390.1	13.6 ± 36.9
1	1600.0	196	340.8 ± 294.2	31.5 ± 29.0
1	4000.0	196	195.1 ± 0.2	18.5 ± 0.2
4	320.0	772	720.2 ± 7538.0	16.3 ± 164.7
4	3200.0	772	616.3 ± 696.3	16.1 ± 15.5
4	6400.0	772	771.1 ± 1159.1	20.5 ± 31.8
4	16000.0	772	771.8 ± 464.6	20.9 ± 13.0
9	720.0	1732	1681.2 ± 1301.7	16.4 ± 13.0
9	7200.0	1732	1729.2 ± 1563.3	18.2 ± 17.2
9	14400.0	1732	1732.1 ± 429.8	19.5 ± 3.5
9	36000.0	1732	1732.9 ± 1.5	19.6 ± 0.5

Table B.1 – Synthetic benchmark mimicking the spike delivery access pattern. We computed the average latency per memory access by diving the runtime of the loop by the number of memory accesses. In the table, we report the median ± max-min over 5 repetitions.

B.4 The LogGP model

The LogGP model is part of a family of analytic performance models developed initially at Berkeley, was adapted and extended by researchers from other centres. Historically, the first model in this family was LogP (Culler et al., 1993). In the LogP model all complex MPI operations are defined on the basis of point-to-point communication, i.e. the cost of sending a message from one compute node to another within the same network. LogP was developed with a focus solely on short (single Byte) messages, but it quickly became clear that the accuracy of its predictions degraded in the case of long messages. Therefore the LogGP model (Alexandrov et al., 1997; Hoefler et al., 2009) was developed to overcome this problem. In the LogGP model, the cost of sending a single message of size m B is given by the analytic formula

$$T_{pt2pt} = L + 2o + G(m-1), \tag{B.25}$$

where

- L is the network latency;
- o is the overhead from non-network operations;
- g is the inverse of the injection rate;
- G is the inverse of the network bandwidth;
- P is the number of processes involved in the communication.

Note that g does not appear above because a single message is considered, while g represents the delay that must occur between two consecutive sends of a message.

The application of the LogGP model is best explained using an example from (Hoefler et al., 2009). Consider the ping-ping-pong benchmark, where a client sends two messages of size m B to a server. Upon completion of the second message reception, the server sends back a single message of size m B to the client. Figure B.6 describes this scenario and the corresponding LogGP model. When the first send is initiated, the client requires $o\,\mu s$ time for non-network operations, such as copying data to the network card. Note that, following (Hoefler et al., 2007a, 2009), in the rest of this work we will always model o as a linear factor of the message size, i.e. $o = o_i + o_s(m-1)$. After this time, the operation of sending data through the interconnect fabric can be initiated. The first byte of the message requires a time $L\,\mu s$ to reach the server, and after that the server receives each subsequent byte with an interval of $G\,\mu s$. Upon receiving the first complete message, the server must spend a CPU time of $o\,\mu s$ to process it. In the meantime, after the client has finished sending the first message, it gets ready to send the second one. Here the LogGP model prescribes the overlap of two operations: on the CPU side, the client must spend $o\,\mu s$ to prepare the second message, while on the network side an

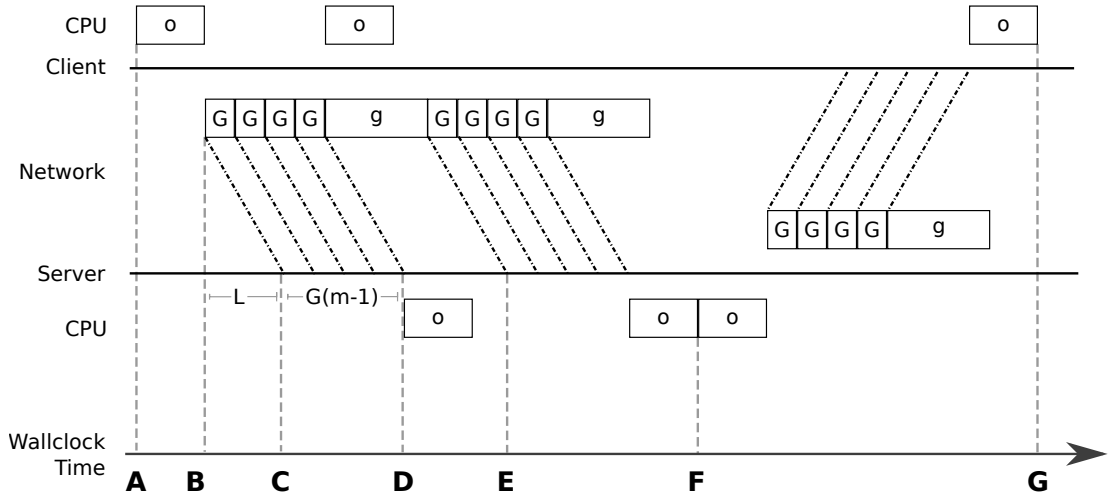

Figure B.6 – Ping-ping-pong illustrative example of the LogGP model. This figure was adapted from the example in (Hoefler et al., 2009). It represents the ping-ping-pong example where the client sends two messages to the server, and after receiving the full messages the server replies with a single message. **A:** beginning of the benchmark. **B:** client needs $o\,\mu s$ of CPU time to prepare the sending of the message. **C:** after an interval of $L\,\mu s$, the first byte of the message reaches its destination. **D:** after $G(m-1)\,\mu s$ the whole message has been delivered. **E:** the injection rate constraint imposes a waiting period of $g\,\mu s$ before the network can process the next message. In the meantime, the CPU time of both the client and server can overlap with this constraint. **F:** both messages have been fully received by the server, which gets ready to send the message back. **G:** end of the benchmark.

injection rate constraint imposes waiting for $g\,\mu s$ before being able to send the next message. In the example in Figure B.6 we have that $g > o$, thus the injection rate represents the most important factor in determining when the next message will be sent. Upon receiving the second message, the server must spend $2o\,\mu s$ to process it and prepare sending the pong message, which will be received by the client fully received by the client after $L+G(m-1)+o\,\mu s$. The total predicted latency of the ping-ping-pong operation is, according to the LogGP model, $2L+4o+g+3G(m-1)\,\mu s$.

B.4.1 LogGP model on Infiniband EDR with HPE-MPI

Even though the performance modelling tools considered here can generalize well to several types of architectures (Hoefler et al., 2009), to validate our performance predictions we restrict our focus to a representative example of an HPC network architecture: a vendor (HPE) MPI implementation based on MPT 2.16 and the MPI 3.0 standard, over an Infiniband EDR 100 GB/s fabric. While it would be possible to take the nominal vendor values for the hardware parameters such as L, g, G, it is highly advised to obtain the values of these parameters through a set of benchmarks. A low-overhead method to compute all the necessary parameters has been proposed (Hoefler et al., 2007a), and ultimately led to the development of the Netgauge tool (Hoefler et al., 2007b). The model parameters can be obtained once and for all, and after that the LogGP model does not require any additional benchmarking efforts. We used the Netgauge v2.4.6 tool introduced in (Hoefler et al., 2007a) to make a first assessment of the LogGP parameters, and the parameters reported in Table 3.10.

Performance penalty for large message sizes In the process of validation, we discovered that the raw LogGP model based on the small-messages parameters from Netgauge was only valid in the context of very small messages, while for messages larger than $P \times 65$B the communication incurred a penalty in both latency and bandwidth. The source of this penalty is unclear, as it could be due to a switch in point-to-point protocols, a change in the underlying algorithm or additional communication to ensure synchronization. Since the details of the underlying communication protocol are not published, one can make at best an educated guess about the underlying protocol switches. We found that, even though the Netgauge tool does detect a change in parameters for large messages, the new parameters reported by the tool were not satisfactory for two reasons: first the message size threshold at which it detects a change in protocol is much larger (around 16 kB) than what we observe; second, the L and G parameters it reports for large sizes are lower by roughly a factor 1.5x than the ones we observe. Therefore, to account for this effect, we introduce two penalty terms in the original LogGP parameters: a latency penalty $p_L = 0.593\mu$s and a bandwidth penalty $p_G = 1.875 \times 10^{-4}\mu$s/B. These parameters were fitted on the data from a benchmark using only double-precision values with $P = 32$, but we find that they generalize quite well to other data types and other cluster sizes. In light of this modification, the formula to obtain a prediction for the latency of

the Allgatherv operation becomes:

$$
\begin{aligned}
T_{comm}(m < 65P) =& 2(P-1)(L+2o_i) + \frac{P-1}{P}(G+2o_s)\left[Nf\delta_{\min}(m_{ID}+m_t)-1\right], \\
T_{comm}(m \geq 65P) =& (P-1)(L+2o_i+p_L) + \frac{P-1}{P}(G+2o_s+p_G)\left[Nf\delta_{\min}(m_{ID}+m_t)-1\right].
\end{aligned}
\tag{B.26}
$$

C Supplementary material for Chapter 5

Figure C.1 presents the scatter plot matrix of the predicted ECM dimension for all kernels in Table 5.1, colour-coded by kernel type (i.e. *current* or *state*). Note that all axis in the Figure have the same dimensions (cycle per scalar iteration) and the same range. This enables visual verification that the variability in the T_{OL} dimension (i.e. the in-core execution) is much larger than the variability in the other, data-traffic related, dimensions.

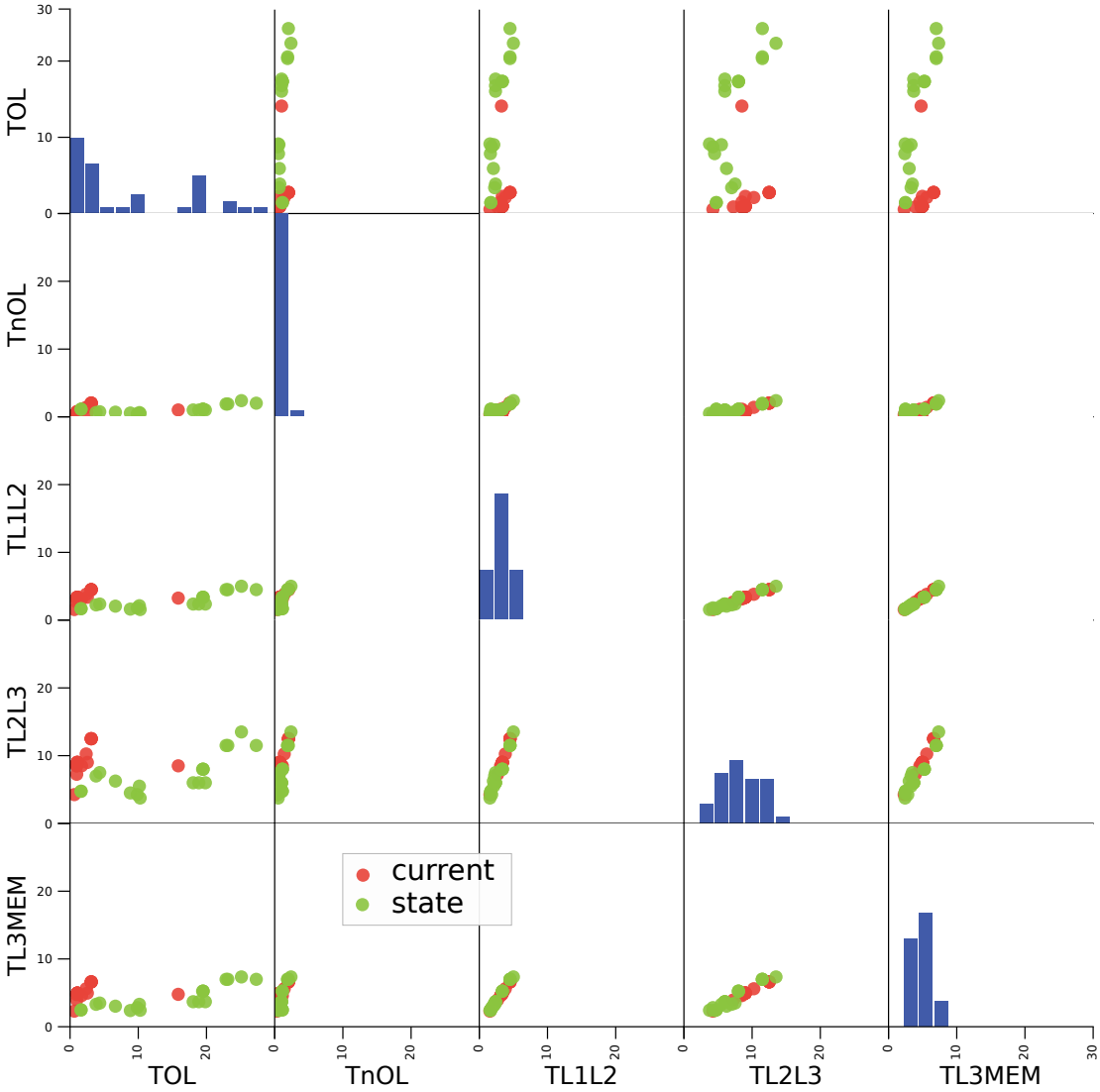

Figure C.1 – Scatter plot matrix of ECM model description of state and current kernels in morphologically detailed neurons. The x and y axis represent the ECM components measured in cycles per scalar iteration. Green dots represent state kernels, and red dots represent current kernels. The histograms in the diagonal represent the distribution of that particular ECM component over the aggregated current and state kernels. The limits on the y axis for the histograms are not shown.

kernel name	type	T_{ECM}^{L1}	T_{ECM}^{L2}	T_{ECM}^{L3}	T_{ECM}^{Mem}
K_Pst	current	2.13	4.87	13.87	18.83
Ca_HVA2	current	2.13	4.87	13.87	18.83
SK_E2	current	2.36	5.21	15.46	21.06
ProbGABAAB_EMS	current	2.75	4.75	13.25	17.84
ProbAMPANMDA_EMS	current	20	20	20	20
NaTg	current	3.11	6.52	19.02	25.63
K_Tst	current	3.56	4.62	13.62	18.58
KdShu2007	current	1.93	3.93	11.18	15.13
Nap_Et2	current	3.1	6.52	19.02	25.63
SKv3_1	current	1.93	4.27	12.77	17.55
Ih	current	1.25	2.18	6.43	8.73
Ca_LVAst	current	2.13	4.87	13.87	18.83
K_Pst	state	38	38	38	38
Ca_HVA2	state	41	41	41	41
SK_E2	state	12.9	12.9	12.9	12.9
ProbGABAAB_EMS	state	1.61	2.81	7.56	10.04
ProbAMPANMDA_EMS	state	1.61	2.81	7.56	10.04
NaTg	state	50.47	50.47	50.47	50.47
K_Tst	state	41.5	41.5	41.5	41.5
KdShu2007	state	23.38	23.38	23.38	23.38
Nap_Et2	state	57.26	57.26	57.26	57.26
SKv3_1	state	18.36	18.36	18.36	18.36
Ih	state	20.26	20.26	20.26	20.26
Ca_LVAst	state	42.4	42.4	42.4	42.4

Table C.1 – ECM runtime predictions for all cortical microcircuit kernels.

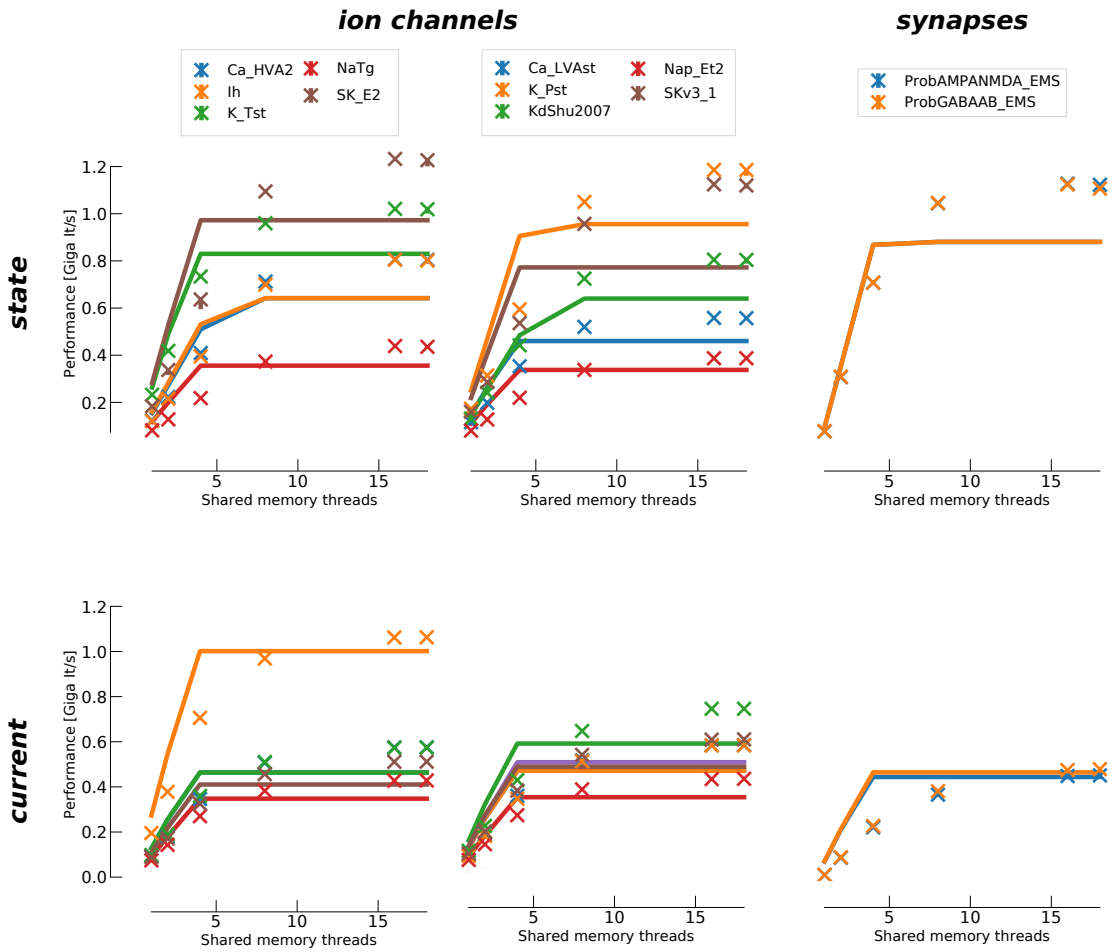

Figure C.2 – Validation of automatic ECM model of all ion channel and synapse kernels in a shared-memory scaling scenario.

D Mathematical formalism for estimating dynamic load imbalance in a spiking neural network

We prove that, under certain assumptions, the load imbalance $\Lambda = max\ \{W_p\} - min\ \{W_p\}$ arising from the different spiking workloads in a distributed spiking neural network simulation can be estimated by:

$$Pr(\Lambda = k) = \sum_{x=k}^{\infty} Pr(\underset{P}{max}\, W = x, \underset{P}{min}\, W = x - k), \tag{D.1}$$

with W i.i.d., $k \geq 0$ and the joint distribution itself given by

$$Pr(\underset{P}{max}\, W = x, \underset{P}{min}\, W = x-k) = \begin{cases} [F_W(x) - F_W(x-1)]^P & k = 0, \\ [F_W(x) - F_W(x-k-1)]^P + [F_W(x-1) - F_W(x-k)]^P \ldots & \\ \cdots - [F_W(x) - F_W(x-k)]^P - [F_W(x-1) - F_W(x-k-1)]^P & k > 0, \end{cases} \tag{D.2}$$

where F_W is the cumulative distribution function of W.

We formalise the problem of finding the dynamic load imbalance as: given a set of P i.i.d. variables $\{W_p\}$, we seek the distribution of $\Lambda = max\ \{W_p\} - min\ \{W_p\}$.

Distribution of the load imbalance The relationship (D.1) is a direct consequence of the law of total probability, which can be used to write $Pr(\Lambda = k)$ as a sum over all possible combinations of values of the maximum. But we are left with the problem of finding the joint distribution of $max\ \{W_p\}$ and $min\ \{W_p\}$.

Marginal distributions of maximum and minimum First we compute the marginal distributions of $max\ \{W_p\}$ and $min\ \{W_p\}$. For the maximum, we use the fact that if the maximum

should be smaller than a certain value x, than *all* the variables in the set should be smaller than x. This implies that

$$Pr(max\,\{W_p\} \leq x) = Pr(W_0 \leq x \cap W_1 \leq x \cap \ldots). \tag{D.3}$$

For the minimum, we use some transformations to obtain

$$Pr(min\,\{W_p\} \leq y) = 1 - Pr(W_0 > y \cap W_1 > y \cap \ldots). \tag{D.4}$$

Let F_W be the cumulative distribution function of W, using the fact that $\{W_p\}$ i.i.d, we have that (D.3) and (D.4) lead to

$$\begin{aligned} Pr(max\,\{W_p\} \leq x) &= F_W(x)^P, \\ Pr(min\,\{W_p\} \leq y) &= 1 - [1 - F_W(y)]^P. \end{aligned} \tag{D.5}$$

Joint distribution The following relationship holds

$$Pr(max\,\{W_p\} \leq x, min\,\{W_p\} \leq y) = Pr(max\,\{W_p\} \leq x) - Pr(max\,\{W_p\} \leq x, min\,\{W_p\} > y). \tag{D.6}$$

From the marginal distribution (D.5) we have that

$$Pr(max\,\{W_p\} \leq x) = F_W(x)^P. \tag{D.7}$$

To estimate the second term in (D.6) we use the fact that if the minimum should be larger that y and the maximum smaller that x, than *all* $\{W_p\}$ must respect these bounds. Thus

$$\begin{aligned} Pr(max\,\{W_p\} \leq x, min\,\{W_p\} > y) &= Pr(W_0 > y \cap W_0 \cap \leq x \cap W_1 > y \cap W_1 \cap \leq x \cap \ldots) \\ &= [F_W(x) - F_W(y)]^P. \end{aligned} \tag{D.8}$$

Since $y \geq x \implies Pr(max\,\{W_p\} \leq x, min\,\{W_p\} > y) = 0$ we can write the joint cumulative distribution function

$$Pr(max\,\{W_p\} \leq x, min\,\{W_p\} \leq y) = \begin{cases} F_W(x)^P - [F_W(x) - F_W(y)]^P, & y < x \\ F_W(x)^P, & y \geq x. \end{cases} \tag{D.9}$$

To obtain the joint probability mass function, we use the relationship

$$\begin{aligned} Pr(X = x, Y = y) &= Pr(x - 1 < X \leq x, y - 1 < Y \leq y) \\ &= F_{(X,Y)}(x, y) - F_{(X,Y)}(x - 1, y) - F_{(X,Y)}(x, y - 1) + F_{(X,Y)}(x - 1, y - 1) \end{aligned} \tag{D.10}$$

to obtain (D.2).

E Scientific papers

This appendix lists scientific contributions published or prepared during the doctoral studies.

Francesco Cremonesi and Felix Schürmann. Telling neuronal apples from oranges: analytical performance modeling of neural tissue simulations. *Neuroinformatics*, 2019. *In review*

Francesco Cremonesi, Georg Hager, Gerhard Wellein, and Felix Schürmann. Analytic performance modeling and analysis of detailed neuron simulations. *The International Journal of High Performance Computing Applications*, 2019a. *In review*

Francesco Cremonesi, Pramod Kumbhar, and Felix Schürmann. Heterogeneity of performance properties in the simulation of a cortical microcircuit. *To be submitted*, 2019b

Timothée Ewart, Francesco Cremonesi, Felix Schürmann, and Fabien Delalondre. Polynomial evaluation on superscalar architecture, applied to the elementary function e^x. *ACM Transactions on Mathematical Software (TOMS)*, 2019. *In review*

Timothée Ewart, Judit Planas, Francesco Cremonesi, Kai Langen, Felix Schürmann, and Fabien Delalondre. Neuromapp: a mini-application framework to improve neural simulators. In *International Supercomputing Conference*, pages 181–198. Springer, 2017. doi:10.1007/978-3-319-58667-0_10

Timothée Ewart, Stuart Yates, Francesco Cremonesi, Pramod Kumbhar, Felix Schürmann, and Fabien Delalondre. Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using morphologically detailed neurons. In *Proc. 6th Int. Workshop on Perfomance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015

Francesco Casalegno, Francesco Cremonesi, Stuart Yates, Michael L Hines, F Schürmann, and F Delalondre. Error analysis and quantification in neuron simulations. In *Proc. VII Europ. Cong.*

on Comput. Meth. in Appl. Sci. and Eng., pages 1366–1380, 2016. doi:10.7712/100016.1892.7366

Aleksandr Ovcharenko, Pramod S Kumbhar, Michael L Hines, Francesco Cremonesi, Timothée Ewart, Stuart Yates, Felix Schürmann, and Fabien Delalondre. Simulating morphologically detailed neuronal networks at extreme scale. In *PARCO*, pages 787–796, 2015

Bibliography

Syed Ahmed Aamir, Yannik Stradmann, Paul Müller, Christian Pehle, Andreas Hartel, Andreas Grübl, Johannes Schemmel, and Karlheinz Meier. An accelerated lif neuronal network array for a large-scale mixed-signal neuromorphic architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, (99):1–14, 2018. doi:10.1109/tcsi.2018.2840718.

Hannelore Aerts, Michael Schirner, Ben Jeurissen, Dirk Van Roost, Eric Achten, Petra Ritter, and Daniele Marinazzo. Modeling brain dynamics in brain tumor patients using the virtual brain. *eNeuro*, 5(3), 2018.

Anant Agarwal, John Hennessy, and Mark Horowitz. An analytical cache model. *ACM Transactions on Computer Systems (TOCS)*, 7(2):184–215, 1989. doi:10.1145/63404.63407.

Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Küsters, Wouter Klijn, Alexander Peyser, and Stuart Yates. Arbor—a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 274–282. IEEE, 2019a. doi:10.1109/empdp.2019.8671560.

Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Küsters, Wouter Klijn, Alexander Peyser, and Stuart Yates. Arbor—a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 274–282. IEEE, 2019b. doi:10.1109/empdp.2019.8671560.

Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model for parallel computation. *Journal of parallel and distributed computing*, 44(1):71–79, 1997. doi:10.1006/jpdc.1997.1346.

Oren Amsalem, Werner Van Geit, Eilif Muller, Henry Markram, and Idan Segev. From neuron biophysics to orientation selectivity in electrically coupled networks of neocortical l2/3 large basket cells. *Cerebral Cortex*, 26(8):3655–3668, 2016. doi:10.1093/cercor/bhw166.

Rajagopal Ananthanarayanan and Dharmendra S Modha. Anatomy of a cortical simulator. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 3. ACM, 2007. doi:10.1145/1362622.1362627.

Rajagopal Ananthanarayanan, Steven K Esser, Horst D Simon, and Dharmendra S Modha. The cat is out of the bag: cortical simulations with 10 9 neurons, 10 13 synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 63. ACM, 2009.

Anton Arkhipov, Nathan W Gouwens, Yazan N Billeh, Sergey Gratiy, Ramakrishnan Iyer, Ziqiang Wei, Zihao Xu, Reza Abbasi-Asl, Jim Berg, Michael Buice, et al. Visual physiology of the layer 4 cortical circuit in silico. *PLoS computational biology*, 14(11):e1006535, 2018. doi:10.1371/journal.pcbi.1006535.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, Berkeley University, 2006.

Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. doi:10.1145/1562764.1562783.

John Backus. Turing award lecture. *CACM Aug*, 1978. doi:10.1145/320435.320497.

David H Bailey, Robert F Lucas, and Samuel Williams. *Performance tuning of scientific applications*. CRC Press, 2010. doi:10.1201/b10509.

Rembrandt Bakker, Thomas Wachtler, and Markus Diesmann. Cocomac 2.0 and the future of tract-tracing databases. *Frontiers in neuroinformatics*, 6:30, 2012. doi:10.3389/fninf.2012.00030.

Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, (5):295–310, 2004. doi:10.1109/tse.2004.9.

Shouvik Bardhan and Daniel A Menascé. Predicting the effect of memory contention in multicore computers using analytic performance models. *IEEE Transactions on Computers*, 64 (8):2279–2292, 2014. doi:10.1109/tc.2014.2361511.

Maxim Bazhenov, Igor Timofeev, Mircea Steriade, and Terrence J Sejnowski. Model of thalamocortical slow-wave sleep oscillations and transitions to activated states. *Journal of neuroscience*, 22(19):8691–8704, 2002. doi:10.1523/jneurosci.22-19-08691.2002.

Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul A Merolla, and Kwabena Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014. doi:10.1109/jproc.2014.2313565.

Upinder S Bhalla. Multi-compartmental models of neurons. In *Computational Systems Neurobiology*, pages 193–225. Springer, 2012. doi:10.1007/978-94-007-3858-4_7.

Upinder S Bhalla. Molecular computation in neurons: a modeling perspective. *Current opinion in neurobiology*, 25:31–37, 2014a. doi:10.1016/j.conb.2013.11.006.

Upinder S Bhalla. Multiscale modeling and synaptic plasticity. In *Progress in molecular biology and translational science*, volume 123, pages 351–386. Elsevier, 2014b. doi:10.1016/b978-0-12-397897-4.00012-7.

UPINDER S Bhalla and JAMES M Bower. Exploring parameter space in detailed single neuron models: simulations of the mitral and granule cells of the olfactory bulb. *Journal of neurophysiology*, 69(6):1948–1965, 1993. doi:10.1152/jn.1993.69.6.1948.

Upinder Singh Bhalla. Synaptic input sequence discrimination on behavioral timescales mediated by reaction-diffusion chemistry in dendrites. *Elife*, 6:e25827, 2017. doi:10.7554/elife.25827.

Justas Birgiolas, Sharon Crook, Rick Gerkin, and Suzanne Dietrich. Neuroml database, 2019.

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. doi:10.1016/s0006-3495(95)80092-5.

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. doi:10.1152/jn.00686.2005.

Romain Brette and Dan FM Goodman. Vectorized algorithms for spiking neural network simulation. *Neural computation*, 23(6):1503–1535, 2011. doi:10.1162/neco_a_00123.

Romain Brette and Dan FM Goodman. Simulating spiking neural networks on gpu. *Network: Computation in Neural Systems*, 23(4):167–182, 2012. doi:10.3109/0954898x.2012.730170.

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

Maximilien B Breughe, Stijn Eyerman, and Lieven Eeckhout. Mechanistic analytical modeling of superscalar in-order processor performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):50, 2015. doi:10.1145/2678277.

Nicolas Brunel. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of computational neuroscience*, 8(3):183–208, 2000. doi:10.1016/s0925-2312(00)00179-x.

Doug Burger, Todd M Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1996.

Nicholas Cain, Ramakrishnan Iyer, Christof Koch, and Stefan Mihalas. The computational properties of a simplified cortical column model. *PLoS computational biology*, 12(9): e1005045, 2016. doi:10.1186/1471-2202-15-s1-p92.

Ana Calabrese and Sarah MN Woolley. Coding principles of the canonical cortical microcircuit in the avian brain. *Proceedings of the National Academy of Sciences*, 112(11):3517–3522, 2015. doi:10.1073/pnas.1408545112.

Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*, pages 1–12. ACM, November 2013. doi:10.1145/2503210.2503277.

Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

Nicholas T Carnevale and Michael L Hines. *The NEURON book*. Cambridge University Press, 2006. doi:10.1017/cbo9780511541612.

Alexandra Carpen-Amarie, Sascha Hunold, and Jesper Larsson Träff. On expected and observed communication performance with mpi derived datatypes. *Parallel Computing*, 69: 98–117, 2017. doi:10.1016/j.parco.2017.08.006.

Francesco Casalegno, Francesco Cremonesi, Stuart Yates, Michael L Hines, F Schürmann, and F Delalondre. Error analysis and quantification in neuron simulations. In *Proc. VII Europ. Cong. on Comput. Meth. in Appl. Sci. and Eng.*, pages 1366–1380, 2016. doi:10.7712/100016.1892.7366.

A Cassidy, Jun Sawada, P Merolla, J Arthur, R Alvarez-Icaze, Filipp Akopyan, B Jackson, and D Modha. Truenorth: A high-performance, low-power neurosynaptic processor for multisensory perception, action, and cognition. In *Proceedings of the Government Microcircuits Applications & Critical Technology Conference, Orlando, FL, USA*, pages 14–17, 2016.

Andrew S Cassidy, Rodrigo Alvarez-Icaza, Filipp Akopyan, Jun Sawada, John V Arthur, Paul A Merolla, Pallab Datta, Marc Gonzalez Tallada, Brian Taba, Alexander Andreopoulos, et al. Real-time scalable cortical computing at 46 giga-synaptic ops/watt with. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 27–38. IEEE Press, 2014.

Adrian Castelló, Manuel Dolz, Enrique Quintana-Ortí, and Jose Duato. Theoretical scalability analysis of distributed deep convolutional neural networks. In *Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019. doi:10.1109/ccgrid.2019.00068.

Lydia Y Chen, Danilo Ansaloni, Evgenia Smirni, Akira Yokokawa, and Walter Binder. Achieving application-centric performance targets via consolidation on multicores: myth or reality? In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 37–48. ACM, 2012. doi:10.1145/2287076.2287083.

WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. Loggpo: An accurate communication model for performance prediction of mpi programs. *Science in China Series F: Information Sciences*, 52(10):1785–1791, 2009. doi:10.1007/s11432-009-0161-2.

Kit Cheung, Simon R Schultz, and Wayne Luk. A large-scale spiking neural network accelerator for fpga systems. In *International Conference on Artificial Neural Networks*, pages 113–120. Springer, 2012. doi:10.1007/978-3-642-33269-2_15.

Ami Citri and Robert C Malenka. Synaptic plasticity: multiple forms, functions, and mechanisms. *Neuropsychopharmacology*, 33(1):18, 2008. doi:10.1038/sj.npp.1301559.

Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*, pages 213–224. IEEE, 2015. doi:10.1109/iiswc.2015.32.

JW Cooley and FA Dodge Jr. Digital computer solutions for excitation and propagation of the nerve impulse. *Biophysical journal*, 6(5):583–599, 1966. doi:10.1016/s0006-3495(66)86679-1.

Charles L Cox, Winfried Denk, David W Tank, and Karel Svoboda. Action potentials reliably invade axonal arbors of rat neocortical neurons. *Proceedings of the National Academy of Sciences*, 97(17):9724–9728, 2000. doi:10.1073/pnas.170278697.

Francesco Cremonesi and Felix Schürmann. Telling neuronal apples from oranges: analytical performance modeling of neural tissue simulations. *Neuroinformatics*, 2019. *In review*.

Francesco Cremonesi, Georg Hager, Gerhard Wellein, and Felix Schürmann. Analytic performance modeling and analysis of detailed neuron simulations. *The International Journal of High Performance Computing Applications*, 2019a. *In review*.

Francesco Cremonesi, Pramod Kumbhar, and Felix Schürmann. Heterogeneity of performance properties in the simulation of a cortical microcircuit. *To be submitted*, 2019b.

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993. doi:10.1145/155332.155333.

Bruno Da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. Performance modeling for fpgas: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013:7, 2013. doi:10.1155/2013/428078.

Beniaguev David, Segev Idan, and London Michael. Single cortical neurons as deep artificial neural networks. *bioRxiv*, page 613141, 2019. doi:10.1101/613141.

Hugo De Garis, Chen Shuo, Ben Goertzel, and Lian Ruiting. A world survey of artificial brain projects, part i: Large-scale brain simulations. *Neurocomputing*, 74(1-3):3–29, 2010. doi:10.1016/j.neucom.2010.08.004.

Maria del Mar Quiroga, Adam P Morris, and Bart Krekelberg. Adaptation without plasticity. *Cell reports*, 17(1):58–68, 2016. doi:10.1016/j.celrep.2016.08.089.

Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi:10.1109/jproc.1999.752522.

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

Markus Diesmann, Marc-Oliver Gewaltig, Stefan Rotter, and Ad Aertsen. State space analysis of synchronous spiking in cortical neural networks. *Neurocomputing*, 38:565–571, 2001. doi:10.1016/s0925-2312(01)00409-x.

Pradeep K Dubey, George B Adams III, and Michael J Flynn. Exploiting fine-grain concurrency analytical insights in superscalar processor design. 1991.

Egidio D'Angelo. The urgent need for a systems biology approach to neurology. *Functional neurology*, 29(4):221, 2014.

Hubert Eichner, Tobias Klug, and Alexander Borst. Neural simulations on multi-core architectures. *Frontiers in neuroinformatics*, 3:21, 2009. doi:10.3389/neuro.11.021.2009.

Stefan Eilemann, Fabien Delalondre, Jon Bernard, Judit Planas, Felix Schuermann, John Biddiscombe, Costas Bekas, Alessandro Curioni, Bernard Metzler, Peter Kaltstein, et al. Key/value-enabled flash memory for complex scientific workflows with on-line analysis and visualization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 608–617. Ieee, 2016. doi:10.1109/IPDPS.2016.23.

Gaute T Einevoll, Alain Destexhe, Markus Diesmann, Sonja Grün, Viktor Jirsa, Marc de Kamps, Michele Migliore, Torbjørn V Ness, Hans E Plesser, and Felix Schürmann. The scientific case for brain simulations. *Neuron*, 102(4):735–744, 2019. doi:10.1016/j.neuron.2019.03.027.

Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT press, 2004. doi:10.1109/tnn.2004.826381.

Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *science*, 338(6111): 1202–1205, 2012. doi:10.1126/science.1225266.

Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012. doi:10.1145/2024723.2000108.

Timothée Ewart, Stuart Yates, Francesco Cremonesi, Pramod Kumbhar, Felix Schürmann, and Fabien Delalondre. Performance evaluation of the IBM POWER8 architecture to support computational neuroscientific application using morphologically detailed neurons. In *Proc. 6th Int. Workshop on Perfomance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015.

Timothée Ewart, Judit Planas, Francesco Cremonesi, Kai Langen, Felix Schürmann, and Fabien Delalondre. Neuromapp: a mini-application framework to improve neural simulators. In *International Supercomputing Conference*, pages 181–198. Springer, 2017. doi:10.1007/978-3-319-58667-0_10.

Timothée Ewart, Francesco Cremonesi, Felix Schürmann, and Fabien Delalondre. Polynomial evaluation on superscalar architecture, applied to the elementary function e^x. *ACM Transactions on Mathematical Software (TOMS)*, 2019. *In review*.

Egidio Falotico, Lorenzo Vannucci, Alessandro Ambrosano, Ugo Albanese, Stefan Ulbrich, Juan Camilo Vasquez Tieck, Georg Hinkel, Jacques Kaiser, Igor Peric, Oliver Denninger, et al. Connecting artificial brains to robots in a comprehensive simulation framework: the neurorobotics platform. *Frontiers in neurorobotics*, 11:2, 2017. doi:10.3389/fnbot.2017.00002.

Babak Falsafi and David A Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(1):104–130, 1997. doi:10.1145/244804.244808.

Xue Fan and Henry Markram. A brief history of simulation neuroscience. *Frontiers in neuroinformatics*, 13:32, 2019.

Dirk Feldmeyer, Michael Brecht, Fritjof Helmchen, Carl CH Petersen, James FA Poulet, Jochen F Staiger, Heiko J Luhmann, and Cornelius Schwarz. Barrel cortex function. *Progress in neurobiology*, 103:3–27, 2013.

Carlos Fernandez Musoles, Daniel Coca, and Paul Richmond. Communication sparsity in distributed spiking neural network simulations to improve scalability. *Frontiers in neuroinformatics*, 13:19, 2019. doi:10.3389/fninf.2019.00019.

Andreas K Fidjeland, Etienne B Roesch, Murray P Shanahan, and Wayne Luk. Nemo: a platform for neural modelling of spiking neurons using gpus. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 137–144. IEEE, 2009. doi:10.1109/asap.2009.24.

Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961. doi:10.1016/s0006-3495(61)86902-6.

Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. technical university of denmark, 2017.

Csaba Földy, Jonas Dyhrfjeld-Johnsen, and Ivan Soltesz. Structure of cortical microcircuit theory. *The Journal of physiology*, 562(1):47–54, 2005. doi:10.1113/jphysiol.2004.076448.

Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978. doi:10.1145/800133.804339.

Geoffrey C Fox, Shantenu Jha, Judy Qiu, and Andre Luckow. Towards an understanding of facets and exemplars of big data applications. In *Proceedings of the 20 Years of Beowulf Workshop on Honor of Thomas Sterling's 65th Birthday*, pages 7–16. ACM, 2014. doi:10.1145/2737909.2737912.

Paul J. Fox. Massively parallel neural computation. Technical Report 830, 2013.

Richard Frackowiak and Henry Markram. The future of human cerebral cartography: a novel approach. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 370(1668): 20140171, 2015. doi:10.1098/rstb.2014.0171.

Erik Fransén, Angel A Alonso, and Michael E Hasselmo. Simulations of the role of the muscarinic-activated calcium-sensitive nonspecific cation currentincm in entorhinal neuronal activity during delayed matching tasks. *Journal of Neuroscience*, 22(3):1081–1097, 2002. doi:10.1523/jneurosci.22-03-01081.2002.

Nicolas Frémaux and Wulfram Gerstner. Neuromodulated spike-timing-dependent plasticity, and theory of three-factor learning rules. *Frontiers in neural circuits*, 9:85, 2016. doi:10.3389/fncir.2015.00085.

Thomas R Furlani, Matthew D Jones, Steven M Gallo, Andrew E Bruno, Charng-Da Lu, Amin Ghadersohi, Ryan J Gentner, Abani Patra, Robert L DeLeon, Gregor von Laszewski, et al. Performance metrics and auditing framework using application kernels for high-performance computer systems. *Concurrency and Computation: Practice and Experience*, 25(7):918–931, 2013. doi:10.1002/cpe.2871.

Irene S Gabashvili, Bernd HA Sokolowski, Cynthia C Morton, and Anne BS Giersch. Ion channel gene expression in the inner ear. *Journal of the Association for Research in Otolaryngology*, 8 (3):305–328, 2007.

Michael R Garey and David S Johnson. "strong"np-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978. doi:10.1145/322077.322090.

Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2 (4):1430, 2007.

M.O. Gewaltig. aeif_cond_alpha, 2015. https://github.com/nest/nest-simulator/blob/c2d265019e60daa06338f54850464394e618e457/models/aeif_cond_alpha.h.

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. doi:10.1371/journal.pcbi.1000815.

Nigel H Goddard and Greg Hood. Parallel genesis for large-scale modeling. In *Computational Neuroscience*, pages 911–917. Springer, 1997. doi:10.1007/978-1-4757-9800-5_141.

Ronald L Graham. Bounds on multiprocessor timing anomalies. *SIAM J. Appl. Math.*, 17: 263–269, 1969. doi:10.1137/0117039.

Sergey L Gratiy, Yazan N Billeh, Kael Dai, Catalin Mitelut, David Feng, Nathan W Gouwens, Nicholas Cain, Christof Koch, Costas A Anastassiou, and Anton Arkhipov. Bionet: A python interface to neuron for modeling large-scale networks. *PloS one*, 13(8):e0201630, 2018. doi:10.1371/journal.pone.0201630.

Michael Graupner and Nicolas Brunel. Stdp in a bistable synapse model based on camkii and associated signaling pathways. *PLoS computational biology*, 3(11):e221, 2007. doi:10.1371/journal.pcbi.0030221.eor.

Thomas Gruber et al. LIKWID: A multicore performance tool suite, 2018. URL http://tiny.cc/LIKWID.

Espen Hagen, David Dahmen, Maria L Stavrinou, Henrik Lindén, Tom Tetzlaff, Sacha J van Albada, Sonja Grün, Markus Diesmann, and Gaute T Einevoll. Hybrid scheme for modeling local field potentials from point-neuron networks. *Cerebral Cortex*, pages 1–36, 2016. doi:10.1186/1471-2202-16-s1-p67.

Espen Hagen, Solveig Næss, Torbjørn V Ness, and Gaute T Einevoll. Multimodal modeling of neural network activity: Computing lfp, ecog, eeg, and meg signals with lfpy 2.0. *Frontiers in neuroinformatics*, 12, 2018. doi:10.3389/fninf.2018.00092.

Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010. doi:10.1201/ebk1439811924.

Georg Hager and Gerhard Wellein. The execution-cache-memory model. Seminar on efficient numerical simulation on multi- and manycore processors, 2016. URL https://moodle.rrze.uni-erlangen.de/pluginfile.php/12401/mod_resource/content/1/07_06_2016_ECM_Model.pdf.

Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, 2016. doi:10.1002/cpe.3180.

Georg Hager, Jan Eitzinger, Julian Hornich, Francesco Cremonesi, Christie L Alappat, Thomas Röhl, and Gerhard Wellein. Applying the execution-cache-memory model: Current state of practice. poster at Supercomputing 2018, 2018.

Jan Hahne, Moritz Helias, Susanne Kunkel, Jun Igarashi, Matthias Bolten, Andreas Frommer, and Markus Diesmann. A unified framework for spiking and gap-junction interactions in distributed neuronal network simulations. *Frontiers in neuroinformatics*, 9:22, 2015. doi:10.3389/fninf.2015.00022.

Jan Hahne, David Dahmen, Jannis Schuecker, Andreas Frommer, Matthias Bolten, Moritz Helias, and Markus Diesmann. Integration of continuous-time dynamics in a spiking neural network simulator. *Frontiers in neuroinformatics*, 11:34, 2017. doi:10.3389/fninf.2017.00034.

Julian Hammer, Georg Hager, and Gerhard Wellein. Performance modeling and engineering using kerncraft. In *Conference on Supercomputing. ACM*, pages 207–216, 2015.

Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: a tool for analytic performance modeling of loop kernels. In *Tools for High Performance Computing 2016*, pages 1–22. Springer, 2017. doi:10.1007/978-3-319-56702-0_1.

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011. doi:10.1109/mm.2011.77.

Günter Haring and Johannes Lüthi. Analysis of computer and communication systems with uncertainties and variabilities in workload. 1996.

Arnold Hayer and Upinder S Bhalla. Molecular switches at the synapse emerge from receptor and kinase traffic. *PLoS computational biology*, 1(2):e20, 2005. doi:10.1371/journal.pcbi.0010020.

Moritz Helias, Susanne Kunkel, Gen Masumoto, Jun Igarashi, Jochen Martin Eppler, Shin Ishii, Tomoki Fukai, Abigail Morrison, and Markus Diesmann. Supercomputers ready for use as discovery machines for neuroscience. *Frontiers in neuroinformatics*, 6:26, 2012. doi:10.3389/fninf.2012.00026.

John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. doi:10.1016/0026-2692(93)90111-q.

Herodotos Herodotou. Hadoop performance models. Technical report, 2011.

Sean Hill and Giulio Tononi. Modeling sleep and wakefulness in the thalamocortical system. *Journal of neurophysiology*, 93(3):1671–1698, 2005. doi:10.1152/jn.00915.2004.

Michael Hines. Efficient computation of branched nerve equations. *International journal of bio-medical computing*, 15(1):69–76, 1984. doi:10.1016/0020-7101(84)90008-4.

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. doi:10.3389/fncom.2011.00049.

Michael L Hines and Nicholas T Carnevale. Expanding neuron's repertoire of mechanisms with nmodl. *Neural computation*, 12(5):995–1007, 2000. doi:10.1162/089976600300015475.

Michael L Hines and Nicholas T Carnevale. Discrete event simulation in the neuron environment. *Neurocomputing*, 58:1117–1122, 2004. doi:10.1016/s0925-2312(04)00180-8.

Michael L Hines, Thomas Morse, Michele Migliore, Nicholas T Carnevale, and Gordon M Shepherd. Modeldb: a database to support computational neuroscience. *Journal of computational neuroscience*, 17(1):7–11, 2004. doi:10.1023/b:jcns.0000023869.22017.2e.

Michael L Hines, Henry Markram, and Felix Schürmann. Fully implicit parallel simulation of single neurons. *Journal of computational neuroscience*, 25(3):439–448, 2008. doi:10.1186/1471-2202-8-s2-p6.

Arend Hintze, Jeffrey A Edlund, Randal S Olson, David B Knoester, Jory Schossau, Larissa Albantakis, Ali Tehrani-Saleh, Peter Kvam, Leigh Sheneman, Heather Goldsby, et al. Markov brains: A technical introduction. *arXiv preprint arXiv:1709.05601*, 2017.

Rebecca D Hodge, Trygve E Bakken, Jeremy A Miller, Kimberly A Smith, Eliza R Barkan, Lucas T Graybuck, Jennie L Close, Brian Long, Osnat Penn, Zizhen Yao, et al. Conserved cell types with divergent features between human and mouse cortex. *BioRxiv*, page 384826, 2018. doi:10.1101/384826.

Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4): 500–544, 1952. doi:10.1016/s0092-8240(05)80004-7.

Torsten Hoefler, Andre Lichei, and Wolfgang Rehm. Low-overhead loggp parameter assessment for modern interconnection networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007a. doi:10.1109/ipdps.2007.370593.

Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. Netgauge: A network performance measurement framework. In *International Conference on High Performance Computing and Communications*, pages 659–671. Springer, 2007b. doi:10.1007/978-3-540-75444-2_62.

Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Loggp in theory and practice–an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations. *Simulation Modelling Practice and Theory*, 17(9):1511–1521, 2009. doi:10.1016/j.simpat.2009.06.007.

Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Loggopsim: simulating large-scale applications in the loggops model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, 2010.

Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, page 6. ACM, 2011.

Johannes Hofmann, Jan Eitzinger, and Dietmar Fey. Execution-cache-memory performance model: Introduction and validation. *arXiv preprint arXiv:1509.03118*, 2015.

Johannes Hofmann, Georg Hager, Gerhard Wellein, and Dietmar Fey. An analysis of core- and chip-level architectural features in four generations of intel server processors. In *International Supercomputing Conference*, pages 294–314. Springer, 2017. doi:10.1007/978-3-319-58667-0_16.

Johannes Hofmann, Georg Hager, and Dietmar Fey. On the accuracy and usefulness of analytic energy models for contemporary multicore processors. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *International Conference on High Performance Computing*, pages 22–43, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-92040-5_2.

Johannes Hofmann, Christie L Alappat, Georg Hager, Dietmar Fey, and Gerhard Wellein. Bridging the architecture gap: Abstracting performance-relevant properties of modern server processors. *arXiv preprint arXiv:1907.00048*, 2019.

Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72, 2007. doi:10.1109/mm.2007.56.

Mark D Humphries and Kevin Gurney. Network 'small-world-ness': a quantitative method for determining canonical network equivalence. *PloS one*, 3(4):e0002051, 2008. doi:10.1371/journal.pone.0002051.

Elisabetta Iavarone, Jane Yi, Ying Shi, Bas-Jan Zandt, Christian O'reilly, Werner Van Geit, Christian Rössert, Henry Markram, and Sean L Lewis Hill. Experimentally-constrained biophysical models of tonic and burst firing modes in thalamocortical neurons. *BioRxiv*, page 512269, 2019. doi:10.1101/512269.

Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2013. doi:10.1109/l-ca.2013.6.

Giacomo Indiveri, Bernabé Linares-Barranco, Tara Julia Hamilton, André Van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Häfliger, Sylvie Renaud, et al. Neuromorphic silicon neuron circuits. *Frontiers in neuroscience*, 5:73, 2011. doi:10.3389/fnins.2011.00073.

Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. Loggps: a parallel computational model for synchronization analysis. In *ACM SIGPLAN Notices*, volume 36, pages 133–142. ACM, 2001.

Intel. Intel Architecture Code Analyzer, 11 2017. URL https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005. doi:10.1007/11549468_24.

Tammo Ippen, Jochen M Eppler, Hans E Plesser, and Markus Diesmann. Constructing neuronal network models in massively parallel environments. *Frontiers in neuroinformatics*, 11:30, 2017. doi:10.3389/fninf.2017.00030.

Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003. doi:10.1109/tnn.2003.820440.

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. doi:10.1073/pnas.0712231105.

Eugene M Izhikevich, Joseph A Gally, and Gerald M Edelman. Spike-timing dynamics of neuronal groups. *Cerebral cortex*, 14(8):933–944, 2004. doi:10.1093/cercor/bhh053.

Ben H Jansen and Vincent G Rit. Electroencephalogram and visual evoked potential generation in a mathematical model of coupled cortical columns. *Biological cybernetics*, 73(4):357–366, 1995. doi:10.1007/s004220050191.

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

A Kagi, James R Goodman, and Doug Burger. Memory bandwidth limitations of future microprocessors. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 78–78. IEEE, 1996. doi:10.1145/232974.232983.

Erich L Kaltofen. The "seven dwarfs" of symbolic computation. In *Numerical and symbolic scientific computing*, pages 95–104. Springer, 2012. doi:10.1007/978-3-7091-0794-2_5.

Lida Kanari, Srikanth Ramaswamy, Ying Shi, Sebastien Morand, Julie Meystre, Rodrigo Perin, Marwan Abdellah, Yun Wang, Kathryn Hess, and Henry Markram. Objective morphological classification of neocortical pyramidal cells. *Cerebral Cortex*, 29(4):1719–1735, 2019. doi:10.1093/cercor/bhy339.

Eric R Kandel, Henry Markram, Paul M Matthews, Rafael Yuste, and Christof Koch. Neuroscience thinks big (and collaboratively). *Nature Reviews Neuroscience*, 14(9):659, 2013. doi:10.1038/nrn3578.

Richard M Karp and Vijaya Ramachandran. *Parallel algorithms for shared-memory machines*. Elsevier, 1990. doi:10.1016/b978-0-444-88071-0.50022-9.

Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 39–39. IEEE, 2001. doi:10.1145/582034.582071.

David J Ketchen and Christopher L Shook. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal*, 17(6): 441–458, 1996. doi:10.1002/(sici)1097-0266(199606)17:6<441::aid-smj819>3.0.co;2-g.

James Courtney Knight and Thomas Nowotny. Gpus outperform current hpc and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. *Frontiers in neuroscience*, 12:941, 2018. doi:10.3389/fnins.2018.00941.

Jeanette Hellgren Kotaleski and Kim T Blackwell. Computational neuroscience: Modeling the systems biology of synaptic plasticity. *Nature reviews. Neuroscience*, 11(4):239, 2010. doi:10.1038/nrn2807.

James Kozloski. Closed-loop brain model of neocortical information-based exchange. *Frontiers in neuroanatomy*, 10:3, 2016. doi:10.3389/fnana.2016.00003.

James Kozloski and Guillermo A Cecchi. A theory of loop formation and elimination by spike timing-dependent plasticity. *Frontiers in neural circuits*, 4:7, 2010. doi:10.3389/fncir.2010.00007.

James Kozloski and John Wagner. An ultrascalable solution to large-scale neural tissue simulation. *Frontiers in neuroinformatics*, 5:15, 2011. doi:10.3389/fninf.2011.00015.

Konstantinos Krommydas, Wu-chun Feng, Christos D Antonopoulos, and Nikolaos Bellas. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, 85(3):373–392, 2016. doi:10.1007/s11265-015-1051-z.

Pramod Kumbhar and Michael Hines. CoreNeuron: morphologically detailed neuron simulations. In *Nvidia GPU Technology Conference*, 2016.

Pramod Kumbhar, Michael Hines, Aleksandr Ovcharenko, Damian A Mallon, James King, Florentino Sainz, Felix Schürmann, and Fabien Delalondre. Leveraging a cluster-booster architecture for brain-scale simulations. In *International Conference on High Performance Computing*, pages 363–380. Springer, 2016. doi:10.1007/978-3-319-41321-1_19.

Pramod Kumbhar, Omar Awile, Liam Keegan, Jorge Blanco Alonso, James King, Michael Hines, and Felix Schürmann. An optimizing multi-platform source-to-source compiler framework for the neuron modeling language. *arXiv preprint arXiv:1905.02241*, 2019a.

Pramod Kumbhar, Michael Hines, Jeremy Fouriaux, Aleksandr Ovcharenko, James King, Fabien Delalondre, and Felix Schürmann. Coreneuron: An optimized compute engine for the neuron simulator. *arXiv preprint arXiv:1901.10975*, 2019b.

Pramod S Kumbhar, Subhashini Sivagnanam, Kenneth Yoshimoto, Michael Hines, Ted Carnevale, and Amit Majumdar. Performance analysis of computational neuroscience software neuron on knights corner many core processors. In *Workshop on Software Challenges to Exascale Computing*, pages 67–76. Springer, 2018. doi:10.1007/978-981-13-7729-7_5.

Susanne Kunkel and Wolfram Schenck. The nest dry-run mode: efficient dynamic analysis of neuronal network simulation code. *Frontiers in neuroinformatics*, 11:40, 2017. doi:10.3389/fninf.2017.00040.

Susanne Kunkel, Maximilian Schmidt, Jochen M Eppler, Hans E Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann, et al. Spiking network simulation code for petascale computers. *Frontiers in neuroinformatics*, 8:78, 2014. doi:10.3389/fninf.2014.00078.

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.

Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for Intel and AMD microarchitectures. *CoRR*, abs/1809.00912, 2018. URL http://arxiv.org/abs/1809.00912. accepted for publication.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

Benjamin C Lee, David M Brooks, Bronis R de Supinski, Martin Schulz, Karan Singh, and Sally A McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007. doi:10.1145/1229428.1229479.

David Levinthal. Cycle accounting analysis on intel core 2 processors. Technical report, 2014. Whitepaper.

KA Lindsay, JR Rosenberg, and G Tucker. From maxwell's equations to the cable equation and beyond. *Progress in Biophysics and Molecular Biology*, 85(1):71–116, 2004. doi:10.1016/j.pbiomolbio.2003.08.001.

William W Lytton and Michael L Hines. Independent variable time-step integration of individual neurons for network simulations. *Neural computation*, 17(4):903–921, 2005. doi:10.1162/0899766053429453.

Ronald MacGregor. *Neural and brain modeling*. Elsevier, 1987. doi:10.1016/B978-0-12-464260-7.X5001-9.

David MacNeil and Chris Eliasmith. Fine-tuning and the stability of recurrent neural networks. *PloS one*, 6(9):e22885, 2011. doi:10.1371/journal.pone.0022885.

Bruno Magalhães and Felix Schürmann. Fully-asynchronous cache-efficient simulation of detailed neural networks. 2019. doi:10.1007/978-3-030-22744-9_33.

Bruno Magalhaes, Michael Hines, Thomas Sterling, and Felix Schürmann. Asynchronous branch-parallel simulation of detailed neuron models (under review). *Frontiers in Neuroinformatics*, 2019a. doi:10.3389/fninf.2019.00054.

Bruno Magalhaes, Michael Hines, Thomas Sterling, and Felix Schürmann. Exploiting flow graph of system of odes to accelerate the simulation of biologically-detailed neural networks. In *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019b.

Bruno Magalhaes, Michael Hines, Thomas Sterling, and Felix Schürmann. Fully-asynchronous fully-implicit variable-order variable-timestep simulation of neural networks (under review). In *Proceedings of the 2019 ACM/IEEE conference on Supercomputing*. ACM, 2019c.

Liam P Maguire, T Martin McGinnity, Brendan Glackin, Arfan Ghani, Ammar Belatreche, and Jim Harkin. Challenges for large-scale implementations of spiking neural networks on fpgas. *Neurocomputing*, 71(1-3):13–29, 2007. doi:10.1016/j.neucom.2006.11.029.

Hyacinthe Nzigou Mamadou, Takeshi Nanri, and Kazuaki Murakami. Collective communication costs analysis over gigabit ethernet and infiniband. In *International Conference on High-Performance Computing*, pages 547–559. Springer, 2006. doi:10.1007/11945918_52.

Gary Marcus, Adam Marblestone, and Thomas Dean. The atoms of neural computation. *Science*, 346(6209):551–552, 2014. doi:10.1126/science.1261661.

Valentin Markounikau, Christian Igel, Amiram Grinvald, and Dirk Jancke. A dynamic neural field model of mesoscopic cortical activity captured with voltage-sensitive dye imaging. *PLoS computational biology*, 6(9):e1000919, 2010. doi:10.1371/journal.pcbi.1000919.

Henry Markram. The blue brain project. *Nature Reviews Neuroscience*, 7(2):153, 2006. doi:10.3389/conf.neuro.11.2008.01.143.

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

Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. Spike-timing-dependent plasticity: a comprehensive overview. *Frontiers in synaptic neuroscience*, 4:2, 2012. doi:10.3389/fnsyn.2012.00002.

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. doi:10.1016/j.cell.2015.09.029.

Diego Rodriguez Martinez, Jose Carlos Cabaleiro, Tomas Fernandez Pena, Francisco Fernandez Rivera, and Vicente Blanco Perez. Performance modeling of mpi applications using model selection techniques. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 95–102. IEEE, 2010. doi:10.1109/pdp.2010.78.

Michael V Mascagni, Arthur S Sherman, et al. Numerical methods for neuronal modeling. *Methods in neuronal modeling*, 2, 1989.

John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. doi:10.1126/science.1254642.

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. doi:10.1007/s10827-006-7949-5.

Dharmendra S Modha, Rajagopal Ananthanarayanan, Steven K Esser, Anthony Ndirango, Anthony J Sherbondy, and Raghavendra Singh. Cognitive computing. *Communications of the ACM*, 54(8):62–71, 2011. doi:10.4018/ijssci.2018010101.

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.

Abigail Morrison, Carsten Mehring, Theo Geisel, AD Aertsen, and Markus Diesmann. Advancing the boundaries of high-connectivity network simulation with distributed computing. *Neural computation*, 17(8):1776–1801, 2005. doi:10.1162/0899766054026648.

Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, 2016. doi:10.1109/tpds.2016.2543725.

A Mundy. *Real time Spaun on SpiNNaker*. PhD thesis, PhD thesis, University of Manchester, 2016.

Andrew Mundy, James Knight, Terrence C Stewart, and Steve Furber. An efficient spinnaker implementation of the neural engineering framework. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015. doi:10.1109/ijcnn.2015.7280390.

Manjusha Nair, Shan Surya, Revathy S Kumar, Bipin Nair, and Shyam Diwakar. Efficient simulations of spiking neurons on parallel and distributed platforms: Towards large-scale modeling in computational neuroscience. In *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*, pages 262–267. IEEE, 2015. doi:10.1109/raics.2015.7488425.

Rajeevan T Narayanan, Daniel Udvary, and Marcel Oberlaender. Cell type-specific structural organization of the six layers in rat barrel cortex. *Frontiers in neuroanatomy*, 11:91, 2017. doi:10.3389/fnana.2017.00091.

Surya Narayanan and André Seznec. Sequential and parallel code sections are different: they may require different processors. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, pages 13–18. ACM, 2015.

Richard Naud, Nicolas Marcille, Claudia Clopath, and Wulfram Gerstner. Firing patterns in the adaptive exponential integrate-and-fire model. *Biological cybernetics*, 99(4-5):335, 2008. doi:10.1007/s00422-008-0264-7.

Javier Navaridas, Mikel Luj'n, Luis A Plana, Jose Miguel-Alonso, and Steve B Furber. Analytical assessment of the suitability of multicast communications for the spinnaker neuromimetic system. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 1–8. IEEE, 2012. doi:10.1109/hpcc.2012.11.

James R Nechvatal. On the performance evaluation and analytical modeling of shared-memory computers. Technical report, 1988.

Mark E Nelson and James M Bower. Brain maps and parallel computers. *Trends in neurosciences*, 13(10):403–408, 1990. doi:10.1016/0166-2236(90)90119-u.

Taylor H. Newton, Marwan Abdellah, Eilif Muller, Felix Schürmann, and Henry Markram. In silico voltage sensitive dye imaging in a digital reconstruction of somatosensory cortex. In *Society For Neuroscience SFN*, 2016.

Max Nolte, Michael W Reimann, James G King, Henry Markram, and Eilif B Muller. Cortical reliability amid noise and chaos. *bioRxiv*, page 304121, 2018. doi:10.1101/304121.

Cedric Nugteren and Henk Corporaal. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th conference on Computing Frontiers*, pages 203–212. ACM, 2012.

Aleksandr Ovcharenko, Pramod S Kumbhar, Michael L Hines, Francesco Cremonesi, Timothée Ewart, Stuart Yates, Felix Schürmann, and Fabien Delalondre. Simulating morphologically detailed neuronal networks at extreme scale. In *PARCO*, pages 787–796, 2015.

Vijay S Pai, Parthasarathy Ranganathan, and Sarita V Adve. Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors. *IEEE Technical Committee on Computer Architecture Newsletter*, 1997.

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. doi:10.1109/jssc.2013.2259038.

I Parnas and I Segev. A mathematical model for conduction of action potentials along bifurcating axons. *The Journal of physiology*, 295(1):323–343, 1979. doi:10.1113/jphysiol.1979.sp012971.

William Penny. Bayesian models of brain and behaviour. *ISRN Biomathematics*, 2012, 2012.

Dorina C Petriu, Shikharesh Majumdar, Jinping Lin, and Curtis Hrischuk. Analytic performance estimation of client-server systems with multi-threaded clients. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 96–100. IEEE, 1994. doi:10.1109/mascot.1994.284440.

Alexander Peyser and Wolfram Schenck. The nest neuronal network simulator: Performance optimization techniques for high performance computing platforms. In *Society for Neuroscience Annual Meeting*, number FZJ-2015-06261. Jülich Supercomputing Center, 2015.

Thomas Pfeil, Tobias C Potjans, Sven Schrader, Wiebke Potjans, Johannes Schemmel, Markus Diesmann, and Karlheinz Meier. Is a 4-bit synaptic weight resolution enough?–constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in neuroscience*, 6:90, 2012. doi:10.3389/fnins.2012.00090.

Thomas Pfeil, Andreas Grübl, Sebastian Jeltsch, Eric Müller, Paul Müller, Mihai A Petrovici, Michael Schmuker, Daniel Brüderle, Johannes Schemmel, and Karlheinz Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in neuroscience*, 7:11, 2013. doi:10.3389/fnins.2013.00011.

Stephen C Phillips, Vegard Engen, and Juri Papay. Snow white clouds and the seven dwarfs. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 738–745. IEEE, 2011. doi:10.1109/cloudcom.2011.114.

Lingyan Ping, Duc M Duong, Luming Yin, Marla Gearing, James J Lah, Allan I Levey, and Nicholas T Seyfried. Global quantitative analysis of the human brain proteome in alzheimer's and parkinson's disease. *Scientific data*, 5:180036, 2018. doi:10.1038/sdata.2018.36.

Judit Planas, Fabien Delalondre, and Felix Schürmann. Accelerating data analysis in simulation neuroscience with big data technologies. In *International Conference on Computational Science*, pages 363–377. Springer, 2018. doi:10.1007/978-3-319-93698-7_28.

Hans E Plesser, Jochen M Eppler, Abigail Morrison, Markus Diesmann, and Marc-Oliver Gewaltig. Efficient parallel simulation of large-scale neuronal networks on clusters of multiprocessor computers. In *European Conference on Parallel Processing*, pages 672–681. Springer, 2007. doi:10.1007/978-3-540-74466-5_71.

Sabri Pllana and Thomas Fahringer. Performance prophet: A performance modeling and prediction tool for parallel and distributed programs. In *2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 509–516. IEEE, 2005. doi:10.1109/icppw.2005.72.

Sabri Pllana, Ivona Brandic, and Siegfried Benkner. Performance modeling and prediction of parallel and distributed computing systems: A survey of the state of the art. In *First International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'07)*, pages 279–284. IEEE, 2007. doi:10.1109/cisis.2007.49.

Dimitri Plotnikov, Bernhard Rumpe, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, and Abgail Morrison. Nestml: a modeling language for spiking neurons. *arXiv preprint arXiv:1606.02882*, 2016.

Panayiota Poirazi, Terrence Brannon, and Bartlett W Mel. Arithmetic of subthreshold synaptic summation in a model ca1 pyramidal cell. *Neuron*, 37(6):977–987, 2003. doi:10.1016/s0896-6273(03)00148-x.

Tobias C Potjans and Markus Diesmann. The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cerebral cortex*, 24(3):785–806, 2012. doi:10.1093/cercor/bhs358.

Christian Pozzorini, Richard Naud, Skander Mensi, and Wulfram Gerstner. Temporal whitening by power-law adaptation in neocortical neurons. *Nature neuroscience*, 16(7):942, 2013. doi:10.1038/nn.3431.

Christian Pozzorini, Skander Mensi, Olivier Hagens, Richard Naud, Christof Koch, and Wulfram Gerstner. Automated high-throughput characterization of single neurons by means of simplified spiking models. *PLoS computational biology*, 11(6):e1004275, 2015. doi:10.1371/journal.pcbi.1004275.

Ivan Raikov, Robert Cannon, Robert Clewley, Hugo Cornelis, Andrew Davison, Erik De Schutter, Mikael Djurfeldt, Padraig Gleeson, Anatoli Gorchetchnikov, Hans Ekkehard Plesser, et al. Nineml: the network interchange for ne uroscience modeling language. *BMC neuroscience*, 12(S1):P330, 2011. doi:10.1186/1471-2202-12-s1-p330.

Wilfrid Rall. Electrophysiology of a dendritic neuron model. *Biophysical journal*, 2(2):145–167, 1962. doi:10.1016/s0006-3495(62)86953-7.

Srikanth Ramaswamy, Jean-Denis Courcol, Marwan Abdellah, Stanislaw R Adaszewski, Nicolas Antille, Selim Arsever, Guy Atenekeng, Ahmet Bilgili, Yury Brukau, and Athanassia et al. Chalimourda. The neocortical microcircuit collaboration portal: a resource for rat somatosensory cortex. *Frontiers in neural circuits*, 9:44, 2015. doi:10.3389/fncir.2015.00044. URL https://bbp.epfl.ch/nmc-portal/downloads.

S Ramón y Cajal. Histologie du systeme nerveux de l'homme et des vertebres. *Maloine, Paris*, 774838, 1909. doi:10.1097/00005072-199809000-00011.

Michael Reimann. *The In-Silico Neocortical Microcircuit: From Structure to Dynamics*. PhD thesis, 2014.

Michael W Reimann, Costas A Anastassiou, Rodrigo Perin, Sean L Hill, Henry Markram, and Christof Koch. A biophysically detailed model of neocortical local field potentials predicts the critical role of active membrane currents. *Neuron*, 79(2):375–390, 2013. doi:10.1016/j.neuron.2013.05.023.

Michael W Reimann, James G King, Eilif B Muller, Srikanth Ramaswamy, and Henry Markram. An algorithm to predict the connectome of neural microcircuits. *Frontiers in computational neuroscience*, 9:28, 2015. doi:10.3389/fncom.2015.00120.

Michael W Reimann, Michael Gevaert, Ying Shi, Huanxiang Lu, Henry Markram, and Eilif Muller. A null model of the mouse whole-neocortex micro-connectome. *BioRxiv*, page 548735, 2019. doi:10.1101/548735.

Michael J Rempe and David L Chopp. A predictor-corrector algorithm for reaction-diffusion equations associated with neural activity on branched structures. *SIAM Journal on Scientific Computing*, 28(6):2139–2161, 2006. doi:10.1137/050643210.

Sylvie Renaud-Le Masson, Gwendal Le Masson, Ludovic Alvado, S Saıghi, and Jean Tomas. A neural simulation system based on biologically realistic electronic neurons. *Information Sciences*, 161(1-2):57–69, 2004. doi:10.1016/j.ins.2003.03.007.

César Rennó-Costa, Ana Cláudia Costa da Silva, Wilfredo Blanco, and Sidarta Ribeiro. Computational models of memory consolidation and long-term synaptic plasticity during sleep. *Neurobiology of learning and memory*, 160:32–47, 2019. doi:10.1016/j.nlm.2018.10.003.

Vinicius Facco Rodrigues, Gustavo Rostirolla, Rodrigo da Rosa Righi, and Cristiano André da Costa. Analyzing performance and efficiency of hpc applications in the cloud. *WSPPD*, 2015.

Christian Rössert, Christian Pozzorini, Giuseppe Chindemi, Andrew P Davison, Csaba Eroe, James King, Taylor H Newton, Max Nolte, Srikanth Ramaswamy, Michael W Reimann, et al. Automated point-neuron simplification of data-driven microcircuit models. *arXiv preprint arXiv:1604.00087*, 2016.

Arnd Roth and Mark CW van Rossum. Modeling synapses. *Computational modeling methods for neuroscientists*, 6:139–160, 2009. doi:10.1007/978-1-4614-6675-8_240.

Stefan Rotter and Markus Diesmann. Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological cybernetics*, 81(5-6):381–402, 1999. doi:10.1007/s004220050570.

Wolfram Schenck, AV Adinetz, YV Zaytsev, Dirk Pleiter, and A Morrison. Performance model for large–scale neural simulations with nest. In *Extended Poster Abstracts of the SC14 Conference for Supercomputing*, 2014.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117, 2015. doi:10.1016/j.neunet.2014.09.003.

Eric Schnarr and James R Larus. Fast out-of-order processor simulation using memoization. In *ACM SIGOPS Operating Systems Review*, volume 32, pages 283–294. ACM, 1998. doi:10.1145/291006.291063.

Felix Schürmann, Fabien Delalondre, Pramod S Kumbhar, John Biddiscombe, Miguel Gila, Davide Tacchella, Alessandro Curioni, Bernard Metzler, Peter Morjan, Joachim Fenkes, et al. Rebasing i/o for scientific computing: leveraging storage class memory in an ibm bluegene/q supercomputer. In *International Supercomputing Conference*, pages 331–347. Springer, 2014. doi:10.1007/978-3-319-07518-1_21.

Tilo Schwalger, Moritz Deger, and Wulfram Gerstner. Towards a theory of cortical columns: From spiking neurons to interacting neural populations of finite size. *PLoS computational biology*, 13(4):e1005507, 2017. doi:10.1371/journal.pcbi.1005507.

Ruggero Scorcioni. Gpgpu implementation of a synaptically optimized, anatomically accurate spiking network simulator. In *2010 Biomedical Sciences and Engineering Conference*, pages 1–3. IEEE, 2010. doi:10.1186/1471-2202-11-s1-p133.

Thomas Sharp, Cameron Patterson, and Steve Furber. Distributed configuration of massively-parallel simulation on spinnaker neuromorphic hardware. In *The 2011 International Joint Conference on Neural Networks*, pages 1099–1105. IEEE, 2011. doi:10.1109/ijcnn.2011.6033346.

Julian C Shillcock. Vesicles and vesicle fusion: coarse-grained simulations. In *Biomolecular simulations*, pages 659–697. Springer, 2013. doi:10.1007/978-1-62703-017-5_26.

Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. Exascaling your library: Will your implementation meet your expectations? In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 165–175. ACM, 2015.

Tom Simonite. Moore's law is dead. now what. *MIT Technology Review*, 2016.

Francesco Simula, Elena Pastorelli, Pier Stanislao Paolucci, Michele Martinelli, Alessandro Lonardo, Andrea Biagioni, Cristiano Capone, Fabrizio Capuani, Paolo Cretaro, Giulia De Bonis, et al. Real-time cortical simulations: Energy and interconnect scaling on distributed systems. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 283–290. IEEE, 2019. doi:10.1109/empdp.2019.8671627.

Jaswinder Pal Singh, John L Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, 26(7):42–50, 1993. doi:10.1109/MC.1993.274941.

Daniel J. Sorin, Jonathan L Lemon, Derek L. Eager, and Mary K. Vernon. Analytic evaluation of shared-memory architectures. *IEEE transactions on Parallel and Distributed Systems*, 14(2): 166–180, 2003. doi:10.1109/tpds.2003.1178880.

DT Sorin, Vijay S Pai, Sarita Adve, MK Vemon, and David A Wood. Analytic evaluation of shared-memory systems with ilp processors. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 380–391. IEEE, 1998. doi:10.1109/isca.1998.694797.

Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, New York, NY, USA, 2015. ACM. doi:10.1145/2751205.2751240. URL http://doi.acm.org/10.1145/2751205.2751240.

Thomas Sterling, Daniel Kogler, Matthew Anderson, and Maciej Brodowicz. Slower: A performance model for exascale computing. *Supercomputing frontiers and innovations*, 1(2): 42–57, 2014. doi:10.14529/jsfi140203.

Robert D Stewart and Wyeth Bair. Spiking neural network simulation: numerical integration with the parker-sochacki method. *Journal of computational neuroscience*, 27(1):115–133, 2009. doi:10.1007/s10827-008-0131-5.

Terrence Stewart, Feng-Xuan Choo, and Chris Eliasmith. Spaun: A perception-cognition-action model using spiking neurons. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 34, 2012.

Marcel Stimberg, Dan FM Goodman, Victor Benichoux, and Romain Brette. Equation-oriented specification of neural models for simulations. *Frontiers in neuroinformatics*, 8:6, 2014. doi:10.3389/fninf.2014.00006.

Erich Strohmaier and Hongzhang Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 467–474. IEEE, 2004. doi:10.1109/mascot.2004.1348302.

Evangelos Stromatias, Francesco Galluppi, Cameron Patterson, and Steve Furber. Power analysis of large-scale, real-time neural networks on spinnaker. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2013. doi:10.1109/ijcnn.2013.6706927.

Chetan Singh Thakur Thakur, Jamal Molin, Gert Cauwenberghs, Giacomo Indiveri, Kundan Kumar, Ning Qiao, Johannes Schemmel, Runchun Mark Wang, Elisabetta Chicca, Jennifer Olson Hasler, et al. Large-scale neuromorphic spiking array processors: A quest to mimic the brain. *Frontiers in neuroscience*, 12:891, 2018. doi:10.3389/fnins.2018.00891.

Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005. doi:10.1177/1094342005051521.

Llewellyn Hilleth Thomas. Elliptic problems in linear difference equations over a network. *Watson Sci. Comput. Lab. Rept., Columbia University, New York*, 1, 1949.

William Thompson and L Kelvin. On the theory of the electric telegraph. In *Proc. Royal Soc. London*, volume 7, pages 382–399, 1855. doi:10.1017/cbo9780511996016.009.

Roger D Traub, Robert K Wong, Richard Miles, and Hillary Michelson. A model of a ca3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. *Journal of Neurophysiology*, 66(2):635–650, 1991. doi:10.1152/jn.1991.66.2.635.

J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Parallel Processing and Applied Mathematics*, pages 615–624. Springer, 2010. doi:10.1007/978-3-642-14390-8_64.

Jan Treibig, Georg Hager, and Gerhard Wellein. Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. In *European Conference on Parallel Processing*, pages 451–460. Springer, 2012. doi:10.1007/978-3-642-36949-0_50.

Misha V Tsodyks and Henry Markram. The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proceedings of the national academy of sciences*, 94(2):719–723, 1997. doi:10.1073/pnas.94.2.719.

T Tsuei and Wayne Yamamoto. A processor queuing simulation model for multiprocessor system performance analysis. In *Proc. of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 58–64, 2002.

Henry Clavering Tuckwell. *Introduction to theoretical neurobiology. Vol. 1, Linear cable theory and dendritic structure*. Cambridge University Press, 1988. doi:10.1016/0300-9629(89)90176-x.

Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33 (8):103–111, 1990. doi:10.1145/79173.79181.

Sacha Jennifer van Albada, Andrew G Rowley, Johanna Senk, Michael Hopkins, Maximilian Schmidt, Alan Barry Stokes, David R Lester, Markus Diesmann, and Steve B Furber. Performance comparison of the digital neuromorphic hardware spinnaker and the neural network simulation software nest for a full-scale cortical microcircuit model. *Frontiers in neuroscience*, 12:291, 2018. doi:10.3389/fnins.2018.00291.

A Van Amesfoort, A Varbanescu, and H Sips. Metrics to characterize parallel applications. In *15th Workshop on Compilers for Parallel Computing, Vienna, Austria*, 2010.

Alexander S van Amesfoort, Ana Lucia Varbanescu, and Henk J Sips. Parallel application characterization with quantitative metrics. *Concurrency and Computation: Practice and Experience*, 24(5):445–462, 2012. doi:10.1002/cpe.1882.

Werner Van Geit, Erik De Schutter, and Pablo Achard. Automated neuron model optimization techniques: a review. *Biological cybernetics*, 99(4-5):241–251, 2008. doi:10.1007/s00422-008-0257-6.

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. doi:10.3389/fninf.2016.00017.

János Végh. How amdahl's law limits the performance of large artificial neural networks. *Brain informatics*, 6(1):4, 2019. doi:10.1186/s40708-019-0097-2.

Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kastner. Configcrusher: White-box performance analysis for configurable systems. *arXiv preprint arXiv:1905.02066*, 2019.

Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. Intel memory latency checker, 2018. version 3.6.

Julien Vitay, Helge Ülo Dinkelbach, and Fred H Hamker. Annarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in neuroinformatics*, 9:19, 2015. doi:10.3389/fninf.2015.00019.

John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993. doi:10.1007/978-3-642-61812-3_30.

Mingchao Wang, Boyuan Yan, Jingzhen Hu, and Peng Li. Simulation of large neuronal networks with biophysically accurate models on graphics processors. In *The 2011 International Joint Conference on Neural Networks*, pages 3184–3193. IEEE, 2011. doi:10.1109/ijcnn.2011.6033643.

Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

Gehan Weerasinghe, Imad Antonios, and Lester Lipsky. A generalized analytic performance model of distributed systems that perform n tasks using p fault-prone processors. In *ipdps*, 2002. doi:10.1109/ipdps.2002.1016524.

F Wendling, J Bellanger, and F Bartolomei. Computational models and eeg dynamics. contribution of neurophysiologically relevant models to the analysis of epileptic signals. In *First International IEEE EMBS Conference on Neural Engineering, 2003. Conference Proceedings.*, pages 414–417. IEEE, 2003. doi:10.1109/cne.2003.1196849.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009. ISSN 0001-0782. doi:10.1145/1498765.1498785. URL http://doi.acm.org/10.1145/1498765.1498785.

Terry L Wilmarth, Gengbin Zheng, Eric J Bohm, Yogesh Mehta, Nilesh Choudhury, Praveen Jagadishprasad, and Laxmikant V Kale. Performance prediction using simulation of large-scale interconnection networks in pose. In *Proceedings of the 19th workshop on principles of advanced and distributed simulation*, pages 109–118. IEEE Computer Society, 2005. doi:10.1109/pads.2005.20.

Felix Wolf, Christian Bischof, Torsten Hoefler, Bernd Mohr, Gabriel Wittum, Alexandru Calotoiu, Christian Iwainsky, Alexandre Strube, and Andreas Vogel. Catwalk: a quick development path for performance models. In *European Conference on Parallel Processing*, pages 589–600. Springer, 2014. doi:10.1007/978-3-319-14313-2_50.

Timo Wunderlich, Akos F Kungl, Andreas Hartel, Yannik Stradmann, Syed Ahmed Aamir, Andreas Grübl, Arthur Heimbrecht, Korbinian Schreiber, David Stöckel, Christian Pehle, et al. Demonstrating advantages of neuromorphic computation: A pilot study. *arXiv preprint arXiv:1811.03618*, 2018. doi:10.3389/fnins.2019.00260.

Esin Yavuz, James Turner, and Thomas Nowotny. Genn: a code generation framework for accelerated brain simulations. *Scientific reports*, 6:18854, 2016. doi:10.1038/srep18854.

Dmitri Yudanov, Muhammad Shaaban, Roy Melton, and Leon Reznik. Gpu-based simulation of spiking neural networks with real-time performance & high accuracy. In *The 2010 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2010. doi:10.1109/ijcnn.2010.5596334.

Semir Zeki. A massively asynchronous, parallel brain. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 370(1668):20140174, 2015. doi:10.1098/rstb.2014.0174.

Friedemann Zenke and Wulfram Gerstner. Limits to high-speed simulations of spiking neural networks using general-purpose computers. *Frontiers in neuroinformatics*, 8:76, 2014. doi:10.3389/fninf.2014.00076.

W Zuberek. Performance analysis of shared-memory bus-based multiprocessors using timed petri nets. *Petri Nets in Science and Engineering, InTechOpen, London*, pages 75–91, 2018. doi:10.5772/intechopen.75589.

Francesco Cremonesi

PhD candidate in computational neuroscience and high performance computing.

LINKS

Github:// **sharkovsky**
LinkedIn:// **francesco-cremonesi**
Twitter:// **@checc000**
Blog:// **sharkovsky.github.io**
StackOverflow:// **sharkovsky87**
Scholar:// **Francesco Cremonesi**

COURSEWORK

TEACHING ASSISTANTSHIP

Google Summer of Code 2015 and 2017
In Silico Neuroscience
Linear Algebra
Unsupervised and Reinforcement Learning on Biological Neural Networks

PROJECTS

Coursework projects:

- Minimum Information Partition with Submodular Minimization *(Python, NEST, NEURON)*.
- Text Generation using Reservoir Computing *(Python, NEST)*.
- Medical Image Segmentation using RSFE-Split Bregman *(C++)*.

Final master project:

- Monolithic Fluid-Structure Interaction and Anisotropic Mesh Adaptation *(C++, FEM)*.

Final bachelor project:

- Super-Resolution algorithm for gelogical images *(Matlab)* .

SKILLS

PROGRAMMING

Python • C/C++ • MPI • OpenMP
Matlab/Octave • R • LaTeX
numpy • pandas • keras • NEST
NEURON/nmodl • BlueGene/Q
Shell • vim • gdb • pdb • vTune

LANGUAGES

Italian *(mothertongue)* • English *(fluent)*
French *(fluent)* • Spanish *(basic)*

HOBBIES

Basketball • Guitar • Photo • Travel
Top 3 books: *the unbearable lightness of being; love in the time of cholera; for whom the bell tolls.*

CURRENT POSITION

BLUE BRAIN PROJECT | PHD CANDIDATE

August 2014 - Present | EPFL, Switzerland
Research in the field of computational neuroscience, under the supervision of **Felix Schürmann**. Focused on understanding the relationship between the simulation of biological or artificial neural networks and computer architectures.

EDUCATION

ECOLE DES MINES DE PARIS

SPECIALIZED MASTER IN MATERIAL PROCESSING AND MODELING
September 2013 | CEMEF, Nice

POLITECNICO DI MILANO

MASTER OF SCIENCE IN MATHEMATICAL ENGINEERING
December 2013 | Milano

ECOLE CENTRALE PARIS

DOUBLE DEGREE - MASTER OF SCIENCE IN ENGINEERING
September 2011 | Paris

POLITECNICO DI MILANO

BACHELOR OF SCIENCE IN MATHEMATICAL ENGINEERING
September 2010 | Milano

EXPERIENCE

BLUE BRAIN PROJECT | MATHEMATICAL MODELING INTERN

November 2013 - June 2014 | EPFL, Switzerland

- understand and document application-specific numerical techniques and distributed implementation;
- numerical validation of neuroscientific simulations using NEURON software;
- analysis of algorithmic complexity and simulation performance;
- C/C++ (pre C++11), MPI, OpenMP;

EADS | NUMERICAL SIMULATIONS INTERN

March 2013 - September 2013 | CEMEF, Nice

- finite element fluid dynamics simulation;
- implementation of Spalart-Allmaras turbulence model;
- C++ (pre C++11);

EGONOCAST | VIDEO RECOGNITION INTERN

June 2010 - September 2010 | Paris

- Video classification using pre-DL revolution: e.g. histogram comparison;
- C++, pthread, CUDA 3.0;

PUBLICATIONS

Francesco Cremonesi et al. “Analytic Performance Modeling and Analysis of Detailed Neuron Simulations”. In: *The International Journal of High Performance Computing Applications* (2019). *In review*

Francesco Cremonesi and Felix Schürmann. “Telling neuronal apples from oranges: analytical performance modeling of neural tissue simulations”. In: *Neuroinformatics* (2019). *In review*

Francesco Casalegno et al. “Error Analysis and Quantification in NEURON Simulations”. In: *Proc. ECCOMAS Conf.* (2016)

Timothée Ewart et al. “Polynomial evaluation on superscalar architecture, applied to the elementary function e^x”. In: *ACM Transactions on Mathematical Software (TOMS)* (2019). *In review*

Timothee Ewart et al. “Performance Evaluation of the IBM POWER8 system to Support Computational Neuroscientific Application Using Morphologically Detailed Neurons”. In: *proc. Supercomputing Conf.* (2015)

Timothee Ewart et al. “Neuromapp: a Mini-Application Framework to Improve Neural Simulators”. In: *Proc.Intl. Supercomputing Conf.* (2017)

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**