Optimizing Extended Hodgkin-Huxley Neuron Model Simulations for a Xeon/Xeon Phi Node

George Chatzikonstantis, Dimitrios Rodopoulos, Christos Strydis, Chris I. De Zeeuw, and Dimitrios Soudris

Abstract—Brain modeling has been receiving significant attention over the years, both for its neuroscientific potential and for its challenges in the context of high-performance computing. The development of physiologically plausible neuron models comes at the cost of increased complexity. In this work, we have selected a highly computationally demanding model of the Inferior-Olivary Nucleus (InfOli) based on the Hodgkin-Huxley (HH) neuron model. This brain region, functionally coupled with the cerebellum, is of vital importance for motor skills and time-sensitive cognitive functions. The computing fabric of choice is an Intel Xeon/Xeon Phi system, which is a typical node of modern computing infrastructure. The target application is parallelized with various combinations of MPI and OpenMP and performance is measured on the target platform. The different implementations are compared and the best one is chosen. Further optimization of this implementation is presented in detail. Its behaviour is then examined when scaling up to neuron populations representative of realistic, human Inferior-Olivary neuronal networks. The evaluation's results highlight the importance of examining a network's size and density before choosing the best platform for its simulation. All the parallelization and vectorization options presented in the current paper are available on a public repository for further examination.

Index Terms—MPI, Neuron Modeling, OpenMP, Xeon Phi, Performance, Vectorization

1 Introduction

Neuroscientists have been gradually revealing details of neuron operation. Software has been developed for single-neuron, and eventually, brain-wide simulations. Porting of such simulators on various platforms is an active field of research [1], [2]. Aiming at larger, more accurate neuron networks, neuroscientists require more memory and extended execution times to produce relevant results.

Similarly to other compute-intensive fields [3], multi- and many-core platforms can speed up neuron simulators, such as through the NEuronal Simulation Tool (NEST) [4]. The current paper features a simulator for biophysically plausible inferior-olivary neuron models. It is an extension of a previously published paper discussing the initial experiences from porting the application on an Intel Xeon/Xeon Phi node [5]. The modeling accuracy is at the cell conductance level (Hodgkin and Huxley models [6]), allowing us to expose fine details of the neuron mechanisms. This workload is an excellent candidate for parallelization on (co)processor fabrics, such as the Intel Xeon/Xeon Phi system [7], due to the large inherent parallelism of the models. Additionally, it constitutes a realistic worst-case scenario in terms of model complexity, hence a benchmark for neuron modeling workloads.

The Xeon Phi [8] (hereon referenced as simply "Phi" for clarity) is an Intel accelerator platform arranged in a host-and-coprocessor fashion. The machine examined in this paper belongs to the Knight's Corner (KNC) generation and features 61 cores, each with four instruction streams. It supports traditional parallel-programming paradigms, such as MPI [9] and OpenMP [10], in contrast to Graphics Processing Units (GPU) requiring platform-specific programming paradigms [11]. After the Xeon host boots Linux on the Phi, the latter may be used independently, for native workload execution. The Phi accelerator features vectorization processing units (VPU) [8], which can parallelize multiple floating-point (FP) operations.

In the current paper, we explore porting of the HH neuron simulator, using combinations of MPI and OpenMP, on the target platforms. By varying the problem size (i.e. number of simulated neurons), the optimal implementation and target fabric (Xeon host or Phi) may also change. The contributions of the current paper are especially geared towards identifying the correct implementation based on problem size: (i) Three different, non-platform-specific implementations are presented for the InfOli simulator (MPI, OpenMP, and hybrid); (ii) Their performance and scaling capabilities for various workload parameters is evaluated natively on the Xeon host and the Phi, allowing optimal combinations of implementation and fabric to be identified; (iii) The optimal design points are scaled up to form realistic neuron population sizes encountered in the human inferior olive [12] and (iv) The most promising implementation undergoes extensively analyzed code transformations in order to boost the underlying fabric's performance.

The paper is organized as follows: Section 2 presents simulators and many-core platforms that are typically used in the computational-neuroscience domain. Section 3 elaborates on the deployment of the InfOli simulator on the Xeon/Xeon Phi system. Section 4 presents performance measurements of the various implementations and the trends observed. Section 5 presents the concept of vectoriza-

- *G. Chatzikonstantis and D. Soudris are with the Laboratory of Microprocessors and Digital Systems (MicroLab), Department of Electrical and Computer Engineering (ECE), National Technical University of Athens (NTUA), Greece. E-mail: {georgec, dsoudris}@microlab.ntua.gr*
- *D. Rodopoulos is with imec, Belgium. E-mail: dimitrios.rodopoulos@imec.be*
- *C. Strydis and C. I. De Zeeuw are with Erasmus Medical Center Rotterdam (EMC), Netherlands. E-mail: {c.strydis, c.dezeeuw}@erasmusmc.nl*

Manuscript received Month Day, Year.

tion in detail, as well as related techniques used to increase performance. Section 6 outlines the conclusions of this work. The simulator, along with all the coding options presented herein, is available under GPL [13].

2 Prior Art and Motivation

The neuroscientific community focuses on various aspects of neuronal activity, featuring models at different abstractions. A reference tool in this domain is NEURON [14], which provides the user with many accurate model simulators. NEST [4] is a lighter tool that mostly aims at simulating large networks of much simpler neuron models. A different approach is explored by the European research project FACETS [15], whereby instead of using software-based numerical methods, analog neuromorphic hardware directly simulates complex neuron models.

In the current Section we firstly present different neuron models to offer some insight on typical neuroscientific processing workloads, as well as the model used in this work. Then, we discuss prior efforts of simulating neuronal networks on various fabrics. Finally, we elaborate on our fabric of choice, the Xeon/Xeon Phi system.

2.1 Background on Neuron Models

Spiking Neural Networks (SNNs) [16] focus on input-current patterns and spike-transfer delays, tracking processes common in biological neural networks. These models are used by neuroscientists to study complex brain mechanisms and test hypotheses that *in vitro* and *in vivo* experiments cannot verify. They are broadly categorized either as Integrate and Fire (I&F) or as conductance-based models.

I&F models are the simplest SNN models, primarily focusing on receiving a spike input and determining the neuron's response based on a voltage threshold. They are widely used due to their simplicity and extensibility, resulting in a large range of I&F variants in the literature (e.g. leaky I&F [17], [18] and exponential I&F [19] models). *Conductance-based models* lie on the opposite side of the spectrum, using complicated differential equations. They offer valuable insight into the electrochemical properties of the neuron and the ability to study its ion channels. However, the high modeling accuracy they offer comes at the cost of significant computational complexity, often deeming them too expensive to use in daily experiments. The Hodgkin and Huxley model [6] can be considered as the most prominent example of this class. A mathematical reduction of this model has been proposed by Richard FitzHugh and Nagumo et al. [20].

The Izhikevich model [21] bridges the gap between simpler I&F models and complex conductance-based ones. This model is computationally simple (like I&F) and, although it does not expose the electrochemical details of biological neurons, Izhikevich neurons display biologically plausible input-output interactions. They feature computational complexity significantly smaller than that of HH models. To put their differences in perspective, an Izhikevich neuron requires 13 floating-point operations to simulate 1 ms of its operation, whereas an HH neuron will need thousands of floating-point operations for the same amount of activity

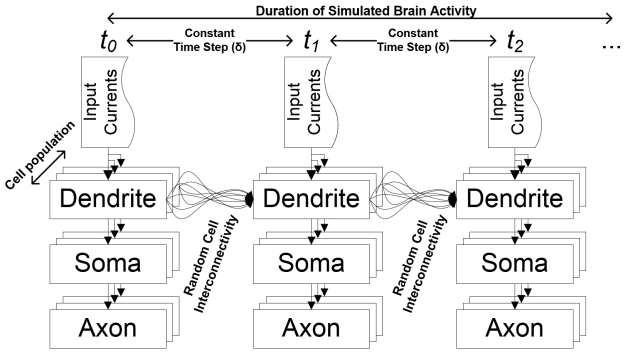


Fig. 1. Flowchart of the InfOli simulator [23].

[22]. A lighter version of the HH-based model discussed in the current paper, featuring significantly less complicated inter-neuron communication mechanisms, has been ported on NVIDIA GPUs and scaled up to one million neurons [2]. Finally, a thorough classification of available neuron models and simulators has been made by Brette et al. [16].

2.2 Target Model and Simulator

The model used in the current work belongs to the complex class of conductance-based (HH) models. Its computational requirements make it a valid candidate for many-core platform porting, especially when the simulated population of neurons needs to be scaled. Contrary to related work, where a custom implementation of this simulator was ported on a research-grade chip [23], we discuss implementations using programming paradigms that are not platform-specific.

The InfOli model is used to describe neurons of the inferior-olive region, which is a small part of the brain linked to learning of movements and motor function [30]. This particular model is an HH extension, originally designed by De Gruijl et al. [31]. The model splits the function of each neuron into three compartments: the soma, the axon and the dendrite. The soma serves as the neuron's main computational body. The axon connects to other parts of the brain (through climbing fibers) and can be thought of as the neuron's output port. Dendrite structures called dendritic spikes form electrotonic connections with other neurons in the network. These electrical synapses are known as gap junctions and are responsible for inter-neuron communication [32]. This model feature represents a potential parallelization bottleneck, since gap junctions require heavy data communication between processing threads, particularly in densely interconnected networks.

The InfOli simulator of Figure 1, performs the following tasks: First, the state of the neurons is initialized with values assigned to ion-channel concentrations and compartmental membrane voltage potentials. Then, the next state of each neuron is calculated, given a connectivity map, a set of input currents and ordinary differential equations (ODEs) for the mechanisms in each neuron compartment, solved via the Euler forward method [33]. The latter is an iterative process, repeated in steps with a duration of δ milliseconds, until the simulated brain time requested by the user is reached. This is thus, a time-driven simulator the state and output of which are calculated precisely at every simulation step.

TABLE 1
Relevant prior art.

Reference	Platform	Neuron Model	Network Size	Network Density	Reported Results
[24]	Supercomputer K	I&F	1.8 bil.	6,000 connections/neuron	270 hrs/second of Activity
[25]	GPU	Izhikevich	300,000	300 connections/neuron	15× slower than Real Time
[26]	Xilinx FPGA	Izhikevich	1,000	1,000 connections/neuron	Real-time Simulation
[27]	GPU	Izhikevich	40,000	1,000 connections/neuron	Real-time Simulation
[28]	Supercomputer Dawn BG/P	Izhikevich	900 mil.	10,000 connections/neuron	5 minutes/second of Activity
[1]	GPU and Intel Xeon	Izhikevich and HH	5.8 mil. Izh., 50 HH	Full Connectivity Between Two Layers	Speedups of 10× for Izh. and 120× for HH
[29]	GPU	HH-based	400,000	8 connections/neuron	1.33 hrs/second of Activity
[2]	GPU	HH-based	1 mil.	8 connections/neuron	6.66 minutes/second of Activity
Our work	**Intel Xeon and Phi**	**HH-based**	**1 mil.**	100 **connections/neuron**	24 **minutes/second of Activity**

2.3 Neuron Simulation on Many-Cores

As mentioned above, accelerators and many-core fabrics are an attractive option for neuroscientific workloads. Fidjeland et al. [27] have successfully deployed densely connected neuronal networks on GPUs, reaching network sizes of $40,000$ Izhikevich neurons. Bhuiyan et al. [1] use various platforms to scale up to millions of Izhikevich neurons, coupled with 50 HH neurons in a 2-level neuronal structure. Furthermore, their work aims at character recognition rather than providing biophysiological value. Choi et al. [26] have developed 1,000 silicon spiking neurons on a Xilinx Field Programmable Gate Array (FPGA), based on the Izhikevich model. Ananthanarayanan et al. have documented one of the largest simulation efforts to date, porting a network of 1 billion Izhikevich neurons on a cutting-edge supercomputer [28]. Partners of the Human Brain Project (HBP) have also used GPUs and FPGAs for HH modeling of the human cerebellum [29]. Their implementation scales up to 400,000 neurons, simulates brain activity for 3 s and imposes a static nearest-neighbour neuron interconnectivity.

In addition to porting standard SNNs on accelerators, complete toolkits, aimed at the development of neuronal models, have been ported to such computing platforms. An FPGA toolbox for simulating SNNs in hardware has been developed by Qingxiang et al. [34]. CARLsim [25], on the other hand, is a GPU-oriented library for SNN simulation and model-testing. Finally, NEST allows MPI, multithreading and hybrid usage thereof as parallelization methods, thus providing another interesting alternative for high-performance neuron modeling that has been tested on high-end supercomputers [24].

The information presented in this Subsection is summarized in Table 1, following an order of simplest to most elaborate neuronal model. The Table does not include a full review of all related work in the literature; an extensive survey is outside the scope of this paper.

2.4 Target Platform

Phi belongs to the Many Integrated Core (MIC) architecture; the processor model used in this work features 61 cores with multithreading capabilities and VPUs, allowing Single Instruction Multiple Data (SIMD) execution of FP operations.

The Intel Phi card is treated as an accelerator and requires a Xeon host to boot a Linux image on it; however it can also be thought of as a standalone processor, executing any series of instructions independently from the main processor. A developer can use the Phi to code via traditional tools of parallel programming, such as the OpenMP library [10]. This fact differentiates it from other acceleration platforms such as GPUs by allowing the programmer to avoid using specialized libraries, such as CUDA [11] or OpenCL [35], allowing for rapid code development. This benefit is partially tempered when attempting to exploit the platform's SIMD instructions, a task which is not trivial for complicated codebases. To that end, there exist specialized tools that aid with the use of the VPUs and enhance vectorization [36]. We will initially evaluate implementations that are generic enough to be seamlessly portable on both the processor and the co-processor. The most promising implementation will be enhanced with fine-grain vectorization in order to take full advantage of the underlying platform.

3 Implementation Details

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

3.1 MPI Implementation

MPI is a library for distributed-memory systems where coordination between cores is achieved by message-passing through fast-memory buffers. While both the Xeon host and

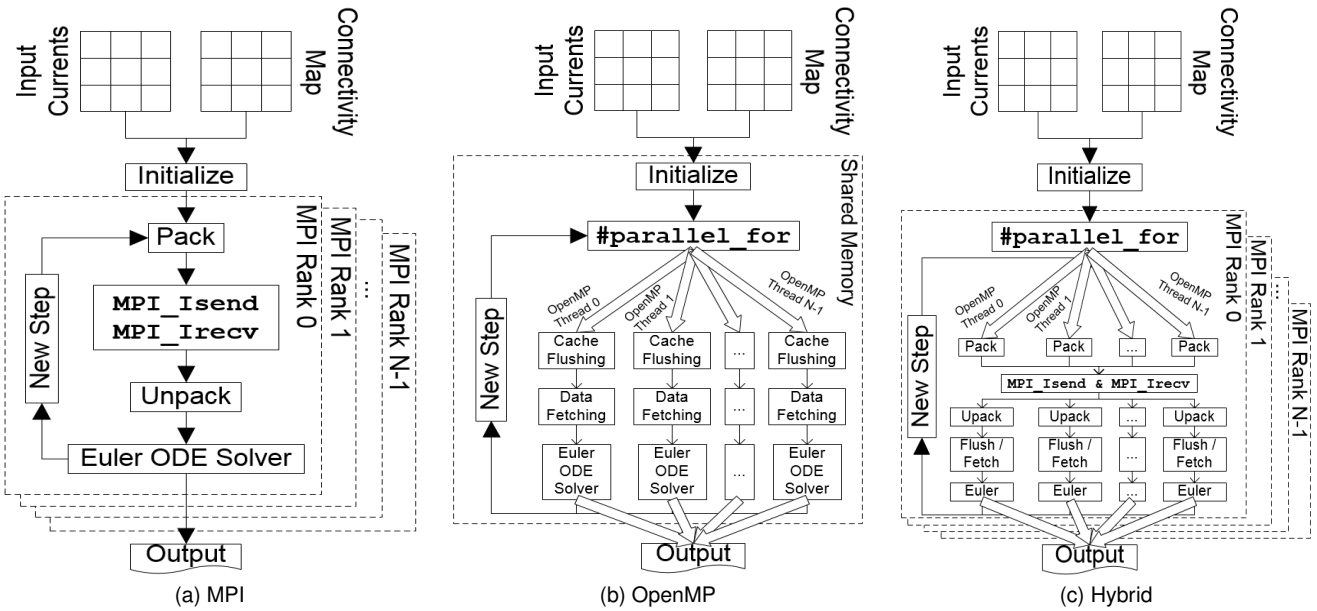


Fig. 2. Flowchart of the implementations discussed in the current paper [5].

the Phi co-processor share memory between their cores, the MPI implementation is still useful for evaluating the shared-memory performance and serves as a baseline towards the hybrid method. It is also a well-supported and continuously updated tool which can aid in multi-node-system implementations [37], [38]. In such multi-node systems, message passing is achieved over TCP or Infiniband. In a single-node system, shared memory is used instead.

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

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

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

Figure 2a describes the MPI implementation. The simulator starts by initializing the neuron states and processes the connectivity map. The neuron population is divided and assigned to MPI ranks. The execution then proceeds to the main loop which lasts for a fixed, user-defined number of simulation steps. In each step, the neurons receive input stimulus current. The cores then pack their data in $k-1$ buffers. These buffers are exchanged via asynchronous MPI communication and unpacked. Each core checks for the completion of all relevant MPI communication functions to ensure that neurons have access to updated data regarding their connections to other neurons. Only then can the neuronal network be updated to its new state while avoiding stale data-propagation. Each core performs a set of calculations for each neuron it handles and stores the neuron's new state values locally. The simulation step then ends and the cycle begins anew.

3.2 OpenMP Implementation

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

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

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

3.3 Hybrid Implementation

An OpenMP implementation, as proposed in the previous Subsection, may appear to be the most intuitive parallelization strategy for the InfOli model. Given both implementations' strengths and weaknesses, it may be interesting to explore a hybrid implementation, combining both MPI and OpenMP. This course of action is even more compelling for the Phi, since it combines the platform's multithreading capabilities and the option to distribute the workload across multiple Phi cards. The hybrid implementation developed stems primarily from our MPI implementation (Subsection 3.1). While the primary unit of execution is an MPI rank, each MPI rank further spawns OpenMP threads to create a hybrid porting. These threads are used to boost packing and unpacking, as well as the main computation process.

In Figure 2c, we organize the cores of a platform into *groups*. Within each group, all cores communicate over shared memory. They spawn OpenMP threads to perform and accelerate computations. Each group is perceived as a single MPI rank in the MPI environment. Within the group, one "master" core handles MPI communication and sends necessary data from the entire group to another shared-memory group on every simulation step. The packing and unpacking of this data is performed by the OpenMP threads spawned by the entire group. Only the actual MPI calls are performed in single-threaded fashion by one core per group.

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

4 Implementation Evaluation

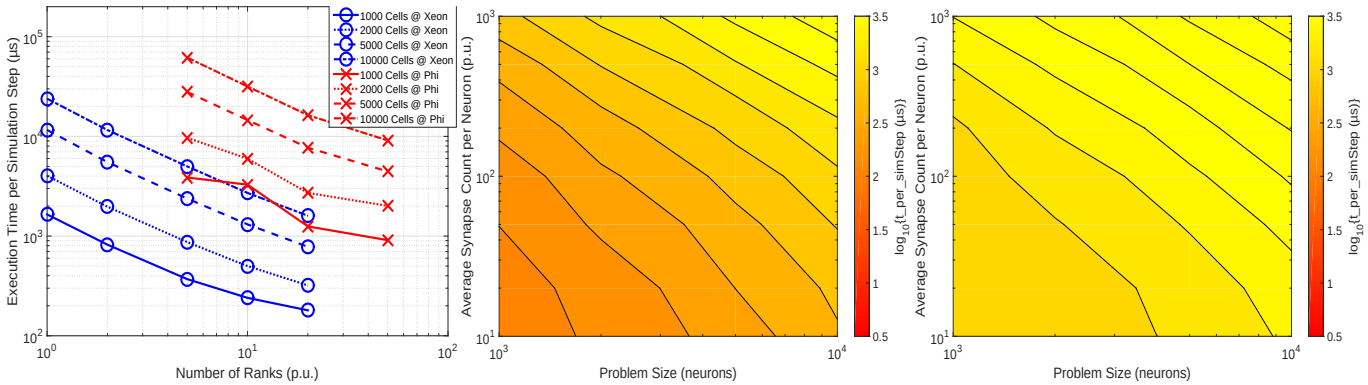
4.1 Experimental Setup

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

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

Connections are created at a pre-processing stage (based on the aforementioned patterns) and are stored in the connectivity map of the simulation, represented as an adjacency matrix. To decrease input file sizes, sparse-matrix formats are used for large neuronal networks ($\geq 10,000$ neurons).

The entirety of our experiments has been carried out in the Blue Wonder cluster, at the Hartree Center of the Science & Technology Facilities Council (STFC) in the United Kingdom. Access to Xeon/Xeon Phi systems (one single node) is enabled over `ssh`. Each node contains an Intel Xeon E5-2697v2 processor (dual-socket arrangement) with 64 GB of RAM, operating at 2.7 GHz and one Intel Phi 5110P accelerator. All measurements presented in the current paper have been taken from execution of the target application on a single node of the cluster. The Intel MPI compiler (Intel MPI-5.0.3) is used for the MPI and hybrid implementations. The Intel C compiler (Intel compiler 15.0.2) is used for the OpenMP implementation, as well as the related Intel OpenMP runtime library (following the OpenMP 4.0 standard). Performance measurements of small-scale runs (maximum network size of 10,000 neurons) have been per-



(a) MPI (b) MPI Network Density Exploration - Xeon (c) MPI Network Density Exploration - Phi

Fig. 3. Depiction of MPI measurements on the host and the Phi.

formed using the Intel VTune performance analyzer (Intel VTune Amplifier XE 2016). Since VTune collects hardware events during execution, its overhead (both in execution time and disk space) is prohibitive for larger simulations; the Linux default `time` command (GNU time 1.7) has been used in large-scale simulations instead. We present performance measurements in execution time per InfOli simulation step, so that we mitigate transient effects at simulation start/end, as well as depict the simulator's performance in a manner more easily comparable to related work in the literature.

4.2 Experimental Results

The three implementations presented in Subsections 3.1, 3.2, and 3.3 have been tested natively both on the Xeon host and the Phi co-processor, creating a total of six different evaluations. We explore the granularity of each implementation and evaluate a variety of problem sizes (neuron populations, network density). Findings are reported next.

4.2.1 MPI Implementation

In Figure 3, we present results for the pure MPI implementation. Figure 3a explores the granularity of the implementation by varying the number of MPI ranks. We used up to 20 ranks for the Xeon processor and up to 50 ranks for the Phi co-processor, since it offers more hardware cores than the Xeon processor. The figures reveal that the Phi is outperformed by the dual-Xeon processor host. One of the primary reasons for this is the inability of a strictly MPI-based implementation to take advantage of the entirety of the Phi's computing resources. The Phi accelerator cards feature cores that base much of their processing power on their multithreading capabilities, capable of supporting up to 4 instruction streams in parallel. The MPI implementation however, only uses a single thread per rank.

From the figure, we also observe a difference in performance gains as we employ more MPI ranks. There is an irregularity in efficiency when simulating relatively small networks of 1,000 to 2,000 neurons on the Phi device. On the Xeon host, there is reduced performance gain for a network of only 1,000 neurons as we reach 20 MPI ranks. This behavior suggests that such problem sizes pose relatively small workloads that do not fully exploit the computational resources of each platform, especially the Phi. The aforementioned trend disappears for problem sizes increase to at least $5,000$ neurons. When solving for such networks, we get a near-linear performance gain when we increase the number of MPI ranks employed. This behavior is interesting since, as explained in Subsection 3.1, an increase in MPI ranks increases communication overheads in data exchange as well as in data packing and unpacking. Linear performance gains, on the Phi device in particular, indicate the following: given that the MIC architecture focuses on high memory bandwidth, it can handle the scaling message exchanging demands imposed by as many as 50 communicating MPI ranks, as long as the workload per rank is large enough.

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

4.2.2 OpenMP Implementation

In Figure 4, we illustrate the performance assessment of the OpenMP implementation. For network sizes between 1,000 and 10,000 neurons the shared-memory implementation works better on the Phi device compared to the purely MPI equivalent. Particularly for smaller networks, OpenMP runs for a fraction of the execution times reported for MPI. By design, Phi supports many more threads (up to 240 when fully utilizing 60 cores) than the Xeon processor. Thus, in the OpenMP paradigm, we can exploit the accelerator's resources much more aggressively. This leads to a performance improvement when compared to MPI in the case of the Phi. Besides, the Xeon host can be expected to have similar performance between MPI and OpenMP, given that the number of threads that can be supported is smaller.

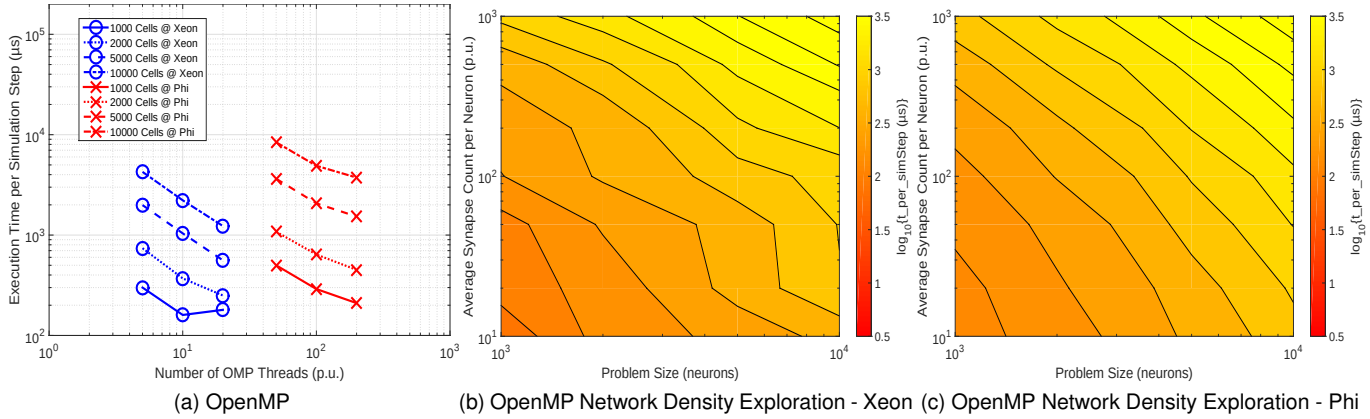


Fig. 4. Depiction of OpenMP measurements on the host and the Phi.

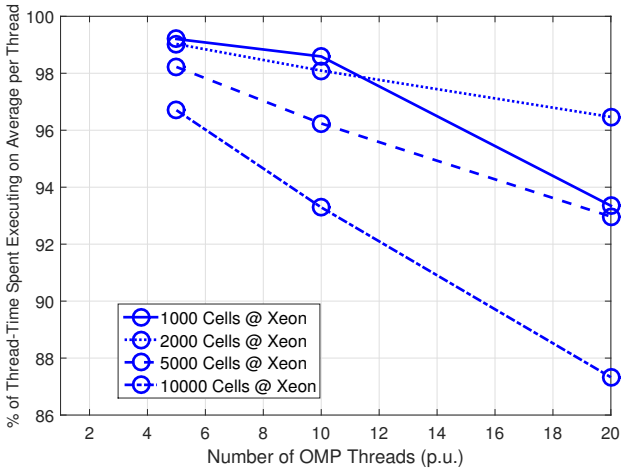


Fig. 5. OpenMP thread activity on the Xeon host.

According to Figure 4a, the Phi accelerator's performance increases in a near-linear fashion with the amount of OpenMP threads invoked. This is an expected observation if we consider the design of the Phi as a platform for massive multithreading. In contrast, the Xeon host does not exhibit consistent scaling as more OpenMP threads are added for small populations of simulated neurons. More specifically, after using half of its resources, employing more OpenMP threads does not speed the processor further up.

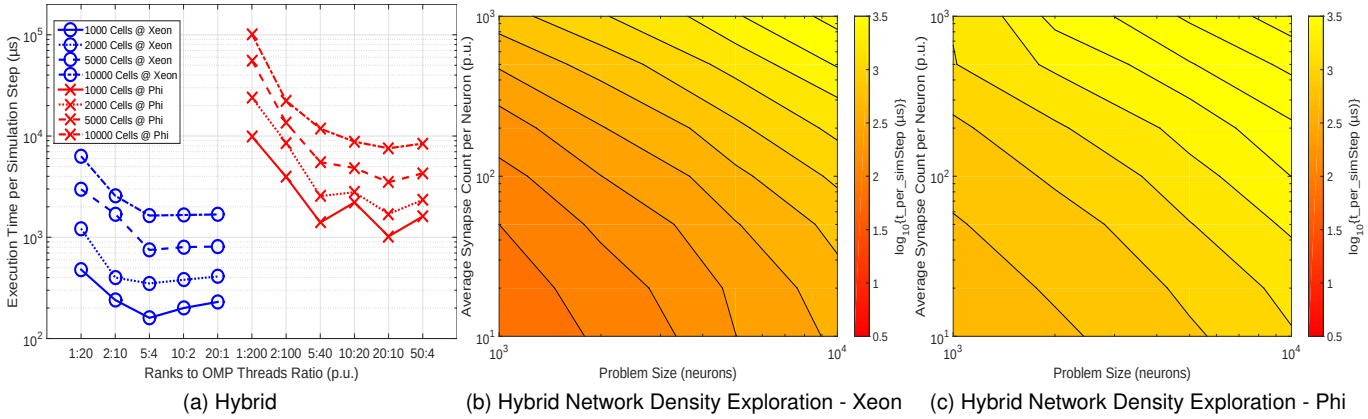
We can generally form the following hypothesis for non-linear scaling of small neuron populations on the Xeon host: The *benefit* of the OpenMP implementation is that, for sufficiently large problem sizes, increasing the available thread count reduces the computational burden assigned per thread. In other words, more threads means less simulated neurons per thread. On the other hand, the *cost* of an OpenMP implementation is related to the overhead of shared-memory operations. More OpenMP threads on the same platform results in thread concurrency taking a larger hit due, for instance, to race conditions on shared resources. Conclusively, and as the Xeon host's performance in 4a indicates, small problem sizes can be efficiently tackled with a small number of threads. On the contrary, when the problem size is sufficiently large, initializing more OpenMP threads is beneficial, the coherency penalties notwithstanding.

The Xeon host's OpenMP performance issues for small workloads are further supported by data in Figure 5. By using Intel VTune to analyze the application performance on the host, we collected data concerning the OpenMP threads CPU time, which is defined as: "the amount of time a thread spends executing on a logical processor and, for multiple threads, the CPU time of the threads is summed" [39]. By dividing the collective CPU time with the number of threads employed by an application, we thus calculate the time spent executing on the processor, averaged across all threads. We then compare this time against the real elapsed time of the workload to calculate the percentage of time spent executing on the processor, averaged across all threads. It is then, demonstrated that, for $1,000$ neurons, using 20 threads drastically decreases the average time of thread activity. When not executing on the processor, the threads are idle, as would be the case of waiting for thread synchronization. This idleness appears to be prevalent when spawning multiple OpenMP threads for small workloads.

In addition, in Figure 4b, the increase in execution time per simulation step of the OpenMP implementation is erratic when examining smaller and sparser networks. This observation builds further upon the statement that the Xeon host's performance is relatively inefficient when using the maximum number of OpenMP threads for small workloads. On the contrary, for larger and denser networks, the Xeon host performs in a more predictable manner. Furthermore, the Phi accelerator, in Figure 4c, features a more linear increase in execution time as the workload increases.

4.2.3 *Hybrid Implementation*

In Figure 6, we present the performance assessment of the hybrid implementation. We examine a range of ratios between the number of MPI ranks and the number of OpenMP threads each rank spawns. For this implementation, multiplying the number of MPI ranks employed by the corresponding number of OpenMP threads a rank utilizes yields the total amount of OpenMP threads spawned across the platform. In all measurements presented for this implementation, this number will always be equal to 20 for the Xeon host and 200 for the Phi accelerator. In this manner, the implementation takes advantage of each platform's assets in



(a) Hybrid (b) Hybrid Network Density Exploration - Xeon (c) Hybrid Network Density Exploration - Phi

Fig. 6. Depiction of Hybrid measurements on the host and the Phi.

a consistent manner throughout the ratio-sweep. In general, measurements shown in 6a indicate that a "middle-of-the-road" ranks-to-threads ratio yields the best performance.

In the case of the Xeon host, spawning 5 MPI ranks, with each rank using 4 OpenMP threads, offers the best results. Using more MPI ranks does not offer any additional benefit. A similar behavior is observed on the Phi co-processor. A configuration of 20 MPI ranks, each spawning 10 OpenMP threads, offers the best performance. Additionally, simulations of reduced neuron populations using the hybrid implementation on the Phi exhibit performance unpredictability beyond the 5:40 rank-to-thread ratio. In general, we observe that both platforms perform better with a moderate balance between MPI ranks and OpenMP threads. Using these ratios, extensive measurements for networks of varying size and complexity are depicted in Figures 6b and 6c.

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

4.2.4 *Comparing Implementations*

Having swept the parameters of the discussed implementations on the Xeon host and Phi co-processor, we attempt to scale the best of them to neuroscientifically-relevant problem sizes, namely beyond half a million neurons [12]. From each of the discussed implementations, we isolate the configurations behaving optimally in Figures 3a, 4a, and 6a and increase the simulated neuron populations. The results of this final set of experiments are illustrated in Figure 7.

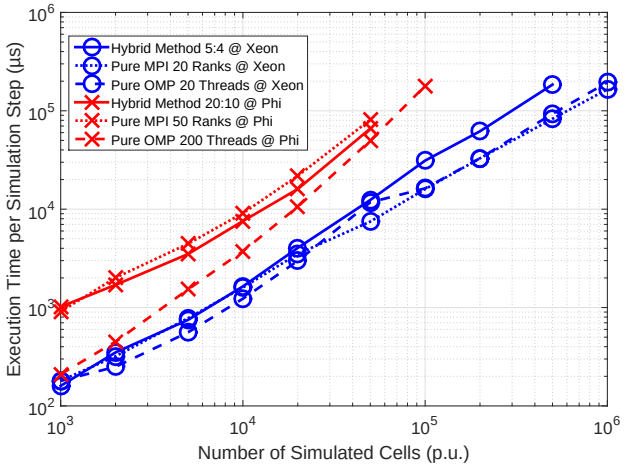


Fig. 7. Comparing the best implementations on host and accelerator, before manual AVX-oriented optimizations.

As expected from Figures 3a, 4a, and 6a, the Phi accelerator cannot compete with the host for native execution of these implementations. Clearly, the only way for potentially gaining more performance from the MIC accelerator is by performing source code vectorization [36] after identifying the implementation of choice for the Phi.

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

Comparing the two message-passing methods on the accelerator, the hybrid implementation outperforms the pure MPI method by drawing on more of the platform's

resources. We observe a relatively small performance gain created by spawning OpenMP threads in each MPI rank, that slowly grows as network size increases. The hybrid approach was expected to improve on the MPI implementation by a wider margin. The overhead of spawning OpenMP threads on each simulation step appears be a limiting factor to its efficiency.

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

In conclusion, all three implementations can use the entirety of the Xeon computing capacity and the larger workloads demand a pure approach, rather than a hybridized one. There is no significant difference between MPI and OpenMP when aiming at simulations of more than 10^5 neurons. We discern a slight inclination for MPI to outperform OpenMP when the network reaches the 1 million neurons barrier.

5 Vectorization Study

In order to exploit the Phi device to its fullest potential, extensive micro-optimizations, as well as code transformations are needed. Section 4 showed clearly that the Phi's performance is inadequate without spending development time in optimizing the codebase. The initial study of the un-optimized code presented evidence that the OpenMP programming paradigm provides the most efficient porting solution. Thus, the efforts of drawing the platform's resources were focused on the shared-memory version of the simulator. This decision was further reinforced by the fact that the MPI message-exchanging functions are incapable of using the platform's Vector Processing Units (VPUs) for acceleration.

5.1 Basics of Vectorization

VPUs are units that enable fine-grain parallelism and are present on Intel's Xeon architecture - both on Xeon processors and Phi cards. At their core, VPUs are registers that allow the execution of a specific instruction set, named Advanced Vector Extensions (AVX) [40], which is an extension to the well-known x86 instruction set. AVX originated in an older extension to the x86 instruction set designed to support Single Instruction Multiple Data (SIMD), named Streaming SIMD Extensions (SSE) [41]. AVX features multiple versions and varying width in the vectorization registers, with current Intel Phi models (Knight's Corner - KNC) supporting AVX2 and future models (i.e. Knight's Landing - KNL), supporting AVX512.

The Knight's Corner Phi that is under evaluation in this work utilizes 512-bit wide VPUs. They allow up to 16 single-precision or 8 double-precision operations to be carried out simultaneously by each of the 240 hardware threads of the device. There already are multiple case studies taking advantage of VPUs and AVX instructions to significantly boost the performance of evaluated applications from a variety of scientific fields [42], [43].

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

The developer is assisted in vectorizing his code by the compiler's optimizations, which circumvent some of these limitations. In other cases, regions of code are designated as not vectorizable, due to dependencies that cannot be avoided by the compiler in an automatic fashion. In this work, we used the Intel C Compiler (ICC) to compile our application for the Phi architecture and enabled the compiler's built-in vectorization assistance by compiling with the `-vec-report` flag for Linux Operating Systems. Furthermore, there are various documents detailing guidelines for efficient vectorization particularly on the Phi [44], [45].

5.2 Optimization Steps

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

5.2.1 User-assisted Dependency Disambiguation (DD)

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

In Figure 8, the compiler may be unable to determine whether pointers *a* and *b* refer to entirely separate memory regions, particularly if there is no pre-compiling information regarding the region sizes. However, when a developer is certain that assumed dependencies and aliasing-caused precautions can be ignored, the compiler may be instructed to override its assumptions and produce vectorized loops.

In the case of the icc compiler, this can be achieved using the `#pragma ivdep` directive, as demonstrated in algorithm 8. This is a *generic modification*; in this particular

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

Fig. 8. An example of preventing aliasing.

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

Fig. 9. Using `_mm_malloc`.

example, the developer is aware that proper coding ensures there is at least $4*upper\_bound$ bytes worth of memory space separating the values of pointers *a* and *b*; thus, there are no memory accesses in this loop for pointer *a* that could interfere with pointer *b*'s and vice versa. Declaring pointers using the `restrict` keyword is also recommended, acting as a way to communicate to the compiler that the developer guarantees exclusive memory accessing.

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

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

In order to ensure aligned memory allocations, the developer is encouraged to avoid using standard C `malloc` function calls and opt instead for Intel's `_mm_malloc`. This icc-compatible function ensures that data allocation will begin at an address that is divisible by the size of the platform's cache line (as supplied by the developer). For the Phi (KNC) architecture, data structures need to be allocated at an address that is divisible by 64, since each cache line has a size of 64 bytes, whereas the Xeon host has 32-byte alignment. An example of the function is given in Figure 9 for the Phi co-processor.

In addition, using function calls in a vectorized loop is discouraged. When vectorizing a loop, an identical instruction pattern must be maintained across all iterations so that their execution can be parallelized. Instructions that alter the flow of a program, such as conditional instructions and function calling, can pose obstacles for efficient vectorization. As such, function inlining is a practice extensively researched [46] and automatically performed by the compiler in many cases; however, in our work, manual inline expansion proved beneficial in regions of code where the compiler did not intervene.

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

Fig. 10. Nested Loop example.

5.2.3 Vectorization-Driven Loop Splitting (LS)

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

In addition, the described behaviour also forces the compiler to ignore any other instructions contained in the nested loop but outside the innermost layer. In many cases, a program's complexity is unaffected by any operations outside the inner loop; however this is not true for all algorithms. In Figure 10, an example from the InfOli simulator presented in this paper is given. The algorithm operates in two phases: for every neuron in the network, the simulator calculates the effect each synapse has on the neuron, represented in the inner loop, and then evaluates the changes in the neuron's state. While the former phase claims a large portion of the total workload, the latter phase is also important due to the number of exponential functions employed in each of the neuron-channel calculations.

In order to produce vectorized code for both phases of the algorithm, we split the simulation loop in two dedicated loops. The first is a two-layer loop for the synapse-evaluation phase whereas the second is a single-layer loop estimating the changes in the neuron's state. The two loops are then vectorized separately; this also allows for exclusive modifications in each loop's code. This *specialized transformation* is presented in Figure 11. The example contains function calls for the sake of clarity and compactness; however, as stated in Subsection 5.2.2, the functions' code has been inlined in the main loop.

5.2.4 Data Restructuring (DR)

Subsection 5.2.2 discussed the importance of matching the execution of vectorized regions of code with cache-line-aligned memory accesses in order to maximize the effectiveness of VPU usage. However, allocating data structures in aligned memory addresses does not guarantee optimal memory-access patterns. It is equally important to ensure that data is accessed in a serial, unit-stride manner across

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

Fig. 11. Split Loop example.

all iterations of the vectorized loop. Unit-stride memory references ensure memory is accessed in a sequential and continuous manner, which is important in the case of vectorized code, since memory accesses happen in parallel.

Since unit-stride memory accesses are paramount to obtaining good performance, data structures need to be designed accordingly. In Figure 12, a data structure is used that contains all relevant information for the main object under examination in this work, a neuron. While the structure presents the data in a meaningful and compact way, its usage in a vectorized algorihm proves to be problematic.

In the main loop, each of the neuron's channels is accessed and processed in a sequential manner. For the unvectorized code, data should be allocated in the memory in such a way that each neuron stores the entirety of its data, such as channel states and membrane voltage levels, as compactly as possible. In this case, the data structure presented in Figure 12 is beneficial to use. In order to generate a network of such neurons, the programmer would allocate an array of the **struct** Neuron.

However, in the case of vectorized code, the order of data accesses changes. Since there are parallel iterations of the loop, data from different neurons will be accessed simultaneously; the processor computes each of the model's parameters for the entire network in parallel. This order of memory accesses points towards storing each parameter's data from the entirety of the network as compactly as possible. In this case, a **struct**, such as Neuron, is unsuitable. In order for memory accesses to happen in unit stride, it is advisable to represent each of the model's parameters as an array that stores data for the entire network, as shown in Figure 13. These arrays can then be packed, if desired, in a different struct that represents the neuron network, rather than each neuron individually. This technique is an Array-of-Structs (AoS) to Struct-of-Arrays (SoA) *specialized transformation* and it, along with other data-structure transformations, has been extensively used in the literature, in multiple fields of HPC and SIMD computing [47], [48].

5.3 Evaluation

The simulator's codebase features a baseline version, which largely ignores the AVX instruction set due to the compiler's conservative strategy concerning assumed dependencies. The techniques mentioned in Subsection 5.2 are successively applied to this version, revealing a steady increase in the efficiency with which the simulator uses the platform's resources. The application's performance is then measured

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

Fig. 12. Data represented as a struct.

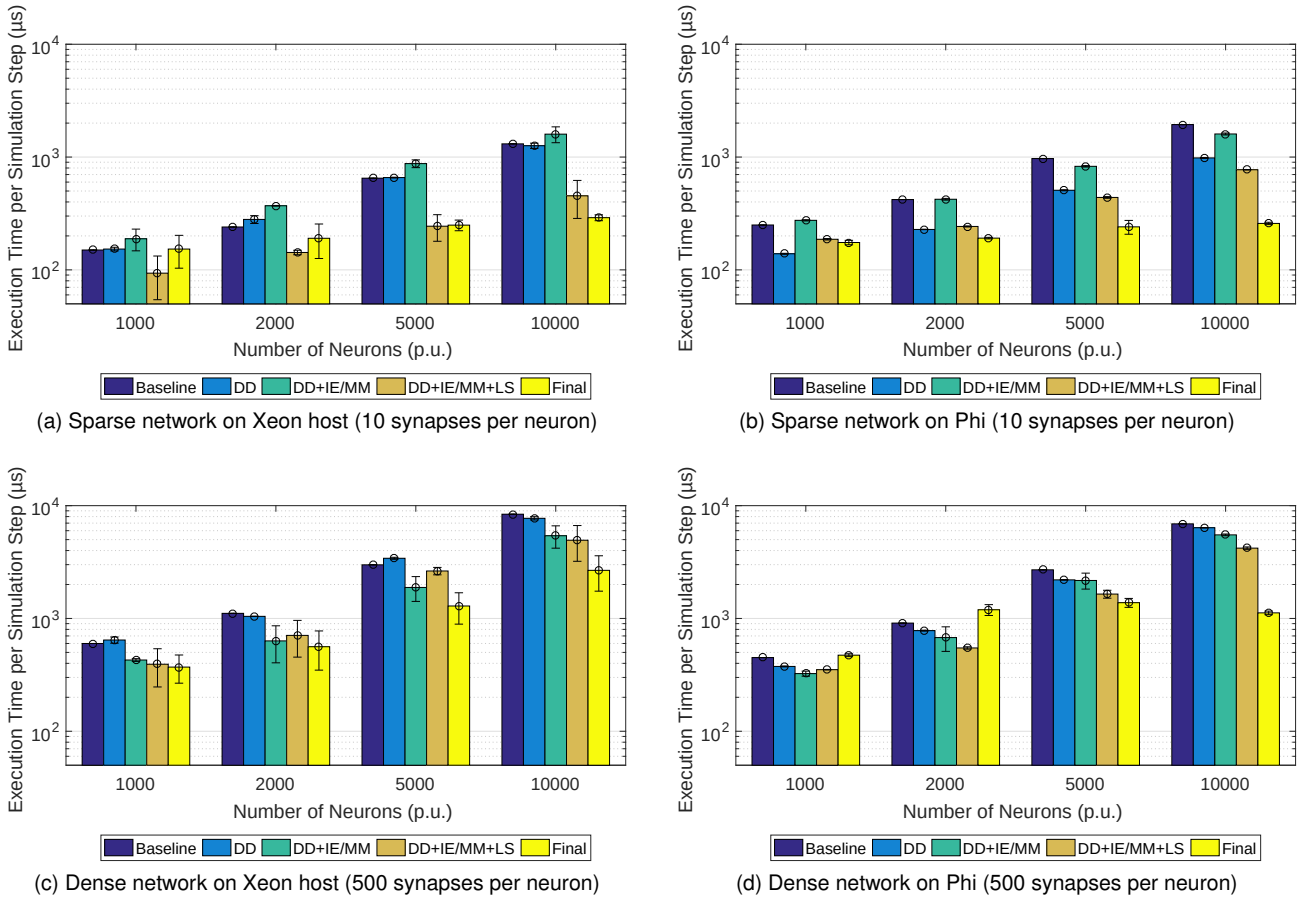
```
1: float *Na = _mm_malloc(NW_Size*sizeof(float), 64);
2: float *K = _mm_malloc(NW_Size*sizeof(float), 64);
3: ...
4: #pragma ivdep
5: for i = 0 to NW_Size do
6:    calculate_channel_Na(Na[i]);
7:    calculate_channel_K(K[i]);
8:    ...
9: end for
```

Fig. 13. Data represented as multiple arrays.

and the contribution of each modification is evaluated. In Figure 14, networks that are both sparsely and densely connected are tested, on both the host and the co-processor. Multiple measurements of each test are conducted and the mean value, along with an error margin corresponding to a confidence interval of 95%, is plotted. Different behaviour patterns are observed based on network complexity.

In the case of sparse networks, performance gains are highly dependent on the vectorization technique used, as well as the size of the network. Small and sparse networks present both platforms with a small workload. Particularly for the Xeon host, using the AVX instruction set does not guarantee a boost in performance. It is observed that techniques mentioned in Subsection 5.2, such as memory alignment and loop splitting, are mandatory in order to attain an improvement in performance; prior to applying them, small-workload-processing is not accelerated via vectorization. In this case, the Xeon host performs better without taking advantage of the VPUs.

This behaviour can be explained by reflecting on the trade-offs made when using the AVX instruction set. Compared to the scalar instruction set, the average vector instruction requires an increased amount of clock cycles until completion. The reward of using vector instructions lies in processing multiple data simultaneously. As Figure 14 demonstrates, this gain diminishes when there is a relatively small amount of neurons to be simulated per OpenMP thread. In such cases, the amount of data processed by each thread is insufficiently large to fill the VPUs. Suitable memory-alignment of aforementioned data also plays a critical role in performance. As a result, scalar instructions may outperform an improperly vectorized codebase, particularly for the Xeon host which features better scalar performance



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

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

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

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

Fig. 14. Effects of vectorization on networks of varying size and complexity.

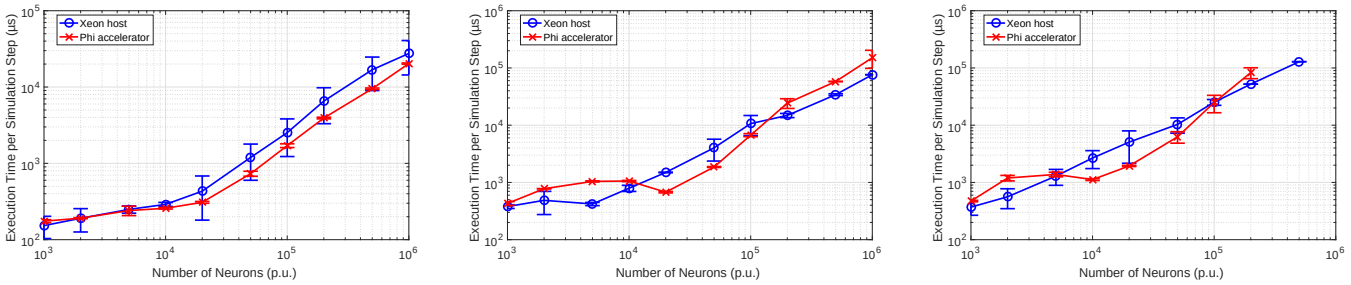
than the Phi.

Dense networks, on the other hand, provide a larger workload and thus, a better opportunity to take advantage of the platform's resources. Cases of not-fully-optimized code outperform the unvectorized codebase, even for the Xeon host. It can be assumed that, in cases where the workload is sufficiently heavy (due to the large amount of calculations required by each neuron's synapses), vector instructions are largely "safe" to use. In Figure 14d, the increase of neuron populations alters execution time only slightly for properly-vectorized code. This is an indication that, with proper manual vectorization, the accelerator can utilize its assets efficiently and can handle increases in workload well, until the entirety of its computational resource pool is expended. Thus, vectorization can potentially yield significant boosts in performance, with larger benefits observed for the Phi accelerator due to a larger amount of available resources.

Figure 15 evaluates the properly-vectorized code when solving for scaled-up networks, on both platforms. In Figure 15a, there is a small and stable performance gap between the Phi device and the Xeon host. The accelerator outperforms the host in a predictable manner. On the contrary, denser networks in Figures 15b and 15c depict a more complicated behaviour. There is a range of neuron populations where the Phi accelerator outperforms the host. Furthermore, as network size increases, approaching populations of realistic, human inferior olivary nucleus' numbers, the performance gap between the two computing fabrics diminishes. For larger populations, the Xeon host may outperform the Phi.

These observations can be justified by the fact that network density is ultimately closely correlated to how parallelizable the code is. From a programmer's point of view, the synapse count per neuron signifies the degree by which the simulator will differ from an embarassingly parallel application. The neuron's dendritic compartment forces OpenMP threads to sync due to shared-memory accesses and MPI ranks to exchange data via messages. In addition, it imposes irregularities in memory access patterns due to the fact that the network connectivity matrix is unknown before the simulation begins and thus, data required by each neuron cannot be stored in a sequential, unit-stride manner. This holds especially true for simulations that would require the connectivity matrix to be constantly changing during a run, in order to study the constantly changing connections forming in the human brain. Non-unit stride accesses have an adverse effect on the efficiency of vectorization, as presented in Subsection 5.2.4.

The results illustrated in Figure 15 can, then, be traced back to the model's shifting behaviour based on network connectivity. Sparse networks can be parallelized and vectorized efficiently; the Phi accelerator will outperform the Xeon host due to an increased amount of available resources, for any non-trivial neuron population. Dense networks, on the other hand, cannot be accelerated as efficiently. The Xeon host is the superior platform for small net-



(a) Networks of 10 Synapses/Neuron (b) Networks of 100 Synapses/Neuron (c) Networks of 500 Synapses/Neuron

Fig. 15. Scaling up the best and properly vectorized implementation on the host and the accelerator.

works of high connectivity due to its better single-threaded performance, as well as the fact that small networks present less opportunities for the Phi accelerator to utilize its available threads and larger VPUs. As the network size increases, the accelerator can use more of its assets. A point is reached where the Phi outperforms the host by meeting workload demands with aggressive usage of its computational assets. Once the Phi's computational resources are working at maximum capacity, a saturation point is reached; in Figure 15c, the performance gap between the two platform gradually narrows for populations beyond $10,000$ neurons, whereas this point is reached at $20,000$ neurons in Figure 15b. From then on, the Xeon host's superior single-threaded performance handles the application in a better manner, allowing it to outperform the accelerator for human inferior-olive numbers ($\geq 100,000$).

6 Conclusion

In this work, we have ported a biophysically accurate simulator of the inferior olivary nucleus on a single-node Xeon/Xeon Phi system. The selected InfOli simulator serves as a significant benchmark for parallelization and scaling of biologically-plausible neuron modeling workloads. We have presented and evaluated three native implementations on the target system: an MPI-based, an OpenMP-based and a combination of both.

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

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

The shared-memory implementation is manually tuned for the underlying platforms. A combination of pragma directives, function inlining and specific memory allocation functions, specialized for cache line alignment, is initially used. These techniques are applicable to any codebase and form the basis of vectorizing any application. Furthermore, modifications that are specifically designed for the simulator's algorithm, are employed. As a result, a sizeable increase in attainable simulation speed was achieved for workload sizes that are eligible for vectorization. Thus, we encourage the usage of these optimizations on codebases that resemble the algorithm, connectivity patterns and problem sizes encountered in the InfOli modeling application.

Overall, the techniques presented in this paper were beneficial for both the accelerator and the host. In particular, for networks that are large and densely-connected enough to saturate the Phi's assets, the difference in performance between manually vectorized code and un-optimized code that relies solely on the compiler is an order of magnitude.

After fine-tuning the application, the platforms perform differently depending on network connectivity density. Sparse networks are a good candidate for acceleration via the Phi co-processor's large pool of computation resources. Furthermore, dense networks feature a range of populations between $5,000$ and $50,000$ neurons where the co-processor can use its computational resources to outperform the host. On the other hand, the host's focus on single-threaded and scalar performance is a better fit for dense networks outside this range due to their less well-parallelizable nature.

Acknowledgments

This work is partially supported by European Commission project H2020–687628–VINEYARD. The STFC Hartree Centre (UK) is acknowledged for computational resources.

References

[1] Bhuiyan, M. et al., "Acceleration of spiking neural networks in emerging multi-core and gpu architectures," in *IPDPSW*, 2010.
[2] Nguyen, H. A. Du et al., "Accelerating complex brain-model simulations on gpu platforms," in *DATE*, 2015, pp. 974–979.
[3] O. Trott and A. J. Olson, "AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *J. of Comp. Chemistry*, 2010.
[4] Plesser, H. E. et al., "Nest: the neural simulation tool," *Enc. of Comp. Neuroscience*, pp. 1849–1852, 2015.

[5] G. Chatzikonstantis *et al.*, "First impressions from detailed brain model simulations on a xeon/xeon-phi node," in *ACM Computing Frontiers*, 2016.

[6] A. L. Hodgkin and A. F. Huxley, "Propagation of electrical signals along giant nerve fibres," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 140, no. 899, pp. 177–183, 1952.

[7] Fang, J. et al., "Test-driving intel xeon phi," in *ICPE*, 2014.

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

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

[10] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *IEEE CSE*, vol. 5, no. 1, pp. 46–55.

[11] "CUDA C Programming Guide," NVIDIA Corporation, Tech. Rep. PG-02829-001_v7.5, 2015.

[12] R. D. Monagle and H. Brody, "The effects of age upon the main nucleus of the inferior olive in the human," *Journal of Comparative Neurology*, vol. 155, no. 1, pp. 61–66, 1974.

[13] https://github.com/GeorgeChatzikonstantis/InfOliFull.

[14] M. L. Hines and N. T. Carnevale, "The NEURON simulation environment," *Neural computation*, vol. 9, no. 6, 1997.

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

[16] Brette, R. et al., "Simulation of networks of spiking neurons: A review of tools and strategies," *J. of Comp. Neuroscience*, 2007.

[17] Y.-H. Liu and X.-J. Wang, "Spike-frequency adaptation of a generalized leaky integrate-and-fire model neuron," *Journal of Comp. Neuroscience*, vol. 10, no. 1, pp. 25–45, 2001.

[18] Chacron, M. J. et al., "Interspike interval correlations, memory, adaptation, and refractoriness in a leaky integrate-and-fire model with threshold fatigue," *Neural Comput.*, pp. 253–278, 2003.

[19] Brette, R. and Gerstner, W., "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *Journal of Neurophysiology*, vol. 94, no. 5, 2005.

[20] W. E. Sherwood, "Fitzhugh–nagumo model," in *Enc. of Comp. Neuroscience*, D. Jaeger and R. Jung, Eds. Springer, 2014, pp. 1–11.

[21] Izhikevich, E. M. et al., "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

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

[23] Rodopoulos, D. et al., "Optimal mapping of inferior olive neuron simulations on the single-chip cloud computer," in *SAMOS*, 2014.

[24] Kunkel, S. et al., "Spiking network simulation code for petascale computers," *Frontiers in Neuroinf.*, 2014.

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

[26] Choi, J. et al., "Implementation of hardware model for spiking neural network," in *ICAI*, 2015, p. 700.

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

[28] R. Ananthanarayanan *et al.*, "The cat is out of the bag: cortical simulations with 109 neurons, 1013 synapses," in *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[29] G. Florimbi *et al.*, "The human brain project: Parallel technologies for biologically accurate simulation of granule cells," *Microprocessors and Microsystems*, 2016.

[30] De Zeeuw, C. I. et al., "Microcircuitry and function of the inferior olive," *Trends in neurosciences*, vol. 21, no. 9, pp. 391–400, 1998.

[31] J. R. De Gruijl *et al.*, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," 2012.

[32] C. I. De Zeeuw *et al.*, "Spatiotemporal firing patterns in the cerebellum," *Nature Reviews Neuroscience*, vol. 12, no. 6, 2011.

[33] W. H. Press *et al.*, *Numerical recipes in C*. Cambridge university press Cambridge, 1996, vol. 2.

[34] Wu, Q. et al., "Development of fpga toolbox for implementation of spiking neural networks," in *CSNT*, 2015, pp. 806–810.

[35] K. O. W. Group *et al.*, "The OpenCL specification," 2008.

[36] Lubin, M. et al., "Efficient Software Development: 4 What's New in Intel® Parallel Studio XE 2013 Service Pack," 2013.

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

[38] L. Dalcín *et al.*, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

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

[40] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, 2011.

[41] P. PENTIUM III, "Implementing streaming simd," 2000.

[42] A. Tanikawa *et al.*, "N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions," *New Astronomy*, 2012.

[43] S. J. Pennycook *et al.*, "Exploring simd for molecular dynamics, using intel® xeon® processors and intel® xeon phi coprocessors," in *International Symposium on Parallel & Distributed Processing*, 2013.

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

[45] M. Barth *et al.*, "Best practice guide intel xeon phi v1." 2013.

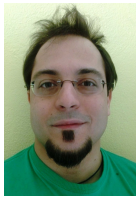
[46] S. P. Jones and S. Marlow, "Secrets of the glasgow haskell compiler inliner," *Journal of Functional Programming*, vol. 12, no. 4-5, 2002.

[47] L. Papadopoulos *et al.*, "Exploration methodology of dynamic data structures in multimedia and network applications for embedded platforms," *Journal of Systems Architecture*, vol. 54, no. 11, 2008.

[48] T. Hussain *et al.*, "Reconfigurable memory controller with programmable pattern support," *HiPEAC WRC*, vol. 67, p. 95, 2011.

George Chatzikonstantis obtained his Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2013. His research interests focus on high-performance computing, multi-core/single-chip multiprocessors and bioinformatics. He is currently conducting research on neuromodeling applications in high-performance computing fabrics as a Ph.D. student in the National Technical University of Athens.

Dimitrios Rodopoulos received his Ph.D. Degree in Computer Science from the National Technical University of Athens and completed his Thesis in imec, Belgium in 2016. His research interests include deca-nanometer device degradation phenomena and mitigation techniques against time-zero/-dependent device variability. Currently, he serves as RnD Engineer at imec, Belgium.



Christos Strydis obtained his Ph.D. Degree in Computer Engineering from the Delft University of Technology in 2011. His interests revolve around the topics of high-performance computational neuroscience and of next-generation implantable medical devices. Currently, he is an assistant professor with the Neuroscience department of the Erasmus Medical Center, the Netherlands, and is also a chief engineer with Neurasmus BV, the Netherlands.

Chris I. De Zeeuw received his PhD with a focus in brain and behavior in 1990 and his MD in 1991 from Erasmus University Rotterdam. He focuses on the nerve cells in the cerebellum responsible for learning and the effect of their electrical activity on movement. He is currently the director of Neurasmus BV, the Chairman of the Department of Neuroscience at Erasmus MC Rotterdam and the Project Director at the Netherlands Institute for Neuroscience in Amsterdam.

Dimitrios Soudris received his Ph.D. Degree in Electrical Engineering from the University of Patras in 1992. His research interests include embedded systems design, reconfigurable architectures, reliability and low power VLSI design. He is currently working as Associate Professor in School of Electrical and Computer Engineering, Dept. Computer Science of National Technical University of Athens, Greece.