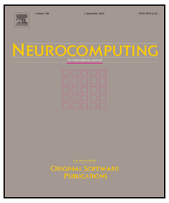
Neurocomputing 598 (2024) 127953

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom



Tricking AI chips into simulating the human brain: A detailed performance analysis

Lennart P.L. Landsmeer [a,b,*], Max C.W. Engelen [c,b], Rene Miedema [b], Christos Strydis [b,a]

[a] Quantum & Computer Engineering Department, Delft University of Technology, Delft, The Netherlands
[b] Department of Neuroscience, Erasmus Medical Center, Rotterdam, The Netherlands
[c] Maxeler IoT-Labs BV, Delft, The Netherlands

ARTICLE INFO

Communicated by X. Li

Keywords:
AI accelerator
GPU
Brain simulation
computer architecture

ABSTRACT

In recent years, significant strides in Artificial Intelligence (AI) have led to various practical applications, primarily centered around training and deployment of deep neural networks (DNNs). These applications, however, require considerable computational resources, predominantly reliant on modern Graphics-Processing Units (GPUs). Yet, the quest for larger and faster DNNs has spurred the creation of specialized AI chips and efficient Machine-Learning (ML) software tools like TensorFlow and PyTorch have been developed for striking a balance between usability and performance. Simultaneously, the field of computational neuroscience shares a similar quest for increased computational power to simulate more extensive and detailed brain models, while also keeping usability high. Although GPUs have also entered this field, programming complexity remains high, resulting in cumbersome simulations. Inspired by AI progress, we introduce a workflow for easily accelerating brain simulations using TensorFlow and evaluate the performance of various, cutting-edge AI chips – including the Graphcore Intelligence-Processing Unit (IPU), GroqChip, Nvidia GPU with Tensor Cores, and Google Tensor-Processing Unit (TPU) – when simulating a biologically detailed as well as simpler brain models. Our model simulations explore the architectural tradeoffs of a modern-day CPU and these four AI platforms by varying computational density, memory requirements and floating-point numerical accuracy. Results show that the GroqChip achieves the best performance for small networks, yet is unable to simulate large-scale networks. At the scale of mammalian brains, the GPU, IPU and TPU achieve speedups ranging from 29x to 1,208x times over CPU runtimes. Remarkably, the TPU sets a new record for the largest, real-time simulation of the inferior-olivary nucleus in the brain. Reduced-accuracy floating-point implementations make some simulation results unreliable for brain research, notably for the GroqChip. Consequently, this work underscores the potential of ML libraries for accelerating brain simulations as well as the critical role of AI-chip numerical accuracy for biophysically realistic brain models.

1. Introduction

To date, Graphics-Processing Units (GPUs) have achieved spectacularly better performance in deep learning (DL) than Central-Processing Units (CPUs) [1]. Recently, novel, specialized Artificial-Intelligence (AI) hardware platforms have begun to emerge, holding the promise of accelerating training and inference even further. The workloads targeted mainly are artificial, and specifically, deep neural networks (DNNs), which have shown great potential in recent years. On the other hand, highly biologically plausible brain models such as conductance-based (e.g., Hodgkin-Huxley) neurons have not attracted similar attention from AI-chip manufacturers and analysts alike. Nonetheless, simulating these biological brain models is just as valuable. It improves our knowledge of the brain and can lead to advancements in treating brain disorders and, additionally, catalyze the development of more biologically realistic AI models [2,3]. Therefore, it is logical – for neuroscientists and AI researchers alike – to reach for these AI accelerators and deploy them in brain simulations. Yet, at the time of this writing, no feasibility and performance studies exist.

In this work, we evaluate multiple, cutting-edge AI chips – Graphcore Intelligence-Processing Unit (IPU) [4], GroqChip [5], TensorRT-capable Graphics-Processing Unit (GPU) [6] and Google Tensor-Processing Unit (TPU) v3 [7] – when simulating a highly biologically detailed model of a brain region, the Inferior-Olivary nucleus (IO). Biologically detailed brain models, such as the IO, chiefly involve

[*] Corresponding author at: Quantum & Computer Engineering Department, Delft University of Technology, Delft, The Netherlands.
E-mail address: l.p.l.landsmeer@tudelft.nl (L.P.L. Landsmeer).

https://doi.org/10.1016/j.neucom.2024.127953
Received 26 September 2023; Received in revised form 15 February 2024; Accepted 29 May 2024
Available online 7 June 2024

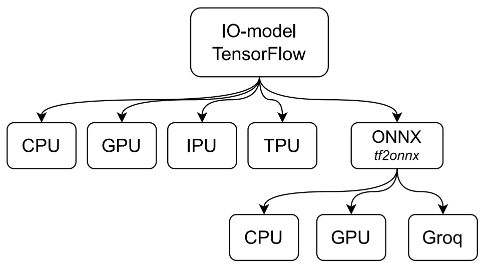


Fig. 1. Overview of the performance-analysis strategy followed in this work.

addition, multiplication, division and exponential operations, arranged as sparse computations. There is, thus, a large operation overlap with artificial networks. Therefore, new AI chips seem a good fit for these types of models. This IO-model simulation represents timely, relevant research and is constructed as an extended-Hodgkin-Huxley (eHH) model. It is very suitable for stress-testing the different AI platforms and highlighting architectural tradeoffs by adjusting the compute density, memory requirements and numerical accuracy of the model. For completeness, we also simulate the simpler but very common Leaky Integrate & Fire (LIF) and Hodgkin-Huxley (HH) models.

Evaluation is performed using the model encoded as a TensorFlow 2 [8] kernel. TensorFlow is a high-level Application-Programming Interface (API), requiring little to moderate intervention from the user. Consequently, TensorFlow is suitable for a wide user base. This is important for neuroscientists as it allows them to benefit from hardware innovations without expert knowledge. For a fair comparison in the analysis of the different accelerators, the exact same TensorFlow model is used. However, not all AI chips – more specifically the GroqChip – support TensorFlow. Therefore, in such cases the exact same model is compiled to its ONNX [9] equivalent, using the Python package `tf2onnx`. ONNX is an intermediary tool used to convert models between different Machine-Learning (ML) frameworks. A schematic overview of this strategy is shown in Fig. 1.

While all accelerators in this study support chip-to-chip communication, this work focuses on single-chip performance comparisons; multi-chip deployment is left as future work. The contributions of this work are as follows:

- We explore the suitability of four cutting-edge AI architectures fo simulating biologically plausible spiking neural networks (SNNs).
- We build the first ML-library-based, efficient implementation of a detailed brain model, the Inferior Olive (IO).
- We deploy the IO as well as less complex models onto the four AI platforms and benchmark their performance and numerical accuracy.
- We demonstrate that modern ML libraries are semantically able to model classical problems in scientific computing, offering large performance gains and reduced development times while remaining hardware-agnostic.
- Lastly, this work is the first to ever simulate a realistic mouse-sized IO model with real-time performance.

The paper is organized as follows: Section 2 presents related works in the field. Section 3 introduces the IO model used as our main benchmarking application, while Section 4 briefly presents the four AI architectures under evaluation and attempts some performance predictions. Section 5 ensures experiment reproducibility by detailing the experiment parameters and platform configurations used to acquire our results presented in Section 6. A general discussion of our findings is included in Section 7. Section 8 concludes this work.

2. Related works

Models of biological neurons come in various levels of detail, ranging from population-level dynamics, from simplified models of single neurons to highly detailed biophysically realistic neurons [10]. Coarse models of single neurons, notably LIF-type models have seen a renewed interest in the DL-community (often referred to as SNNs, here) as an alternative to artificial neural networks (ANNs) [11].

Simplified SNN models have readily available general-purpose GPU (GP-GPU) implementations of LIF and similar models as well. High-level ML libraries like TensorFlow and PyTorch have allowed for the *hardware-agnostic* implementation of their neural dynamics, considerably lowering development efforts to build SNN simulators on GP-GPUs. For example, Nengo DL [12] allows for the GPU-based simulation of existing SNN models defined in the Nengo framework using TensorFlow. Beyond just simulating neural networks on the GPU, novel developments in surrogate gradients for event-based SNNs and automatic gradient calculation provided by ML libraries allowed for the nearly simultaneous appearance of similar SNN deep-learning libraries spyTorch [13], Norse [14], snnTorch [15] and SpikingJelly [16]. BindsNET [17] is another, efficient SNN implementation in PyTorch with a focus on reinforcement learning. Furthermore, simplified SNN models led to the development of specialized "neuromorphic" (or AI) hardware to simulate them, demonstrating promising improvements in energy efficiency and performance. Numerous publications show the benefits of using these chips for simplified SNN simulations. Examples are Loihi 2 [18], TrueNorth [19] and NeuroEngine [20]; for a concise review, we point the reader to [21]. However, these chips do not support conductance-based brain models and, thus, are not suitable for highly biologically plausible brain models, our models of interest.

The ML libraries show the expressive power and performance required to run large-scale SNN models. Moreover, ML-library-based models arguably can be developed faster than hardware-specific, low-level code. However, all these efforts only focus on relatively simple neural models and consequently do not meet the requirements of realistic brain simulations. As computational neuroscience is usually interested in biophysically accurate models that model the underlying biological processes in a way that makes it possible to gain insights about these processes. These conductance-based models can be made more realistic by modeling their 3-D structure (the morphology) using multiple discretized compartments. Multi-compartmental, conductance-based neurons are then simulated by explicit calculation of electrical currents flowing within, between and into discretized compartments [22].

Due to the computational resources needed for large-scale, conductance-level brain simulations, computational neuroscience was an early adopter of GP-GPUs in the High-Performance Computing (HPC) environment. Notable GPU-based examples of large-scale, biologically detailed brain simulators include CoreNeuron [23], which enabled porting of existing conductance-level NEURON [24] models to the GPU, and more recently, Arbor [25], a library-based approach to performance-portable, large-scale brain simulation. Their success shows that the computational problems of neuroscience map well to GP-GPU platforms and result in significant speedups for large-scale brain models. Still, even with hand-optimized CUDA code [26], the IO-model simulation (to be detailed in the next section) at biological sizes runs orders-of-magnitude slower than the biological brain, hampering research.

In addition to GPUs, Field-Programmable Gate Arrays (FPGAs) offer an alternative for brain simulations. At the cost of time-consuming hardware synthesis, FPGAs offer high performance with a relatively low energy consumption. Previous research has largely been limited to simulating relatively simple SNNs, which lack the biological detail required for our studies. Notable projects are [27–31]. Standing out among other FPGA implementations, flexHH [32] is a library which supports the higher level of biological detail required. However, the library is unable to simulate at the size and speed of the biological brain. Moreover, the programming of FPGAs has a steep learning curve. Coupled with their lack of seamless integration with TensorFlow, we decided to exclude FPGAs from our evaluation. Their limitation in usability contrasts with the accessible high-level TensorFlow API, a feature that greatly benefits the user experience on the selected AI platforms. With respect to TensorFlow-based implementations of conductance-level models, there is PymoNNto [33], an attempt to bring the Brian [34] API of neural models to TensorFlow. While faster than the Brian simulator on a GTX1080 GPU, performance was not a primary goal and the architecture prohibits optimizations using TensorFlow's JIT compiler backend, by scattering the computational definitions across the codebase. Although this shows that TensorFlow *does* express the right API surface for neural models, no efficient ML-library-based, conductance-level, GP-GPU simulators exist.

On AI chips that have the semantic power to capture more general HPC workloads, little has been published about both simplified and conductance-level SNNs. With respect to simplified SNN simulations, we found just one preprint targeting an AI chip, introducing an IPU-optimized version of snnTorch [35]. Training throughput of a dense 3-layer LIF network on an image-classification task is 3.4x higher on the IPU than on the A100. The reported performance benefits decrease if the network size is increased, with the A100 being apparently underutilized throughout the entire application. This shows the potential of using the IPU for simple SNN workloads but the performance characteristics of other AI chips or more complex SNNs are not yet obvious.

No works have been published targeting AI chips with conductance-level models or other biologically realistic brain-simulation scenarios, neither using high-level ML libraries or hardware-specific Software-Development Kits (SDKs). To the authors' knowledge, this is the first work to implement an efficient, conductance-level, multi-compartmental neuron in an ML library and also the first to benchmark multiple AI chips on this workload class.

3. The inferior-olive application

The IO is a brain region located in the brainstem and is key to motor control and learning [36]. The estimated neuron population in the mouse brain is approx. 10^4 neurons [37] and in humans between 10^6 – 10^7 neurons [38]. These numbers will be referred to during hardware-performance evaluation (Section 6). In this work, we will capture in TensorFlow two the IO nuclei as an eHH model, first published in [39]. The model is a good example of the computational load of realistic brain models and, also, a good fit for our benchmarking purposes, since it captures complex neuron dynamics and fast interneural communication (in the form of gap junctions), as will be shown next.

We restate the IO-neuron main equations in this section, but refer the reader to [39] for more details. In addition, we model connectivity based on the network described in [40].

3.1. The cable model

$$C_m \frac{dV^{(i)}}{dt} = -\sum_{k\in\text{Channels}} I_k^{(i)} - \sum_{j\in\text{Compartments}} I_{i,j} - \sum_{j\in\text{Gap junctions}} I_{gj,i,j} - I_{\text{app}}^{(i)} \tag{1}$$

The eHH model describes the membrane that envelops the neurons as a capacitor. The cell internal voltage can thus be calculated by integrating currents flowing into and out of the cell (Eq. (1)), detailed in the following sections. Finally, I_{app} is an optional term describing externally applied currents by the experimenter.

Listing 1: Axonal sodium-channel current

```
m_inf = 1/(1+tf.exp(−(V_axon+30)/5.5))
h_inf = 1/(1+tf.exp((V_axon+60)/5.8))
tau_h = 1.5*tf.exp(−(V_axon+40)/33)
dh_dt = (h_inf−h)/tau_h
I_na  = g_Na*(V_axon−V_Na)*m_inf**3*h
```

Listing 2: Sparse gap-junction current

```
Vdiff    = tf.gather(V_dend, gj_src) \
         − tf.gather(V_dend, gj_tgt)
I_per_gj = Vdiff * g_gj * (0.2 + \
              0.8 * tf.exp(−0.01*Vdiff*Vdiff))
I_gapp   = tf.tensor_scatter_nd_add(
              tf.zeros_like(V),
              tf.reshape(gj_tgt, (−1,1)), I_per_gj)
```

3.2. Channel currents

Channels (CaL, h, KCa, Na, Kdr, K, CaH, Na, K) allow currents to flow through the cell membrane. They produce this current as a function of internal state variables changing over time. In general, this current (Eq. (2)) results from the potential difference to a channel-specific reversal potential E multiplied by the product of one or more internal gating variables, each optionally raised to an integer power (Eq. (2)). $\bar{g}$ transforms the gated potential-difference into a current. The gating variables follow an Ordinary Differential Equation (ODE) that brings them to a certain cell voltage-dependent steady state $n^{\infty}\left(V^{(i)}\right)$ at a given speed $1/\tau_n$ (Eq. (3)). These latter equations are usually gaussian or sigmoidal functions of the voltage. For certain, fast-operating channels we set $n(t) = n^{\infty}(V)$ as a numerical-stability optimization.

$$I_j^{(i)} = \bar{g}_j \left[\prod_k n_{j,k}(t)^{m_k}\right] \left(V^{(i)} - E_j\right) \tag{2}$$

$$\tau_n\left(V^{(i)}\right) \frac{dn}{dt} = n^{\infty}\left(V^{(i)}\right) - n(t) \tag{3}$$

3.3. Compartmental currents

A single IO cell consists of three separate compartments: the axon, soma and dendrite. Currents flowing between different compartments are modeled resistively as: $I_{i,j} = g_{i,j}\left(V_j - V_i\right)$.

3.4. Gap-junction currents

Gap junctions are direct electrical connections between different IO cells and allow current to flow between them. They follow experimentally determined Connexin-36 protein dynamics:

$$I_{gj,i,j} = g_{gj}\Delta V_{i,j}\left[0.2 + 0.8\exp\left(-\Delta V_{i,j}{}^2/100\right)\right] \tag{4}$$

with $\Delta V_{i,j}$ being the potential difference between two connected cells.

3.5. Topology

The real IO looks like a large, folded sheet with mostly local connectivity [36]. As approximating this structure is not a focus of this paper; our model neurons are assumed to exist on a discrete 3D grid with wrap-around connectivity (i.e., a hypertorus). This should exhibit the same non-local memory-access patterns as a more realistic model. Connections are sampled as a function of inter-neuron distance r on a radially symmetric distribution: $p(r) \propto u(r_{max} - r)(e^{-r^2} - e^{-r_{max}^2})n(r)$, where $n(r)$ is the density of neurons in the volume shell around r. This distribution is sampled until we have 10 connections per neuron on average.

Table 1
Technical specifications of AI chips used in this work.

Device software	Memory	Base/boost freq. TDP	Process node Transistor cnt.
AMD 3955WX CPU[*] TF 2.11.0	128 GB DDR4	3900-4300 MHz 280 W	7 nm 19.94 Bn
GroqChip TSP Groq SDK 0.9.1[***]	220 MiB on-chip	900 MHz –	14 nm 26.8 Bn
Nvidia A100 GPU TF 2.11.0	80 GB HBM2e	1275–1410 MHz 400 W	7 nm 54.2 Bn
Graphcore IPU (GC200)[**] TF IPU 2.6.3+gc3.0.0	900 MB on-chip	1330 MHz 185 W	7 nm 59.4 Bn
Google TPUv3 TF 2.11.0	32 GiB HBM	940 450 W	16 nm (est.) 11 Bn

* AMD Ryzen Threadripper PRO 3955WX (16-Core).
** Single M2000 in IPU-POD16 (with 4 GC200 chips).
*** TF2ONNX 1.13.0 and ONNX opset 16.

3.6. *TensorFlow translation*

The previous equations sum up to a total of 14 ODEs per neuron. This system of ODEs is translated into a series of TensorFlow operators in Python. By defining the model in TensorFlow instead of using platform-specific APIs we make sure that all platforms have equal optimization opportunities. Furthermore, TensorFlow naturally translates to ONNX models, which is the only high-level API available to the GroqChip at the moment. Straightforward translation to TensorFlow is achieved by storing all state in a large 2D-array and direct substitution of mathematical expressions by their TensorFlow counterparts (see Listing 1). When certain model parameters need to be user-specified (e.g., g_j or I_{app}), they are passed to the TensorFlow kernel, which then needs to be recompiled before running again.

As already mentioned, IO neurons communicate via gap junctions. Translating gap junctions to both TensorFlow and ONNX in a performant way requires expressing them as vector operations, as opposed to more traditional `for-loop`-based approaches [26]. With just 10 connections per IO neuron on average, cell-to-cell communication is sparse. The effective operation from a TensorFlow perspective is two sparse-matrix (SpMV) multiplications. As a novel contribution in computational neuroscience, we model those as `tf.gather` and `tf.tensor_scatter_nd_add` operations (see Listing 2). Apart from being more specific and memory-efficient in describing SpMV multiplications, these functions have a direct mapping to ONNX operators as *Gather* and *ScatterND* since ONNX specification `opset 11`, contrary to SpMV multiplications which currently are not possible in ONNX.

At each timestep, ODEs are integrated using Forward-Euler to produce the next state array, using a *hardware-agnostic* timestepping function that is kept the same for all platforms. For TensorFlow backends, a Just-in-Time (JIT)-compilable TensorFlow function is constructed that executes 40 timesteps at a Δt of 0.025ms, resulting in a 1ms sampling accuracy. For ONNX backends, the timestep function is converted to an ONNX model and either the public onnx-runtime library or Groq compiler is used to compile this into executable code. Direct execution of this device code does not lead to the best possible performance by default, mostly relating to redundant device-host data copies and call-overhead. Corresponding hardware-specific optimizations are discussed in Section 5.2.

4. Target platforms

Hardware platforms were selected from the top-performing AI accelerators in the MLCommons MLPerf training benchmark v2.0 [43]. From this, the Intel Habana Gaudi was not available to us. The GroqChip was included as it was already available through academic channels. An overview of all AI chips is given in Table 1 and they will be presented next. A modern, server-grade CPU is also included as a baseline for our subsequent performance and numerical-accuracy comparisons.

Table 2
Breakdown of all arithmetic operations used in neural models on NeuroML-db [41]. Cell model: Number of cell models (with multiple channels) containing at least one such operation over all single-cell models, calculated from EDEN's [42] intermediate representation. Ion channel: Mean+std arithmetic-operation counts for a single-channel integration step on one single compartment over 146 unique ion channels measured using Arbor's [25] `modcc` tool.

Operation	Cell model (%)	Ion channel (#ops)
add/sub	100	13.5 ± 8.0
neg	100	1.5 ± 0.6
mul	100	20.0 ± 11.1
div	100	10.5 ± 5.2
exp	97.5	3.0 ± 2.1
log	96.5	0.6 ± 0.8

All selected platforms implement the mathematical operations common to neuroscience models (Table 2) and also support the minimum 32-bit floating-point resolution required for neuroscience simulations. The CPU and GPU support 64-bit floating-point arithmetic, as well. Note that floating-point support does not necessarily mean IEEE754 compliance, and post-hoc numerical validation of simulation results will be required, after performance evaluation.

NVIDIA GPUs [6] are well-established in the HPC world because of their excellent performance on parallel problems. This is primarily the consequence of the architecture of a GPU, which consist of multiple Streaming Multiprocessors (SMs), each containing multiple cores. Within each SM, threads are grouped into so-called warps. A warp is a set of threads that execute simultaneously. The architecture of a SM is shown in Fig. 2. Consequently, a GPU contains thousands of cores capable of concurrent processing. A prerequisite is that multiple cores execute the same instructions on different data. This is known as the Single-Instruction Multiple-Data (SIMD) paradigm. Next to the conventional cores, newer generation of GPUs contain Tensor Cores. These Tensor Cores are specialized hardware to efficiently perform matrix multiplications. Part of the efficiency can be obtained using the TensorFloat −32 (TF) data type. A TF32 variable is a floating-point number with the dynamic range of float 32 variable range but the accuracy of float 16 variable.

4.1. *Nvidia GPU*

In addition to computational resources, memory management is critical for the performance on a GPU. A GPU utilizes a hierarchical memory architecture, consisting of different levels of memory including registers, cache memory and RAM. In the hierarchy, the smallest memory (registers) is the fastest and each subsequent level is larger but slower. Therefore, memory management is crucial for the performance

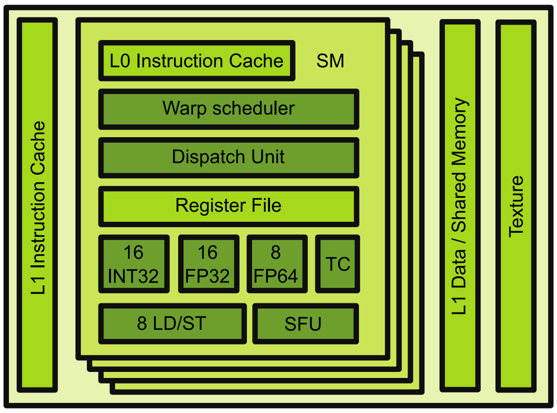


Fig. 2. Architectural overview of a modern-day GPU [6]. SM: Streaming Multiprocessor, TC: TensorCore, SFU: Special Function Unit.

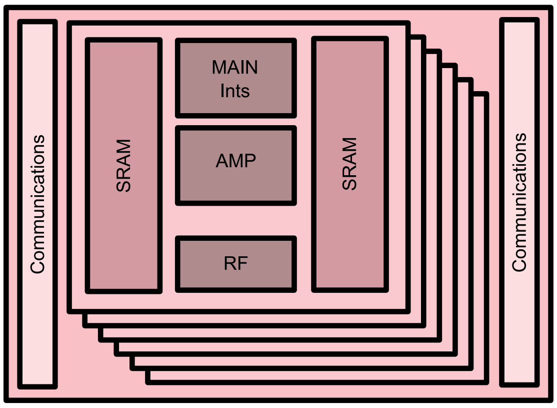


Fig. 4. Architectural overview of the GraphCore IPU MK2 [45]. AMP: Accumulating Matrix Product, RF: Register File.

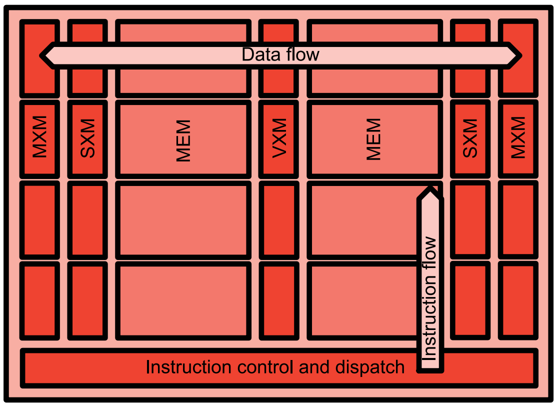


Fig. 3. Architectural overview of the first-generation GroqChip [5]. VXM: Vector Execution Module, MXM: Matrix Execution Module, SXM: Switch Execution Module, MEM: MEmory Module.

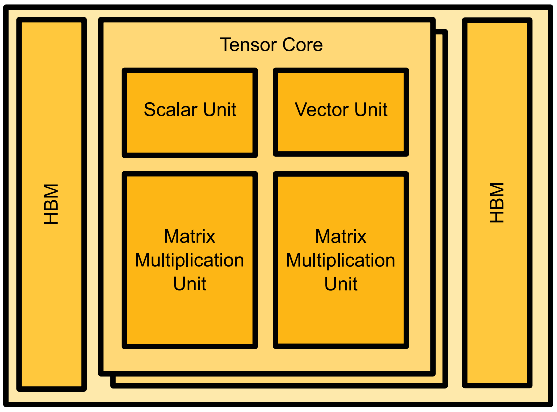


Fig. 5. Architectural overview of the TPU v3 [46]. HBM: High-Bandwith Memory.

of GPU (e.g., frequently used data is placed in registers/cache and the access to RAM is done by coalesced accessed).

For inter-thread communication within the same GPU, shared and global memory can be used on the same GPU. In the context of inter-GPU communication, interconnects such as NVLink offers a solution. Communication between threads, if not possible to hide/overlap with computation on the cores, can degrade the performance as it incurs an overhead. This emphasizes on the importance of explicit parallelism for optimal GPU utilization.

4.2. GroqChip

The GroqChip [5] is a deterministic Tensor Streaming Processor (TSP), resembling a modified systolic-array architecture. The architecture is a 2D mesh of cores, each with its own dedicated functionality, as shown in Fig. 3. A column of these cores – all of the same type – is called a functional slice. The functional slices consist of one vector processor (VXM), two matrix-execution modules (MXM), switch-execution modules (SXM) and memory modules (MEM). Data travels horizontally, executing in 320 SIMD-style lanes. A single instruction can control 16 lanes, effectively creating 20 superlanes that can all be operated independently from each other. Each superlane implements a 4 × 4 mesh of vector Arithmetic–Logic Units (ALUs) capable of doing 16x SIMD operations. Each ALU has 32-bit input operands but, with the exception of additions and multiplications, instructions are done in a reduced-precision FP32 format compared to the IEEE754 standard.

The memory on the GroqChip only consist of the memory modules. These memory units are SRAM blocks, adding up to a total of 220 MiB of on-chip SRAM. The reason for this relatively small, but fast, memory is the decoupling of the memory subsystem from the functional units, which is beneficial for the deterministic-computing paradigm.

Each functional unit (core) accepts a set of instructions; for example, the MEM unit could receive the instruction to put a vector onto one of the data streams or store the results from the data stream in its available SRAM. As soon as data is loaded onto a data stream, it automatically 'flows' in the direction of the stream, which can be either EAST-bound or WEST-bound. When an addition needs to be performed, both inputs need to arrive at the same time as the `add` instruction at the corresponding VXM core. This design choice puts the burden of optimization on the software generating the instructions. This is either done by the Groq compiler automatically from an ONNX-graph input or manually controlled by a user through the exposed Groq API, which has various levels of abstraction on top of the Groq Instruction-Set Architecture (ISA). To support the creation of large-scale systems, the GroqChip has dedicated chip-to-chip modules that are capable of performing off-chip communication without losing their determinism [44].

4.3. Graphcore IPU

The Graphcore IPU [4], whose architecture is shown in Fig. 4, contains multiple tiles. More specifically, the GC200 chip contains 1,472 tiles, each containing a core which supports up to 6 threads, therefore, supporting a total of 8,832 threads on a single IPU. Moreover, each of these cores contains an `Accumulating Matrix Product` (AMP) unit. These units support IEEE754 compliant FP32 arithmetic and are designed to accelerate matrix multiplications and convolutions. The memory of the IPU is located locally with the core within a tile. Each tile contains 624KiB of SRAM memory creating a total of 900 MB on-chip memory. The intra-IPU communication relies on a powerful, low-latency interconnect called `IPU exchange` and allows for all-to-all data exchange. For inter-IPU communications, each chip contains 10 so-called `IPU links`. Overall, the IPU offers true Multiple-Instruction Multiple-Data (MIMD) parallelism. This unique style of parallel-processor design adapts well to fine-grained computations that exhibit irregular data-access patterns. With respect to the programming model, the IPU adopts the Bulk Synchronous Parallel (BSP) model [47] through which it organizes its compute and data-exchange operations. This abstraction for parallel computations consists of multiple sequential supersteps. A superstep consists of a local computation phase; every process (tile, in the IPU case) operates in isolation performing compute only on its local memory, followed by a communication phase where each process can exchange values needed by other tiles. These activities are concluded with a barrier-synchronization phase; only when all processes have reached the barrier can the next superstep be started. Because of this, the IPU can be described as a true BSP machine.

4.4. Google TPU

The TPU (version 1) was designed as a systolic-array processor for inference, only supporting 8/16-bit operations; see Fig. 5. By supporting only matrix-multiply and basic nonlinear activation functions, it was unfit for training neural networks. Consequentially, an HPC application – for example, the one demonstrated in this paper – would also not be a suitable fit for this processor. However, with the TPUv2 whose architectural overview is shown in Fig. 5, Google shifted their focus towards supporting training on their TPU chips. Google added a vector-processing unit (VPU) and changed the matrix-multiply units to support the FP16 format (FP32, with only a 7-bit mantissa). The VPU most likely supports higher precision, as can be deducted from results in this work but no confirmation of this is found in the public domain. These two major (micro)architectural changes made it possible to run a wider range of applications including training neural models on the TPU. All are supported through the Google XLA compiler [48] taking TensorFlow as input. The TPUv3 [7], assessed in this work, is an upgrade in terms of functional-unit count, higher memory speed and optimized chip layout, but did not include any fundamental changes. Recently it has been disclosed that TPUv2 onwards also contains *Sparse Cores* for optimized embedding access [49].

4.5. Performance predictions

In this section, we attempt to predict the performance of the IO workload on the different types of hardware. Therefore, it is important to realize that the IO workload has two components that map differently onto different types of hardware:

1. Neuron local state dynamics
   These dynamics are autonomous for each neuron and, therefore, are embarrassingly parallel calculations. Consequently, these are calculations for updating the state of every single neuron. This boils down to element-wise vector operations.
2. Gap-junction dynamics
   These dynamics require communication in each timestep and, therefore, are not embarrassingly parallel. As described previously, gap-junction communication employs the gather-scatter operations (essentially, SM operations) from TensorFlow.

Based on these components, next we will discuss the expected performance per hardware platform.

4.5.1. CPU

We expect very accurate results because of full FP32 support. Yet, the multi-core CPU is only expected to give decent performance. Simply because of the lack of supported parallelism/ computational power in comparison to the other hardware accelerators. However, because of the limited degree of parallelism, the performance impact for the communication of the gap junctions is expected to be less than for the other platforms. For the execution of the program on the CPU we use two versions: One version uses TensorFlow, that is, JIT compilation through the XLA compiler will be used; it will automatically utilize the many threads available in the CPU over the whole program. The other version uses ONNX instead of TensorFlow; we expect a slight decrease in performance as it does not perform program-wide optimization.

4.5.2. GPU

The GPU architecture featuring one warp execution per SM or multiple Tensor Cores is very well-suited for local neuron-state dynamics because the element-wise vector operations map very well on the architecture of the GPU. As described previously, gap-junction communication employs the gather-scatter operations (essentially, SM operations) from TensorFlow. These operations can be handled within the architecture by design. However, they require synchronization and non-coalesced memory accesses which can negatively affect the performance. The TensorFlow backend for CUDA uses Tensor Cores sacrificing FP32 accuracy. However, this only happens when explicit matrix multiplications are requested and not as an optimization. So in our case, the compiler will only use float32 CUDA operations and CUDA cores.

With TensorFlow as design entry, the XLA compiler is used, which optimizes the graph resulting in a single kernel launch. Among other techniques, it does this by “fusing” the calculations. Moreover, this fusion keeps intermediate values stored in GPU registers [50]. By comparison, the use of ONNX results in small single-kernel invocations without “fusing” the calculations. The invocation overhead for small GPU kernels is expected to hurt the performance of the ONNX-GPU-runtime. TensorRT, which is a NVIDIA backend for improving inference performance, is also a supported backend in ONNX and is expected to outperform the CUDA runtime in performance. However, this comes at a loss of accuracy as the Tensor Cores perform operations at TF32 precision.

4.5.3. GroqChip

The GroqChip is based upon a systolic-array architecture, supporting Matrix-Multiplication and Vector-Operation operations that can be utilized for the calculations of the neuron local state dynamics. In fact, since neuron updates require only 1D-data, the Matrix-Multiplication units (which is the focus of this chip) are effectively underutilized in this architecture.

The strategy for the calculation of the gap-junction dynamics is a naive approach: the calculations enforce dense-matrix operations via one-hot encoding of operands and, then, utilizing the matrix-multiplication hardware. With the use of this approach we expect that performance will deteriorate very rapidly or memory will be depleted with increasing IO-network sizes.

The compiler of the GroqChip takes in the ONNX graph but is not limited to executing this on an operation-per-operation basis, as opposed the use of ONNX on the CPU and GPU, as it recompiles the full ONNX graph at once. Therefore, it can potentially perform the same optimizations as the XLA compiler for the TPU. Besides, the GroqChip VXM is not capable of doing all operations in IEEE754 FP32 arithmetic. Because of this, it can be expected to perform slightly better than the TPU at the cost of reduced accuracy. As this is the first version of the architecture, current compiler development is still exploring ways to map non-standard ML-operations to the hardware. Therefore, future compiler releases will likely improve performance of brain simulations on the GroqChip.

4.5.4. Graphcore IPU

The IPU, with a large amount of very small general-purpose cores, should do well in parallelizing local neuron-state calculations, however, its architecture is geared towards irregular data-access patterns, which is not essential to the particular task. The extra overhead of such advanced features, therefore, will not help performance in terms of computing this embarrassingly parallel part of the simulation. In contrast, for the gap-junction communication which employs the gather-scatter operations, the MIMD architecture and the BSP model is very well-suited. Because of the MIMD architecture, the non-coalesced memory accesses are not expected to introduce a significant overhead, in contrast to the GPU. However, as the topology is given as an unknown parameter to the model, the IPU compiler cannot be expected to allocate neighboring cells on adjacent tiles, resulting in sub-par communication performance. Furthermore, we expect that even though having the smallest available memory among the AI chips under evaluation, the IPU should easily be able to handle large problem sizes.

4.5.5. Google TPU

Similarly to the GroqChip, the TPU is based on a systolic-array architecture which, again, supports Matrix-Multiplication and Vector-Operation operations natively. However, similarly to the GroqChip, the Matrix-Multiplication unit is expected to be underutilized in the case of the local-neuron state calculations.

As Google put much effort into TensorFlow support, the gather-scatter operations (used in the gap-junction calculations) are expected to be optimized both in terms of hardware support and the XLA compiler. We expect that the so-called Sparse Cores created for embedding retrieval in DNNs, as disclosed in the recently published TPUv4-architecture paper [49], can also be heavily utilized for the gap-junction calculations in our model. Therefore, the TPU is expected to outperform the other platforms, especially in this part of the model. Still, the large-scale sparse communication patterns used in the IO network are very atypical of embedding-based ANN workloads, and the exact operations supported by the Sparse-Core vector units are unknown, leaving a lot of prediction uncertainty.

The level of IEEE754 compliance of the chip is not documented, but it is known that matrix-multiplications are performed in 16-bit precision, suggesting that the designers choose to trade some numerical precision for performance. How this translates into accuracy of other operators is not known and the resulting numerical validity of simulation results can thus not be estimated.

5. Experimental setup

5.1. Benchmarking parameters

Each platform is benchmarked for *performance* on a set problem (i.e., network) size as well as for its *performance scalability* by simulating the IO network for small population sizes in the range $[4^3, 5^3, \dots, 20^3]$ and, again, for larger sizes in the range $[30^3, 40^3, \dots, 100^3]$, where the third power is an artifact of the cubic network-topology generation method. These experiments are focusing on four different aspects of each AI platform, discussed next.

5.1.1. Unconnected network

By removing the communication step (gap junctions) from the model, we obtain a (biologically unrealistic) compute-heavy, embarrassingly parallel workload. First, we measure the setup time for each AI platform, including on-chip buffer allocation, Ahead-Of-Time (AOT) compilation or definition of Just-In-Time (JIT)-enabled functions. Next, we simulate an IO network for 100 ms of biological time and take the minimum wall-clock time from 5 runs (including data-transfer times). For JIT targets, the first runtime (if outside the other runtimes' standard deviation) minus follow-up runtimes is taken as the JIT compilation time, such that we can compare setup times between AOT and JIT targets.

5.1.2. Connected network

By restoring gap junctions into the IO network, we assess communication overhead. Runtimes are obtained in an identical way as before, yet the expectation here is that they are markedly longer than the unconnected case.

5.1.3. Numerical validation

Measuring performance is our main focus, yet this must not come at the cost of functional correctness. Here, we simulate connected networks up to 729 neurons for 10 s of biological time and numerically compare the various results to the reference CPU output.

5.1.4. Numerical stress-test

Here, we simulate the IO in a more biologically realistic way that is of interest to neuroscientists: We add more variance to the neural parameters and, most importantly, a lot of external current inputs (simulating other brain regions) that will evoke action potentials (spikes) in the IO dynamics. These fast transients will stress-test the numerical performance of the AI hardware, especially non-IEEE754 targets (Tensor Cores and GroqChip). We perform this experiment on the smallest 64-neuron network and then compare it for numerical accuracy against the CPU.

Benchmarking is implemented in a publicly available and modular, extensible framework, downloadable from GitLab.[1] The main benchmarking script auto-discovers available hardware, runs the appropriate benchmarks and records results. Used software versions are also shown in Table 1.

5.2. Hardware-specific optimizations

As detailed in Section 3.6, the IO is represented as hardware-agnostic TensorFlow kernel updating its internal state from one time-step to the other. While our original goal was not to write platform-specific code, we found that, by default, some of the AI platforms did not perform very well. For example, most platforms defaulted to copying over the entire parameter arrays for each kernel invocation, which was not needed for this mostly constant data. For a fair comparison between hardware platforms, we allowed optimizations to be applied to hardware-specific code that either led to operation fusion across different execution kernels or prevented unnecessary device-host data transfers. The exact optimizations have been applied in close collaboration with Graphcore and Groq for the respective chips, and are as follows:

5.2.1. TensorFlow XLA

The TensorFlow graph executor typically performs each operation separately when a graph is run with a corresponding kernel invocation. A different way to run TensorFlow models is made available by XLA, which turns a TensorFlow graph into a series of kernels created for a particular application. These kernels can take advantage of application-specific information for performing optimizations, e.g., operation fusion. The CPU, GPU, and TPU are the three available backends for the XLA compiler. For the IO application, a TensorFlow wrapper function was implemented that fuses up to 40 timesteps together for each call in order to fully exploit the XLA compiler.

5.2.2. ONNX

Except for the GroqChip, all ONNX implementations build on top of *onnxruntime* or *onnxruntime-gpu*. We enable all backend-supported graph optimizations. Explicit `IOBindings` are used to prevent unneeded host-device data copies. Parameters are copied once to the device at simulation start. Then, state is allocated twice, with each timestep toggling between two buffers, one as the input state and the other as the output (next) state. For TensorRT, we leave the default behavior of using TF32 enabled, otherwise, it will not utilize its Tensor Cores.

[1] https://gitlab.com/neurocomputing-lab/Inferior_OliveEMC/ioperf

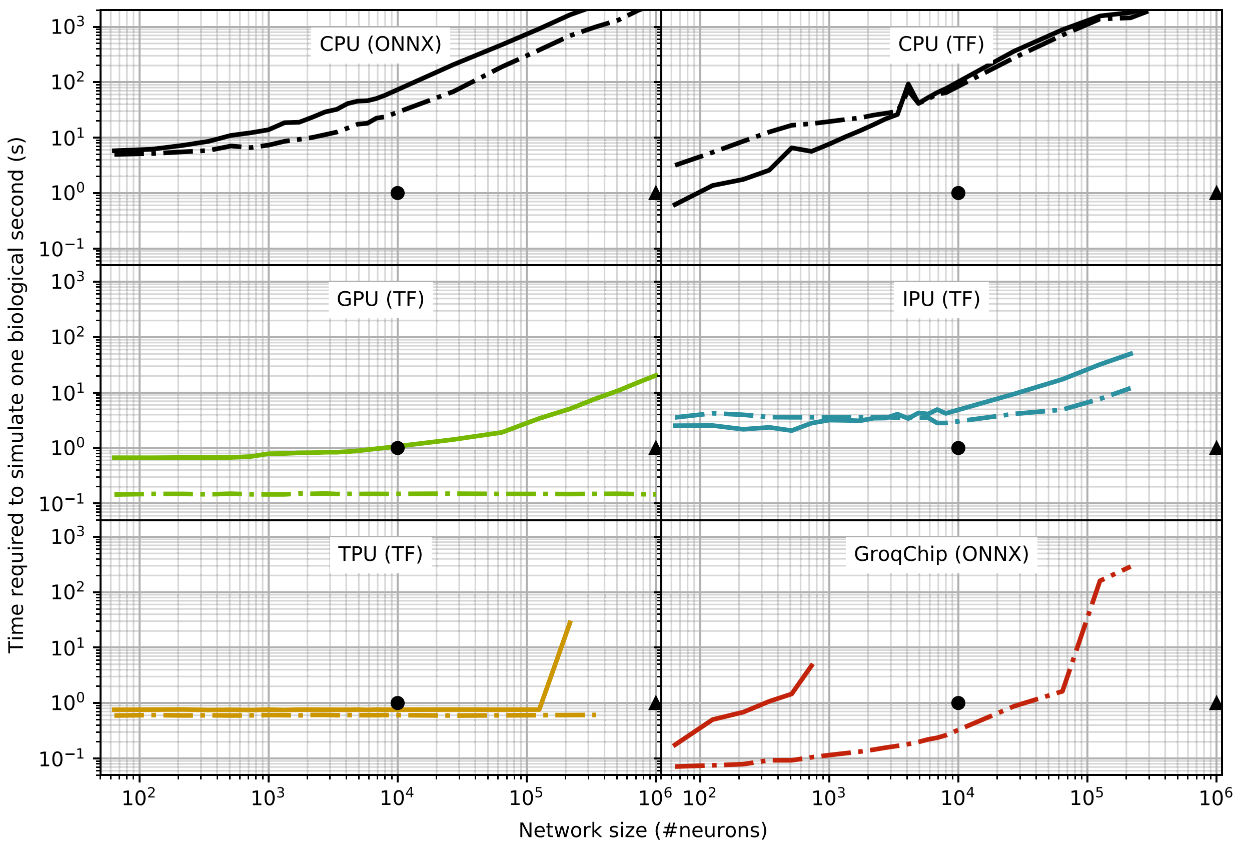


Fig. 6. Runtime performance (lower is better), comparison between CPU baseline, GPU and AI chips. For scale, the mouse (•) and human (▲) Inferior Olive are shown as running in realtime in all figures. The CPU is included twice to explain the observed switching behavior of the IPU. On the CPU, while the XLA optimizer builds a single-core, connected-network simulation, it builds a multicore, unconnected-network simulation (as observed by load-testing), leading to an unexpectedly *slow* simulation for the latter case. The same behavior can be observed for the IPU, which uses the XLA compiler as well.

5.2.3. Groq

After the compilation of an ONNX graph with the Groq Compiler, the binary can be executed directly on the GroqChip. A naive approach here would be to invoke this binary 40 times for 40 timesteps and move the data back and forth continuously since the GroqChip only has SRAM which is fully managed at compile time. However, the Groq Compiler is able to tie input and output tensors together into a *persistent* memory buffer in the on-chip SRAM. Utilizing this results still in 40 invocations of the binary but skips the continuous I/O between host and accelerator. A more radical way to improve the performance is to compile the 40 timesteps into a single ONNX graph that can then be converted with the Groq Compiler; this method will reduce 40 invocations to a single invocation. We implemented all optimizations as long as the compiler was able to compile them. The 40 timesteps at once quickly ran into compiler errors with growing networks.

5.2.4. Graphcore

The IPU has architectural support for *streaming memory*. This means that we can run a single program on-chip for the entire simulation that will stream out samples every 40 timesteps. The inner, unsampled 1-msec 40-timestep loop, is run using `ipu.loops.repeat`, after which the recorded voltages are pushed to an `IPUOutfeedQueue` with a 200-sample size. This is, then, looped once more using `ipu.loops.repeat` for the required amount of milliseconds to simulate and wrapped in a TensorFlow JIT function. Furthermore, the fast-math optimization is enabled, 128 IPU tiles are reserved for I/O with `place_ops_on_io_tiles = True` and program execution is limited to a single IPU.

6. Experimental results

Except for the reference CPU, for brevity we report here either TensorFlow or ONNX results, depending on which of the two leads to better performance. Overall performance plots are shown in Fig. 6 and will be detailed in the next sections. In general, it is found that, for the IO application, the ONNX ports are outperformed by their TensorFlow counterparts. This is due to the fact that the onnx-runtime library currently does not perform as extensive optimizations as the XLA compiler. For example, the CUDA target translates each compute step into a single predefined kernel call. The TensorRT backend performs operator fusion, resulting in multiple kernels that chain arithmetic operations. Still, the CUDA XLA-backend vastly outperforms both ONNX CUDA targets, and as such we removed the corresponding findings from the main analysis. Note that the Groq platform only supports AOT compilation of ONNX models.

6.1. Compilation time

Both software stack and hardware influence program setup time, as illustrated in Fig. 7 for the largest network (729 cells) that could fit in all AI chips. The CPU compiles the fastest across the board as we have a direct translation of ONNX operations to their CPU-optimized callbacks. The TensorFlow (XLA) version, not included in the figure, was much slower due to the increased compiler complexity. Both the IPU and GPU exhibit similar JIT compilation speeds. The GroqChip's AOT compiler takes significantly longer for this workload due to the explicitly concatenated 40 timesteps. The GroqChip version with a single timestep per program compiles much faster than the Graphcore or A100 versions, but at a small performance loss.

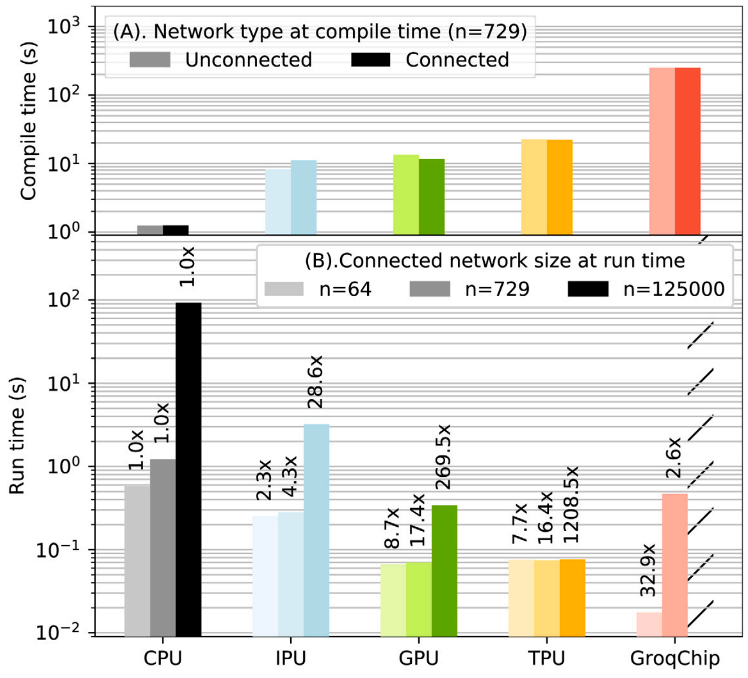


Fig. 7. (A) Setup (AOT+JIT compile + memory allocation) times for a network of 729 neurons in both Unconnected- and Connected-network configurations. JIT compile times are extracted from the first run of 5 performance runs and added to the initial setup time (if outside one standard deviation). **(B)** Performance and speedup of different AI chips vs. the CPU reference on the Connected benchmark, for different network sizes. Sizes were chosen to be the smallest (64) and largest connected networks that could fit on the GroqChip (729) and the TPUv3 (125,000). The rightmost GroqChip bar is absent, corresponding to the model that could not be compiled.

6.2. Runtime performance

6.2.1. Unconnected network (embarrassingly parallel)

For unconnected cells, neural dynamics are expressed only using vectorized operations. As predicted, this fits the compute paradigm of the GPU very well. Performance scales linearly with problem size (horizontal line), showing that the GPU cores are underutilized for all simulated network sizes.

The TPU and GroqChip, as systolic-array-based processors, were expected to be a poorer architectural fit because large parts of the chip would be left unused. Still, the focus on efficient vector operations could result in speedups. We can indeed observe this in Fig. 6, although in different ways. The TPU, similar to the GPU, flatlines across all problem sizes, although being 4.1x slower. Consequently, memory capacity is not a problem for the TPU but performance capping in raw single-cell computations due to architectural design choices. In contrast, the GroqChip starts out 2.0x faster than the GPU, quickly loses this edge and, between 10^3 and 10^4 cells, starts to hit its memory-capacity limits, degrading performance with higher problem sizes. Networks of more than 640,000 cells simply do not fit on the chip. The GroqView analyzer confirms that the problem is core-to-core-memory communication and that most dedicated cores are not used most of the time.

The IPU was expected to perform well given its large core count but the very homogeneous compute load proved a poor fit for its MIMD design, leading to large under-utilization of the chip. With respect to real-time performance, only the GPU followed by the GroqChip (ignoring memory issues) and marginally the TPU makes the 1-sec cut.

6.2.2. Connected network (high communication overhead)

As predicted, communication patterns induced by a small number of gap junctions lead to a large performance reduction of 4.6x for small networks on the GPU. For higher problem sizes, performance drops at a growing rate, with a 141x degradation for networks of 10^6 cells. The AI chips fare much better here, most of which initially shows a less than 20% reduction in performance against their unconnected counterparts.

As an exception, the GroqChip's connected-network simulation runs 2.5x slower than the unconnected version; even so, it outperforms the GPU on very small, connected neural networks by a 3.7x speedup. However, the GroqChip (as expected) converts the SM communication into a dense-matrix multiplication, making the best out of its deterministic-execution hardware. This quickly leads to prohibitively large matrix multiplications and, beyond 729 cells, the scheduler is unable to allocate the necessary instructions. In effect, the GroqChip loses its edge over the GPU for larger networks.

In contrast, the TPU shows nearly identical behavior to the unconnected case and its performance does still not scale with problem size. This changes around networks larger than 10^5, where the *JIT compiler* seems to run into performance problems. Here, we observed large random fluctuations in performance that either led to approx. 1-sec or very long more than 400-sec run-times over the 5 repeated runs. We expect that these originate from memory limits of the TPU and had to stop benchmarking due to impractically large run times. However, we could not determine the true source of variation. The high performance, and absence of performance-scaling with respect to problem size, after adding communication overhead, most likely shows that Sparse Cores are indeed heavily utilized for gather-scatter operations and allow for efficient sparse communication between neurons.

The IPU – severely underutilized for the unconnected case – sees in fact a performance improvement when we increase the communication overhead in small networks. While counterintuitive, this is actually the same effect we see on the XLA-based CPU backend. Here, we see that gap junctions force the simulation to become single-core, which becomes faster than the parallel, multi-core, unconnected case due to the lower synchronization overhead. Around 10^4 cells, this behavior changes, gap-junction communication becomes a fixed overhead on top of normal simulation. At a certain point, this growth becomes exponential and the largest simulated network does not fit on a single IPU anymore.

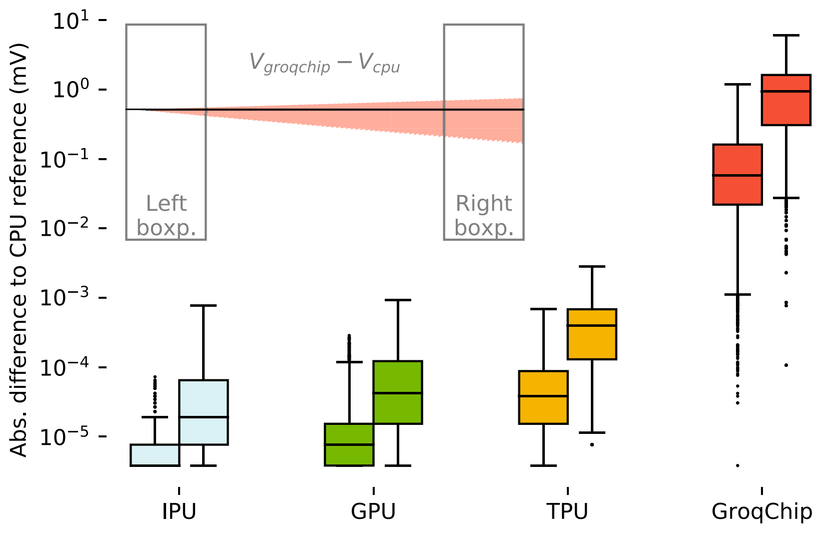


Fig. 8. Numerical-accuracy validation (lower is better). Box plots show deviations from CPU baseline, as recorded over two 1-sec timespans, one at the start (left) and one at the end (right) of the 10-sec numerical-validation simulation. The GroqChip result, showing the largest deviation, is plotted in the upper left corner together with the two recording spans.

6.3. Numerical validation

While all AI chips outperform the CPU baseline, it is wise to also explore any potential decrease in numerical accuracy of the different runs with respect to that of the same CPU. Here, we compare 1-msec sampled cell somatic voltages of an extended, 10-sec simulation for a 729-cell, connected network (the largest population supported by all platforms); results are shown in the box plots of Fig. 8.

As expected, platforms supporting IEEE754 floating-point numerics (IPU, GPU, TPU) show accurate reproduction of voltage traces. The IPU, even with fast-math enabled, is the most faithful to the CPU baseline. The GPU and TPU exhibit increasingly large deviations but still fall within limits explainable by floating-point instruction reordering. The GroqChip, while supporting FP32 number storage, implements certain operations at lower precision including exponent calculation. This is visible by a quite large mV-order deviation from the CPU baseline, for a process that happens at the $10 - 100$ mV-scales. This voltage difference mostly stems from a slowly accrued phase difference for the oscillating cells. TensorRT (not shown in this plot) is by default using Nvidia's TF32, for which accuracy was found similar to that of the GroqChip.

6.4. Numerical stress-test

The numerical stress test increases neuronal variation and adds external inputs that lead neurons to spike. These fast transients cannot be simulated using FP16 precision, but reduced-accuracy FP32 operations as used in Tensor Cores or GroqChip (and possibly the TPU) are still untested. Once more, we compare the deviation of the somatic-voltage traces of the various AI chips against the CPU baseline.

Again, the platforms with native FP32 support show the lowest deviation (over a 10-sec simulation run): The maximum absolute difference from the CPU baseline is 0.087 mV for the IPU, 0.135 mV for the GPU and 0.672 mV for the TPU. These moderate, mV-order differences can be explained by small spike-time differences which, due to the large neuronal-spike sizes, quickly lead to large voltage discrepancies. If there were any differences in neuronal dynamics including missing or surplus spikes, or oscillation phase-lag, then deviations much larger than 10 mV in absolute magnitude would be observed. Importantly, all simulations run stably; i.e., they do not cause this chaotic IO-model simulator to crash. The GroqChip simulation initially starts out the same as in the numerical-validation test, but as soon as input perturbations are applied, it becomes unstable and settles on voltage deviation at a measured maximum of 8.51×10^{36} mV, unacceptable for scientific applications. Notably, the error stabilizes at this point and does not explode to infinity or NaN values, as observed with FP16 simulations. To regain numerical stability, we tried lowering the time-stepping constant Δt 10-fold and 100-fold for the GroqChip simulation but this did not lead to results more closely in range with the CPU ones.

Table 3
Chip-measured, average power consumption during a 100ms simulation over 5 trials, connected, 729-cell simulation, including measured execution time.

Platform	Power consumption (W)	Execution Time for 1-sec simulated time (s)
A100	103.8	0.70
Graphcore	85.8	2.81
GroqChip	68.0	4.71

6.5. Energy usage

Reduced energy usage on deep-learning loads is one of the main selling points of AI chips. Here, we will briefly explore how this feature holds for the Inferior-Olive application. We measured the power consumption of each chip with the caveat that this is a very coarse and possibly biased measurement of actual power usage and not representative for the entire system for a single 729-neuron, gap-junctioned simulation. For each platform, native power-profiling tools have been used; namely, `nvidia-smi` for the GPU, `gc-monitor` for the IPU, `tsp-ctl monitor` for the Groq; regarding the TPU, we were unable to find any power-monitoring tool.

The results are reported in Table 3. After factoring in the simulation performance, the A100 GPU scores the lowest energy cost (73 J) though exhibiting the highest power consumption. The GroqChip does not fare well in this comparison (323 J) but this is mainly due to the bad gather-scatter performance of the chip – picking a smaller network would result in better performance-per-watt metric. The GraphCore chip draws much less power than the A100 during simulation as well, however runs significantly longer, resulting in an overall larger energy draw (241 J).

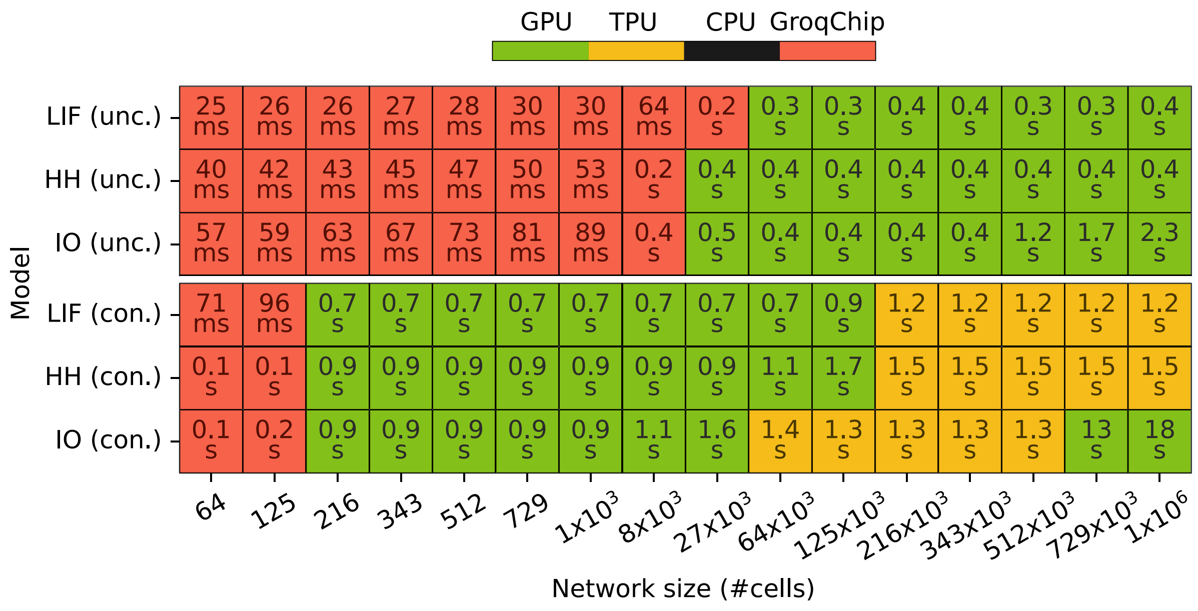


Fig. 9. Best-performing AI platform per model type and network size. Model complexity increases from LIF to HH to IO. A fixed timestep of 25 μs is used across models. LIF and HH communicate via spikes using the same method as outlined in Listing 2.

6.6. Model complexity

Until now, the results focused on IO simulation performance. In Section 4, it was shown that the IO is representative of general conductance-based neural models. While our primary focus with this work lies in biophysically plausible models of the brain, it is still interesting to explore how varying the model complexity affects the results. To that effect, we have extended our experiments to include also the more simplified LIF and HH model.

In terms of characteristics, the (non-conductance-based) LIF model is the simplest model including one state variable (membrane potential) plus an additional one (exponential synaptic current), in its connected form. The HH model is the simplest model exhibiting biophysically realistic spiking, containing only four state variables (membrane potential, two sodium gates and one potassium gate) plus an additional one (exponential synaptic current), when connected. Finally, the IO model is an eHH model with 14 state variables with, as extra characteristics, multiple cell compartments and calcium-concentration modeling.

Performance results are shown in Fig. 9. For the unconnected case, with increasing problem sizes, we observe no significant change in the optimal platform, though for the largest IO networks, the GPU appears to be reaching its compute capacity, signified by the rapidly rising runtimes. For the connected case, behavior changes mildly only for larger problem sizes. Effectively, the TPU is performing best across the board, with the exception of the IO case where it seems to run out of resources faster than the GPU, also denoted by the steeply rising runtimes (obscured by GPU runtimes).

7. Discussion

As this work has shown, utilizing AI platforms for executing highly biologically plausible SNN workloads is made exceedingly user-friendly when using a ML library like TensorFlow. Arguably, even better performances could be obtained by coding via the various hardware SDKs (Software Development Kits), but it is unrealistic to expect computational scientists to learn the low-level optimization options of each hardware platform made available to them these days.

As shown, the added benefits from JIT compilation make a hand-coded CUDA implementation perform on par with the XLA-compiled TensorFlow version while, at the same time, allowing one to move easily to a new piece of hardware when this is released. We expect that, in the future, more classical HPC workloads will see ML-library, that is, tensor-based implementations.

The benchmarks also highlight the various strengths and weaknesses of the platforms with respect to brain simulation. The GroqChip is very fast for small and unconnected networks, but is limited by small on-chip memory size and due to its focus on deterministic-compute, inefficient sparse communication execution and most notably numerical stability due to non-IEEE floating point implementation, limiting its use for biophysical realistic brain simulation. The GPU performs well in general, but for large networks, sparse communication between neurons leads to increasingly large runtimes. The TPU is not limited by sparse communication in large networks but exhibits highly arbitrary compilation and/or run times for very large networks ($> 10^5$). The IPU is architecturally not a good fit for the workloads at hand, across the board. The homogeneous nature of the solved equations is a poor fit for its MIMD architecture.

For promising upcoming accelerators like those by Graphcore and Groq, we believe that future speedups will chiefly come from software and compiler upgrades, as current SDKs are mostly optimized for ML workloads. For instance, gather-scatter operations on the GroqChip could potentially be improved upon, memory can be better utilized, and better support for iterative programs must also be introduced. The TPU which is architecturally similar to the GroqChip, clearly performs gather-scatter operations in a more efficient way than arguably encoding indexing as one-hot vectors as the GroqChip seems to do.

Speedups could be gained by effective use of mixed precision on the IPU or reduced accuracy FP32 operations using Tensor Cores or GroqChip. For the IPU, this would constitute a separate numerical sensitivity analysis to find out which parts of the compute graph can be lowered to (stochastic rounded) FP16. As shown, the accuracy loss on Tensor Cores and GroqChip does in its current form not allow for brain simulation, but these could possibly be put to use by switching the integration scheme or other numerical optimizations.

This work has steered clear from multi-chip topologies. All discussed architectures do support specifically developed, low-latency, chip-to-chip hardware and assorted communication protocols. In many ways, such coherent communication is a bigger and more timely challenge than acceleration speed itself, which would deliver massive benefits for large-scale SNN simulation (or training). However, tapping into those platform-specific interfaces requires SDK-specific coding of the IO application; relying on TensorFlow or ONNX frameworks will, generally, not work. Careful and platform-specific coding is necessary, which we leave as future work.

Other future work related to brain simulations on AI hardware is the support of chemical synapses. Chemical synapses are absent in the IO model and therefore, they were not prioritized in this work. However, we will discuss them here shortly. Chemical synapses are traditionally encoded as discrete time-delayed spike events. None of the existing ML-library-based simulators, Norse [14] and snnTorch [15] implements spikes with delays, but instead deliver spikes instantaneously. At the same time, GPU-based simulators like CoreNEURON [23] or Arbor [25] readily implement spikes using event delivery. This indicates that chemical synapses with delays will require a non-trivial amount of work to implement efficiently, if even possible, using ML libraries. On the other hand, Norse and snnTorch focus on the ability to calculate gradients through spikes, which is not required for conventional brain simulations. Regardless, the highly dynamic and event-based nature of spikes makes them very hard to implement using vector-based instructions. As such, we suggest implementing the spike delivery system outside the ML-library-based kernels. As a small, novel contribution, we suggest the use of TensorArrays as a reasonable performing way to distribute and receive spikes using exclusively TensorFlow. This is implemented in a model consisting of simple LIF neurons, see Listing 3. For the TPU, targeting Sparse Cores for such sparse workloads might lead to high speedups on the TPU in comparison to CPU-based communication [49]. However, at the moment not enough information is known about the Sparse Cores to make it possible to write an efficient event-scheduling system. It is left for future work to evaluate chemical synapses and improve implementation. Furthermore, we would like to note that the development of an algorithm for chemical synapses is a one-time process, after which it can be reused.

8. Conclusions

This study presents a new workflow for conducting compute-intensive simulations of biophysically realistic brain models. The workflow involves the conversion of the model into TensorFlow code, enabling its execution on a range of high-performance AI-accelerated platforms, including Nvidia GPUs equipped with Tensor Cores, the Graphcore IPU, the GroqChip, and the Google TPU. Following model translation, the modeler needs only concern herself with optimizing device-host communication, which involves minimizing data transfers (AI-chip inbound and outbound data) and kernel-call overhead (via on-device looping). This approach notably streamlines the process of conducting high-performance brain simulations, with the simplification attributed to TensorFlow's abstraction of hardware intricacies. The performance evaluation of this workflow on a detailed Inferior-Olivary (IO) model simulation shows that the GroqChip is the fastest to compute the neural dynamics. However, it runs into problems when sparse communication is introduced, making it impossible to fit networks larger than 729 gap-junction connected cells on-chip. At the level of the mammalian IO (125,000 neurons connected via gap junctions), the IPU, GPU and TPU demonstrate a significant boost in performance, ranging from 29 to an impressive 1,208 times, when compared to CPU execution. Notably, the Google TPU is able to accommodate the largest network of IO cells in real time and Sparse Cores seem to be able to lead to the best sparse communication performance as well. However, despite the performance advantages of utilizing AI chips for simulations, a challenge for some of them remains maintaining numerical accuracy with respect to the reference CPU runs since they do not fully support single-floating-point precision. This accuracy reduction may lead to inaccurate brain simulations, although it may not be a limiting factor for other high-performance computing (HPC) workloads. One limitation of our presented work is the absence of an evaluation of chemical synapses, as they pose challenges in integration with our tensor-based workflow. This aspect remains a subject for future investigation. Finally, the use of multiple chips to allow for the simulation of even larger and faster brain networks warrants further exploration also be explored.

Listing 3: Spike-event delivery in TensorFlow

```
import numpy as np, tensorflow as tf

# model parameters
dt = 1; nsteps = 1000; nneurons = 100; max_delay = 20

# connection matrix containing non-zero delays
# for connecting neurons
delay_matrix = tf.constant(
    np.random.randint(10, 20, (nneurons, nneurons)) *
    (np.random.random((nneurons, nneurons)) > 0.8))

# the spikes for each neuron at each timestep
spikes = tf.TensorArray(
    tf.int64, size=0, dynamic_size=True,
    clear_after_read=True, infer_shape=False)

# set up initial array
for i in range(nsteps+max_delay):
    spikes = spikes.write(i, tf.constant([], dtype=tf.
        int64))

# internal state of the neurons
v = tf.constant(20 * np.random.random(nneurons))

def lif(step, v, spikes, delay_matrix, dt, tau=20, vth
     =10):
    i = spikes.read(step)
    i = tf.tensor_scatter_nd_add(
        tf.zeros_like(v),
        tf.reshape(i, (-1, 1)), tf.ones_like(i, dtype=v.
            dtype))
    v = v - v/tau*dt + i
    s = v >= vth

    delays = delay_matrix[s]
    targets = tf.where(delays != 0)[:,1]
    delays = delays[delays != 0]

    def c(i, _): return tf.less(i, len(targets))
    def b(i, spikes):
        spikes = spikes.write(step + delays[i], tf.
            concat([
           spikes.read(step + delays[i]), [targets[i]]
              ], axis=0))
        return tf.add(i, 1), spikes
    spikes = tf.while_loop(c, b, [tf.constant(0), spikes
        ])[1]
    v = tf.where(s, tf.zeros_like(v), v)
    return v, spikes

# simulation loop
for step in range(nsteps):
    v, spikes = lif(step, v, spikes, delay_matrix, dt)
```

CRediT authorship contribution statement

Lennart P.L. Landsmeer: Writing – review & editing, Writing – original draft, Visualization (equal), Software (equal), Methodology (equal), Conceptualization. **Max C.W. Engelen:** Writing – review & editing, Writing – original draft, Visualization (equal), Software (equal), Methodology (equal), Conceptualization. **Rene Miedema:** Writing – review & editing, Writing – original draft, Conceptualization. **Christos Strydis:** Writing – review & editing, Writing – original draft, Validation, Resources, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Co-author employed at both EMC and Maxeler, the latter recently acquired by Groq. Groq did not interfere in research. - M.C.W.E.

Data availability

Data will be made available on request.

Acknowledgments

This paper is partially supported by the European-Union Horizon Europe R&I program through projects SEPTON [Grant agr. no.: 101094901] and SECURED [Grant agr. no.: 101095717], through the HBP ICEI SGA project Arborio [Grant agr. no.: 785907], and through the NWO - Gravitation Programme DBI2 [Grant agr. no.: 024.005.022]. The authors are grateful to NVIDIA for providing two RTX6000s for development with the Hardware Grant, Google for cloud credits for TPU simulation, GraphCore for help with optimizations and dedicated Mk1 access during development, Paperspace for providing an IPU POD16, Gcore cloud for the dedicated IPU machine in their academia program, and Groq for providing a university-priced evaluation system for experiments. Finally, we wish to thank Dr. Mario Negrello for neuroscientific insights and discussions.

Appendix. Supplementary material

https://gitlab.com/c7859/neurocomputing-lab/Inferior_OliveEMC/ioperf/

References

[1] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, J. Kepner, Survey and benchmarking of machine learning accelerators, in: 2019 IEEE High Performance Extreme Computing Conference, HPEC, IEEE, 2019, pp. 1–9.

[2] N. Kasabov, Chapter 6 - evolving and spiking connectionist systems for brain-inspired artificial intelligence, in: R. Kozma, C. Alippi, Y. Choe, F.C. Morabito (Eds.), Artificial Intelligence in the Age of Neural Networks and Brain Computing, Academic Press, 2019, pp. 111–138, http://dx.doi.org/10.1016/B978-0-12-815480-9.00006-2, URL https://www.sciencedirect.com/science/article/pii/B9780128154809000062.

[3] W. Maass, Networks of spiking neurons: The third generation of neural network models, Neural Netw. 10 (9) (1997) 1659–1671, http://dx.doi.org/10.1016/S0893-6080(97)00011-7, URL https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[4] Z. Jia, B. Tillman, M. Maggioni, D.P. Scarpazza, Dissecting the graphcore IPU architecture via microbenchmarking, 2019, http://dx.doi.org/10.48550/ARXIV.1912.03413, URL https://arxiv.org/abs/1912.03413.

[5] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E.R. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, B. Kurtz, Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads, in: Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA '20, IEEE Press, 2020, pp. 145–158, http://dx.doi.org/10.1109/ISCA45697.2020.00023.

[6] R. Krashinsky, O. Giroux, S. Jones, N. Stam, S. Ramaswamy, NVIDIA Ampere Architecture In-Depth, URL https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[7] N. Jouppi, D. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, D. Patterson, A domain-specific supercomputer for training deep neural networks, Commun. ACM 63 (2020) 67–78, http://dx.doi.org/10.1145/3360307.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org, URL https://www.tensorflow.org/.

[9] J. Bai, F. Lu, K. Zhang, et al., ONNX: Open neural network exchange, 2019, https://github.com/onnx/onnx.

[10] W. Gerstner, H. Sprekeler, G. Deco, Theory and simulation in neuroscience, Science 338 (6103) (2012) 60–65.

[11] A. Tavanaei, M. Ghodrati, S.R. Kheradpisheh, T. Masquelier, A. Maida, Deep learning in spiking neural networks, Neural Netw. 111 (2019) 47–63.

[12] D. Rasmussen, NengoDL: Combining deep learning and neuromorphic modelling methods, Neuroinformatics 17 (4) (2019) 611–628.

[13] E.O. Neftci, H. Mostafa, F. Zenke, Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks, IEEE Signal Process. Mag. 36 (6) (2019) 51–63.

[14] C. Pehle, J.E. Pedersen, Norse - A deep learning library for spiking neural networks, 2021, http://dx.doi.org/10.5281/zenodo.4422025, Documentation: https://norse.ai/docs/.

[15] J.K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D.S. Jeong, W.D. Lu, Training spiking neural networks using lessons from deep learning, 2021, arXiv preprint arXiv:2109.12894.

[16] W. Fang, Y. Chen, J. Ding, D. Chen, Z. Yu, H. Zhou, T. Masquelier, Y. Tian, et al., SpikingJelly, 2020, https://github.com/fangwei123456/spikingjelly.

[17] H. Hazan, D.J. Saunders, H. Khan, D. Patel, D.T. Sanghavi, H.T. Siegelmann, R. Kozma, BindsNET: A machine learning-oriented spiking neural networks library in python, Front. Neuroinform. 12 (2018) 89, http://dx.doi.org/10.3389/fninf.2018.00089, URL https://www.frontiersin.org/article/10.3389/fninf.2018.00089.

[18] G. Orchard, E.P. Frady, D.B.D. Rubin, S. Sanborn, S.B. Shrestha, F.T. Sommer, M. Davies, Efficient neuromorphic signal processing with loihi 2, in: 2021 IEEE Workshop on Signal Processing Systems (SiPS), IEEE, 2021, pp. 254–259.

[19] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, et al., Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 34 (10) (2015) 1537–1557.

[20] H. Lee, C. Kim, Y. Chung, J. Kim, NeuroEngine: a hardware-based event-driven simulation system for advanced brain-inspired computing, in: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 975–989.

[21] R.A. Nawrocki, R.M. Voyles, S.E. Shaheen, A mini review of neuromorphic architectures and implementations, IEEE Trans. Electron Devices 63 (10) (2016) 3819–3829.

[22] E.M. Izhikevich, Dynamical Systems in Neuroscience, MIT Press, 2007.

[23] P. Kumbhar, M. Hines, J. Fouriaux, A. Ovcharenko, J. King, F. Delalondre, F. Schürmann, CoreNEURON: an optimized compute engine for the NEURON simulator, Front. Neuroinform. 13 (2019) 63.

[24] N.T. Carnevale, M.L. Hines, The NEURON Book, Cambridge University Press, 2006.

[25] N. Abi Akar, B. Cumming, V. Karakasis, A. Küsters, W. Klijn, A. Peyser, S. 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, IEEE, 2019, pp. 274–282.

[26] M.A.v.d. Vlag, G. Smaragdos, Z. Al-Ars, C. Strydis, Exploring complex brain-simulation workloads on multi-GPU deployments, ACM Trans. Archit. Code Optim. (TACO) 16 (4) (2019) 1–25.

[27] S.W. Moore, P.J. Fox, S.J. Marsh, A.T. Markettos, A. Mujumdar, Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation, in: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, IEEE, 2012, pp. 133–140.

[28] A. Sripad, G. Sanchez, M. Zapata, V. Pirrone, T. Dorta, S. Cambria, A. Marti, K. Krishnamourthy, J. Madrenas, SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture, Neural Netw. 97 (2018) 28–45.

[29] W. Guo, H.E. Yantır, M.E. Fouda, A.M. Eltawil, K.N. Salama, Toward the optimal design and FPGA implementation of spiking neural networks, IEEE Trans. Neural Netw. Learn. Syst. 33 (8) (2021) 3988–4002.

[30] S. Panchapakesan, Z. Fang, J. Li, SyncNN: Evaluating and accelerating spiking neural networks on FPGAs, ACM Trans. Reconfigurable Technol. Syst. 15 (4) (2022) 1–27.

[31] J. Sommer, M.A. Özkan, O. Keszocze, J. Teich, Efficient hardware acceleration of sparsely active convolutional spiking neural networks, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 3767–3778.

[32] R. Miedema, G. Smaragdos, M. Negrello, Z. Al-Ars, M. Möller, C. Strydis, Flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations, IEEE Access 8 (2020) 121905–121919.

[33] M. Vieth, T.M. Stöber, J. Triesch, PymoNNto: A flexible modular toolbox for designing brain-inspired neural networks, Front. Neuroinform. 15 (2021) 715131.

[34] D.F. Goodman, R. Brette, The brian simulator, Front. Neurosci. (2009) 26.

[35] P.-S.V. Sun, A. Titterton, A. Gopiani, T. Santos, A. Basu, W.D. Lu, J.K. Eshraghian, Intelligence processing units accelerate neuromorphic learning, 2022, arXiv preprint arXiv:2211.10725.

[36] J. De Gruijl, L. Bosman, C.I. De Zeeuw, M. De Jeu, Inferior olive: all ins and outs, in: Handbook of the Cerebellum and Cerebellar Disorders, Vol. 3, Springer Netherlands Dordrecht, 2013, pp. 1013–1058.

[37] Y. Yu, Y. Fu, C. Watson, The inferior olive of the C57BL/6J mouse: a chemoarchitectonic study, Anatomical Rec. 297 (2) (2014) 289–300.

[38] V. Braitenberg, A. Schüz, Cortex: Statistics and Geometry of Neuronal Connectivity, Springer Science & Business Media, 2013.

[39] J.R. De Gruijl, P. Bazzigaluppi, M.T. de Jeu, C.I. De Zeeuw, Climbing fiber burst size and olivary sub-threshold oscillations in a network setting, PLoS Comput. Biol. 8 (12) (2012) e1002814.

[40] M. Negrello, P. Warnaar, V. Romano, C.B. Owens, S. Lindeman, E. Iavarone, J.K. Spanke, L.W. Bosman, C.I. De Zeeuw, Quasiperiodic rhythms of the inferior olive, PLoS Comput. Biol. 15 (5) (2019) e1006475.

[41] J. Birgiolas, V. Haynes, P. Gleeson, R.C. Gerkin, S.W. Dietrich, S. Crook, NeuroML-DB: Sharing and characterizing data-driven neuroscience models described in NeuroML, PLoS Comput. Biol. 19 (3) (2023) e1010941.

[42] S. Panagiotou, H. Sidiropoulos, D. Soudris, M. Negrello, C. Strydis, EDEN: A high-performance, general-purpose, NeuroML-based neural simulator, Front. Neuroinform. 16 (2022).

[43] MLPerf Training V2.0, URL https://mlcommons.org/en/training-normal-20/.

[44] D. Abts, G. Kimmell, A. Ling, J. Kim, M. Boyd, A. Bitar, S. Parmar, I. Ahmed, R. DiCecco, D. Han, J. Thompson, M. Bye, J. Hwang, J. Fowers, P. Lillian, A. Murthy, E. Mehtabuddin, C. Tekur, T. Sohmers, K. Kang, S. Maresh, J. Ross, A software-defined tensor streaming multiprocessor for large-scale machine learning, in: Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 567–580, http://dx.doi.org/10.1145/3470496.3527405.

[45] Graphcore, Graphcore Tile Vertex ISA Release 1.2.3.

[46] Google, System architecure, 2023, https://cloud.google.com/tpu/docs/system-architecture-tpu-vm, [Online, accessed 15 sept 2023].

[47] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111, http://dx.doi.org/10.1145/79173.79181.

[48] XLA: Optimizing Compiler for Machine Learning, URL https://www.tensorflow.org/xla.

[49] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, et al., Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings, in: Proceedings of the 50th Annual International Symposium on Computer Architecture, 2023, pp. 1–14.

[50] Pushing the limits of GPU performance with XLA, URL https://blog.tensorflow.org/2018/11/pushing-limits-of-gpu-performance-with-xla.html.

Lennart P.L. Landsmeer received the M.Sc. degree in Nanobiology at Delft University of Technology in 2023 and worked shortly as a scientific coworker at Forschungszentrum Jülich, Germany. He is currently pursuing a Ph.D. degree at the Quantum and Computer Engineering Department of Delft University of Technology working on a joint project with the Neuroscience Department of the Erasmus Medical Center, The Netherlands.

Max C.W. Engelen received his M.Sc. in Computer Engineering at Delft University of Technology in 2021 and worked as a research assistant for the Erasmus MC Department of Neuroscience. He is currently employed at Maxeler IoT labs and still works part-time at the Neuroscience Department of the Erasmus Medical Center, The Netherlands.

Rene Miedema received his M.Sc. Computer Engineering degree in 2019 at the Delft University of Technology. He is currently a Ph.D. student at the Neuroscience Department of the Erasmus Medical Center, The Netherlands.

Christos Strydis obtained his Ph.D. degree in Computer Engineering from the Delft University of Technology in 2011. He is currently an associate professor with the Neuroscience Department of the Erasmus Medical Center and with the Quantum & Computer Engineering Department of the Delft University of Technology, The Netherlands. He is also a Senior Member of the IEEE. He has published work in prominent international conferences and journals. His current research interests include brain simulations, high-performance computing, neural implants, and functional ultrasound imaging.