Department of Physics and Astronomy

University of Heidelberg

Master thesis

in Physics

submitted by

Philipp Spilger

born in Heidelberg

2021

From Neural Network Descriptions to Neuromorphic Hardware

— A Signal-Flow Graph Compiler Approach

This Master thesis has been carried out by

Philipp Spilger

at the

Kirchhoff-Institute for Physics

under the supervision of

Dr. Johannes Schemmel

Von Beschreibungen neuronaler Netze zu neuromorpher Hardware — Ein Signalfluss Graph Kompiler Ansatz:

Der Bedarf an künstlicher Intelligenz (KI), der Fähigkeit von Maschinen zu lernen und Probleme zu lösen, wächst schnell. Spezielle Beschleuniger Hardware wird verwendet, um damit Schritt zu halten. Neben klassischen Beschleunigern, die auf Daten-parallelen numerischen Berechnungen beruhen, wird Event-basierte neuromorphe Hardware entwickelt. Um KI Modelle effizient beschreiben und nutzen zu können, ist eine natürliche Frontend und Backend Software Abstraktion essentiell. In dieser Arbeit wird eine Signalfluss Graph-basierte Experiment Beschreibung als natürliche Abstraktion für BrainScaleS-2 entwickelt und implementiert. BrainScaleS-2 ist eine Event-basierte neuromorphe Hardware Plattform, die spikende und klassische neuronale Netzwerke unter Verwendung von analogen Schaltkreisen emuliert. Daran angegliedert entwickelte Ausführungsmodelle erlauben Just-in-Time Ausführung sowie Kompilation für autarken Einsatz. Darauf aufbauend werden PyTorch und PyNN als Frontends für klassische und spikende Experimente integriert. Die Entwicklungen werden durch eine gründliche Leistungsanalyse unter Verwendung von artifiziellen Testprogrammen und realen Experimenten evaluiert. Die Evaluation wird abgeschlossen durch eine Ende-zu-Ende Anwendung auf energieeffiziente autarke Klassifikation von Elektrokardiogramm Aufnahmen.

From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach:

The demand for Artificial intelligence (AI), the ability of machines to learn and solve problems, is growing rapidly. Special-purpose accelerator hardware is utilized to keep up. Aside classical accelerators relying on dense, data-parallel numerical calculations, "event-based" neuromorphic hardware emulating neural networks directly is emerging. To efficiently describe and use AI models with accelerators, a natural front and back end software abstraction is crucial. In this work, a signal-flow graph-based experiment representation is developed and implemented as a natural abstraction for BrainScaleS-2, an event-based neuromorphic hardware platform emulating spiking and classical neural networks using analog circuits. Accompanying development of execution models allows just-in-time execution and compilation for standalone deployment. Building on top of this, PyTorch and PyNN are integrated as front ends for classical and spiking experiments. The developments are evaluated by a thorough performance analysis using artificial benchmarks and real-world experiments, culminating in an end-to-end application for energy-efficient standalone classification of electrocardiogram recordings.

Contents

1 Introduction **9**

2 Outline **11**

3 Methods and Tools **13**

3.1 Neuromorphic Hardware 13
3.2 Graph-based experiment notation 21
3.3 Front ends 31
3.4 Profiling tools 34
3.5 Development tools 36

4 Implementation **39**

4.1 Graph-based Experiment Notation and Execution — grenade 39
4.2 PyTorch extension — hxtorch 63
4.3 PyNN back end 72
4.4 Profiling tools 75

5 Results **77**

5.1 Profiling tools 77
5.2 Establishing a performance baseline 80
5.3 Graph-based experiment description and execution 84
5.4 PyNN 97
5.5 PyTorch extension 97
5.6 Competition — Application on ECG trace classification 101
5.7 Software organization 105

6 Discussion **107**

7 Outlook **111**

8 Acknowledgments (Danksagungen) **115**

9 References **117**

A Experiment environment **125**

1 Introduction

Artificial intelligence (AI) is the ability of machines to learn and solve problems using (often highly reduced) principles found in biological neural networks. Its influence and demand are growing rapidly in both research and industry. The amount of computation required for the largest AI training doubles approx. every 3.4 months since 2012 [37]. Moreover, this increase results in significance of energy consumption of AI compared to daily expenses of humans. For example, the authors of [69] found training of common AI models to result in up to about five times the carbon dioxide emissions of a car over its lifetime.

The demanded growth in amount of computation can't be met by technological advances of classical central processing units (CPUs) as projected by Moore's Law [49], which grows slower than requested. This discrepancy is reduced by development and introduction of dedicated special-purpose accelerator hardware. Since AI models typically contain potential for high data-parallelism and involve numerical computation, accelerators like GPUs and Google's tensor processing unit TPU [40] offer data-parallel computation intrinsically and lately are accompanied by emergence of application-specific integrated circuitry (ASIC) of similar architecture [61].

In contrast, research is conducted to reduce the amount of computation required for AI, e.g. by changes in the model architecture or quantization of the computations. One such approach is changing the model architecture from consisting of classical numerical computations with dense data-flow to being "event-based" resulting in sparse data-flow. It promises reduction in the required amount of computation and data transport alike and more-closely mimics signal transport in biological neural networks. Dedicated accelerator hardware for such models is being developed, for example Intel Loihi [14] or TrueNorth [16], both relying on localized digital simulation of neurons and synapses. Such hardware is called neuromorphic, originally introduced in [48] as analog emulation of the model's behavior, recently used for all architectures targeting implementation of models of neural networks.

BrainScaleS-2 [63] is the latest prototype neuromorphic hardware developed by the Electronic Vision(s) group residing at the Kirchhoff Institute for Physics of the Heidelberg University. It is event-based and emulates a set of 512 adaptive exponential leaky-integrate-and-fire neurons [29] and 256 current-based synapses per neuron using analog circuits. Therefore, the differential equations describing the behavior of the neurons and synapses are solved by equivalent differential equations of the analog circuits, opposed to simulating their behavior numerically. The latest feature addition introduces the ability to perform

analog multiply-accumulate operations using the synapses as multiplication units and the neurons as accumulator units [63], allowing hybrid operation of both event-based spiking and classical non-spiking experiments. Two single instruction multiple data (SIMD) processors accompany the analog circuits for configuration and the ability to perform local learning [25].

Efficient development of AI models requires an intuitive description and integration of training concepts. The two major frameworks Tensorflow [2] and PyTorch [53] dominate the market with Tensorflow being used more in the industry and PyTorch being used more in research [36]. They are both constructed around implementing efficient numerical operations on multidimensional data. Lately, increasing efforts are made to extend them for spiking models [35, 54]. In contrast, the neuroscience community relies on model descriptions composed of populations of neurons and projections of (synaptic) connections [31, 67].

Execution of AI models described in such frameworks involves multi-stage compilation and optimization. For special-purpose hardware being usable with such models, dedicated additional compilation steps are required. The diversity of frameworks and special-purpose hardware results in development of a multitude of different compilers, like XLA (Accelerated Linear Algebra) [34], MLIR (Multi-Level Intermediate Representation) [44], GLOW (Graph Lowering Compiler Techniques for Neural Networks) [59] and Intel NxTF (specifically for Intel Loihi) [60]. The more the target hardware's execution model diverges from that of classical CPUs, the more differences are visible in the applied compilation process. All these compilers rely on graph-based representations of the to be performed high-level operations. This is beneficial for optimization due to increased knowledge of the logical operations to be performed opposed to instruction-level optimization. For event-based hardware, a natural low-level representation of configuration is also graph-based.

In this thesis, such a natural graph-based abstraction is developed and implemented for the BrainScaleS-2 neuromorphic hardware. It is applicable simultaneously for representation of spiking and non-spiking models and features intrinsic support for models being distributed over multiple hardware instances for increased concurrency and therefore performance or model size. Using this representation, two execution models are developed, one leading to a just-in-time executor and the other targeting standalone execution with a separated compilation process. Furthermore, the developed representation is used as back end for the PyTorch extension hxtorch, already published by the author during the course of this thesis in [66], allowing seamless integration of the hardware into this framework. Similarly, PyNN [15] is targeted as front end for execution of spiking models making use of its population and projection-based interface. It is shown, that the front ends become mere thin adapters to the graph-based representation. A thorough performance analysis highlights overhead introduced by the abstraction and front ends and reveals scalability properties. The analysis is completed by a real-world end-to-end application of energy-efficient classification of electrocardiogram recordings using the non-spiking mode of operation.

2 Outline

This thesis is structured in the following way: Methods and tools used and developed during this thesis are presented in chapter 3. We start by an introduction to the BrainScaleS-2 neuromorphic hardware platform, which is used as target for all developments and experiments. Following, a signal-flow graph based notation of experiments on the platform is formulated as natural form of representation based on requirements given by the hardware and its usage. With this abstraction available, PyNN and PyTorch are introduced as two end-user front ends targeting neuroscience and machine-learning experiments. Rounding up, developed profiling tools for later software performance evaluation and (changes to) the development environment are motivated and described.

Chapter 4 constitutes an in-depth description of the software implementation for the signal-flow graph representation and the two front ends. We start with the graph representation and develop its notation and execution models by explanation of design decisions, obstacles and their their solution. Building on top of this implementation, the front ends' implementation is explained focussing on interface decisions.

Afterwards, the developed software abstractions are evaluated for their performance in chapter 5. Runtime and memory consumption are used as primary performance metrics. We start by establishing baseline measurements of controlling the hardware with the already present and used software. This is followed by an evaluation of the developed profiling tools for their overhead and achieved performance. Using this, the signal-flow graph hardware representation is evaluated with artificial benchmarks to characterize its performance. Continuing, the PyNN front end is evaluated using a soft winner-take-all network as real-world example. For the PyTorch front end, first, artificial benchmarks are used to evaluate introduced interface overhead compared to using the signal-flow graph representation directly. The chapter is concluded by a full-stack evaluation of the PyTorch front end and the signal-flow graph representation for an edge-inference application in form of a competition for energy-efficient classification of electrocardiogram recordings.

The developed software and its performance is discussed in chapter 6. An outlook about potential for optimization and features to come in the future is given in chapter 7.

3 Methods and Tools

This chapter describes the methods and tools used and developed in the following implementation and evaluation. Firstly in section 3.1, the BrainScaleS-2 neuromorphic hardware is introduced. Following a general introduction, routing of digital events on the hardware as well as a description of the state of software support prior to this thesis will be focused. Afterwards in section 3.2, the foundations for development of a graph-based notation of experiments on the neuromorphic hardware are formulated. We start with an introduction of graph-related terms and signal-flow graphs in particular and then focus on how these general pattern can be applied onto a description of experiments on the hardware. This marks the foundation of higher-level abstraction onto which end-user-facing front-end support is built. In section 3.3 we describe choice of two such front ends, PyNN and PyTorch and the approach for integration of support for the BrainScaleS-2 hardware. We conclude the chapter with a description of developed profiling tools in section 3.4 and a general description of as well as improvements made to the development environment in section 3.5.

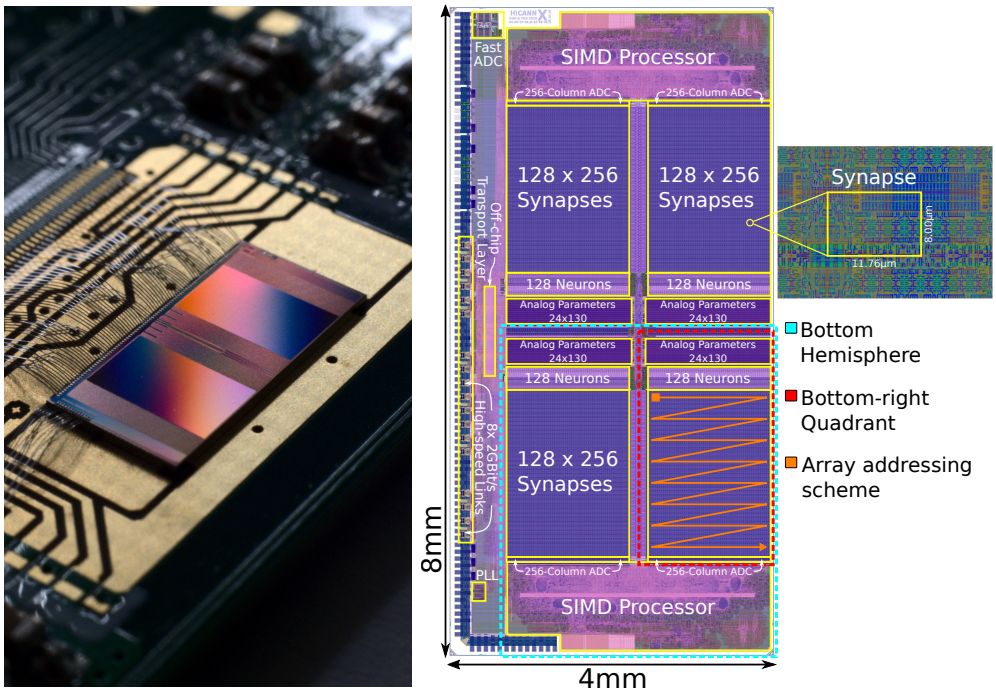
3.1 Neuromorphic Hardware

This section introduces the BrainScaleS-2 neuromorphic hardware specifically geared towards the knowledge needed in the following parts of the thesis. We start with a general introduction of the purpose of the hardware and its constitution in section 3.1.1. In section 3.2 we describe the development of a signal-flow abstraction of the hardware. For this, section 3.1.2 will describe in detail the hardware from the point of view of digital event data flow. Continuing, we describe the lower-level software for hardware configuration and control, which is used as basis for the developments within this thesis in section 3.1.3. Finally, we investigate achievable performance of the non-spiking mode of operation theoretically.

3.1.1 BrainScaleS-2

BrainScaleS-2 is the newest generation mixed-signal neuromorphic hardware developed by the Electronic Visions group in Heidelberg. The currently available single chip is depicted in fig. 3.1. It is composed of 512 analog neuron circuits emulating the adaptive exponential neuron model distributed over two hemispheres on the chip. Each neuron is connected to 256 current-based synapses, which are driven by double-row-wise synapse drivers.

Communication of data to and from this analog neural network core is implemented via



(a) Image of the chip on its carrier board, taken from [50].

(b) Schematic of the chip, displayed over the layout, taken from [50].

Figure 3.1: Left: Image of the BrainScaleS-2 chip; Right: Schematic of the entities on the chip. The top and bottom hemisphere feature 256 neurons each towards the chip middle, which are outwards-connected to the 256 synapses each. At the top and the bottom of the chip, the PPUs are located. At the chip-left physical links connect it to the outside world.

realtime event-based digital transfers. Via multiple digital routing stages, explained in section 3.1.2, event routing to and from the chip and between parts of the analog neural network core is possible. Realtime in this context means events on the hardware have a direct correspondence to their effects occurring in physical time.

Synaptic current pulses are generated by using the synapse's configurable digital weight value to scale the strength, while the pulse length is given by the synapse drivers, enabling them to implement short-term plasticity by scaling the pulse length depending on the history of events. In typical spiking neural networks, events are logically value-less and in hardware they carry only routing information. However, using parts of this event value to scale the pulse length in the synapse drivers allows performing analog multiplication in the synapses. In combination with using the neurons as integrator this allows to implement analog matrix multiplications aside spiking neural networks on the same substrate. This mode of operation is investigated in [74, 75] and put to use in [22].

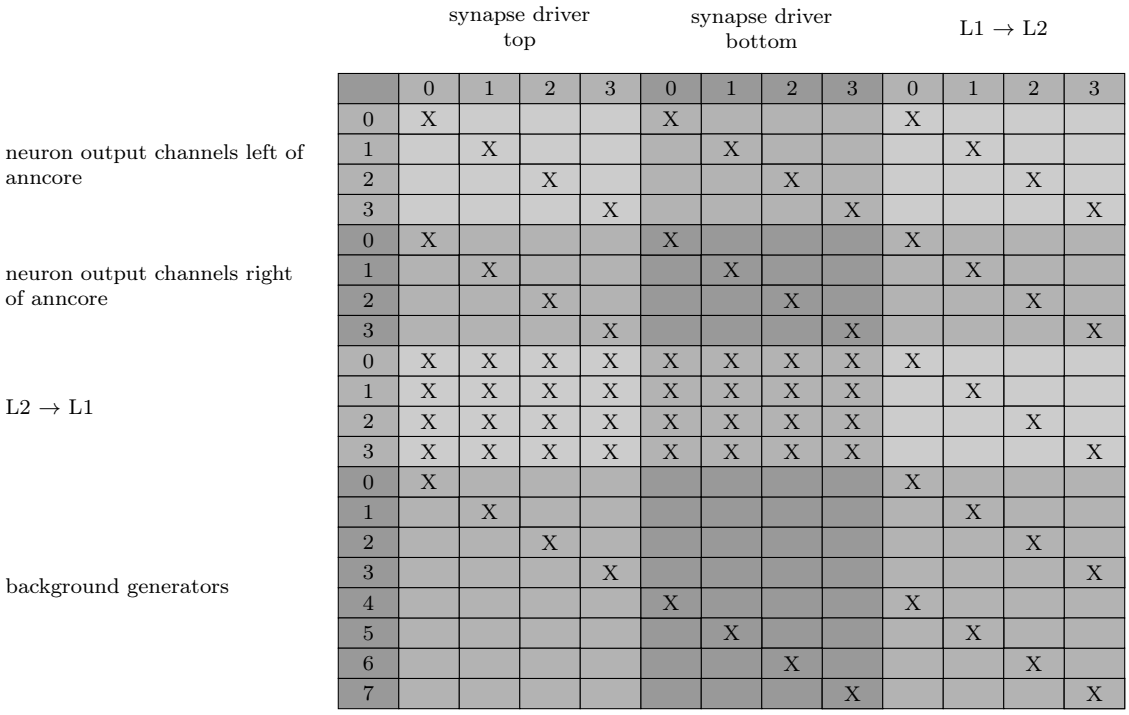
Besides the analog neural network core, one general purpose microprocessor per hemisphere adhering to the Power architecture, developed in [25] is present. Each microprocessor features a weakly coupled single instruction multiple data (SIMD) unit of 128B width. In addition to 16 kB on-chip SRAM memory, each microprocessor has access to off-chip memory via a connected field programmable gate array (FPGA), which serves as controller. These microprocessors are meant to be used to implement on-chip learning algorithms by accessing observables and altering configuration of a neural network, which gives them their name: plasticity processing unit (PPU). Suitability for on-chip learning has already been demonstrated in [6, 76, 65]. The chip-FPGA link features a bandwidth of $8\,\mathrm{Gbit\,s^{-1}}$.

Entities on the chip are configurable via memory accesses over Omnibus [26], which can be accessed by both PPUs and externally. In addition, each PPU has parallel access to configuration in the synapses of its hemisphere.

3.1.2 Digital event routing

Digital routing of events to, from and within the BrainScaleS-2 chip consists of four main elements, a routing crossbar, synapse drivers and synapses to and neuron back ends from the analog neural network core. In the following, only their behavior will be needed. Therefore, the description will aim to be *logically* correct, while omitting implementation details. Congestion effects of multiple events simultaneously sharing a resource are disregarded, because they don't change the possible deterministic path of event propagation.

The crossbar is the central logic for distribution of events on the HICANN-X. It's also called layer-1 in contrast to layer-2 for chip-external events. It has a set of inputs and a set of outputs of the same event type. The event type in the crossbar is 14 bit wide. All bits in an event are treated equally within the crossbar. Its values are never altered within the crossbar. Figure 3.2 shows the crossbar. A crossbar node connects a horizontal input line to a vertical



	0	1	2	3	0	1	2	3	0	1	2	3
0	X				X				X			
1		X				X				X		
2			X				X				X	
3				X				X				X
0	X				X				X			
1		X				X				X		
2			X				X				X	
3				X				X				X
0	X	X	X	X	X	X	X	X	X			
1	X	X	X	X	X	X	X	X		X		
2	X	X	X	X	X	X	X	X			X	
3	X	X	X	X	X	X	X	X				X
0	X								X			
1		X								X		
2			X								X	
3				X								X
4					X				X			
5						X				X		
6							X				X	
7								X				X

Figure 3.2: Schematic of the routing crossbar, adapted from [63]. Inputs coming from the left are broadcasted horizontally, outputs merge incoming events vertically. Each X describes a location of a crossbar node connecting a horizontal input line with a vertical output line. Their location is static, intersections without an X can't be connected. The input channels can be divided into neuron output channels, four for the left and the right half of the analog neural network core, four external input channels (from the L2) and eight on-chip background events generators. The output channels can be divided into four synapse driver input channels per hemisphere, and four external output channels (to the L2).

output line. It conditionally forwards events based on the rule event & mask $\stackrel{!}{=}$ target, where event is the event label and the mask and the target are configurable 14-bit wide values. Therefore, it allows selection of forwarded events based on their content.

Chip-external events (layer-2 or L2) are used for communication with the outside world, e.g. for feeding-in external spike sources or recording a neuron's spikes. The chip has one external event input and one external event output channel. The external event type is 16 bit wide. External events are connected exclusively to the four input and output channels of the crossbar from and to the L2 via value-based split of the one channel to four channels. The single external input event is forwarded to the crossbar channel of the index selected by the value of the highest two bits in the external event's value. The crossbar event forwarded consists of the lower 14 bit of the external event's value. The four crossbar external event output channels are merged to the single external event output channel by taking the crossbar output event and extending its 14 bit value by the crossbar output channel index placed in the two highest bits of the 16 bit external event's value.

The PADI-Bus connects a synapse driver crossbar output channel to synapse drivers. There are four PADI-buses per hemisphere of the chip. Its event type is 11 bit wide. A PADI-event is formed from a crossbar event by discarding the highest three bits.

Each hemisphere of the chip features a block of 128 synapse drivers. Each synapse driver is connected to one PADI-bus. Figure 3.3 shows this static connection. A synapse driver

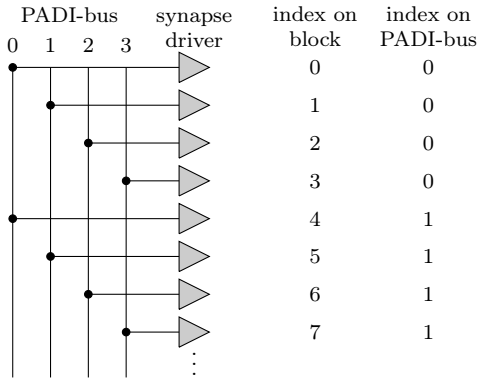


Figure 3.3: Schematic of the synapse driver to PADI-bus connectivity. Each synapse driver is connected to one PADI-bus. Adjacent synapse drivers are connected to different buses. Each PADI-bus is connected to 32 synapse drivers.

filters incoming events based on a static and a configurable entity. The static entity is the index of the synapse driver on its PADI-bus (right in fig. 3.3). The configurable entity is a 5 bit mask. For all bits enabled in the mask the event high-bit has to match the synapse driver's static index on the PADI-bus. If the incoming PADI-event filter forwards the event, the lower six bits may be forwarded as synaptic event to the two synapse rows connecting to one driver. Forwarding of the lower five bits of these six bits can be enabled/disabled, the highest bit is always sent. Events are broadcasted to the synapse driver's two synapse rows.

A synapse is located within one synapse row, of which there are 256 per chip hemisphere (and two per synapse driver). Each synapse locally compares its configurable 6 bit label

value to the incoming event's value and elicits an event on match. A synapse is vertically connected to exactly one neuron.

The neurons are located in one row per hemisphere with 256 neurons each. Each neuron is connected statically to exactly by neuron output channel to the routing crossbar. Figure 3.4 shows the mapping of neurons to neuron output channels. Upon eliciting a spike, the neuron

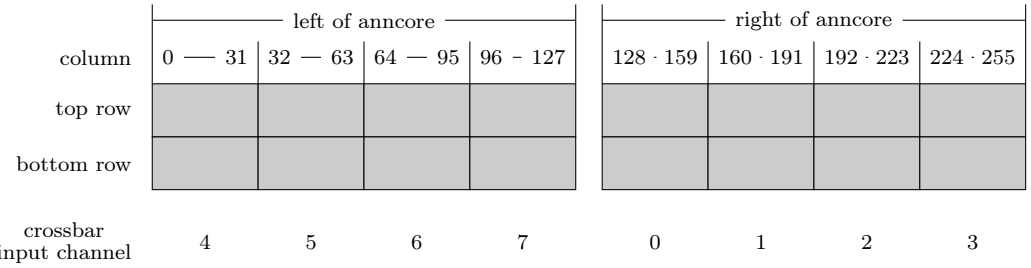


Figure 3.4: Schematic of the connectivity of the neurons. Neurons in horizontal blocks of 32 per row are connected to one neuron output channel into the crossbar. Vertically adjacent neurons are connected to the same output channel.

generates a crossbar event (14 bit wide), where the lower eight bits are configurable per neuron and the highest six bits carry the neuron output channel index.

3.1.3 Software control

The BrainScaleS-2 hardware is configured and controlled via multiple custom software layers. This section describes the purpose and context of the software layers present before this thesis, which are used as a foundation to develop higher-level abstraction. Figure 3.5 shows the software layers. The usage scheme can be split into two parts, a data representation and a control flow part. Due to the realtime nature of the neuromorphic hardware, precise timing during experiment runtime is important. Data representation in this context describes, what is configured where and result data. Control flow describes, what command or configuration is executed when and in which context. [50] explains the concepts of the user-facing hardware abstraction layer in detail. Configuration of chip entities is encapsulated into so-called *container* objects. They provide type-safe, named and possibly ranged access to configurable entities. For example the container corresponding to a synapse contains a ranged integer field for its weight (limited to the range $[0, 64)$). The containers are split into two classes, haldls (hardware abstraction layer for DLS) and lola (logical layer) containers. The former describe the smallest accessible configurable entities, while the latter combine logically connected configurable entities, e.g. analog values corresponding to a certain neuron circuit with the circuits digital settings. A container object describes unplaced configuration and only in conjunction with a so-called coordinate describes a placed configuration on the chip. Unplaced in this context means, that it is not defined which of possibly multiple instances

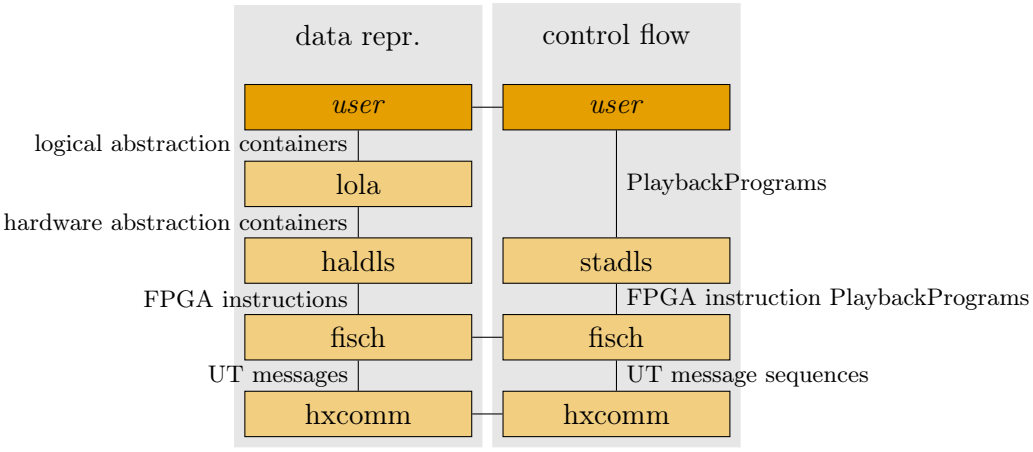


Figure 3.5: Software layers used within this thesis. Left: Configuration data flow abstraction from transport-protocol messages in hxcomm to a register abstraction in fisch to bit-configurable abstraction in haldls and their logical grouping in lola; Right: Runtime control flow abstraction as a linear sequence (also called PlaybackProgram) of objects of the corresponding configuration abstraction level.

of the same type of entity on the hardware are described. Coordinates define the selection of a single entity. They are type-safe, strictly-ordered, (typically) ranged and geometric (e.g. linear or 2d-grid) location descriptors, which ensure that a given configuration can only be accessed at the right memory location, e.g. a synapse can't be falsely accessed at the location of a neuron configuration.

Universal translator (UT) messages [41] are instructions to and responses from the FPGA, which are convertible to and from a bit-stream for transport. Converting a hardware abstraction layer configuration container to or from a sequence UT messages for the FPGA is done in two steps. The hardware abstraction layer containers can be encoded and decoded to and from sequences of register-like word-address pairs. These type-safe heterogeneously-sized words reside in fisch, which provides a back-end-independent translation of word to and from sequences of FPGA commands.

Runtime control is managed by execution of a linear sequence of commands in a state-machine on the FPGA. Compiling such linear sequences, also called playback programs, is abstracted in software by a builder pattern forming a sequence by appending and concatenation. Configuration writes generate a sequence of commands, whereas reads yield a future-like object to access result data after successful execution. In addition to deterministic response data corresponding to configuration reads, non-deterministic event data is accessible via the playback program structure after execution. Compilation is independent of hardware access, allowing separation of program construction and actual execution.

Hardware access is encapsulated by so-called connections in hxcomm (HICANN-X commu-

nication). A general and extensible interface requiring only ability to consume a sequence of commands and returning a sequence of responses, allows providing multiple different types of connections. Currently, a ARQConnection connects to an FPGA via a custom transport-layer protocol and SimConnection connects the full software stack to a software simulation of the hardware at the register-transfer level.

This software interface allows timed type-safe and ranged access to configuration of the hardware. Configuration relations, e.g. which neuron is connected to which synapses, is only visible implicitly via conversion functionality of the coordinates. This is the main limitation of this software representation and leads to a discontinuity between experiment model and actual hardware representation, since experiment models typically are a structured formulation with dependencies and not a flat collection of seemingly independent parts.

3.1.4 Performance expectation for non-spiking experiments

Performance evaluation of non-spiking multiply-accumulate (MAC) operations benefit from a specification for an expectation. As performance metric, the rate of MAC operations is used. To find an upper bound of the achievable rate for BrainScaleS-2, data transport bandwidth to and from the system is proposed, because it yields a hard limit. The actual operation on the hardware is then assumed to be fast compared to data transport, so that it can be neglected. It becomes clear, that using these assumptions, maximal performance is reached when using the full chip area. We describe the shape of a synapse matrix by height h and width w, where h coincides with the number of inputs to be sent to the hardware and w coincides with the number of outputs to be read from the hardware. Input values are 5 bit unsigned, output values are 8 bit signed integers. Data transport is performed with overhead introduced by packing the values into larger messages. Additionally, reads from the hardware might require transport of information about what to read first. To describe this overhead, required data transported per logical value is used, where d_{in} and d_{out} describe input and output data and d_{trigger} describes the data transported to the chip required to initiate a read. Given a raw full-duplex transfer rate r, the achievable rate r_{ops} is calculated like:

$$r_{\text{ops}} = \frac{r \cdot h \cdot w}{\max(h \cdot d_{\text{in}} + w \cdot d_{\text{trigger}}, w \cdot d_{\text{out}})} \tag{3.1}$$

Table 3.1 shows the set of parameters of the current prototype hardware. A maximally achievable rate of $4\,\text{Gop}\,\text{s}^{-1}$ is calculated for this set of parameters. A reasonable altered configuration is constructed, when changing the data transport bandwidth to the chip-FPGA link of $8\,\text{Gbit}\,\text{s}^{-1}$, while conserving all other parameters. This would be achieved by improving the host-FPGA link or connecting to the chip directly. It leads to an expected achievable rate limit of $32\,\text{Gop}\,\text{s}^{-1}$. A way to test this limit is introduced in section 3.4.2.

parameter	value
rate r	$1\,\mathrm{Gbit\,s^{-1}}$ (Ethernet)
height h	128 (signed)
width w	256
data d_{in}	32 bit (two 8 B messages for four 5 bit values)
data d_{out}	16 bit (one 8 B message for four 8 bit values)
data d_{trigger}	16 bit (one 8 B message for four 8 bit values)

Table 3.1: Set of parameters to estimate the upper bound of MAC operations via data transport for the current hardware prototype. We ignore Ethernet-protocol overhead for the data transport rate.

3.2 Graph-based experiment notation

Revisiting the lower-level software description in section 3.1.3, it becomes clear, that a therein described flat representation of an experiment on neuromorphic hardware as a collection of configuration and control-flow statements is feature-complete to describe arbitrary hardware-compatible experiments. However, the relation of such a collection to a logically intuitive description of an experiment is not ideal in that it does not resemble interconnections between elements in the collection. Additionally, other constraints and connections like plasticity rules or digital operations are not contained.

Spiking neural networks are predominantly described as their name suggests as a network or graph structure, where single neurons or collections thereof are described in combination with their connectivity [15, 31, 32]. Similarly, the two major machine-learning frameworks PyTorch and Tensorflow rely on a graph-based description of computation (Tensorflow) or are moving into this direction (PyTorch with their JIT intermediate representation [23]). Moreover, the BrainScaleS-2 neuromorphic hardware is inherently data-flow-centric due to its event-based nature, as described in section 3.1.2.

All this speaks for a graph-based experiment description for the BrainScaleS-2 hardware. The path from the user to the hardware is then described by a graph transformation. We strive to integrate this representation at the lowest level of abstraction possible in order to fully incorporate all hardware features without premature abstraction. This shall allow for all higher-level abstraction to utilize this representation.

In the following we start by an introduction to the graph-theory needed in section 3.2.1 followed by a short introduction to signal-flow graphs in section 3.2.2. Afterwards we focus on the application of these concepts for development of a signal-flow graph based description of experiments on the BrainScaleS-2 hardware in section 3.2.3.

3.2.1 Directed Graph

This section introduces the graph theory needed and establishes nomenclature used throughout the document. The mathematical foundation is based upon [5], while we mix-in nomenclature used by the boost graph library [7] in order to easier understand its usage during the implementation.

A directed graph $D = (V, E)$ is constituted of a finite set $V(D)$ of elements called vertices and a finite set $E(D)$ of ordered pairs of distinct vertices called arcs or edges. The first vertex in the ordered pair of an edge is called tail or source, while the second vertex is called head or target. The vertices of an edge are called its end-vertices. End-vertices are called adjacent to each other. The above definition of the set of edges forbids parallel edges, i.e. multiple edges with the same source and target vertex as well as loops, where the source equals the target. Figure 3.6 shows a exemplary directed graph.

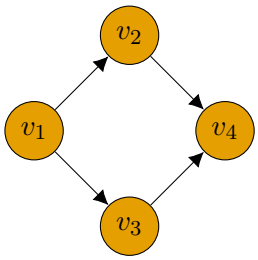


Figure 3.6: Directed graph example. The vertices are $V = \{v_1, v_2, v_3, v_4\}$ and the edges are $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)\}$. The presented graph is acyclic.

For a given vertex v in a directed graph, we describe its surrounding

$$N_v^+ = \{u \in V - v : (v, u) \in E\}, \quad N_v^- = \{u \in V - v : (u, v) \in E\}, \tag{3.2}$$

where N_v^+ is its out-neighborhood, the vertices which are connected by edges with v as source, and N_v^- is its in-neighborhood, the vertices which are connected by edges with v as target. The number of vertices in the out-neighborhood describe the vertex v's out-degree, while the number of vertices in the in-neighborhood describe its in-degree.

A directed graph H is called a subgraph of a directed graph D, if $V(H) \subset V(D)$ and $E(H) \subset E(D)$ such that $\forall(u, v) \in E(H) : u, v \in V(H)$.

A directed walk of length k is a sequence $W = v_1e_1v_2e_2v_3...v_{k-1}e_kv_k$ of vertices v_i and edges e_i, where v_i is the source of e_i and v_{i+1} is its target. If the vertices $\{v_i\}$ are distinct, $k \geq 3$ and $v_1 = v_k$, then the directed walk is called a directed cycle. A directed cycle of length 2 is formed by $v_1a_1v_2a_2v_1$ and a directed cycle of length 1 is a loop. A directed graph is called acyclic, if there don't exist directed cycles. It can be shown (e.g. [5]), that the vertices of every acyclic directed graph D can be in acyclic ordering, which means, that for $\forall(v_i, v_j) \in E(D) : i < j$. This allows visitation of vertices in a directed acyclic graph such that for each visited vertex the in-neighborhood is guaranteed to have been visited already. This property is ideally suited for execution of a graph, where availability of incoming data

is prerequisite for execution of a vertex.

3.2.2 Signal-flow graph

A signal-flow graph [47] is a directed graph, where each vertex is interpreted as receiving input signals from its in-neighbors and transmitting a output signal generated by using the input signals in some manner to its out-neighbors. Figure 3.7 shows a exemplary signal flow graph. While historically signal-flow graphs are used for describing analog electric

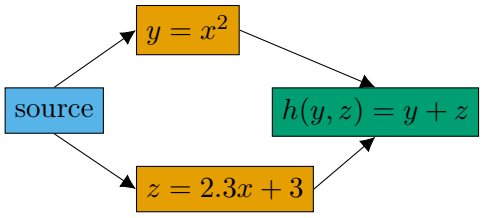


Figure 3.7: Signal-flow graph example: following a source vertex, data is transformed by two parallel operations (y and z) at the top and bottom vertex in the middle. Their results are transformed in the right vertex (h).

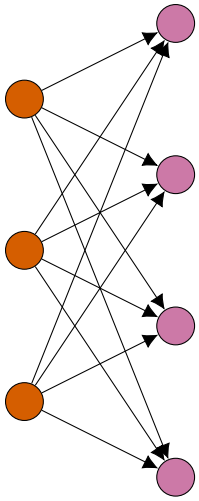
circuitry [47], the model is not limited to this type of signals.

3.2.3 Neuromorphic hardware as signal-flow graph

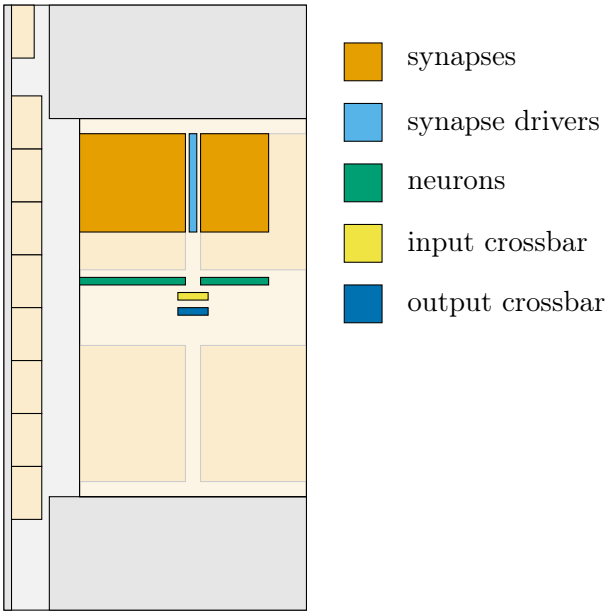
Using the BrainScaleS-2 neuromorphic hardware can be described in form of a signal-flow graph. Firstly, a given placed static network configuration can equivalently be described in a signal flow graph. Secondly, virtualized digital computation (on the PPUs) will be integrated. Afterwards, data-flow between multiple realtime experiment executions on multiple chips or at multiple times will be described. Following, advantages of such a representation are highlighted. Lastly, a time evolution model for realtime execution sections is described.

Static network configuration

Section 3.1.2 describes the digital event routing entities on the BrainScaleS-2 platform. Analog signals from the synapse drivers over the synapses to the synaptic inputs of neurons onto their membranes bridge the digital events inside the analog neural network core, cf. section 3.1.1. Figure 3.8 shows a placed configuration of a simple feed-forward network on the hardware. Signal-flow is only implicitly known by knowledge of which placed entity is connected to which other entities. Representation as a signal-flow graph is displayed in fig. 3.9 and explicitly tracks connectivity between used entities on the hardware. We choose the configurable entities on the hardware to be represented as vertices in this graph, while the graph edges track the connectivity between these entities. Additionally, the set of allowed linked vertex and edge combinations is ensured. Table 3.2 shows an overview of the different vertices, their expected neighborhood and a brief explanation of the transformation they perform to received signals.



(a) Abstract feed-forward network of one input layer (blue) and one output layer (red) with all-to-all connectivity in-between.



(b) Feed-forward network placed on the BrainScaleS-2 hardware. Aside the synapses and neurons, synapse drivers and parts of the crossbar are used. The configuration is flat, the connectivity between the configured entities is only known implicitly, e.g. because the synapse column of a specific synapse coincides geometrically with the location of a specific neuron.

Figure 3.8: Left: Simple abstract feed-forward network; Right: Same network placed onto (mostly the northern hemisphere of) the BrainScaleS-2 chip.

	signal type		
entity	input	output	transformation
ext. event input	ext. event label	crossbar input label	digital match-based splitter
crossbar node	crossbar input label	crossbar output label	digital mask-based filter
PADI-bus	crossbar output label	PADI label	digital narrowing forwarding
synapse driver	PADI label	synapse label, analog pulse	digital mask-based filter, configurable analog pulse generation
synapse	synapse label, analog pulse	synaptic analog pulse	digital match-based filter, analog signal weighting
neuron	synaptic analog pulse	membrane potential	adaptive exponential leaky integrate and fire analog neuron
neuron back end	membrane potential	crossbar input label	spike detector
MADC readout	membrane potential	14 bit samples	ADC
CADC readout	membrane potential	8 bit samples	ADC
ext. event output	crossbar output label	ext. event label	digital origin-annotating merger
on-chip event generator	-	crossbar input label	regular or Poisson spike source (configurable)

Table 3.2: Hardware entities representing parts of a static network configuration as signal-graph vertices. For each entity the input and output signal types are given together with a short description of its performing transformation. For a detailed explanation of the transformation and the propagated digital signals see section 3.1.2.

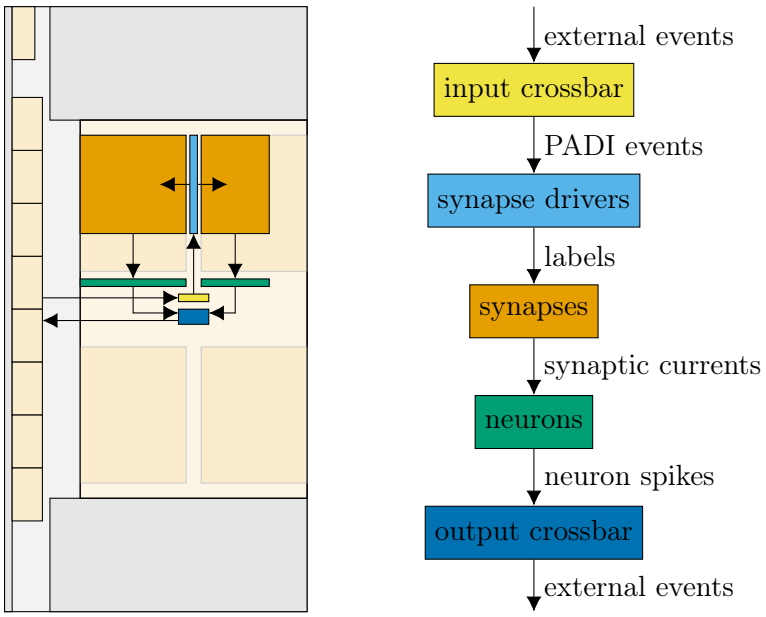


Figure 3.9: Placed feed-forward network (cf. fig. 3.8) represented as signal-flow graph. Left: Actual layout on the chip, the arrows represent the graph edges; Right: The network graph structure enlarged with signal type annotation on the edges.

Virtualized digital computation

In addition to the static routing, neuromorphic experiments typically involve digital computation (e.g. [6, 76, 65]), because BrainScaleS-2 incorporates embedded general purpose processors. For example parts of a learning rule to update weights are implemented as digital operation. In the non-spiking operation mode of BrainScaleS-2, operations like rectified linear units (ReLU) or bias addition can be represented as digital operations.

Data-heavy digital computation greatly benefits from data locality [71]. Therefore, usage of the inherently local embedded general purpose processors, the PPUs, is proposed for implementing such operations.

To express the locality of such operations, they are represented as vertices in the same signal-flow graph as the static routing. Therefore, the neighborhood of such vertices incorporates all data paths from and to an operation and therefore completely expresses its locality-property towards other operations and entities in the graph. This procedure has been shown to be useful additionally in intermediate representations for locality-aware optimization [12, 59], when the operations are known in an abstract form.

Digital operations which calculate a result from given inputs require these inputs to be present upon calculation. This implies, that such operations can only be embedded in acyclic subgraphs of the signal-flow graph representation.

In the course of executing an experiment, the PPUs are also involved in control-flow. To

merge this usage with the proposed digital operations, a virtualized representation is chosen. This means, that the detailed sequence of operations and their time-share on a given PPU is not explicitly represented in the signal-flow graph, but subject to optimization based only on the signal-flow for a given compiler. Figure 3.10 shows the signal-flow graph representation of a matrix multiplication using the non-spiking mode of the analog neural network core followed by a digital ReLU operation.

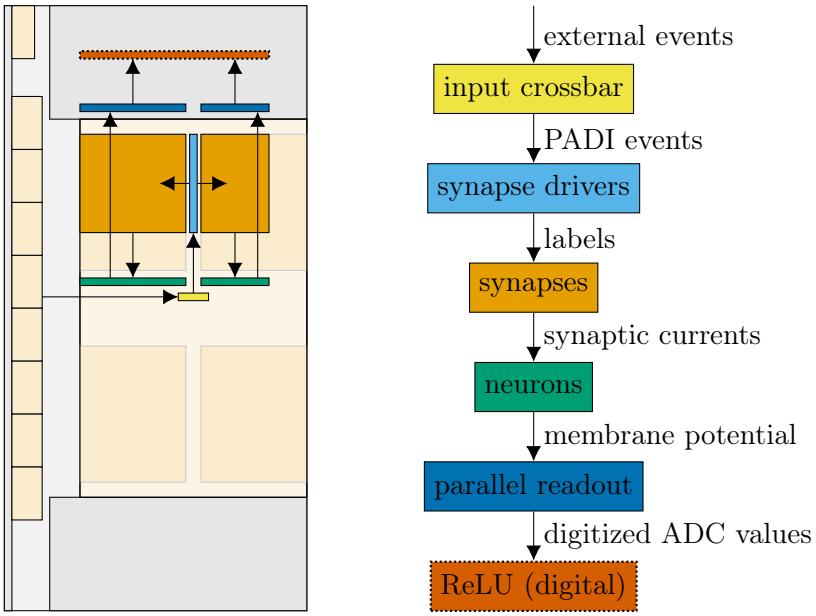


Figure 3.10: Placed matrix multiplication in the analog neural network core followed by a digital ReLU operation represented as signal-flow graph. Left: Actual layout on the chip, the arrows represent the graph edges. The digital operation placement is artificial, since it is virtualized; Right: The network graph structure enlarged with signal type annotation on the edges. The parallel readout performs the transition from analog to digital values.

Multiple chips and/or multiple executions

A single experiment might involve multiple physical chip instances and temporal reuse of these instances. For example a static-parameter sweep of a given experiment results in multiple realtime executions. Likewise, implementing a learning rule in such a sequence of realtime executions involves data dependencies between the individual runs, because for example the weights of the next execution might be a result of an observable of the last. However, the connectivity between individual realtime executions can't be realtime, so recurrent spiking connections are bound reside on one physical chip currently due to lack of realtime inter-chip communication. Individual non-recurrent layers can however be placed on different physical chip instances or temporal instances of the same chip at the expense of

executing sequentially with regard to data-flow. Figure 3.11 shows an experiment consisting of multiple realtime executions distributed across two physical chips. In the signal-flow

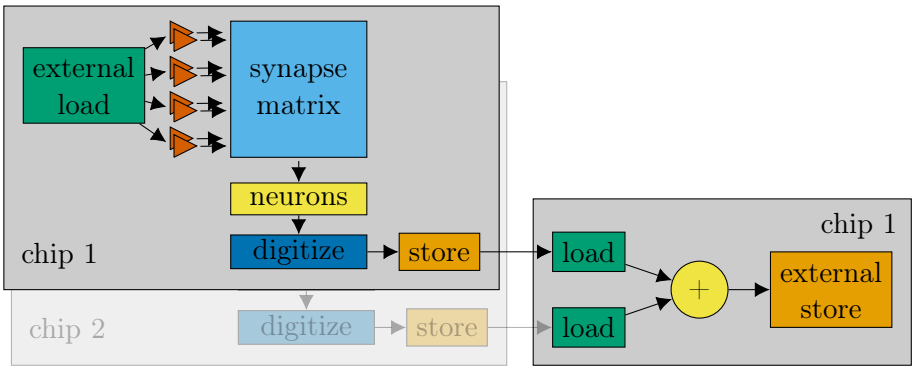


Figure 3.11: Non-spiking network distributed over two physical chips, adapted from [66]. Two matrix multiplications on chips 1 and 2 follows a digital addition of the results, executed on chip 1. The latter execution instance depends on the output of the two former instances.

graph, all vertices are placed in a single graph, where their physical and temporal location is represented as a vertex property. In the following, a specific physical and temporal chip instance will be called *execution instance*. Filtering for a specific execution instance then results in a subgraph, which can for example be used to compile its localized part of a graph. Edges between subgraphs of different execution instances thereby represent the directed data-flow between these executions.

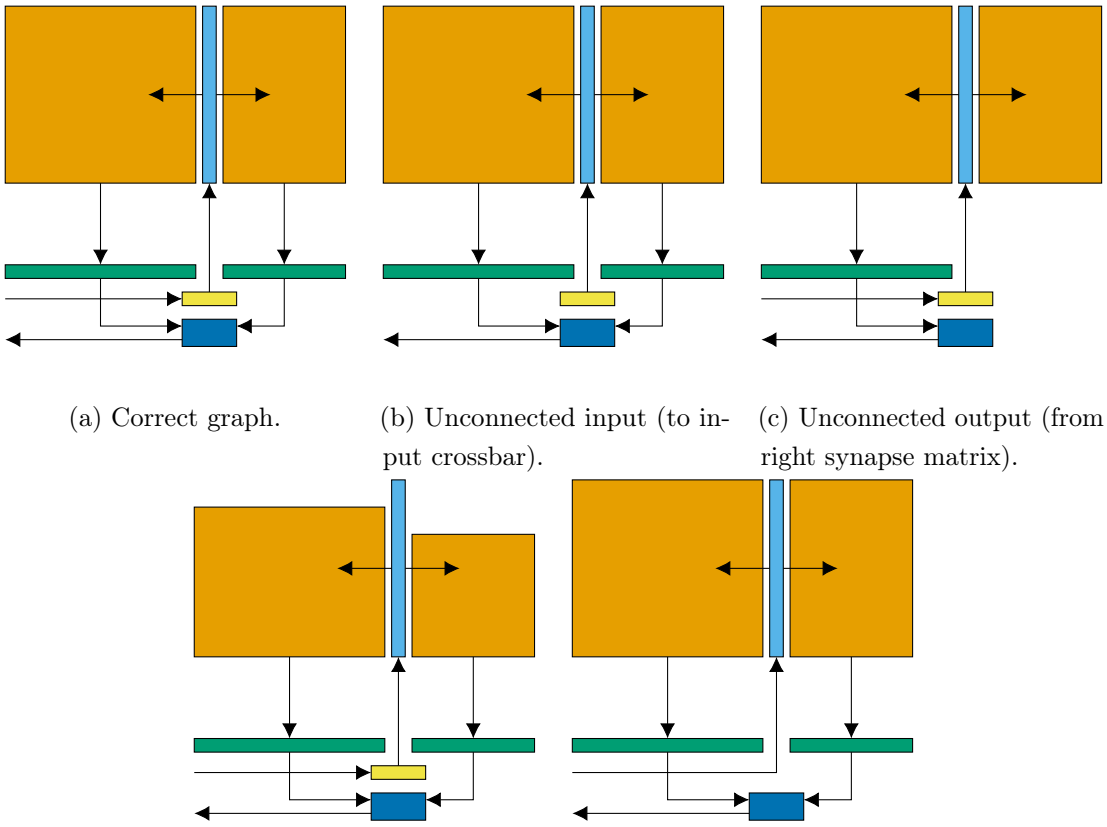
Advantages of graph-based hardware representation

Representation of an experiment on neuromorphic hardware via a signal-flow graph allows for several possible advantages.

Firstly, because of the explicit description of signal flow, its validity can be evaluated and checked. For example given a placed vertex, its expected incoming neighborhood is known and can be verified to match its actual incoming neighborhood in the graph. Similarly, unconnected inputs to or outputs from vertices can be avoided. Additionally, acyclicity for subgraphs containing digital operations can be ensured. These checks are useful in avoiding inherently dysfunctional experiments due to missing pieces of configuration or logically impossible constraints. Figure 3.12 visualizes the described checks.

Already touched on at the description of the static network configuration, representation as a signal-flow graph directly allows visualization as such. This allows for a human-readable representation of a network, which eases debugging. Therefore, it is a typical feature in graph-based frameworks, e.g. boost.graph [7] allows export for a graphviz-based visualization [27]. An example of such a representation can be seen in the later implementation in section 4.1.1.

The signal-flow graph representation inherently describes the relationship between entities



(a) Correct graph.

(b) Unconnected input (to input crossbar).

(c) Unconnected output (from right synapse matrix).

(d) Shape mismatch (synapse driver too large for synapses).

(e) Missing vertex (input crossbar).

Figure 3.12: Visualization of possible checks of vertex neighborhood for a signal-flow graph representing an experiment on neuromorphic hardware.

on the hardware. It is therefore the ideal source of information for optimization algorithms, because they rely on this relationship information. In contrast, only knowing the flat collection of configured entities on the hardware would complicate finding out, which *surrounding* entities to change upon alteration of one entity, because the notion of surrounding is only given via implicit knowledge. Figure 3.13 exemplarily shows an optimization of used synapse drivers, which is only possible because of knowledge of the connected synapses.

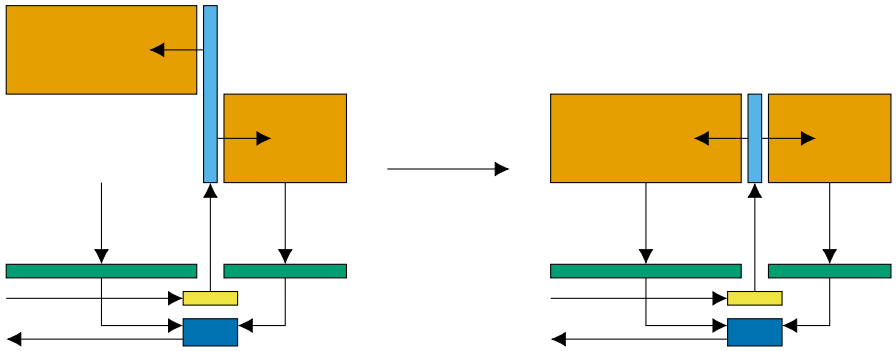


Figure 3.13: Optimization example based on the signal-flow representation of the hardware configuration. The amount of used synapse drivers can be reduced by moving synapses. This is possible via knowledge of the relationship between synapses and synapse drivers as well as routing-specific configuration, which might need adaptation, e.g. synapse labels because of overlap once the synapse drivers are shared between left and right. Left: Unoptimized configuration; Right: Optimized configuration.

Realtime time evolution

While the signal-flow graph representation describes the signal flow, the actual signals are not part of the representation. They are either to be provided separately (e.g. input spike-trains), will only be present locally upon execution (e.g. synaptic current pulses or intermediate digital results) or will be generated by an execution (e.g. recorded output spike-trains). Typical experiments consist of an initial static configuration followed by realtime time evolution (e.g. [65, 13]). Given the approx. 1000-fold faster time constants compared to biological time constants of the hardware [1], the initial static configuration duration might already be within the same order of magnitude as an actual realtime execution. For example only configuring the synapse memory requires the transmission of 1 MB of data via the $1\,\mathrm{Gbit\,s^{-1}}$ Ethernet to the FPGA, which roughly corresponds to 8 ms transmission time, translating to 8 s biological time equivalent.

Therefore, we link one initial static configuration to a collection of sequentially executed realtime executions. In the non-spiking mode of operation, each element of the collection of realtime executions then represents a batch entry, trace or image to be presented. Likewise,

in spiking experiments, each element of the collection of realtime executions can for example be used to present different input spike-trains. This greatly reduces the relative duration of the initial configuration for each realtime execution with $\frac{1}{N}$ for a collection of N realtime executions. An example for such a collection is a test or training dataset, where each entry corresponds to one realtime execution. Figure 3.14 shows a single execution consisting of one initial configuration followed by a collection of realtime executions with input spike-trains.

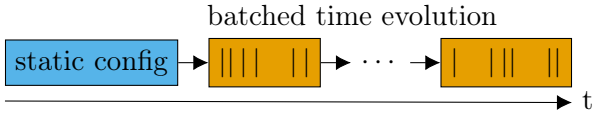


Figure 3.14: Time evolution of a single execution. One initial static configuration is sequentially followed by possibly multiple realtime executions with input spike-trains displayed by the vertical lines within the executions.

A realtime time evolution is described by a contiguous time interval and a collection of time-series data, for example an input spike-train. We define each realtime time interval to start at logical time 0, therefore the time interval is fully described by specification of the interval duration, also called *runtime*.

3.3 Front ends

The signal-flow graph representation of an experiment on neuromorphic hardware as described in section 3.2.3 using configurable hardware entities can be used directly. However, direct usage requires explicit knowledge of the location and configuration of all hardware entities. For example, it does not allow description of abstract unplaced networks and their algorithmic lowering to a placed graph representation. This limits the reachable level of abstraction of this representation. Reoccurring pattern in the hardware graph are not deduplicated. These limitations especially become apparent when the network size increases.

To overcome these limitations we aim to provide support for front end software, which allows abstract network specification and is widely used in the machine learning or neuroscience community. This then allows implementation of algorithmic mapping and routing in-between the front ends and the hardware graph representation. Used in the neuroscience community, PyNN [15] is targeted as front end for spiking neural networks, describing them via populations, collections of neurons, which can be connected via projections, collections of synapses. Tensor-based libraries are used most commonly [36] for machine learning, predominantly PyTorch [53] and Tensorflow [2]. We found Pytorch is simpler to integrate in the group's build flow, because it relies on the standard `cmake` build tool and tracks out-of-tree dependencies with standard mechanisms. Additionally, its documentation is deemed sufficient for extending. Therefore, we target PyTorch as second front end [66].

3.3.1 PyNN

PyNN [15] is a framework to describe spiking neural networks and simulate/emulate their behavior. Its building blocks are populations as collection of neurons and projections as collection of (synaptic) connections between neurons.

A Population is a collection of (possibly) multiple neurons of homogeneous type, but possibly heterogeneous configuration. A non-owning PopulationView allows masked access to a subset of neurons of an existing population. Similarly, collections of population(-view-)s are accessible via an Assembly.

A Projection is a collection of (possible) multiple connections between single neurons of two (not necessarily different) population(-view-)s or assemblies with homogeneous type but possibly heterogeneous configuration. Specification of the set of connections to construct is done via a Connector, which forms a generation-rule for single connections. For example the AllToAllConnector construct a connection for every pair of neurons between the two populations, while the FixedProbabilityConnector constructs connections randomly with fixed probability.

A Recorder allows registering access to observables of neurons in populations. Examples are spikes or the membrane potential.

Execution can be triggered (and continued) for a specified runtime interval. Figure 3.15 shows the structure of the representation for an exemplary network.

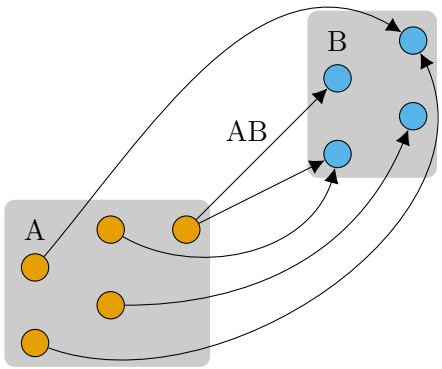


Figure 3.15: Network as represented in PyNN. Population A consisting of five neurons is connected to a population B consisting of four neuron via a projection AB. The projection only connects a subset of neurons of both populations, presynaptic neurons might be connected to multiple post-synaptic neurons as can post-synaptic neurons feature multiple presynaptic partners.

PyNN allows selection from a multitude of back ends for implementing the simulation like Nest [31] or Brian [67]. Similarly, this group has already provided support for emulation on the two predecessor platforms Spikey [62] and BrainScaleS-1 [51].

As can be seen in fig. 3.15, the PyNN representation forms a graph structure with populations as vertices and projections as edges. We therefore will provide a back end for BrainScaleS-2 by linking this graph to a hardware graph representation according to section 3.2.3. Acquisition of the hardware graph representation forms the mapping and

routing algorithm. Each population and projection in the PyNN graph is connected to a subset of vertices in the hardware graph, for example a projection will be connected to a collection of synapses in the synapse array. Figure 3.16 illustrates this connection between PyNN graph and hardware graph description at a network example.

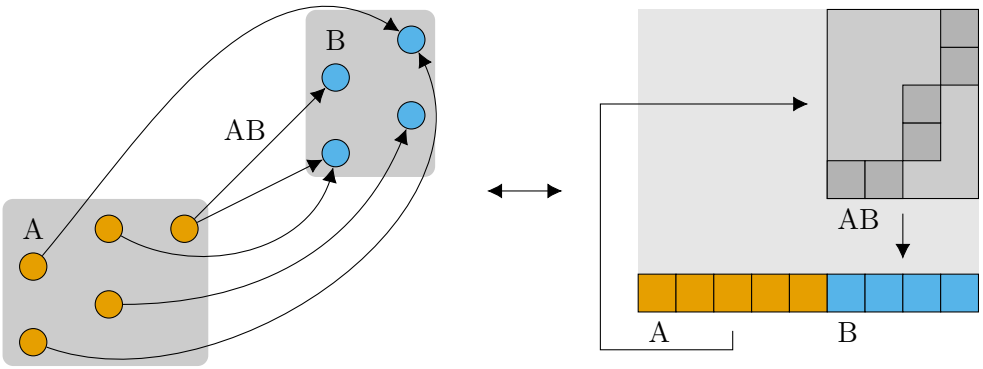


Figure 3.16: Network as represented in PyNN (left) aside mapped hardware graph representation (right). The populations A and B correspond to a set of hardware neurons (blue and red squares). The projection AB corresponds to a set of synapses in the synapse array. The connection between PyNN entities and hardware graph vertices is bidirectional, allowing in addition to transforming from PyNN to hardware also backward-mapping from the hardware representation to the PyNN entities.

3.3.2 PyTorch

PyTorch [53] is a framework for performing tensor operations. It is geared towards usage for machine learning especially due to its integrated support for gradient calculation. This section gives an introduction to basic building blocks which will be used and then will focus on the roadmap for integration of the BrainScaleS-2 hardware, which will be oriented along the reasoning provided by the Author in [66].

Values are stored and accessible as Tensor, which represents a multi-dimensional array of homogeneous type. Calculations are performed on tensor values via operations like for example `torch.matmul` performing a matrix multiplication.

Neural networks are described via a so-called Module. It is a stateful object, which contains parameters like weights alongside a procedure to apply itself onto given input values by calling its `forward` method, which consists of tensor modification(s) via operation(s) using the module's parameters.

The true power lies in the integrated support for gradient calculation, which is called

autograd in PyTorch. When describing a network as compute graph, the parameters and inputs are root vertices (with in-degree of zero). Each operation in the compute-graph (i.e. each non-root vertex) features a formula for the forward direction for calculating the output from the inputs. In addition, it also features a formula for the backward direction (calculating the inputs' gradients from a gradient given for the output). Traversing the compute-graph backwards allows collection of all these backward formulas (called gradient function in PyTorch) for every parameter and input of a network. These functions are registered in the parameter or input tensor objects.

Therefore, adding a new operation by providing a forward and backward formula allows complete integration of new functionality into this framework. In the non-spiking mode of operation, BrainScaleS-2 provides support for (analog) matrix multiplication and thereby also convolution, in the analog neural network core, in conjunction to (performance-wise limited) support for additional digital operations via the PPUs. By adding these as operations in PyTorch, they can directly be used alongside built-in operations. While the forward pass implementation is clear in that it is performed using the BrainScaleS-2 hardware, the backward pass can't easily be implemented on the hardware due to temporal and fixed-pattern noise and limited digital precision [75, 74]. However, using the results obtained from the forward pass in conjunction with a model allows calculating an approximation for the gradient in software. This approach is adapted from [64], where a spiking neural network is trained by using the results from a hardware execution on BrainScaleS-1 in combination with a software model. It is called training with *hardware in the loop*. The model used here is however much simpler in that it only resembles a matrix multiplication.

3.4 Profiling tools

During development of performance-oriented software, optimization is a vital part. As will be laid out in section 3.4.1, this additionally stays true for production use. In our case, program runtime is used as primary performance metric. Therefore, we investigate how to properly add runtime information with minimal overhead for both development and production use. Secondly, we want to investigate performance of the full software stack under artificial alteration of the performance of the hardware. Therefore, a simple fast mock will be introduced in section 3.4.2.

3.4.1 Runtime tracing in production software

The BrainScaleS-2 hardware is designed to be used as *accelerator* for emulating spiking and non-spiking neural networks.

Firstly, this implies, propagating information about the time spent executing on the hardware is essential. Optimization goal therefore is maximizing this time in relation to the

time spent in surrounding software both during development and during usage.

The to be developed software interfaces will allow usage resulting in variable degree of efficiency. For example given a fixed abstract network can be represented by a signal-flow graph of varying vertex granularity. Algorithms operating on the collection of all vertices will most certainly be slower for a fine-grained description with many vertices, e.g. describing a single synapse each, while they may be faster on a description with few vertices, describing multiple synapses as a block each. This implies, propagating information about the time spent executing on the hardware is essential to raise awareness of performance impact.

Therefore, manual instrumentation of interesting sections is used to provide both developer and user with runtime information. Listing 1 shows pseudo code for such a runtime tracing. Since as explained above this instrumentation is to reside in the program also for production

```
1  begin = now()
2  // Interesting section.
3  end = now()
4  log(end − begin)
```

Listing 1: Pseudo code for manual instrumentation of runtime logging.

use, minimizing its own runtime overhead is crucial. This is ensured by choosing coarse enough sections to trace runtime-wise compared to the overhead introduced by one interval measurement. The measurement method is evaluated in section 5.1.1.

3.4.2 Hardware mock

A intrinsic property of developing software to configure and control special hardware is that typically full-stack performance evaluations and integration tests are limited to using the special hardware currently available. Reasons to try to circumvent this restriction are that on the one hand the hardware might not always be available for testing and on the other hand that the performance of the available hardware prototypes does not (fully) match the target of the software development.

In the case of this thesis the latter is the main motivation. The currently available hardware features a $1\,\mathrm{Gbit\,s^{-1}}$ Ethernet connection between FPGA and host computer, while the chip itself is connected by a $8\,\mathrm{Gbit\,s^{-1}}$ link. The Ethernet bottleneck is temporary and therefore the developed software shall be capable to cope with speeds comparable to the chip link speed. This is particularly interesting for the non-spiking mode, where results are transported back to the host computer solely as response to deterministic read instructions.

The software will be designed in a way to perform independently of the actual payload of the transferred result data. Therefore, the simplest complete model of the hardware for this

use-case is to ignore all instructions but read instructions and yield a compile-time constant (e.g. all bits of the payload set to zero) for each read instruction. Listing 2 shows this model's execution as pseudo code. This model ignores all other instructions like sleeps or other

```
EXECUTE(instructions, t_message)
1  t_begin = now()

2  responses = EMPTY-LIST
3  for i = 1 to instructions.length
4       if is-read(instructions[i])
5            append(responses, 0)

6  t_actual = now() − t_begin
7  t_total = t_message · instructions.length
8  wait(t_total − t_actual)

9  return responses
```

Listing 2: Pseudo code for simple hardware mock model execution which handles reads by responding with zero and ignoring all other instructions.

time-consuming operations. Additionally, pipelining and buffering on different stages like on the FPGA is disregarded. The model therefore is an upper bound against an actual system with given connectivity.

By recording the amount of time spent for actually processing the given instruction sequence t_{actual} of length N and waiting afterwards t_{wait}, a customizable average amount of time per processed message $t_{\text{message}} \cdot N$ can be set like $t_{\text{total}} \stackrel{!}{=} t_{\text{actual}} + t_{\text{wait}}$. Given that the actually spent time t_{actual} is smaller than the target total time t_{total}, this approach allows precise simulated performance of the mock for a sequence of instructions. Precise timing for a sequence of instructions is sufficient, because all users of the API, cf. fig. 3.5 operate on sequences and therefore precise timing of a single instruction on the host computer is not required. The hardware mock performance is evaluated in section 5.1.2.

3.5 Development tools

The software developments made within this thesis involve interaction with and choices about the development environment. Firstly, we reason about the choice of `C++` as primary programming language in section 3.5.1. In section 3.5.2, we describe the development environment, which is present within the group, specifically the process of creating and verifying changes to software. Following, two advancements made within the course of this

thesis are highlighted. In section 3.5.3 we describe integration of automated code style verification as a method to increase homogeneity of code to ease human readability. In section 3.5.4 we evaluate static code analysis tools for C++, which are similarly integrated for automated evaluation. Finally, we conclude by developing a strategy for verification of the developed software's function in section 3.5.5.

3.5.1 Language(s)

The lower software layers described in section 3.1.3 are implemented in C++ for performance reasons and its facilities for compile-time correctness evaluation [50]. Additionally, the interfaces are exposed to Python as scripting language for interactive usage [50]. This is realized using pybind11 [38] wrapping, which is automatically generated via the group-developed genpybind [42]. PyTorch is implemented in C++ as well using also pybind11-based Python wrapping. PyNN is implemented solely in Python.

The software developed within this thesis greatly benefits from the ability to perform compile-time checks. Additionally, minimizing overhead when using lower software layers is crucial, when the overall performance is important as described in section 3.4.

Because of its ability to easily interface both lower-level software and the planned front end PyTorch as well as its potential for good performance, C++ is chosen as primary language for development. For interfacing PyNN and providing Python bindings usable alongside PyTorch, pybind11-based Python wrapping will be used similar to the lower software layers.

3.5.2 Development environment

The group uses code-review for all (software and hardware) development in the form of Gerrit [28]. So-called change sets are uploaded aside the production state and only merged into the production stack upon positive review. Code-review can be divided into two parts, human review and automated review. The former is supposed to perform review on the content of a change, while the latter in the case of software builds it, executes and evaluates tests and performs a binary success vote. This automated continuous integration (CI) is performed with Jenkins [4]. Being very customizable it allows for development of fine-grained verification [68].

3.5.3 Code-style

Consistent style of code eases readability for humans and therefore improves efficiency in trying to understand unknown code. The group already used the Clang-Format [9] formatting tool for C++ on a voluntary basis, i.e. only enforced via human code-review. Similarly, Python code-style is ensured using Pycodestyle [58], however enforced via CI. During this thesis it became apparent, that automated verification of code-style is beneficial

in that it relieves human reviewers from doing this and leads to more consistency, because it is never overlooked. In order to ease gradually improving consistency of code-style within existing projects, change-based-difference style-checking is used via Git integration. This is integrated into CI by enforcing no alteration when applying the formatting tool onto a change.

3.5.4 Code-linting

Static code analysis can help reduce overlooked mistakes and thereby improve code quality. The group uses Gcc [70] as main compiler and enforces change sets to be free of compiler warnings in CI. This already greatly reduces the possibility for easily-fixable errors like missing return types or narrowing casts. Furthermore, dedicated static code analysis tools can be used to improve coverage. Two such tools are Cppcheck [46] and Clang-Tidy [10]. Both are integrated into CI during this thesis to allow automated evaluation. The first one by default uses a custom parser (there is experimental support to use Clang instead). Application of Cppcheck on the already existing software stack caused preprocessor parsing errors, which were found hard to circumvent. The latter is expected to be free of such problems, since the software stack can already be parsed by Clang, because it is used for the automated Python binding generation.

3.5.5 Test strategy

For verification of the developed software a combination of unit tests and integration tests is used. For increased test coverage, randomization of the test parameterization is applied as much as possible. Depending on the part of the software at hand, full test coverage by randomization is not an achievable goal. For example testing the interface of the signal-flow graph description via randomized graphs would require a complete re-implementation of the construction constraints described in section 3.2.3. Therefore, while striving for randomized tests, we resort to interface testing via fixtures when the configuration space is too complex to easily randomize sampling from it. Integration tests are used for verification of the compilation and execution process of the signal-flow graph representation and for the front ends.

The signal-flow graph layer is tested almost exclusively in `C++` using the Googletest testing library [33]. Contrary, the front ends for PyNN and PyTorch are tested in Python with the Unittest [56] testing library due to easier parameterization and them being either exclusively based in (which is the case for PyNN) or targeting Python.

Both testing libraries are used in the group for testing in their respective language and their results are convertible to a format supported by Jenkins. This allows direct integration of all tests into CI for automated verification of changes and developments.

4 Implementation

This chapter describes the implementation performed as part of this thesis. Firstly in section 4.1, the signal-flow graph hardware representation is investigated. We cover the graph description interface, implementation design decisions, its execution and higher-level abstraction in detail. This forms the foundation onto which the two front ends PyNN and PyTorch are built.

Consequently, continuing in section 4.2, the implementation of integrating the BrainScaleS-2 hardware into PyTorch is described. This is the main front end of the hardware targeting machine learning. Following, implementation of BrainScaleS-2 as back end for PyNN is described in section 4.3. Here, we make use of higher-level abstract neural network description from section 4.1.4.

We conclude the chapter by shortly describing implementation of the hardware mock connection in section 4.4.

4.1 Graph-based Experiment Notation and Execution — grenade

This section describes the implementation of the signal-flow graph-based experiment notation and its execution for the BrainScaleS-2 neuromorphic hardware. It is called grenade as acronym. The concepts described in section 3.2 are used to provide a signal-flow graph abstraction of hardware usage and virtualized digital operations on one or multiple execution instances in section 4.1.1. In section 4.1.2 we describe implementation of two execution models for the graph representation, one oriented on close-to-host-computer training and development and the other for standalone deployment.

For implementation of the PyTorch and the PyNN front ends, higher-level abstraction for creation of the signal-flow graph representation is feasible. In section 4.1.3 we describe a subgraph-generator interface, which allows a coarser-than-vertex formulation of parts of a graph. It is used extensively in the back end of the PyTorch front end for non-spiking operations. In section 4.1.4 we describe a builder-pattern interface, which closely mimics the network description from PyNN via populations and inter-population projections, which is specifically suited for spiking networks.

4.1.1 Hardware graph description

The logical hardware graph content is described in section 3.2. This section describes its implementation, design decisions and their implications.

Firstly, the interface will be investigated, afterwards, notable implementation decisions under-the-hood are highlighted.

Graph construction

The hardware graph structure of a typical experiment will potentially consist of many vertices and edges. It is therefore convenient to be able to gradually construct it. Other graph-based frameworks solve this by providing means to mutate a given graph instance, for example the MutableGraph concept in boost.graph supports adding or removing elements [7] or the Sequential model in Tensorflow allows gradual construction via adding single layers [2]. We implement the same and provide means to add a new vertex to an existing graph instance.

In section 3.2.3 we describe validity checks, which can be performed on the signal-flow representation. They ensure a functionally correct experiment. For such checks to be of most use, early error detection is important. The earliest possibility to detect errors in a graph representation is upon insertion of a vertex or edge. In particular, we want to detect unconnected inputs to a vertex, missing vertices, signal-shape mismatch between vertices and acyclicity of subgraphs containing digital operations. In order to implement these checks, we restrict the mutability of the graph to compound addition of a vertex with full specification of its in-neighborhood. Figure 4.1 shows such an addition into an existing graph instance. Because all the inputs have to be present when a new vertex is inserted, this scheme can be equivalently identified to be static single assignment [3], when we identify each vertex with its output being a calculation from all its inputs.

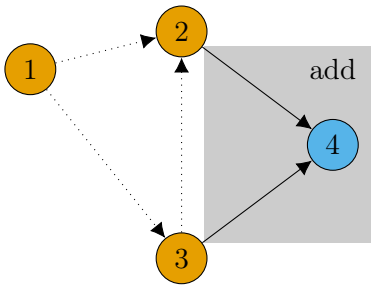


Figure 4.1: Mutable insertion operation of a vertex (4) into an existing graph (1, 2, 3). Alongside the vertex, its in-neighborhood is specified. This enforces its source vertices to be already present in the graph. Therefore, it allows checking the in-neighborhood upon insertion of a vertex and inhibits missing incoming vertices as well as enables shape checking and expected input vertex types.

Vertices

In section 3.2.3 we identify vertices in the signal-flow graph representation as being either hardware entities, which perform some kind of operation or virtualized digital operations, both which given input values produce output values. Table 3.2 shows, that each member of

the heterogeneous set of hardware entities has distinct properties in its function as well as the type of signals it operates on. The validity checks described in section 3.2.3 operate on the vertices' types and their expected neighborhood. Therefore, we choose to represent each vertex as a unique type. The collection of all vertex types is then realized via a type-safe union (in our case in form of `std::variant` as realization), see listing 3.

```
typedef variant<Vertices...> Vertex;
```

Listing 3: Arbitrary vertex represented by type-safe union over all possible vertices.

The expected neighborhood of all hardware entity vertices can be described as featuring an in-neighborhood of possibly different sources and producing one type of output signal. In order to be able to implement checks against the expected in-neighborhood, we provide each vertex instance with a set of expected in-neighbor signal types and one output signal type. Upon insertion of a new vertex into the graph, this allows checking the set of in-signals against the set of out-signals from the specified source vertices. We call each such allowed signal connection Port. Figure 4.2 visualizes this interface. This scheme can be identified

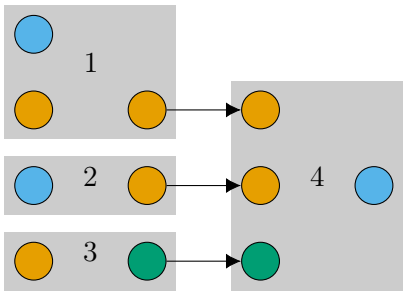


Figure 4.2: Vertex neighborhood matching via ports. The vertices (1, 2, 3) serve as sources for the vertex 4. Each port is depicted as colored circle, where equal color means equal port properties. Comparison of the set of source out-ports (left of arrows) to the set of target in-ports (right of arrows) allows checking for match, which is depicted by matching colors.

with being equivalent to the signature of a free function, where the return type is the equivalent to the out-port and the function argument types is the equivalent of the in-ports.

Allowing $I \geq 0$ in-ports and $O = 1$ out-ports is sufficient for arbitrary entities. If we have an entity, which generates more than one output, it can be represented either via a dedicated port value or by a collection of vertices, which all have the same in-neighborhood. In particular, the one-to-one relation between vertex and out-port allows identification of a source's out-port via the source descriptor.

For the purpose of matching signal shape and type, it suffices for the Port to carry these information, listing 4 shows its interface. We use a one-dimensional shape in the current implementation, because it suffices for all signals between hardware entities, and a type identification via a type-safe enumeration.

For some entities, the number of sources is fixed. For example a synapse driver has exactly one source PADI-bus. Other entities have a variable number of sources. For example a PADI-bus takes input from possibly multiple crossbar nodes (maximally four). In order to

```
enum class ConnectionType;
typedef size_t Shape;

struct Port {
  Shape shape;
  ConnectionType type;
};
```

Listing 4: Port structure for identification of a signal. The shape is one-dimensional, which suffices for all signals between hardware entities.

express both cases easily, we allow the input port of a vertex to be either of fixed number or to be of variadic number (including zero) in the last port:

$$(A, B, C_{\text{variadic}}) = \left\{(A, B) + (C)^N : N \in \mathbb{N}\right\}, \tag{4.1}$$

where for a vertex with input ports A, B, C and variadic last port, the sources matching A, B as well as A, B, C or $A, B, C, C, ...$ are valid.

The interface described so far fully supports the requested port matching. But this does not suffice for placed hardware entities. For example a placed set of synapses might yield an output of size N being connectable to a set of neurons of same amount N. The actual placement on the chip however has to match as well, in this case their residing hemisphere as well as their columns have to match, because e.g. a left-most synapse can't be connected to a right-most neuron physically. In order to allow for such checks, an optional vertex method will be used as shown in listing 5.

```
bool A::supports_input_from(B) const;
```

Listing 5: Function to check a matching source vertex based on its configuration, e.g. placement. Given a vertex `A` only supports input from vertex `B` with certain configuration, then this function will provide the decision whether this is the case.

In section 3.2.3 we describe that multiple realtime executions shall be representable in a single graph structure. The signal-flow between most hardware entities however only works within a single realtime execution. For example the signals from synapses can't be generated and recorded in one realtime execution and fed to neurons in another. Therefore, we have to restrict connections between different realtime executions in the graph based on the entities to be connected. This annotation is implemented as compile-time constant for each vertex, see listing 6.

This concludes the interface of a vertex, which is shown fully in listing 7. It is verified for all vertices using compile-time assertions (as replacement for C++-20 concepts).

```
constexpr static bool can_connect_different_execution_instances;
```

Listing 6: Compile-time static boolean value whether this vertex type can take part in connections between different execution instances, i.e. realtime executions. For a connection between different execution instances to be possible, both vertices have to feature this possibility. This value being positive is basically restricted to vertices describing movement of digital data.

```
struct SomeVertex {
        // whether this vertex can take part in connections between
        // different realtime executions
        constexpr static bool can_connect_different_execution_instances;

        // input ports with specification whether last port is variadic
        // the array size might be dynamic
        constexpr static bool variadic_input;
        array<Port> inputs() const;

        // single output port
        Port output() const;

        // check for matching source taking configuration into account
        // optional, maybe also for different source types
        bool supports_input_from(SomeOtherVertex) const;
};
```

Listing 7: Interface of a vertex. The input ports and output port are available via methods in order to allow dynamic generation, e.g. depending on the vertex configuration. Optionally, one or multiple methods to check matching source additionally with regard to its configuration are possible.

Edges

Edges in the graph can't be added individually, but are added in combination to adding a vertex by connecting the specified sources with this new vertex. As can be seen in fig. 4.2, the port matching between vertices implicitly adds a property to each edge, because both sides of the edge feature the same port as (output or input)-property.

Additionally, using the scheme implemented for the vertices, it becomes apparent, that the possibility to connect only a subset of a source-vertex output port to a target vertex input port is necessary. This becomes clear for example for neuron back end event outputs. Given a number of neurons of different neuron event output blocks, cf. fig. 3.4, this vertex' output port will be of type crossbar input label, but possibly of size larger than one channel. In order to connect one crossbar node (featuring one crossbar channel as input) from this vertex, the port has to be restricted. Figure 4.3 shows this described example. Such a

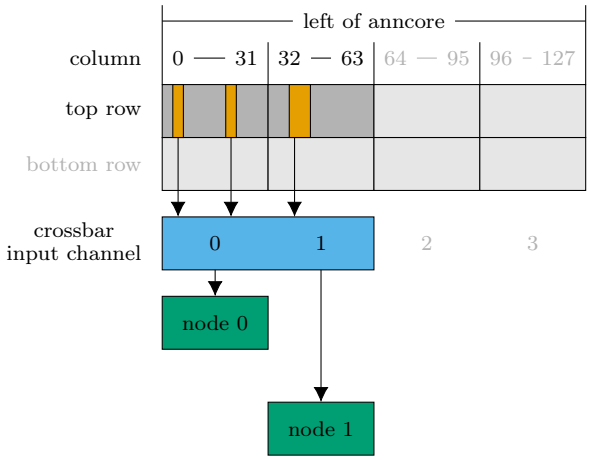


Figure 4.3: Depiction of the necessity for restrictions of ports. A set of neurons (red) is connected to its neuron event output channels (yellow). Each event output channel is then connected to a crossbar node (green). In order to express which channel is connected to which crossbar node, a restriction of the shape of the port is necessary. The first node is to be connected to the first channel, the second node to the second channel.

restriction can easily be implemented as edge property. Listing 8 shows the interface of such a port restriction property. Since only the port's shape can be restricted and the signal type stays constant, the restriction describes a subset of the shape. Currently, we only allow contiguous intervals within the port's interval, which is sufficient.

Execution instance

Section 3.2.3 describes the representation of an experiment containing multiple realtime executions in a single graph. Each vertex in the graph belongs to exactly one such realtime execution. Therefore, the execution instance is a vertex property. However, in contrast

```
struct PortRestriction {
        size_t min;
        size_t max;
};
```

Listing 8: Restriction of a port to the interval [min, max].

to other configuration of a vertex, it describes its physical and temporal placement in the dependency graph of realtime executions. For the lower software layer haldls, it has shown beneficial to split the configuration and placement information into there called container and coordinate [50]. We adopt this idea here by separation of the execution instance information, which is treated as the equivalent of a coordinate, and the vertex's configuration. Each execution instance contains information about its physical chip instance and temporal index. Both entities are in principle unordered but representable as natural number, because there will be a countable number of physical chip entities and temporal realtime executions involving each chip. Listing 9 shows the interface of an execution instance value. We use unique types for both the physical and temporal index to inhibit mix-up from the coordinates halco [18].

```
strong_typedef size_t PhysicalChip;
strong_typedef size_t TemporalIndex;

struct ExecutionInstance {
        PhysicalChip chip;
        TemporalIndex index;

        bool operator==(ExecutionInstance const& other) const;
        bool operator!=(ExecutionInstance const& other) const;
};
```

Listing 9: Interface of a execution instance value consisting of the pair of physical chip identifier and temporal index. It is only comparable equal/unequal to other values, since ordering is not meaningful. The unique number-like types are denoted by (non-existent) `strong_typedef` for simplicity.

Graph interface

The specification of the vertices and edges and construction of the graph leads to an interface description of the graph object. As described, we allow mutability via adding a vertex once all its sources are already part of the graph by specification of all sources. In addition, as laid out in section 3.2.3, every vertex is to be placed onto an execution instance to be identified with a realtime execution.

To fully describe a source vertex and its connection to a target, its vertex descriptor is needed and an optional port restriction of the vertex's output port can be specified. This information is called Input, its interface is shown in listing 10. Listing 11 shows the complete

```
struct Input {
        Graph::vertex_descriptor descriptor;
        optional<PortRestriction> port_restriction;
};
```

Listing 10: Input vertex description for addition of a new vertex to a graph instance. The vertex descriptor is used to identify the source vertex. The optional port restriction can be used to restrict the vertex's output port.

signature for the addition of a vertex to a graph object.

```
struct Graph {
        typedef size_t vertex_descriptor;

        vertex_descriptor add(
          Vertex config,
          ExecutionInstance instance,
          vector<Input> inputs);
};
```

Listing 11: Graph interface for addition of a new vertex. In addition to its configuration, an execution instance is specified for placement in a realtime execution. The list of specified inputs have to refer to existing vertices in the graph. The returned vertex descriptor uniquely identifies this added vertex in the graph.

Restricting addition to the graph to static single assignment results in the inability to directly express recursion. Recursion is however needed inside realtime execution instances for example to describe a recursive neural network. It would require statements of the following:

$$a = f(b),\ b = g(c),\ ...,\ c = h(a), \tag{4.2}$$

where the last equation leads to reassignment of the variable a. In order to resolve this, two solutions are proposed. First, static single assignment could be relaxed to allowing reassignment of existing vertices in the graph. This would not diminish the ability to perform the validity checks on the inputs, because still all inputs would need to be present prior to assignment. Second, static single assignment could be kept and new vertices could be allowed to be added by reference to an existing vertex. As such, its configuration would stay constant and be reused and only the in-neighborhood of the new vertex would change.

Logically both vertices are identified with the same (hardware) entity. The second approach is chosen, because this graph representation is directly convertible to the representation, where reassignment is allowed, however it additionally contains the history of construction (i.e. first assignment had these inputs, second assignment had these other inputs). When displaying such a representation for a recursive loop, a three-dimensional description is suitable, where the vertex references are in the third plane. Conversion to a real graph with circles then is equivalent to removing this third dimension. Figure 4.4 shows this exemplarily. This completes the mutable interface of the graph. Listing 12 shows the complete signature

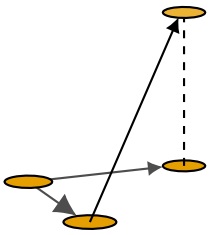


Figure 4.4: Recursion via reference in a graph. The resulting graph with edges depicted as arrows can be built with static single assignment. The reference depicted as dashed line logically connects two vertices on top of each other. When projecting to the two-dimensional plane, a cyclic graph results.

for the addition of a vertex via reference of an existing vertex to a graph object.

```
struct Graph {
        vertex_descriptor add(
          vertex_descriptor reference,
          ExecutionInstance instance,
          vector<Input> inputs);
};
```

Listing 12: Graph interface for addition of a vertex via reference to an existing vertex. The existing vertex is referenced by its descriptor. Compared to listing 11, the only difference is that instead of a new configuration to be specified, we specify the existing vertex.

The immutable interface of the graph consists of accessors to parts of the graph content. The properties of a vertex as well as an edge are queryable as well as the underlying graph structure, which will be explained in the following.

Storage implementation

The graph object stores the vertices, edges and their properties. As underlying implementation for the graph connectivity, we use `adjacency_list` from boost.graph [7]. It is feature-rich and especially offers a vast amount of utility functions for iteration over (parts of) a graph. Instead of only one underlying graph, two are used. One contains all vertices and edges as added to the graph object, the other contains execution instances as vertices and their dependency as edges. Figure 4.5 visualizes this concept. In combination with storing the vertices in the full graph which belong to a certain vertex in the execution

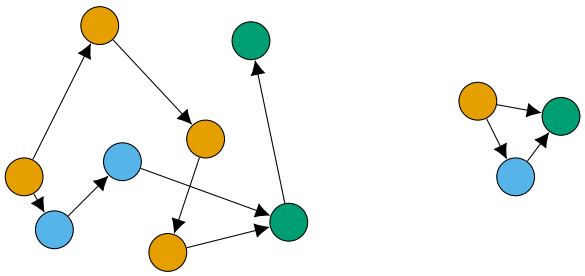


Figure 4.5: Storage of the graph information in two graphs. Left: Full graph as constructed, the colored vertices represent their execution instance. Right: Graph only describing the execution instance dependencies. It allows for easy check of acyclicity of the right graph. In addition, storing the edges between left and right graph (where the execution instance is the same) allows for direct iteration over all vertices of the same execution instance, which will be useful for compilation.

instance graph, this allows direct iteration of a subgraph of one execution instance on the one hand and checking acyclicity of only the execution instance dependency graph on the other.

Visualization

As described in section 3.2.3, visualization of the signal-flow graph description eases understanding and debugging. The chosen graph implementation from boost.graph directly allows generation of a graphviz-readable description of adjacency lists in the dot format [27]. It allows customizable annotation of vertices and edges, which is used to provide names to vertices. In the visualization, we annotate vertex references (i.e. vertices referencing the same hardware entity) by assigning these vertices the same description. The dot format as used here is both machine-readable and human-readable, the latter because it is only a definition of vertices and edges. This allows using the same generation as console-printable format. Figure 4.6 shows a visualized exemplary graph of a recurrent network.

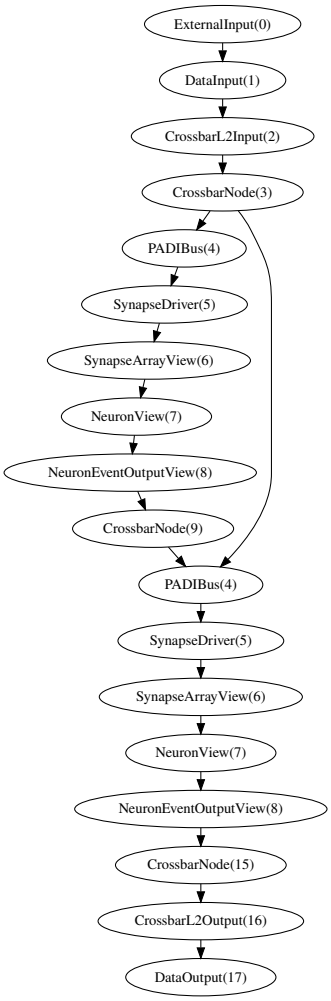
4.1.2 Hardware graph execution

Compilation and eventually execution of an experiment description is a task separated from the actual representation of the description. The compilation and execution process depends on the application and there will be multiple optimal solutions for different environments. In the following, two solutions, which are implemented as part of this thesis will be described in detail.

On the one hand, a just-in-time combined compiler and executor is developed supporting all features. Inherently it leads to close coupling of the executing host computer and the accelerator hardware. Therefore, it is ideally suited for training and development. This

```
digraph G {
0[label="ExternalInput(0)"];
1[label="DataInput(1)"];
2[label="CrossbarL2Input(2)"];
3[label="CrossbarNode(3)"];
4[label="PADIBus(4)"];
5[label="SynapseDriver(5)"];
6[label="SynapseArrayView(6)"];
7[label="NeuronView(7)"];
8[label="NeuronEventOutputView(8)"];
9[label="CrossbarNode(9)"];
10[label="PADIBus(4)"];
11[label="SynapseDriver(5)"];
12[label="SynapseArrayView(6)"];
13[label="NeuronView(7)"];
14[label="NeuronEventOutputView(8)"];
15[label="CrossbarNode(15)"];
16[label="CrossbarL2Output(16)"];
17[label="DataOutput(17)"];
0->1 ;
1->2 ;
2->3 ;
3->4 ;
4->5 ;
5->6 ;
6->7 ;
7->8 ;
8->9 ;
3->10 ;
9->10 ;
10->11 ;
11->12 ;
12->13 ;
13->14 ;
14->15 ;
15->16 ;
16->17 ;
}
```

(a) Graph visualization in dot export format. Each vertex is annotated with a label, the edges are defined below the vertices.



(b) Rendering of the same graph with graphviz. Referenced vertices are annotated via the same label. The recurrence is visible when identifying the vertices of same label with each other.

Figure 4.6: Left: dot-format representation of recurrent network; Right: Same network visualized with graphviz using the exported data from the left.

execution model is already introduced by the author in [66] and will be described here in more detail.

On the other hand, a compiler for standalone execution of the hardware is developed. While its supported feature set will currently be limited due to missing support for inter-chip data transport, its application is deployment, e.g. for embedded usage.

Just-in-time execution

The signal-flow graph experiment representation described in section 3.2.3 can be divided into two granularities for compilation and execution. The coarse granularity of an experiment are the execution instances, which each represent a realtime execution. The inside of such a realtime execution is described via the subgraph corresponding to all vertices with the same execution instance property. Currently, all input data is an immediate, its value is part of an instruction's payload in the realtime program transferred to the FPGA. Therefore, in order to compile a realtime execution, all external data dependencies need to be resolved beforehand.

By associating each execution instance with a compiler and following execution, this can be implemented by just-in-time compilation and execution. Once all data for a given execution instance are present, it can be compiled and executed leading to more data available for the remaining execution instances. Because we force the graph of execution instances to be acyclic, this scheme is guaranteed to be successful. Figure 4.7 shows the execution model for an exemplary graph on a single physical chip.

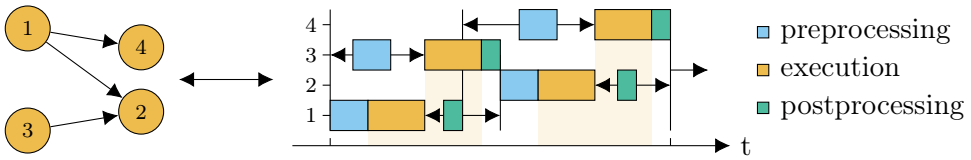


Figure 4.7: Just-in-time compilation and execution of a graph, taken from [66]. Left: Execution instance graph containing four realtime executions, which are all to be performed on the same physical chip instance; Right: Compilation and execution of the graph. For each execution instance (separated on the y-axis), preprocessing incorporates the compilation process, execution follows and postprocessing describes parsing the response data such that they are available for further execution instances. The realtime executions (depicted in red) are serialized, since they are performed on the same physical chip instance. However, the compilation and postprocessing of response data can be performed concurrently.

It becomes clear, that the execution instance graph can directly be executed just-in-time when treating it as a dependency graph, where each vertex is executed once its in-neighborhood was executed. Compilation and result processing for each execution instance can be performed concurrently for multiple execution instances. The execution is serialized

for each physical chip instance. Figure 4.8 shows the execution model for multiple physical chips, where additionally different physical chips can be used concurrently.

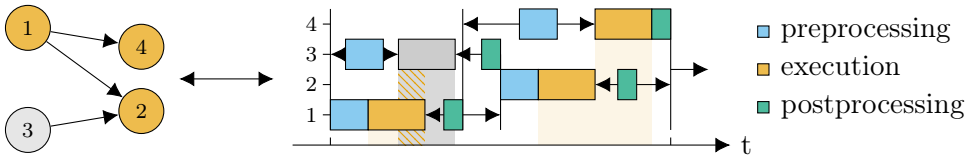


Figure 4.8: Just-in-time compilation and execution of a graph on multiple (two) physical chips, adapted from [66]. Left: Execution instance 3 is to be executed on another physical chip than the other execution instances. Right: The execution of instance 3, depicted in gray, can be performed concurrently, here with execution of instance 1.

For implementation of the dependency graph execution, `tbb::flow_graph` from the intel-tbb library is used [73]. It allows direct conversion from the execution instance graph and transparently handles possible concurrent execution and dependency tracking, i.e. it is possible to configure, that a vertex is only executed once all its inputs are present. For exchange of the result data between execution instances, a centralized object is used. This is efficient, since the majority of accesses are immutable and mutable access time is small (typically a set of moves). The `flow_graph` would also allow transport of data along its edges. However, this would involve potentially many data copies, when data from an execution instance is used in more than one other instance.

Each vertex in the dependency graph contains a compiler and execution procedure for the local execution instance subgraph. An execution instance represents exactly one realtime execution as depicted in fig. 3.14 containing initial configuration followed by a batch of time evolutions. Building the initial configuration is implemented via visiting all vertices of the execution instance subgraph and applying potential hardware entity configuration. Similarly, the time evolution of the realtime execution is extracted via registering visited hardware entities. All surrounding digital operations are executed on the host computer (which performs the compilation). Digital operations needed before the realtime execution are performed early, while digital operations, which require the realtime execution's results are delayed until after execution on the BrainScaleS-2 hardware. Figure 4.9 shows the compilation and execution process for a single execution instance subgraph.

The interface of the whole just-in-time executor therefore requires a graph, a set of input data and connection handles to the hardware and results in a set of output data. The set of input or output data, called `IODataMap` , is implemented as map-like structure, where vertex descriptors are used as key to lookup data corresponding to the specified vertex. Listing 13 shows the executor's interface.

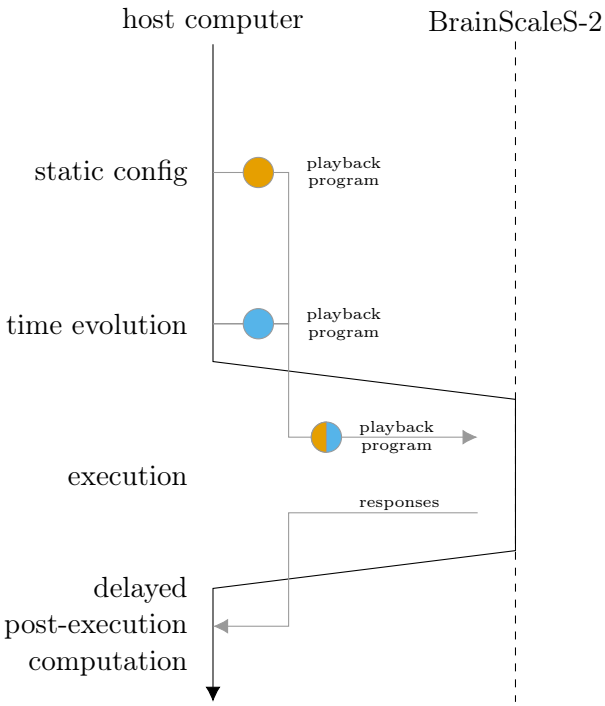


Figure 4.9: Compilation and execution of a single execution instance subgraph in the just-in-time executor. First, the static configuration is extracted by visiting all vertices and applying hardware configuration where applicable. Then, the realtime execution is built by again visiting all vertices. This built program is executed on the BrainScaleS-2 hardware and results are transported back to the host computer. Finally, delayed digital operations are processed, which require the results from the execution. They are performed on the controlling host computer.

```
struct JITGraphExecutor {
        // hardware connection handles
        typedef map<PhysicalChip, Connection&> Connections;

        static IODataMap run(
          Graph const& graph,
          IODataMap const& inputs,
          Connections const& connections);
};
```

Listing 13: Interface of the just-in-time executor for hardware graphs. The executor requires a graph to execute, a set of inputs to use and a set of connection handles to hardware. The result contains data for each vertex which is some kind of data output.

Compilation for standalone execution

In contrast to the just-in-time executor described above, for standalone experiment execution the interaction with a host-computer is to be minimized or even to be avoided altogether. Here, compilation and execution are to be separated for the whole graph and not to be (concurrently) interleaved like in the just-in-time executor. We define standalone as transferring a program to the hardware once and expecting results only after completion in the following. In principle this is not standalone in the sense that no host computer is involved, but that no host computer is involved during the experiment described by the signal-flow graph, which implies possibility for true standalone usage.

We chose the same granularity of execution instance subgraphs and their dependencies for compilation. However, the transfer of data between these compilations has to be implemented standalone on the hardware opposed to reading back results to a host computer and feeding them into the depending on execution instances' compilers.

This statement causes a vast limitation for the compilation model with the current state of the hardware. There is currently no way to route event data (spike and ADC) generated from the chip to a storage, which is accessible from within the hardware, but they are streamed-out directly to the controlling host computer. Similarly, arbitrary spike-train injection is not possible with comparable performance from within the hardware as is via playback sequences generated on a host computer (but event generation for the non-spiking mode of operation is possible via dedicated circuits currently residing on the FPGA). This effectively implicates, that a compiler for standalone execution can currently only be used for exclusively non-spiking experiments. Additionally, multiple physical chips currently don't feature means to communicate other than via a host computer and therefore compilation is limited to graphs residing on a single physical chip (but still with possibly multiple realtime executions).

On the upside, the improving the data-flow locality by restricting it to be near the hardware instead of transporting all data between hardware and host computer lets expect improvements in performance of the execution. Moreover, the separation of compilation and execution removes compilation time from the execution time in the limit of many executions, where in contrast the just-in-time executor will repeat the whole compilation for every execution.

First, compilation of a single execution instance is investigated and afterwards, compilation of graphs containing more than one execution instance is described.

Compilation of a single execution instance can be split into four aspects. The hardware is to be initialized for the realtime execution, expected input data is to be loaded to the hardware, the realtime is to be performed and result data is to be read out for evaluation. The first and second aspect are independent and therefore interchangeable and in principle aspects are optional if they don't occur. Figure 4.10 visualizes this result of compilation. We

Figure 4.10: Compilation result of compilation for standalone execution of a single execution instance. The result is split into four parts. The initialization of the hardware and transfer of input data are interchangeable and precede the realtime execution orchestrated by the PPUs. Last, possible result data is stored after the execution.

use the two PPUs as experiment master for the standalone execution model. They initiate the time evolution during the realtime execution and perform all internal data transfer and digital operations. Comparing the just-in-time executor, the digital operations are implemented on the host computer, which can be seen there as the experiment master. All four parts of the compilation result are generators of a playback sequence to be executed on the FPGA. Especially the generator for loading input data can only be transformed to a playback sequence with knowledge of the actual data, because data is an immediate in playback instructions, cf. section 3.1.3. This inhibits *complete* separation of compilation and execution, if this transformation is seen as part of the compilation process. For transfer of data for the load and store execution section, static memory allocation is used within either the DRAM on the FPGA or the PPUs' internal memory. This then allows access within the realtime execution from the PPUs. All digital operation and the execution of the realtime time evolution are implemented on the PPUs. Parameterization of a precompiled PPU binary including all such operations is used. This is easier than code generation and on-demand compilation of the PPU programs and is therefore favored as initial approach. Parameterization is done via a sequence of commands describing single operations, which are processed during execution by the PPUs. They consist of a description of location of input data, target location for operation results and additional parameters needed for the operation. Listing 14 shows the interface of such a command. Upon execution on the PPU, the correct operation is chosen via runtime dispatch. Therefore, the provided commands are interpreted by the PPUs rather than used for compile-time code generation. Figure 4.11 shows the complete compilation process and resulting execution in detail for an exemplary network.

Until now, we described compilation of a single execution instance. A graph containing multiple execution instances can be compiled by compiling execution instances iteratively, since they can be ordered topologically because of acyclicity. The same separation of compilation and execution is thereby directly given by the single execution instance compiler. However, sequential compilation leads to interleaving of loads and stores in-between realtime executions. Therefore, we reorder memory allocation of these parts of the execution instance compilations such that they surround all realtime executions. Figure 4.12 shows this

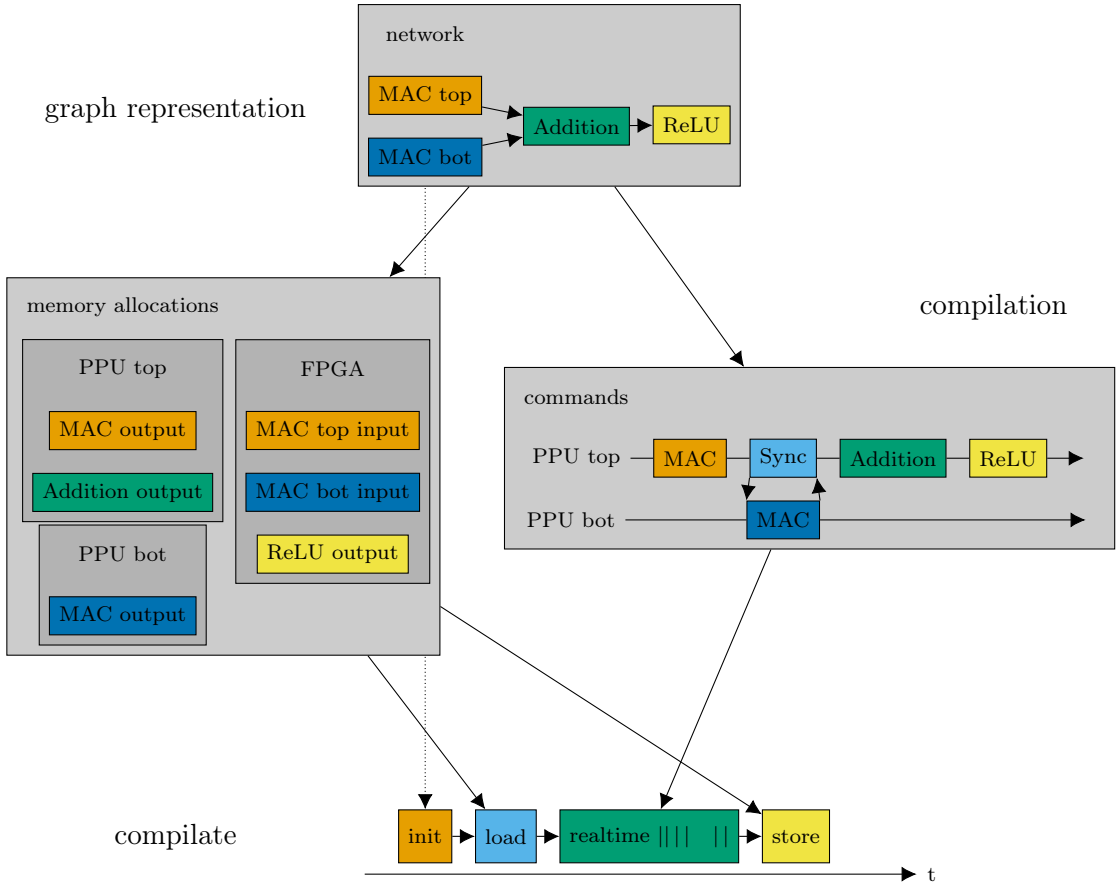


Figure 4.11: Compilation process for an exemplary network on a single execution instance consisting of a matrix multiplication on each hemisphere, followed by an element-wise addition and rectified linear unit operation (at the top). During the compilation process (in the middle), necessary memory allocations are statically placed (on the left) and commands for the PPUs are generated (on the right). While large allocations for input and output data is allocated on the DRAM on the FPGA, temporary data in-between operations is placed at the PPUs' SRAM for improved locality. The memory allocations are then used to generate the load and store part of the compilate. The commands are put into the realtime part of the compilate. Initialization of the chip is omitted for simplicity.

```
typedef void* GlobalAddress;

struct SomeCommand
{
        maybe_array<GlobalAddress> input;
        GlobalAddress output;

        AdditionalParameterForSomeCommand params;
};
```

Listing 14: Interface of a command for interpretation on the PPU. It features (possibly multiple) accessors for input data and an accessor for output data. In addition, parameters specifically needed for the command at hand can be provided. Therefore, such a command describes arguments and return value location of a function invocation.

reordering process and the resulting compilate for a graph containing multiple execution instances. It shows, that such reordered compilation results again in a standalone executable by above definition. This completes the description of the compiler for standalone execution.

4.1.3 Composeability and reuseability with subgraph-insertables

Using the signal-flow hardware graph description directly for large-scale experiments can lead to a large amount of repetition of (almost) the same structures or subgraphs. For example a non-spiking experiment might involve a multitude of matrix multiplications, which are connected in some manner. Therefore, a higher-level abstraction is needed in order to reduce this repetition and allow easy construction and use of larger-than-vertex graph elements.

One solution for this is to provide an interface for subgraph generators, which can be inserted into an already existing graph instance. These generators are called `Insertable` in the following due to their ability to be *inserted* into an existing graph.

The interface of a subgraph to the rest of a graph can be described by a set of in-edges and a set of (possible) out-edges.

In contrast to single vertices, a user of such an interface does not necessarily have control over the complete placement of all subgraph-vertices. Moreover, with increasing insertable complexity complete control over placement is not feasible anymore. Therefore, we provide the insertable generator with a resource manager, which is an entity queryable for placement information. Currently, a very simple allocator-based resource manager is used, which only allows request of a next free chip hemisphere. In the future, it may be as complex as allowing requesting for example single neurons. This completes the `Insertable` interface, which is shown in listing 15.

Insertables are used extensively as interface for the back end in the implementation of the PyTorch front end.

```
struct InsertableIO {
        vector<Input> inputs;
        vector<Input> outputs;
};

struct Insertable {
        template <typename... Args>
        Insertable(Args...);

        InsertableIO operator()(
          Graph& graph,
          vector<Input> const& inputs,
          ResourceManager& resource_manager) const;
};
```

Listing 15: Interface of insertable subgraph-generators. Insertion into an existing graph instance requires a set of inputs to use and a mutable resource manager from which to request resources needed for insertion, e.g. a free execution instance. Result of an insertion (aside mutation of the graph) is a description of the (input and) output accessors of the subgraph. In contrast to insertion of a vertex, insertables insert themselves into a graph, which allows for extension without alteration of the core code-base. It is necessary for the result to contain input accessors in addition to the output accessors, because an insertable might add additional input vertices (without in-neighborhood) for which a user will have to provide input data upon execution.

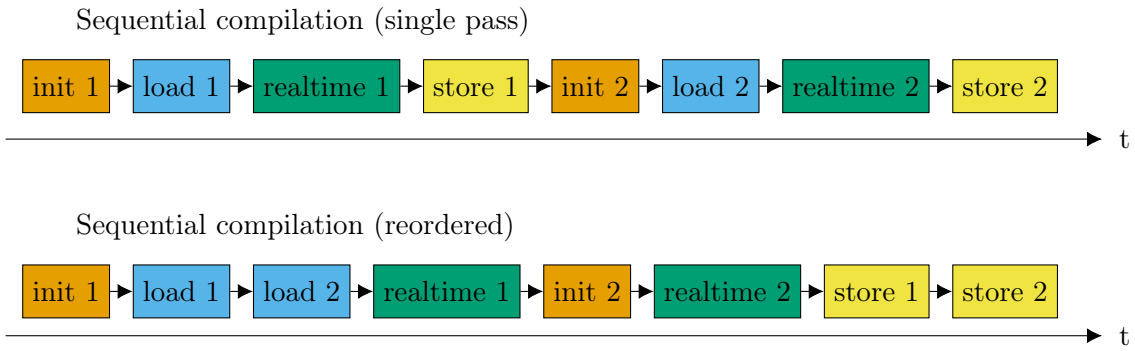


Figure 4.12: Compilate of a graph containing multiple (two) execution instances. Top: Sequential compilation of the execution instances. While this leads to a functional experiment, host-computer interaction is not separated from realtime execution, because the realtime executions are interleaved by load and store operations. This results in the compilate being not suited for standalone execution. Bottom: Reordering parts of the execution instances' compilation process results in complete separation of load/store and realtime execution. This is achieved by first compiling allocation of loads, then compiling the realtime executions and finally allocating store memory. Therefore, this reordered compilation for multiple execution instances enables execution in a standalone way for arbitrary (exclusively non-spiking and using only one physical chip) graphs.

4.1.4 Building networks from populations and projections

In contrast to the non-spiking mode of operation, the spiking mode of operation on BrainScaleS-2 typically requires non-trivial and highly heterogeneous configuration of the event routing facilities described in section 3.1.2. This implies, that for a higher-level abstraction of spiking networks, some kind of automation process for finding the correct configuration and usage of the hardware entities participating in routing of events for an abstract network description is necessary. This statement contains two parts, which require development, an abstract network description and the automation process for finding a hardware configuration for a given abstract model. We start with describing the abstract network description and continue with the automated process of finding a hardware representation.

Abstract network description

The abstract description of spiking neural networks is defined to lack information about the actual representation on the hardware. This specifically involves placement information and the routing configuration, which as described shall be provided by a separate (automated) step. We strive towards ability to easily interface it to user-facing frameworks like PyNN and therefore let this description be inspired by such frameworks. Computational neuroscience frameworks like PyNN [15], Brian [67] or Nest [31], but also machine learning libraries for

spiking neural networks like BindsNET [35] provide means for abstract network description via collections of neurons and collections of connections between them. This facilitates a concise network description even for large-scale networks, while also allowing for small networks. We therefore adapt this widely used general interface idea. In the following, nomenclature of PyNN is used like introduced in section 3.3.1, i.e. we call a collection of neurons population and a collection of connections between neurons projection.

A collection of populations and projections forms a graph, where populations are vertices and projections edges between them. Similarly to the signal-flow hardware graph description in section 4.1.1, we chose a mutable graph interface, now however allowing insertion of both vertices and edges individually. Listing 16 shows its interface.

```
strong_typedef size_t PopulationDescriptor;
strong_typedef size_t ProjectionDescriptor;

struct Network {
        template <typename Population>
        PopulationDescriptor add(Population&& population);

        template <typename Projection>
        ProjectionDescriptor add(
          Projection&& projection,
          PopulationDescriptor source,
          PopulationDescriptor target);
};
```

Listing 16: Interface of the abstract network graph object. Populations and projections can be added to this network resulting in a number-like unique identifier. In the case of adding projections, these are used to identify source and target population. The unique number-like types are denoted by `strong_typedef` for simplicity.

On BrainScaleS-2, three general types of populations are available. First there are populations consisting of on-chip neuron circuits, which can serve both as source and as target of events. Then external spike-trains can be used to provide source-only populations with arbitrary time evolution. Last, the on-chip background generators allow for configurable source-only populations with regular or Poisson event generation. In principle these restricted spike-trains can also be provided externally at the expense of increased traffic, possibly congestion and a decreased upper-rate-limit, which is the reason for background-generator populations being not implemented yet. Listing 17 shows the interface of the external and internal populations. Opposed to PyNN, where external spike source populations contain the actual spike-trains for their neurons, we chose to separate the description of data-flow from the actual data as is done for the signal-flow graph representation of the hardware.

A projection contains a collection of single-neuron connections between two populations.

```
struct ExternalPopulation {
        size_t size;
};

struct InternalPopulation {
        struct NeuronProperties;
        vector<NeuronProperties> neurons;
};
```

Listing 17: Interface of populations in the abstract spiking neural network description. Both external spike-source populations and internal populations provide access to their size (the `InternalPopulation` via its number of neuron properties), i.e. number of neurons, and possibly additional properties. A neuron within a population is identified by its (zero-based) index.

On BrainScaleS-2, connections can only be implemented between on-chip neurons or from off-chip spike sources or on-chip background sources to on-chip neurons. This is ensured by the network graph object upon insertion of a projection into the graph. In principle, a multitude of synaptic connection types can be thought of and implemented, e.g. unsigned static connections (excitatory or inhibitory), signed connections (excitatory or inhibitory within the same projection), connections with short-term plasticity, connections with spike-time dependent plasticity or other. However, each connection contains information about which neuron to connect to which neuron. Listing 18 shows the generic interface of a connection. A projection is then simply a collection of connections with homogeneous

```
template <typename Synapse>
struct Connection {
        size_t index_source;
        size_t index_target;
        Synapse synapse;
};
```

Listing 18: Interface of a generic connection between two single neurons. The neurons are identified within the projection's source or target population by zero-based index. Additionally, the connection stores information about the synapse type and parameterization.

synapse type. Listing 19 shows its interface. In addition to this interface featuring possibility to represent sparse projections with few connections, it is thought of to provide a dense representation for improved performance in the case of many connections. Then such a dense projection incorporates a matrix of synapses of homogeneous types, as shown in listing 20.

This completes an abstract description of spiking neural networks. In the following, the process of translation of such a description to a hardware emulation on BrainScaleS-2 is

```
template <typename Synapse>
struct Projection {
        vector<Connection<Synapse>> connections;
};
```

Listing 19: Interface of a projection between two populations. It contains a collection of single-neuron connections.

```
template <typename Synapse>
struct Projection {
        struct Offset {
                size_t source;
                size_t target;
        };
        Offset offset;

        matrix<Synapse> synapses;
};
```

Listing 20: Interface of a dense projection between two populations. It contains a dense collection of single-neuron synapses. The shape of the translated synapse matrix has to be within the possible index range of the source and target population respectively. For a given synapse at `synapses[i][j]`, its source index is calculated as `index_source = offset.source + i` and its target index is given by `index_target = offset.target + j`. Thereby it allows all-to-all connectivity between a subset of the populations.

described.

Mapping and routing towards a hardware representation

The process of transforming an abstract network description to a hardware representation involves two parts, which are in-separable, finding hardware entities, which allow the abstract network's topology in principle and then solving their connectivity. The first part is necessary, since for example there exist neuron combinations, which don't allow routing events between them. The latter is needed because of multiple shared resources in-between different parts of typical networks. We call this process mapping and routing, where mapping describes placement and routing describes connectivity. It has already been successfully applied to the wafer-scale predecessor system BrainScaleS-1 [39, 51].

We want this process to be automatable and to allow user-provided algorithms for that task. The result of the mapping and routing process shall be (canonically translatable into) a signal-flow graph hardware description following section 4.1.1. Figure 4.13 visualizes the layering-wise location of the map and route process. The mapping and routing process

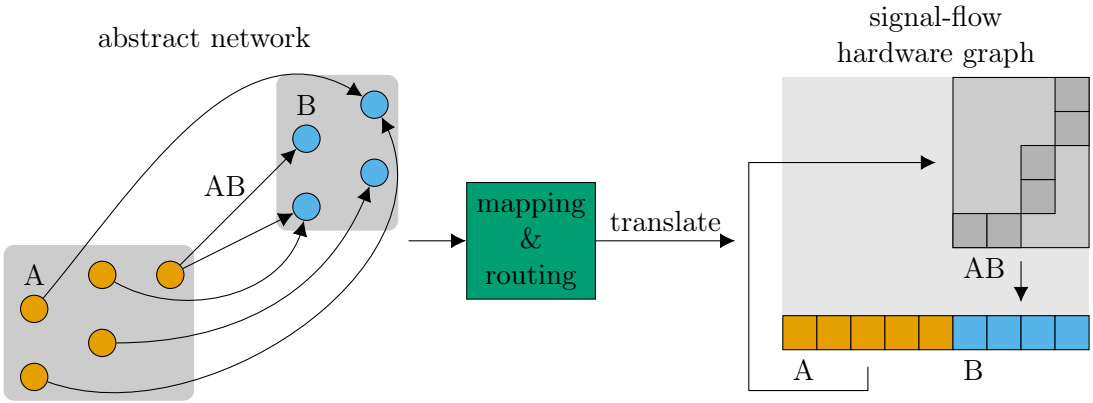


Figure 4.13: Layering of abstract spiking neural network description, mapping and routing and the signal-flow graph hardware representation. Given an abstract network, mapping and routing is inserted as a black-box algorithm resulting in placement, which is canonically translatable into a signal-flow graph hardware representation.

therefore results in annotation of the given abstract network with placement and configuration information for the hardware representation. Given this result, it is then canonically possible to build a signal-flow graph hardware representation. During this, identification of abstract parts of the network with the hardware graph representation is possible, which allow backward annotation of actual representation on the hardware to the abstract network, which is facilitated by both representations being structured in a graph-based way. This is especially helpful during development of models and map and route algorithms. Listing 21

shows the complete interface of the map and route process and its translation to the signal-flow graph representation.

```
struct Placement
{
        struct PopulationPlacement;
        map<PopulationDescriptor, PopulationPlacement> populations;

        struct ProjectionPlaecment;
        map<ProjectionDescriptor, ProjectionPlacement> projections;
};

// A (user-providable) map and route algorithm
Placement map_and_route(Network const& network);

// Canonical construction of signal-flow graph
// from placement and abstract network
Graph build_graph(Placement const& placement, Network const& network);
```

Listing 21: Interface of mapping and routing for abstract network descriptions. The placement contains annotation of hardware entities to the abstract network. It is to be generated by a mapping and routing algorithm given an abstract network. This result can then be canonically transformed to a signal-flow graph hardware representation.

4.2 PyTorch extension — hxtorch

This section describes the implementation of the PyTorch extension hxtorch, which allows using the BrainScaleS-2 neuromorphic hardware as an accelerator within this framework. The concepts described in section 3.3.2 are used to provide operation-level integration of the hardware with PyTorch. The non-spiking part of this section is based on the implementation description provided by the author in [66]. We start by investigating integration of operations, cf. section 4.2.1. Continuing, specific aspects of this integration are described in detail, namely data type conversion, cf. section 4.2.2, partitioning of operations to hardware, cf. section 4.2.3, execution, cf. section 4.2.4, hardware access, cf. section 4.2.5 and handling of hardware parameters, cf. section 4.2.6. Afterwards, we explain a way to record multiple hardware operations for fused optimized execution in section 4.2.7. We complete the section by laying out the foundation for integration of the spiking mode of operation into the PyTorch extension in section 4.2.8

4.2.1 Operation

As described in section 3.3.2, the lowest-level user-facing interface in a PyTorch extension is the operation. In the following, we describe the process of interfacing the BrainScaleS-2

hardware in the non-spiking mode as the back end of an operation. The matrix multiplication operation `matmul` is used as example throughout this section, because it is the most basic operation, which involves the analog neural network core, but still can be used to describe all aspects of the integration. In order to make use of the automatic gradient calculations, the `torch::autograd::Function` interface is to be fulfilled. It requires provision of a free function callable in the forward direction and a free function callable in the backward pass of the operation.

Forward pass

The forward function contains a linear sequence of data-transformations to and from the hardware. Figure 4.14 shows its internal implementation. The sequence can be split into

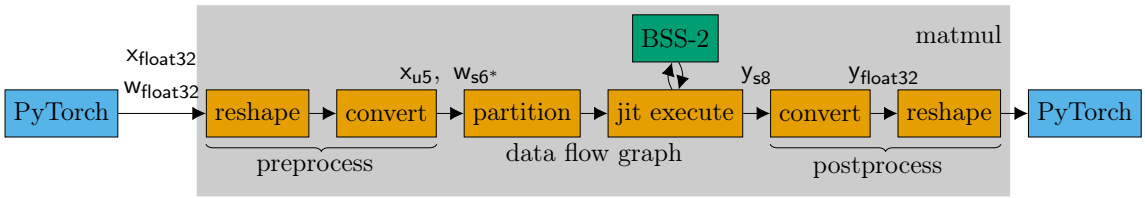


Figure 4.14: Internal sequential implementation of the `matmul` operation's forward pass with hardware back end, taken from [66]. The function is provided with `torch::Tensor` data for the inputs and the weights. They are reshaped to match the dimensionality of the matrix multiplication on the hardware (one-dimensional inputs and two-dimensional weights) and type-converted into hardware units, which are 5 bit unsigned inputs and 6 bit weights with additional sign bit. Following, eager partitioning of the operation onto available hardware and just-in-time execution, cf. section 4.1, is employed. The result data from the hardware execution, being signed 8 bit integer values, are again converted to `torch::Tensor` data structures.

three main parts, shape and type conversion between PyTorch data and back-end data format, cf. section 4.2.2, partitioning onto available hardware, cf. section 4.2.3 and the actual execution, cf. section 4.2.4.

Compared to the digital equivalent operation, the multiply-accumulate operation on BrainScaleS-2 is adjustable especially via parameters altering the gain of the operation. In section 4.2.6, they are described in more detail including their supplication to the operation. For the backward-pass implementation, access to the supplied information is needed to correctly model the operation, cf. section 3.3.2. Listing 22 shows the signature and implementation of saving these parameters in the forward-function of an operation.

```
torch::autograd::variable_list forward(
  torch::autograd::AutogradContext* ctx,
  torch::autograd::Variable var, ...)
{
        ctx->save_for_backward(var, ...);
        return forward_impl(var, ...);
}
```

Listing 22: Interface of the forward function of an operation using the `autograd` interface of PyTorch. It is supplied with an `AutogradContext`, which serves as state between the forward and backward function. It allows saving parameters in the forward pass for the backward pass via `save_for_backward`.

Backward pass

The backward pass is used during training for calculation of the back-propagation of gradients using the automated differentiation framework autograd in PyTorch. As described in section 3.3.2, the backward pass is calculated digitally via a software model of the actual analog operation. In the current implementation, a linear model of the multiply-accumulate operation is used. The gain is supplied via a singleton-pattern like the hardware access, cf. section 4.2.4. In addition, the gain-adjusting parameters from the forward pass are used in the calculation. Listing 23 shows the signature and implementation of retrieving these parameters in the backward-function of an operation. The model is implemented entirely

```
torch::autograd::variable_list backward(
  torch::autograd::AutogradContext* ctx,
  torch::autograd::variable_list grad_output)
{
        auto saved_variables = ctx->get_saves_variables();
        return backward_impl(saved_variables, grad_output);
}
```

Listing 23: Interface of the backward function of an operation using the `autograd` interface of PyTorch. It is supplied with a `AutogradContext`, which serves as state between the forward and backward function. It allows retrieving parameters in the backward pass via `get_saved_variables`, which were saved in the forward pass. The gradient of the output `grad_output` and the saved parameters altering the gain of the operation are supplied to the linear model of the `matmul` operation to calculate the gradient of the operation's inputs.

using PyTorch's `C++`-API.

4.2.2 Tensor data conversion for hardware

As already described in fig. 4.14, non-spiking operations on BrainScaleS-2 features a collection of differently quantized data types. On the other hand, PyTorch also supports different tensor element types like `float` or `int8_t`. Between these two collections of data types, only one is present in both, the 8 bit signed integer for the digitized membrane potentials. While the others are also integer-like and could be represented in a PyTorch format, which allows also storage of larger values, this is infeasible, because ensuring correct ranges of the supplied data is important for performing operations on the hardware. Another point to take into consideration is the PyTorch data type's ability to be used with the automatic differentiation framework for training. During the training process it is crucial to allow for sub-quantization-precision value updates for correct representation of many small updates. Therefore, we chose to use `float` as expected and returned tensor element type for all operations. This implies necessity of conversion to hardware values, where we chose to round to the next integer value for conversion to integer types. Additionally, we chose to perform range checks on the resulting integer values opposed to clamping to the allowed range. This removes the possibility of silent modelling mistakes like assuming the hardware's ability to accept signed input values. All these conversions and checks are potentially costly. Therefore, in section 4.2.7, we describe a possibility to get rid of intermediate conversions via tracing of operations.

4.2.3 Partitioning of operation to hardware

The physical shape of a single synapse array on the hardware is fixed to $N = 256$ rows (128 for signed weights) and $M = 256$ columns. This limits the size of a single multiply-accumulate operation. However, operations, that don't fit into these limits can be implemented by splitting them up into smaller operations and reconstructing the complete result by combining the temporary operations' results. This is achievable via either physical distribution onto multiple hardware instances or via temporal distribution onto the same physical hardware instance or a combination thereof.

For representation of an operation, we want to make use of the signal-flow graph hardware representation, cf. section 4.1.1. The insertable interface, cf. section 4.1.3, which describes a generator for a subgraph, is the ideal candidate for representation of the partitioning of an operation. Its interface allows access to a resource manager, which is used here for allocation of hardware executions for distribution of an operation onto multiple execution instances.

In the following, the partitioning of two frequently used operations, the matrix multiplication and the convolution is described in detail.

The implementation of partitioning of a matrix multiplication is shown in fig. 4.15. In order to support more columns, the matrix is split into parts with directly supported width and their results are concatenated. In order to support more rows, the matrix is split again

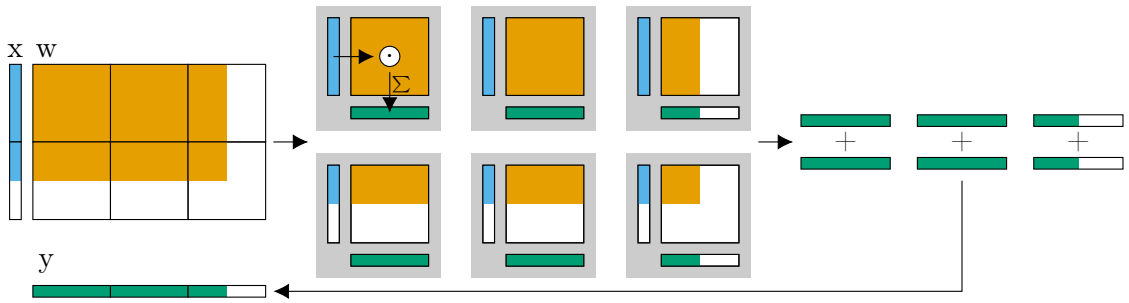


Figure 4.15: Partitioning a matrix multiplication, which is too large to fit on a single synapse array, taken from [66]. Top left: the input x is multiplied with the weight matrix w. Inputs and weights are split at the black boundaries representing the shape of a hardware synapse array. Middle: as the smaller split operations are independent, they are allocated and executed individually. Right: the split operations' results in the row dimension are summed digitally, results in the column dimension are concatenated leading to the result y (bottom left).

into parts with directly supported height, but their results are digitally accumulated:

$$y_j = \sum_i^N x_i w_{ij} = \left(\sum_i^{N_1} x_i w_{ij}\right) + \ldots + \left(\sum_i^{N_R} x_i w_{ij}\right), \quad N = \sum_r^R N_r, \tag{4.3}$$

where the number of rows N is split into R ranges N_r of analog computation $\sum_i^{N_r} x_i w_{ij}$, which are then accumulated digitally. We expect this partitioning to be comparable to execution on a larger synapse array, if boundary effects like analog saturation or digital overflow are negligible. In the limit of large weight matrices in both dimensions, this scheme leads to optimal chip area usage, because the number of only partially used synapse arrays scales with the matrices' edges like $\mathcal{O}(N + M)$, while the number of fully used synapse arrays scales with the area like $\mathcal{O}(N \cdot M)$.

The implementation of the partitioning for a convolution is shown in fig. 4.16 for the example of a two-dimensional convolution. It transforms the convolution into a matrix multiplication for which the partitioning described beforehand is utilized. The kernel is unrolled into the vertical matrix dimension, where then all kernel channels are placed aside each other. The input is then traversed with given stride such that the resulting operation is equivalent to application of the original kernel at a certain position in the input. This scheme is efficient, because it leads to a constant weight matrix, while posing the necessity to resend inputs multiple times for different kernel positions. However, the latter is far less expensive as a matrix reconfiguration. All these transformations are implemented via PyTorch tensor operations, which automatically generates the backward pass for these convolution operations.

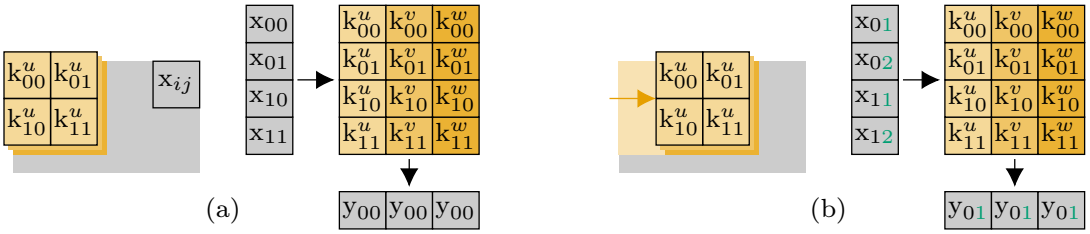


Figure 4.16: Transformation of a 2-d convolution ((a) left, (b) left) of inputs x_{ij} with kernel k_{ij} to a multiplication ((a) right, (b) right), taken from [66]. The kernel has three channels (u, v, w) and is moved from (a) to (b) with a stride of 1. The resulting matrix is constant for all kernel positions, which is efficient in terms of reconfiguration of the weights while leading to overlapping inputs for different kernel positions.

Additionally, a scheme for increased parallelism for one-dimensional convolutions has been developed and implemented by Arne Emmel in his master thesis [22]. There, the kernel is placed multiple times aside each other vertically shifted by the stride. This allows performing multiple operations described above in parallel, increases the amount of inputs to be sent in one multiply-accumulate operation and reduces the amount of inputs which are sent multiple times for different kernel positions.

4.2.4 Execution in hardware

As described in section 4.2.3, the result of partitioning of an operation is an insertable, cf. section 4.1.3 into a signal-flow graph hardware representation, cf. section 4.1.1. In order to complete this graph for execution, the insertable's subgraph is surrounded by a load of input data before and a store of result data after the insertable. The resulting signal-flow graph can then be used for execution. PyTorch expects the results of an operation to be available upon return from the forward function. Therefore, eager execution on the hardware is required. We use the just-in-time executor, described in section 4.1.2, for this task. Since the forward function of an operation is a free function, we need to provide access to the hardware via a side-effect. For this, a singleton-pattern is chosen to manage access to hardware handles, i.e. connection handles from hxcomm, cf. section 3.1.3. Initialization and tear down of this singleton hardware management is described in section 4.2.5.

4.2.5 Hardware initialization and tear down

In section 4.2.4, we describe, that access to hardware handles is given via a singleton-pattern. This is chosen over performing initialization and tear down for every operation, because these two processes are time-consuming. Therefore, initialization of the hardware is only performed once before using it in operations. We allow a user to modify the initialization

process by supplying a custom calibration. Initialization and tear down are performed via free functions, which modify the singleton, listing 24 shows their interface.

```
void init(optional<string> calibration = nullopt);

void release();
```

Listing 24: Interface of the hardware initialization and tear down. The initialization can optionally be provided with a path to a serialized calibration. If no custom calibration is given, the group's hardware database [19] is queried based on the hardware available and a default calibration generated automatically by Jenkins, cf. section 3.5.2, is used. The tear down releases all initialized hardware for use outside the PyTorch extension.

4.2.6 Handling hardware parameters

The analog computation on BrainScaleS-2 features parameters, which alter the dynamic range of operations. The two parameters currently present in the interface are the wait time between successive input events and the number of times to send each input. Their effect is described in detail in [75]. They can both be used to optimize the precision of an operation to the problem at hand. For example a small operation might benefit of sending inputs multiple times to increase the absolute change in membrane potential for better signal-to-noise ratio. Like this, these parameters are meant to be tuned for each operation individually. Therefore, we provide them side-by-side to the other parameters of the operation, which are already present for the original PyTorch operation. Listing 25 shows this at the example of the `matmul` operation.

```
torch::Tensor matmul(
  torch::Tensor tensor1, torch::Tensor tensor2,
  int64_t num_sends = 1, int64_t wait_between_events = 8);
```

Listing 25: Interface of supplication of additional parameters for the hardware operations. They are provided as additional arguments to the operation aside the original PyTorch operation's parameters.

4.2.7 Tracing operations for inference

Providing hardware support at the level of PyTorch operations provides fine granularity for development. However, for a model consisting of a large amount of operations this approach scales badly performance-wise due to increased overall accumulated latency of hardware access from all the operations and the large number of intermediate data transformations, cf.

section 4.2.2. Figure 4.17 visualizes these expenses and the decreased runtime when fusing multiple operations.

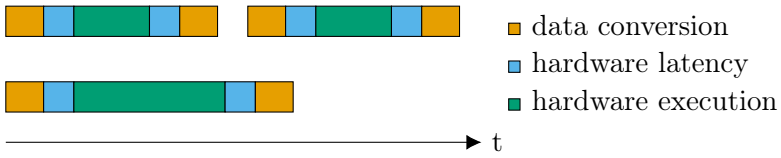


Figure 4.17: Runtime expenses of two operations executed on the hardware when called individually (top) and when fused (bottom). The hardware latency between operations as well as the data transformations between operations are saved when fusing operations.

The eager execution model of PyTorch does not directly allow accessing the compute graph. Therefore, we need the ability to record the compute-graph during an execution of a model, and register all hardware executions. For registering of hardware operations, we chose the granularity of insertables in the signal-flow graph, cf. section 4.1.3. Using this approach, while tracing the forward pass, the backward pass is neglected, which results in an operation, which can be used for inference, but not anymore for training. A singleton-based registering of recorders is used, which hook into the hardware operations. This allows tracing models without alteration, which is especially helpful for complex models. This tracing of operations only works for hardware operations without intermediate classical PyTorch operations. In order to ensure this, for each successive recorded hardware operation, we compare the input values to the newly recorded operation to the output values of the last recorded operation. By enforcing identity between the values, it is ensured, that in-between these two operations, only alterations resulting in identity are made, which can be dropped without change. For example such an identity would be tensor reshape followed by an inverse reshape. Listing 26 shows the implemented concept of tracing hardware operations. The recorded sequence of insertables can be serialized for export. This allows using the tracing for construction of a deployable version of the model for standalone execution. A single graph for compilation is generated by insertion of all insertables one after another. On the other hand, the sequence can be used from within PyTorch via a dedicated operation, which supports execution of arbitrary traced models, see listing 27, which is especially useful for integrated high-performance inference.

4.2.8 Towards spiking operation

PyTorch's classical field of application are artificial neural networks. However, the benefits described in section 3.3.2 are equally applicable for spiking neural networks. Frameworks line BindsNET [35] or Norse [54] allow construction and simulation of spiking neural networks in the ecosystem of PyTorch. In this thesis we focus on prerequisites of successfully integrating BrainScaleS-2 as hardware back end for spiking experiments while using PyTorch. The

```
// Construct and register tracer
Tracer tracer();
// {
//   singleton_tracer = *this;
// }

// Model
torch::Tensor input;
auto out1 = op1(input);
// {
//   out = op1_insertable(input);
//   singleton_tracer.add(op1_insertable);
//   singleton_tracer.last_output = out;
//   return out;
// }
auto out2 = op2(out1);
// {
//   assert(singleton_tracer.last_output == out1);
//   out = op2_insertable(out1);
//   singleton_tracer.add(op2_insertable);
//   singleton_tracer.last_output = out;
//   return out;
// }

// Save serialization of recorded insertable sequence
tracer.save("some/path");
// {
//   file << {op1_insertable, op2_insertable};
// }
```

Listing 26: Interface of tracing of hardware operations with pseudocode for the internal implementation. Construction of a tracer leads to registration within the singleton. This singleton is then used in each hardware operation to access the tracer. In the first operation, we add the insertable to the tracer and save the operation's output. In the second operation, this saved output is compared to the provided input to ensure no (non-recorded) alteration is made in-between the two traced operations. Lastly, the tracer supports saving the recorded insertables via serialization to a file.

```
torch::Tensor inference_trace(torch::Tensor input, string model_path);
```

Listing 27: Interface of the operation supporting replay of traced models for inference.

abstract spiking neural network description developed in section 4.1.4 as well as its mapping and routing to a hardware representation are planned to be used as basis for gradual construction of spiking operations or models. This then results in a single interface for abstract spiking neural networks for both the PyTorch front end and the PyNN front end, cf. section 4.3. A second prerequisite to be answered is efficient data transformation between PyTorch data and hardware compatible counterparts. For spiking neural networks these are predominantly spikes. PyTorch's tensors are intrinsically dense data structures, whereas spike data typically is sparse in both the time and the location (source or target) dimension. It however offers experimental support for sparse tensors [24], where data is represented by a list of entries and their position instead of a dense tensor of entries. Their true benefit is ability for dense iteration, e.g. for simulators, which is however not of interest for interfacing the hardware, since the sparsity is preserved there. Additionally, it is currently not possible to freely specify a sparse tensor's shape, which leads to the inability to specify the complete runtime of an experiment through the tensor's size in the time dimension. We adopt the idea of sparse representation of spike data by a list-like tensor and specify the runtime information separately. This leads to a spike data representation as two-dimensional tensor, where the outer dimension resembles the list of spikes and the inner dimension contains data associated to a single spike, namely its time stamp and location, the latter being a pair of population descriptor and neuron index. In contrast to non-spiking activation data, we resort to using integer data types in the PyTorch tensors here directly, because sub-integer precision for location data is not necessary (e.g. there is no neuron at index 2.4).

Using these two basic building blocks, Elias Arnold is continuing integration and development of an interface for construction and specification of spiking neural networks to PyTorch in his upcoming Master thesis.

4.3 PyNN back end

This section describes the implementation of PyNN as a front end for using the BrainScaleS-2 hardware in computational neuroscience. In section 3.3.1 we introduce PyNN and describe the general scheme for integrating the hardware as new back end. In the following, we focus on two main aspects of the integration, decisions about the user-facing API and the back end implementation of providing execution on the hardware. A large portion of what is described below (everything except using the signal-flow graph as back end) is developed conjointly and implemented solely by Milena Czierlinski in her bachelor thesis [13].

4.3.1 User-facing API

At the current state of support for the hardware, we lack the ability to transform between hardware neuron parameters and parameters of a biological neuron model. The calibration

would need to provide a translation between the equivalent parameters and additionally, a multitude of technical parameters would need to be set automatically. This infeasibility prevents usage of already existing neuron models from PyNN for the BrainScaleS-2 hardware. Therefore, a new neuron model, which directly maps (a subset of) the hardware parameters as described in the lower software layer lola, cf. section 3.1.3, is developed [13] and provides access to hardware properties without further abstraction.

The PyNN Python package does not allow for hooking-in new functionality from outside the module (opposed to e.g. PyTorch. Therefore, we provide the hardware support in a separated module, called `pynn_brainscales` [21], which adheres to the upstream interface and re-uses upstream implementation, where applicable. This also allows providing early checks at other re-used parts of the interface against hardware compatibility, e.g. checking provided synaptic weights against the range restrictions on hardware, and therefore improving user experience.

4.3.2 Back end implementation

PyNN uses a singleton-pattern for registration of populations and projections to be part of the simulation or emulation. A free function `run(runtime)` then triggers the simulation or emulation and uses the network state aggregated in the singleton.

Each back end has to provide its own implementation for registration and transformation to its interface. The point at which the transition between PyNN structures and back end structures occurs is not decided, e.g. the Nest back end chooses to directly fill the Nest data structures upon creation of populations or projections [55]. In contrast, we choose to perform this transition as late as possible, which results in performing it within the `run` function. This has several advantages for the hardware back end. On the one hand, it allows performing the mapping and routing to hardware locations, when all entities, which shall reside on the hardware are known. This allows optimizations, which would not be possible with partial knowledge and prevents necessity for multiple iterative mapping and routing calls after each alteration to the network. On the other hand, only one transformation of parameters between the PyNN data structures and lower software layers is necessary. This prevents a direct relation of number of calls into lower-level software to network alterations in the PyNN API, which results in faster execution due to missing back-and-forth transformation between `C++` implementation of the lower level software and the Python front end software.

Adhering to this design decision, a back end implementation using the lower level software layers haldls, lola and stadls, cf. section 3.1.3, directly is performed in [13]. Figure 4.18 shows the layer structure. For this, a general purpose mapping and routing algorithm is developed and implemented in [13] using the Python-API of the lower level software layers directly and Numpy [72] for calculations. It fills available hardware neurons and the synapse array linearly. Using the lower level software directly allows decoupled concurrent development of the PyNN

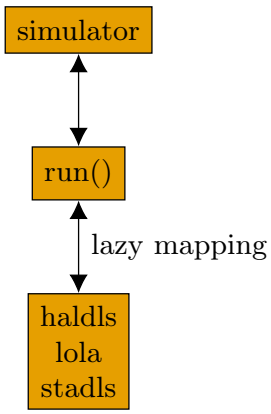


Figure 4.18: Layer structure of the PyNN back end implementation using the lower level software directly. Network state is given by the `simulator` singleton. Inside the run function, lazy mapping and routing is used to fill the lower level software data structures, cf. section 3.1.3, directly.

front and back end implementation and the signal-flow graph hardware representation.

In this thesis, a smoother transition between PyNN and hardware representation is developed and integrated. As described in section 3.3.1, the PyNN data structures describe an abstract graph. The abstract network description and its mapping and routing to a signal-flow graph hardware description described in section 4.1.4 is ideally suited for this task. Its network description interface allows a one-to-one relation to (and from) PyNN populations and projections. The mapping and routing then is a black-box algorithm on this network description resulting in a signal-flow graph hardware representation, which is executable. Like for the PyTorch back end in section 4.2.4, we use the just-in-time graph executor described in section 4.1.2 in order for the results to be available upon return of the `run` function and because it is the only executor, which currently supports the spiking mode of operation. The mapping and routing algorithm developed and implemented in [13] is translated to a `C++`-based implementation, which fits the expected interface of section 4.1.4. Figure 4.19 shows the software layers with the signal-flow graph as intermediate representation.

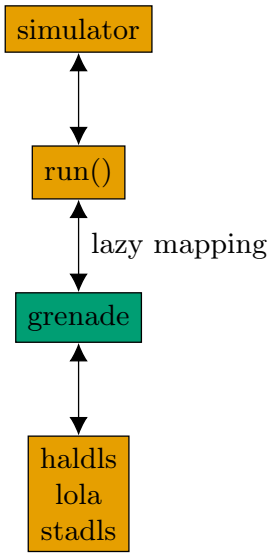


Figure 4.19: Layer structure of the PyNN back end implementation using the abstract network description from section 4.1.4 and signal-flow graph hardware description, cf. section 4.1.1 as intermediate representation. Network state is given by the `simulator` singleton. Inside the run function, the abstract network representation is constructed. Following, lazy mapping and routing is used to construct a signal-flow graph hardware representation. During execution in the just-in-time executor, cf. section 4.1.2, is used for lowering the signal-flow graph to configuration in the lower level software layers.

4.4 Profiling tools

This section covers the implementation of the proposed profiling tools in section 3.4. The timing facilities for runtime duration tracing, cf. section 3.4.1, were already implemented and only used and therefore not described in detail here.

4.4.1 Hardware mock

The hardware mock shall be a drop-in replacement for real hardware with limitations described in section 3.4.2. As described in section 3.1.3, the communication layer hxcomm provides so-called connection handles for hardware access, but also already for simulation access. Similarly, we provide the ZeroMockConnection for the hardware mock, which yields zero payload for each read instruction and ignores all other instructions. Contiguous memory allocation for the responses is preferred over possibly distributed chunk allocation for cache locality. The maximal amount of responses is known upon execution, which allows pre-allocation of the response space at the cost of possibly allocating too much. This reduces the number of allocations to one, independent of the amount of instructions to process. The connection is used via a type-safe union in the other software layers like stadls. Therefore, this new connection type can be used without any additional integration work. Selection of which connection type to use was implemented by querying the environment for the already existing connections. We integrate this new connection into this framework, which allows using it without necessity of compile-time changes. In section 5.1.2, we evaluate the mock with regard to achieved runtime performance.

5 Results

In this chapter, the developed and implemented software concepts are evaluated. Since functional verification is a necessity and not a feature, we will focus on investigating performance measurements. The framework is already in use [66, 22]. We combine evaluation of real-world experiments with artificial benchmarks.

As already described in section 3.4, the BrainScaleS-2 hardware is supposed to be used as accelerator for experiments in the spiking and non-spiking mode of operation. The primary metric of performance therefore is the runtime of the developed software in comparison to existing prototype hardware as well as thought-of artificial alterations of this performance.

We start by evaluating the proposed tools for profiling of section 3.4 in section 5.1. They are then used throughout the other section sin this chapter. Continuing, baseline measurements of the lower-level software layers, cf. section 3.1.3, are established as reference against which to compare the developments.

Continuing, the signal-flow graph-based experiment description and execution, cf. sections 3.2 and 4.1 is evaluated using the formerly measured tools and baselines in section 5.3. For the PyTorch extension, cf. sections 3.3.2 and 4.2, we evaluate additional interfacing performance in section 5.5 to ensure minimal overhead. Continuing, the PyNN implementation using the signal-flow graph hardware representation, cf. section 4.3 is evaluated using a soft-winner-take-all network from [13] as real-world experiment. Furthermore, the PyTorch extension and graph-based experiment description and execution is evaluated in-depth at the application of a competition for energy-efficient classification of electrocardiogram recordings in section 5.6.

We complete the chapter by description of the organization and availability of the software developments.

The software and FPGA state used in the conducted experiments is documented in appendix A.

5.1 Profiling tools

This section contains measurements regarding the overhead introduced by the implemented runtime tracing and the achieved performance (processing speed) of the hardware mock.

5.1.1 Runtime tracing in production software

To evaluate feasibility of the time interval measurement proposed in section 3.4.1 the isolated overhead induced by the measurement is investigated. A number N of back-to-back interval measurements with empty body in-between the start and end measurement is conducted and its time interval measured. Using the same measurement method for the duration of multiple measurements allows to mitigate overhead for the outer measurement by increasing the number. To ensure no congestion effects when using the measurement concurrently, this experiment is conducted in parallel with the hardware concurrency. 100 measurements are taken for each N in every thread and the results are averaged. Figure 5.1 shows the resulting time duration for a single interval measurement for different numbers of back-to-back measurements to range slightly below 0.1 µs. The time measurement is based on

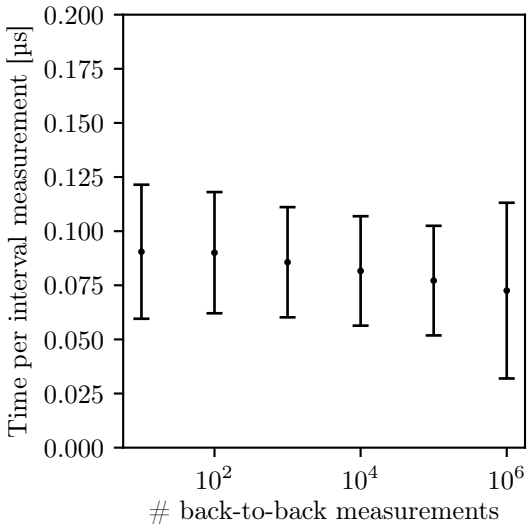


Figure 5.1: Time interval measurement overhead in dependence of the number of back-to-back interval measurements taken. The measurements are taken with an AMD Ryzen 7 3800X featuring a hardware concurrency of 16. 100 measurements are averaged and displayed alongside the statistical standard deviation. The duration does not significantly depend on the number of back-to-back interval measurements and ranges slightly below 0.1 µs.

using the `gettimeofday` function provided by the operating system [30]. Its digital time resolution is 1 µs. Since the measurement overhead of 0.1 µs is one order of magnitude below the resolution, the measurement method can be used for the full representable duration spectrum. For a measured section of 1 µs, the overhead is expected to be smaller 10 % and for measured sections longer than 10 µs, the overhead is expected to be below 1 %.

5.1.2 Hardware mock

To evaluate the implemented hardware mock connection, its achieved performance in terms of duration per UT message is investigated. A sequence of random instructions is generated and executed using an instance of the hardware mock for different settings of the expected duration per message in the range of 1 ns to 15 ns. Random instructions yield a probability of 17 % to be a read instruction, because there are 12 instruction types of which one is

always a read and two are a potential read with 50 % probability leading to $\frac{2}{12}$ probability to generate a read. Additionally, we measure for the corner cases of no read instructions and only read instructions. Potentially there are multiple hardware mock instances to be used in parallel. Therefore, the achieved performance is additionally evaluated against the number of concurrently used connections. Figure 5.2 shows the resulting achieved measured performance for one and multiple hardware mocks. The measurements show that

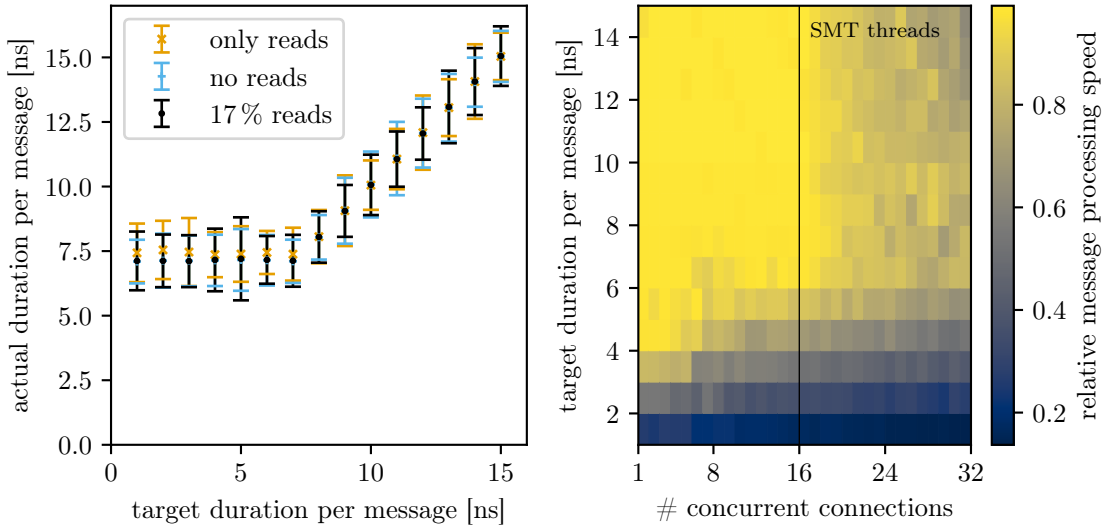


Figure 5.2: Achieved performance of the hardware mock in terms of actual duration per UT message compared to the expected set duration. Left: Single hardware mock actual duration $\frac{t_{\text{actual}}}{\#\text{instructions}}$ against set target duration t_{message}. Until a duration of 8 ns per message, the hardware mock is able to fulfill the expected target. The minimally reachable duration settles around 7 ns. We don't observe significant deviations for no reads and only a small performance drop of 0.5 ns for only read instructions. The measurements are taken for 10^5 instructions averaging the results of 10^4 executions. Right: Multiple concurrent hardware mocks' message processing duration relative to the target. A value of 1 means the target is reached, while lower values express the slow-down against the target. The measurements are taken with a AMD Ryzen 7 3800X featuring a hardware concurrency of 16, averaging the results of 10^4 executions for 10^4 instructions. Only above this number of connections a slight decrease in performance is visible. It is to be noted that for all configurations the to be processed UT message instructions fit into the level-3-cache of the used processor.

the target duration of 8 ns, which relates to a wire speed of 8 Gbit s^{-1} currently available at the chip-FPGA link, is reached for different concurrency configuration up to the hardware concurrency. This target coincides with the chip's link speed. Therefore, the hardware mock can be used to characterize the developed software against the chip's design limitations.

5.2 Establishing a performance baseline

In order to be able to reason quantitatively about the performance of the developed software, we establish a baseline for the lower level software used, which is introduced in section 3.1.3. This allows determining introduced overhead and comparing it to the overall performance achieved by using the lower level software.

Usage of the lower level software can be summarized as follows. We build playback programs from haldls and lola container data and halco coordinates by *encoding* their information into a sequence of UT messages. These are then transported to the FPGA and executed there resulting in some mutation of the hardware state. Following, result data is transported back and *decoded* into container data again.

We investigate these four steps individually in the following.

5.2.1 Encoding

Encoding describes the process of constructing a linear sequence of UT messages from container data structures in haldls and lola in conjunction with placement information from halco. These UT messages, i.e. instructions, can then be transported to the FPGA, at which they are executed. As described in section 3.1.3, encoding takes place for the two main operations, write and read operations. While for the write operation, no responses are expected, read operations trigger response generation. In fig. 5.3, we show the encoding rate for write operations for all currently available container structures. Random container data is used to simulate a real-world use-case. It becomes clear, that lola data structures offer better performance than haldls containers. This is due to the fact, that the former offer tighter loops over the latter than invoking write operations using the small haldls container structures directly. We see an average encoding rate for writes of lola containers of 38.1 MHz of UT messages. Converted to data to be transported, this amounts to $304\,\mathrm{MB\,s^{-1}}$, because for encoding, every UT message features a width of 8 B currently. Figure 5.4 shows the encoding rate for read operation for all currently available container structures. Again, haldls container encoding is slower than lola container encoding. We see an average encoding rate for reads of lola containers of 19.8 MHz of UT messages. This is equivalent to $158\,\mathrm{MB\,s^{-1}}$ by the same calculation as above. However, compared to write operations, the distribution of reads will mostly be limited to reading memory from the PPUs or the FPGA DRAM. These result in a mean rate of 35.9 MHz, which is equivalent to $287.2\,\mathrm{MB\,s^{-1}}$.

5.2.2 Decoding

Decoding is used to construct haldls and lola container data from response UT messages resulting from an execution on hardware. In section 3.1.3, we explain, that decoding takes place for reads in so-called tickets, which are comparable to `std::future` objects. Figure 5.5

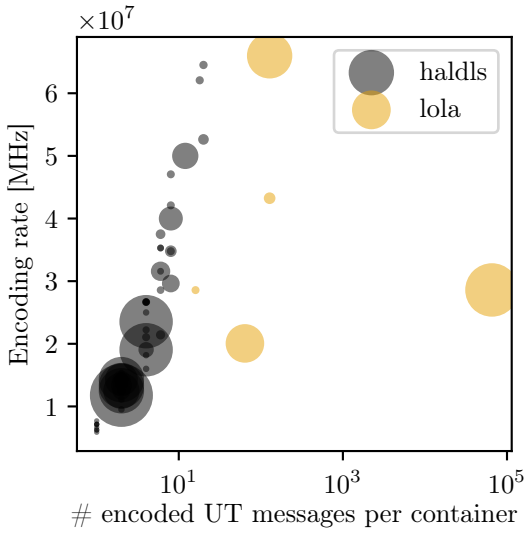


Figure 5.3: Encoding rate of write operation on random haldls and lola container data. Measurements are taken by averaging the time for 10^2 random container values per type. The scatter plot shows all currently available containers. The marker size represents the container type's portion of a complete system configuration in UT message count accumulated for all coordinate values, i.e. instances of the container on the hardware. We see an increase in encoding rate for larger containers. The weighted average encoding rate is 38.1 MHz for lola containers and 16.0 MHz for haldls containers. All measurements are taken on a machine with an AMD Epyc 7402P processor.

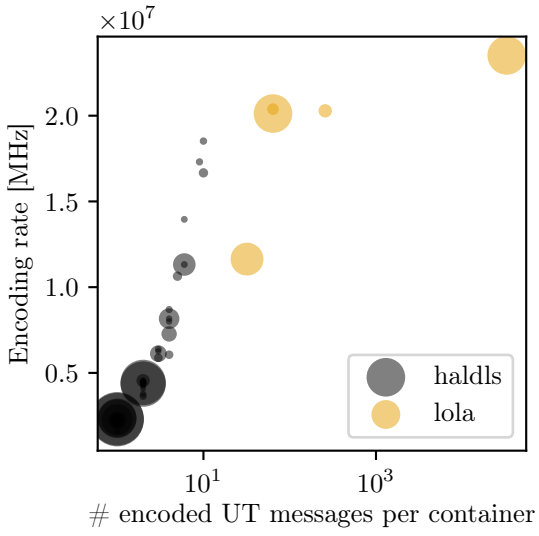


Figure 5.4: Encoding rate of read operation on random haldls and lola container data. Measurements are taken by averaging the time for 10^2 random container values per type. The scatter plot shows all currently available containers. The marker size represents the container type's portion of reading back a complete system configuration in UT message count accumulated for all coordinate values, i.e. instances of the container on the hardware. We see an increase in encoding rate for larger containers. The weighted average encoding rate is 19.8 MHz for lola containers and 3.0 MHz for haldls containers. All measurements are taken on a machine with an AMD Epyc 7402P processor.

shows the rate of decoding UT messages into container data. We see an average decoding

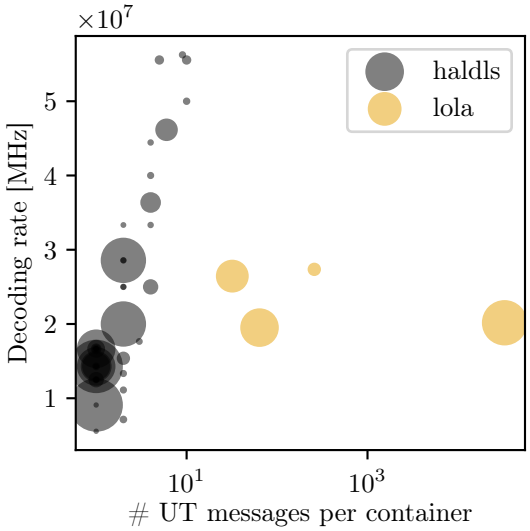


Figure 5.5: Decoding rate for haldls and lola container data. The hardware mock introduced in section 3.4.2 is used to generate the response data. Measurements are taken by averaging the time for 10^2 random container values per type. The scatter plot shows all currently available containers. The marker size represents the container type's portion of decoding reads of a complete system configuration in UT message count accumulated for all coordinate values, i.e. instances of the container on the hardware. We see an increase in decoding rate for larger containers. The weighted average decoding rate is 20.9 MHz for lola containers and 16.1 MHz for haldls containers. All measurements are taken on a machine with an AMD Epyc 7402P processor.

rate of lola containers of 20.9 MHz. This is equivalent to 167.2 $\mathrm{MB\,s^{-1}}$ again because the response UT message width is 8 B for all deterministic responses.

5.2.3 Transport

Transport of encoded UT messages to and from the FPGA is currently implemented via 1 $\mathrm{Gbit\,s^{-1}}$ Ethernet (full-duplex). We therefore expect to reach this rate for large-enough amount of transported data. The transport layer offers the ability to perform loop back transactions, which allows throughput benchmarking. Figure 5.6 shows a loop back measurement for a range of data sizes. A maximal rate of 115 $\mathrm{MB\,s^{-1}}$ is observed, which is close to the expectation of 117 $\mathrm{MB\,s^{-1}}$ (Using a maximal frame size of 1538 B, which carry $180 \cdot 8$ B words results in an expected rate of 1 $\mathrm{Gbit\,s^{-1}}/8\,\mathrm{bit\,B^{-1}}/1538\,\mathrm{B} \cdot 180 \cdot 8\,\mathrm{B} = 117\,\mathrm{MB\,s^{-1}}$.).

5.2.4 Execution

Quantification of execution speed is dependent on the content of the instructions to be processed on the FPGA. For example a wait instruction might take many FPGA cycles to complete. In order to nonetheless quantify the performance of the execution, we investigate

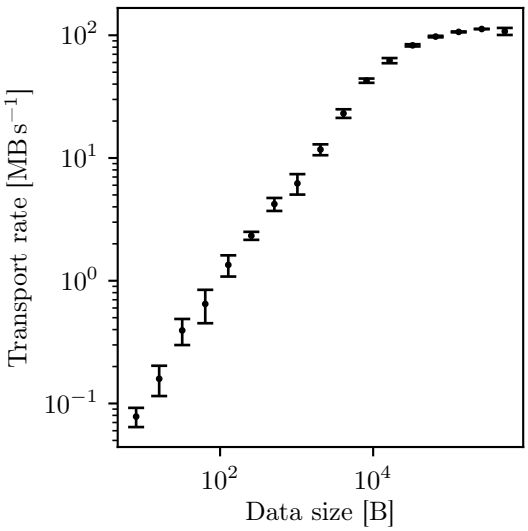


Figure 5.6: Rate of looped-back transport of data between host computer and FPGA via 1 Gbit s^{-1} Ethernet. The measurement is done 5 times for a data size between 8 B and 512 kB. For large-enough data to be transported, the rate approaches the expectation. We see a maximal rate of 115 MB s^{-1}.

execution of an instruction, which is guaranteed to be processed within one FPGA cycle. The FPGA's clock frequency is 125 MHz. The execution time includes encoding UT messages to and from words, which are being transported. Therefore, this time is included in the measurement. Figure 5.7 shows the throughput measurement. A write-instruction is used, therefore no additional responses are generated. This is a worst-case of an experiment consisting of only writes, e.g. configuration. A maximal rate of 109 MB s^{-1} is observed

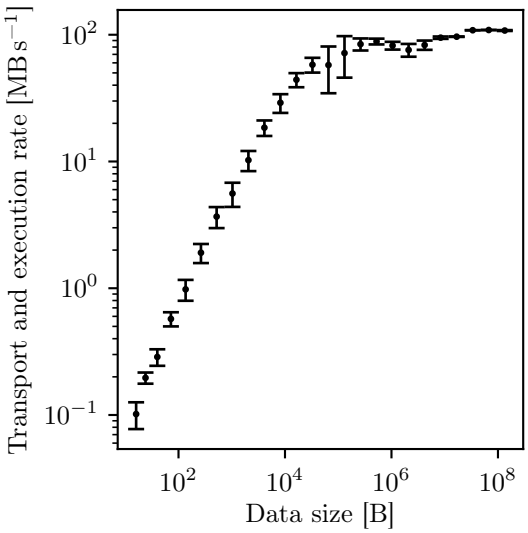


Figure 5.7: Rate of execution of instructions on the FPGA including encoding and decoding of UT messages from a word stream. The measurement is done 5 times for a data size between 16 B and 128 MB. For large-enough data to be transported, the rate approaches the expectation. We see a maximal rate of 109 MB s^{-1}.

for 64 MB instruction sequence size. Execution therefore does not pose an additional performance penalty compared to the isolated transport performance.

5.2.5 Notes on concurrency

The steps described beforehand offer different ability to be performed concurrently. Encoding can in principle be performed concurrently for a single resulting sequence, because concatenation of encoded sequences is supported. However, this concatenation comes with similar cost than the encoding itself and therefore concurrent encoding offers no benefit. Decoding can be performed concurrently on responses of execution of a single sequence of instructions. For the use-cases evaluated, the amount of read responses related to a single sequence of instructions is however too small for the benefit of concurrency to overcome its cost of data and task distribution. Transport of completed instruction sequences is serialized for a single physical hardware setup. The same holds for its execution. The combination of steps is fully serialized, so a sequence can only be transported, when it is complete and responses can only be decoded once all of them are present on the host computer. All steps can be performed fully concurrently for independent instruction sequences, their responses and hardware setups.

5.3 Graph-based experiment description and execution

To evaluate the signal-flow graph-based experiment description and execution formulated in section 3.2 and implemented in section 4.1, we use a collection of artificial benchmarks to highlight performance properties of different parts of the developed framework. First, we investigate the graph-based description implementation in terms of required time and memory for construction in section 5.3.1. Next, the just-in-time execution is benchmarked for both spiking and non-spiking experiments in section 5.3.2. Following, the compilation for standalone execution is evaluated for non-spiking experiments as its currently only supported mode of operation. Last, we investigate the abstract network description and mapping and routing process, cf. section 4.1.4.

5.3.1 Graph construction

The gradual build-up of the graph is the only occasion, where its content is mutated. By adding vertices and edges in-between them, the graph is gradually completed. We expect construction of the graph to feature a linear dependency of time consumption in dependence of the number of vertices to be added. In particular, the runtime checks described in section 4.1.1 performed for each addition shall not lead to large time expenditure compared to other operations necessary during graph construction like partitioning or construction of vertex values. One check is expected to perform in a non-linear fashion, the acyclicity check for every added vertex. This is due to the fact that we try to topologically order the full graph for this check. Therefore, we perform time expenditure measurements with and

without this check for every added vertex (it still has to be checked once before compilation or execution).

We use building differently-sized non-spiking matrix multiplications as artificial benchmark. They use a multitude of different vertices and allow easy scaling of different numbers of vertices simply by changing the shape of the matrix. The partitioning-scheme explained in section 4.2.3 is used. Figure 5.8 shows duration measurements. We see no strong non-linear scaling effects in the range of number of vertices measured. The acyclicity check for every vertex addition becomes dominant (i.e. same order of magnitude as all other checks) only at the top end of the measurement for 1.6×10^4 execution instances.

Additionally, we expect the memory consumption of the constructed graph to depend linearly on the number of vertices. For this, peak memory consumption is measured using Valgrind with its heap memory profiler Massif [52]. Peak heap memory allocation yields a sensible measurement, because the graph data is allocated completely on the heap, and we know, that during construction, outside the graph structure, only temporary values are allocated, which are small compared to the complete graph. Figure 5.9 shows the peak memory consumption for the differently-sized matrix multiplications. As expected, we observe a linear dependency of memory consumption on the number of vertices.

5.3.2 Just-in-time graph execution

The just-in-time executor, developed in section 4.1.2 offers full support for both spiking and non-spiking experiments on BrainScaleS-2. While most of the implementation is shared between the two modes, some parts are simply not touched by executing an experiment, which only used one or the other. Therefore, specific properties are evaluated in dedicated experiments below.

Spiking experiments

Experiments involving spike events are characterized by consuming and producing such data. Ideally, the executor shall yield minimal overhead of such data transport when compared to using the lower-level software layers directly. Since the actual network topology does not influence the data flow, we use a loop back experiment. The external spike event transfer saturates the FPGA-chip connection for sending two events per UT message [57]. Therefore, we use this rate for the comparison, since it is an upper limit for the rate of transferred spike events. In contrast to the lower-level software layers, the just-in-time graph executor offers sorted response events with relative timing annotation. Figure 5.10 shows spike loop back time in comparison between the just-in-time graph executor and using the lower-level software directly. We see a performance decrease by a factor of ≈ 2 for using the just-in-time graph executor with its features for spike event loop back. The performance of

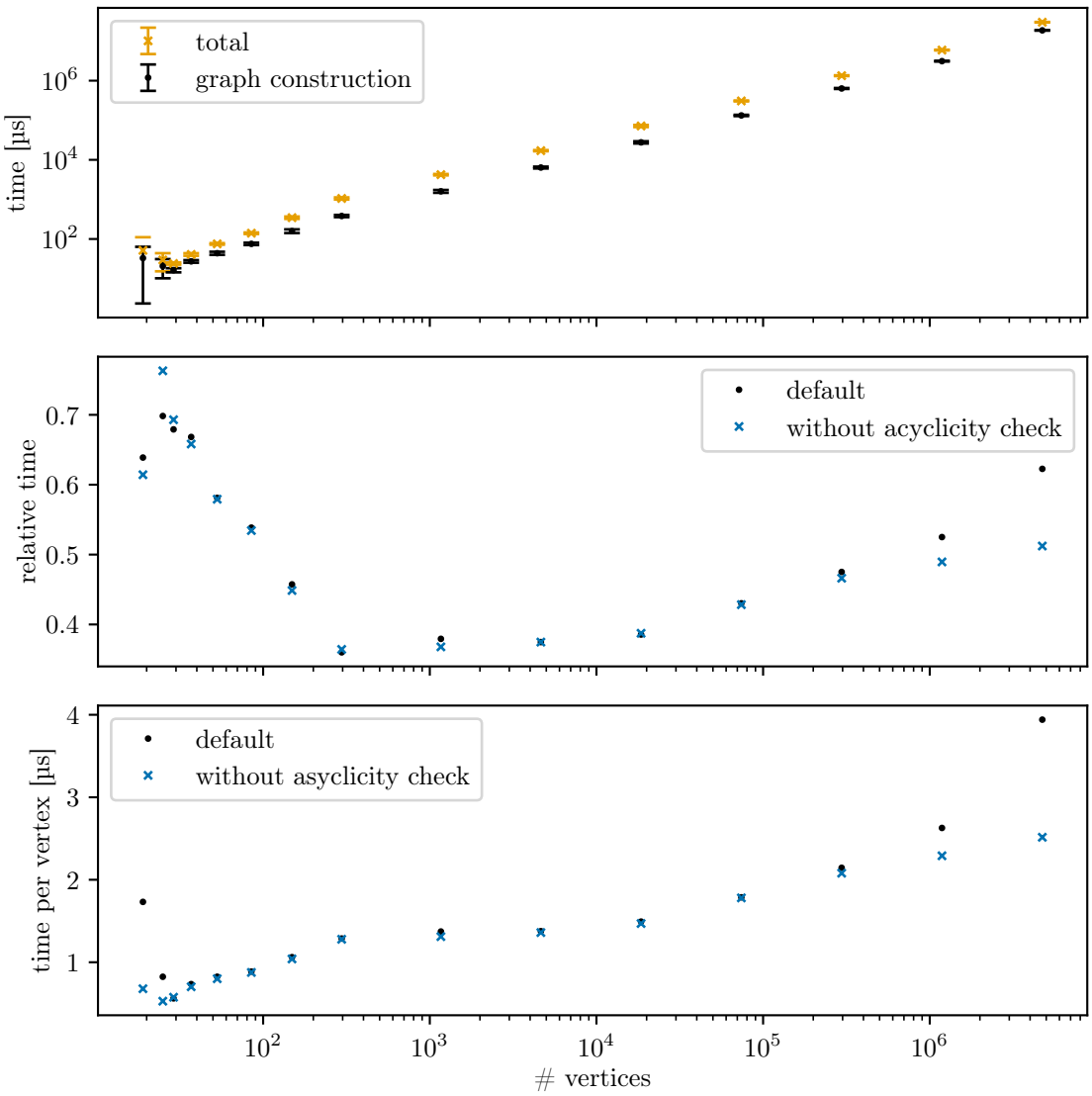


Figure 5.8: Time expenditure of graph construction for differently-sized matrix multiplications. The matrix size is quadratic with its order varied between 1 and 3.3×10^4. This is equivalent to be between (below) one and 1.6×10^4 full chip allocations. Top: absolute time of graph construction and total time expenditure including partitioning decisions and vertex value construction; Middle: Time of graph construction relative to total time; Bottom: Average time per vertex addition. We see, that the graph construction requires time in the same order of magnitude than surrounding operations. Below ≈ 300 vertices, the graph construction becomes more dominant. This can be explained in that for these numbers, the matrix multiplication constructed is smaller than a full chip, which results in different surrounding overhead. Until $\approx 10^4$ vertices, relative time as well as time per vertex stays constant. Above, we see a non-linear scaling, but still following a power-law. When disabling the acyclicity check for every vertex addition, this increase rate can be lowered. Still, also when enabled, the time per vertex only increases by a factor of ≈ 4 in the range of number of vertices tested.

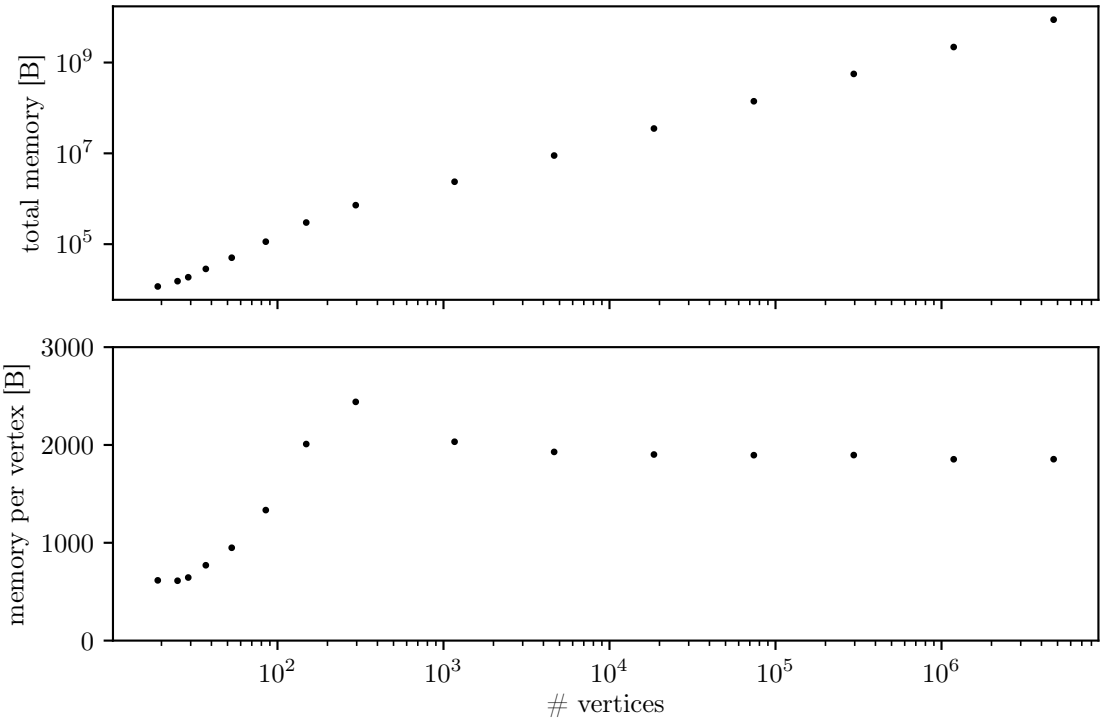


Figure 5.9: Peak memory consumption during graph construction for differently-sized matrix multiplications corrected by the constant offset of an empty measurement. The matrix size is quadratic with its order varied between 1 and 3.3×10^4. This is equivalent to be between (below) one and 1.6×10^4 full chip allocations. Top: Total peak memory consumption; Bottom: Average memory consumption per vertex in the complete graph. It is to be noted, that the actual memory requirement of a single vertex varies greatly depending on its type and use. We see a linear dependency of the memory consumption on the number of vertices. The non-linear behavior for small graphs is explained by that the vertices for a multiplication smaller than a full chip carry fewer data. The measured data is without uncertainty because the execution is deterministic and multiple execution results in exactly the same peak memory consumption.

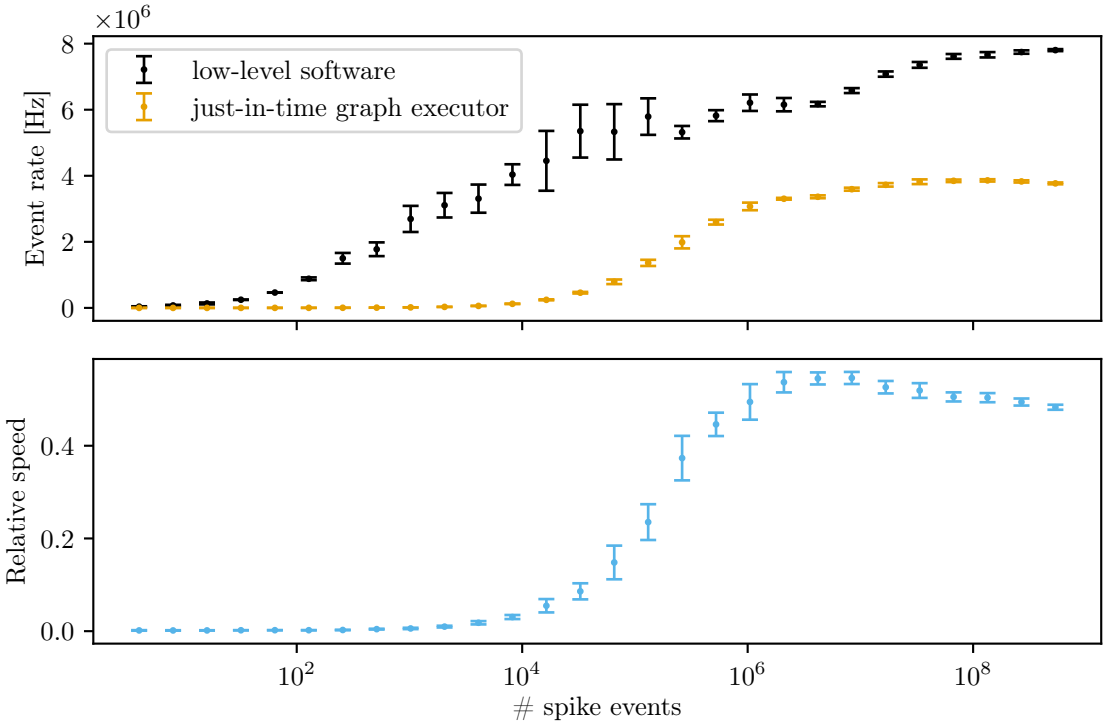


Figure 5.10: Spike loop back performance of the just-in-time executor in comparison to using the lower-level software, cf. section 3.1.3, directly. An average of five measurements is used. Top: Event rate; Bottom: Relative speed of just-in-time graph executor compared to the lower-level software implementation. We see a relative performance of approx. 50 % for many spike events. For few spikes, the just-in-time executor performs significantly slower, because application of the static configuration becomes significant. This is to be seen as a feature in that it ensures reproducible configuration, which can be mitigated by execution of a collection of experiments in one run, cf. fig. 3.14. For both, the experiment using the lower-level software directly and the just-in-time executor, the measured event rate is significantly below the expectation of 27 MHz (using the execution throughput of 109 MB s^{-1} from section 5.2.4 knowing, that two events fit into a UT message of width 8 B). It is to be noted, that the actual realtime execution on the hardware remains unaffected by surrounding software slow-down due to buffering on the FPGA.

both experiments is systematically lower than expected from calculation. Therefore, further investigation is advised, suggesting however a common cause of the slowdown.

Non-spiking experiments

For evaluation of non-spiking experiment performance of the just-in-time graph executor we choose to again use matrix multiplications as primary artificial benchmark. We use back-to-back events and no resending of such for minimized execution time in the following, cf. section 4.2.6 or [75] for explanation of their effect. Non-spiking experiments allow to easily use multiple execution instances, be it via different physical chips or using the same chip(s) multiple times. Additionally, the data transferred and time computed for a single operation is small compared to the initial configuration of the system. Therefore, it is important to make use of the batched-execution support of the just-in-time executor. The measure of performance used in the following is the number of multiply-accumulate operations performed per time. There are multiple different evaluations possible, which are performed in the following.

First, we investigate operations-rate for a multiplication using a single physical chip. This follows the possible concurrency seen in fig. 4.7 and yields a baseline performance. We expect saturation for increasing number of batches executed after one-another and increasing matrix size, because it increases potential for parallelism. Figure 5.11 shows the results. We see an increase in performance towards larger batch size and matrix order as expected. The maximal performance is reached at $710\,\mathrm{Mop\,s^{-1}}$ operation rate. Chip utilization reaches above 99 % of total execution time for large matrices. This implies, that the developed just-in-time executor poses minimal runtime overhead. Comparing to the theoretical data transport limit constructed in section 3.1.4 of $4\,\mathrm{Gop\,s^{-1}}$, the achieved rate is approx. a factor of 5.6 smaller. Together with the high hardware utilization this suggests, that the majority of time is spent on the actual operation instead of data transport.

In the following, a fixed batch size of 8192 is used to counteract overhead introduced by initial configuration. Continuing, we use round-robin allocation of multiple physical chip instances. This allows concurrency additionally in the execution, cf. fig. 4.8. Since currently, only a limited amount of physical hardware is available, we make use of the software mock introduced in section 3.4.2, setting its transmission speed to $1\,\mathrm{Gbit\,s^{-1}}$ to mimic the currently available hardware. Finally, we increase the speed of the software mock to mimic a direct $8\,\mathrm{Gbit\,s^{-1}}$ connection to the chip(s) as explained in section 3.4.2. Figure 5.12 shows the results for a number of physical chips between one and 48. The upper limit is chosen, because it is the number of chips on a planned multi-chip wafer-like setup, it also matches the hardware concurrency of the used host computer. We achieve a speed-up of ≈ 8 between one and 48 simulated chips with $1\,\mathrm{Gbit\,s^{-1}}$ wire speed, where from ≈ 24 chips onward a plateau is reached. The maximal operation rate is measured as

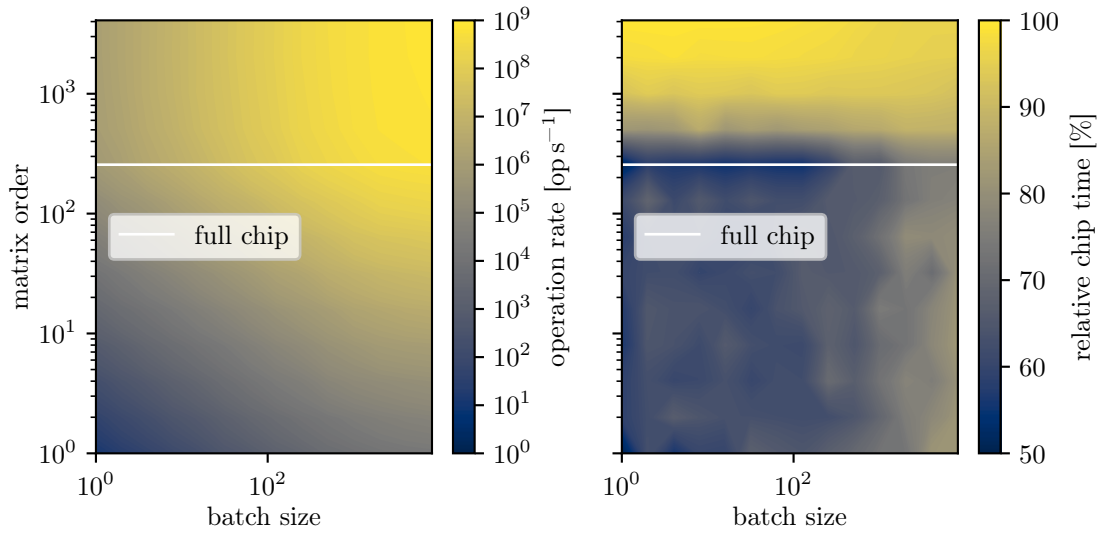


Figure 5.11: Square-matrix multiplication using the just-in-time executor for a single physical chip. Left: Operation rate; Right: Chip utilization time relative to total execution time. Increase in batch size increases performance. However, only above the full chip size, potential concurrency in preprocessing and postprocessing, cf. fig. 4.7 occurs, further increasing the performance. A maximal rate of operations is reached at 710 Mop s^{-1}. Chip utilization rises until 80 % for matrix sizes smaller than a full chip and reaches >99 % for matrices larger than the chip area. All measurements are taken on a machine with an AMD Ryzen 3800X processor with a hardware concurrency of 16.

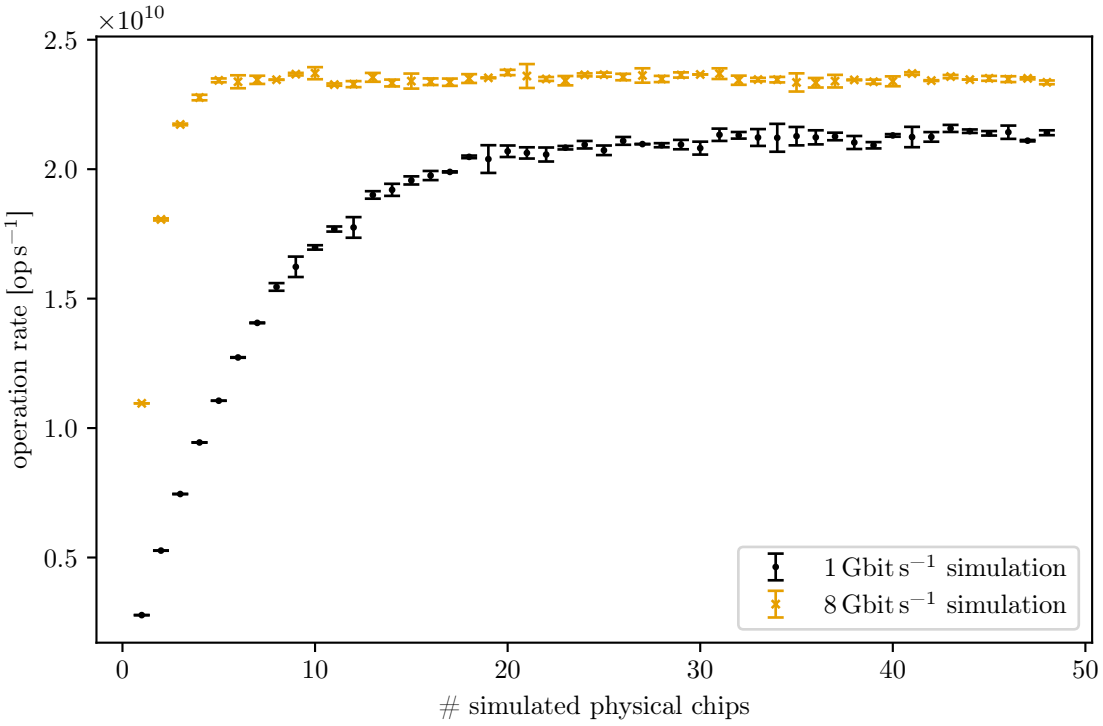


Figure 5.12: Operation rate of a square-matrix multiplication using the just-in-time graph executor and a collection of simulated physical chips. A batch size of 8192 is used in conjunction with a matrix order of 4096. The simulated hardware mock from section 3.4.2 is used with wire speed of $1\,\mathrm{Gbit\,s^{-1}}$ and $8\,\mathrm{Gbit\,s^{-1}}$. The operation rate for a single chip is given as $2.8\,\mathrm{Gop\,s^{-1}}$ at $1\,\mathrm{Gbit\,s^{-1}}$ and $11.0\,\mathrm{Gop\,s^{-1}}$ for $8\,\mathrm{Gbit\,s^{-1}}$ respectively. This is higher than usage of actual hardware for the $1\,\mathrm{Gbit\,s^{-1}}$ simulation, cf. fig. 5.11, which can be explained by missing execution time. We see, that until a number of chips of ≈ 24 the performance increases and reaches a plateau at $21.6\,\mathrm{Gop\,s^{-1}}$ for the $1\,\mathrm{Gbit\,s^{-1}}$ simulation. The $8\,\mathrm{Gbit\,s^{-1}}$ simulation reaches its plateau for fewer chips of ≈ 5 at $23.7\,\mathrm{Gop\,s^{-1}}$. Missing further increase can be explained by that then concurrent pre- and post-processing as depicted in fig. 4.8 lacks (more) free processor time on the host computer. All measurements are taken on a machine with an AMD Epyc 7402P processor featuring a hardware concurrency of 48.

21.6 Gop s^{-1}. The 8 Gbit s^{-1} simulation leads to a further increase of maximal operation rate to 23.7 Gop s^{-1}. The earlier performance plateau for higher wire speed suggests, that there the computation necessary on the host computer becomes dominant. Comparing the highest speed achieved for the simulated chip collection with the performance of the actual hardware, cf. fig. 5.11, a speed-up of 33.4 is achieved, promising scalability of 70 % of maximally achievable scalability for a real multi-chip system of 48 chips with an identical host computer hardware concurrency. Comparing to the theoretical data transport limit constructed in section 3.1.4 of 4 Gop s^{-1} for a single setup connected via 1 Gbit s^{-1}, the achieved rate of 2.8 Gop s^{-1} is slightly decreased. For the 8 Gbit s^{-1} connection, the difference of the theoretical limit of 32 Gop s^{-1} and the achieved rate of 11 Gop s^{-1} grows. This suggests, that at these rates, overhead from en- and decoding and compilation of playback sequences from the graph becomes dominant.

5.3.3 Compilation for standalone execution

In contrast to the just-in-time executor evaluated beforehand, the compilation for standalone execution, developed in section 4.1.2, only offers support for non-spiking experiments on BrainScaleS-2. In this section we evaluate its performance.

Since the compilation process is separated from the execution and the execution itself can be split into initialization, loading data, the realtime execution and storing results, cf. fig. 4.10, we investigate the time consumption of these different parts individually. A matrix multiplication operation is used as for the just-in-time executor as artificial benchmark. In contrast to there, we restrict the matrix size to fit onto one full chip hemisphere here. This leads to only one initial configuration and allows projection onto the full chip, since operations on hemispheres are serialized in the current implementation. Additionally, we expect the locality of input-data to vastly influence achieved performance. However, local space on the PPUs is limited, therefore we test this using a constant input value, which simulates a use-case, where the input data is generated locally in some manner. Figure 5.13 shows the time distribution for local and non-local input data of the multiple steps involved in compilation and execution for a range of input samples. A rate of multiply-accumulate operations of 490 Mop s^{-1} is reached for a local data source. This is less than but comparable to the performance achieved for the just-in-time executor of 710 Mop s^{-1} in fig. 5.11. In section 5.6, we evaluate the compilation for standalone execution further in the context of a real-world application classifying ECG traces.

5.3.4 Building networks from populations and projections

In this section, we evaluate the abstract spiking neural network description and mapping and routing process towards a signal-flow graph hardware representation developed in section 4.1.4.

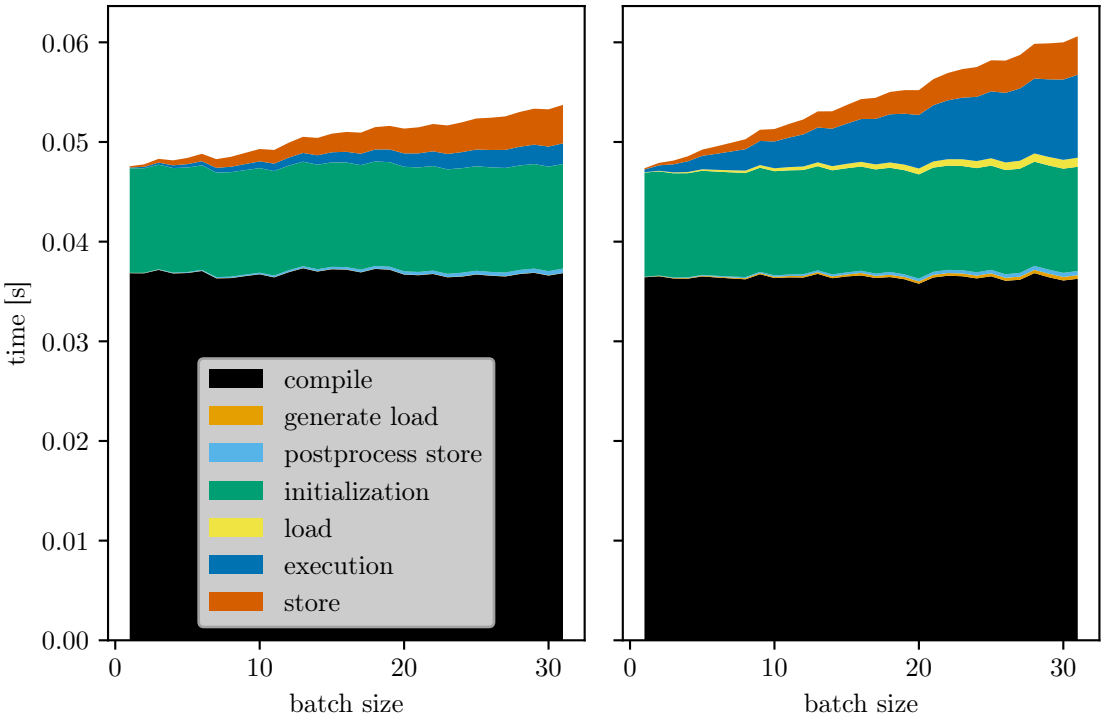


Figure 5.13: Time duration measurement for compilation and execution of a matrix multiplication using the compiler for standalone execution for a range of inputs between 1 and 31. Compilation, generating the load program and postprocessing the store results after execution are performed on the host computer. Initialization, load, execution and store are each executed individually and in linear order on the BrainScaleS-2 hardware. They are measured on the FPGA without transfer duration from and to the host computer to mimic actual standalone operation. Left: Input values are replaced by a constant directly on the PPU to simulate a local data source; Right: Input values as well as output values are placed on the FPGA's DRAM. As expected, compilation and initialization feature no visible dependence on the number of inputs samples. All other durations scale linearly with the number of input (and output) samples. Compilation is the most-costly operation of all, but is only to be performed once and therefore not considered when evaluating the performance of the compilate. Initialization as well is to be done only once, load and store will change depending on the actual application. Therefore, to investigate the performance of the performed computation, only the execution duration is important. We see, that locality reduces the runtime for 31 samples from 8.3 ms to 2.1 ms. The faster execution is equivalent to 490 $\mathrm{Mop\,s^{-1}}$, which is below the performance measured for the just-in-time executor, cf. fig. 5.11, but within the same order of magnitude. In contrast to there however, the complete execution can be performed without host computer interaction. The host computer used features a AMD Ryzen 3800X processor.

First, the abstract network construction is evaluated. Afterwards, the mapping and routing algorithm as well as the construction of a signal-flow graph hardware representation are investigated.

Abstract network description

We use an all-to-all connected single-layer network as artificial benchmark network. It features many single-neuron connections and allows simple ranged measurements for the number of populations and projections by splitting the source and target population gradually and by that also increasing the number of projections. Figure 5.14 shows this scheme of gradually increasing the number of projections by splitting the populations into smaller populations at the example of four neurons in total. Figure 5.15 shows time and memory

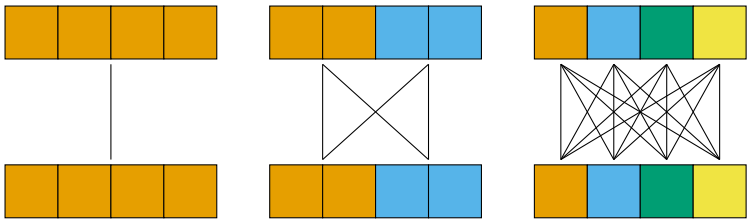


Figure 5.14: Different granularity of populations (and projections) of the same feed-forward network connecting four neurons in each population with all-to-all connections. The projection(s) are depicted by lines, where each projection itself connects all neurons of its source and target population via all-to-all single-neuron connections (not displayed). Left: one population (with all four neurons) per layer; Middle: two populations (with two neurons each) per layer; Right: four populations (with one neuron each) per layer.

requirement for construction of a feed-forward all-to-all connected network for different numbers of populations (and projections). For many projections, we expect a quadratic memory requirement due to the quadratic growth of projection count. We observe weak dependencies (within the same order of magnitude) between granularity of the network description and memory as well as construction duration footprint.

Mapping and routing

The same network configuration is used as worst-case estimation for the process of mapping and routing an abstract network to hardware. The mapping and routing algorithm tested is the ported one from [13]. For the conversion of the mapping and routing result to a signal-flow graph hardware representation, this also investigates scalability of the signal-flow graph representation for varying granularity. Figure 5.16 shows the results. We see constant time expenditure until a certain amount of populations and projections for both measurements with polynomial continuation for higher counts. The mapping and routing

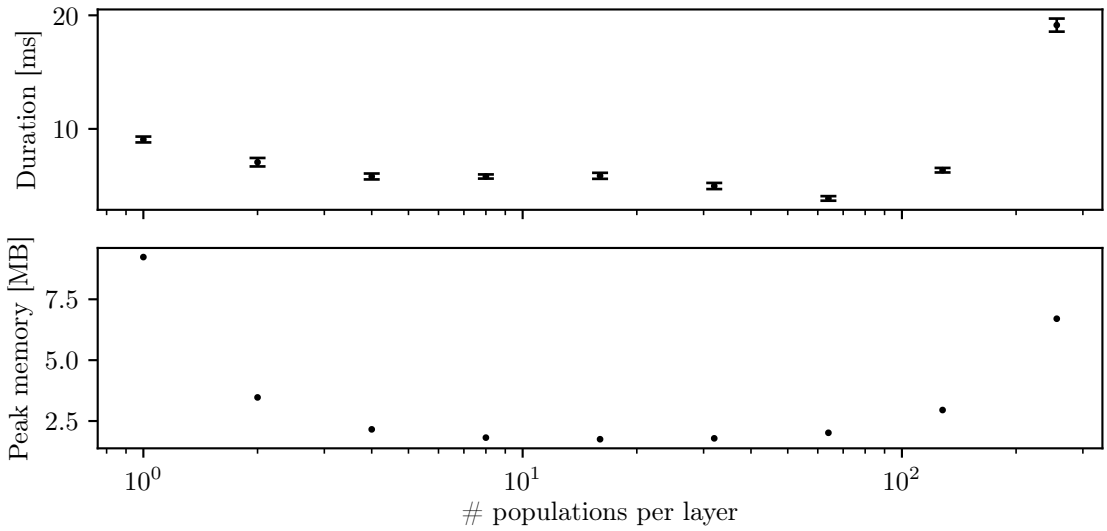


Figure 5.15: Time expenditure (top) and memory consumption (bottom) of abstract network construction for a feed-forward network with varying population granularity projecting 256 external sources onto 256 internal neuron circuits. We see a time consumption between 4 ms and 19 ms, where only for 256 individual populations (i.e. one population for every neuron and one projection for every single-neuron connection) the time consumption increases by a factor of more than two compared to all other configurations. The peak memory consumption ranges between 1.7 MB and 9.2 MB, showing, that different granularity does not have a great effect on the storage and construction efficiency.

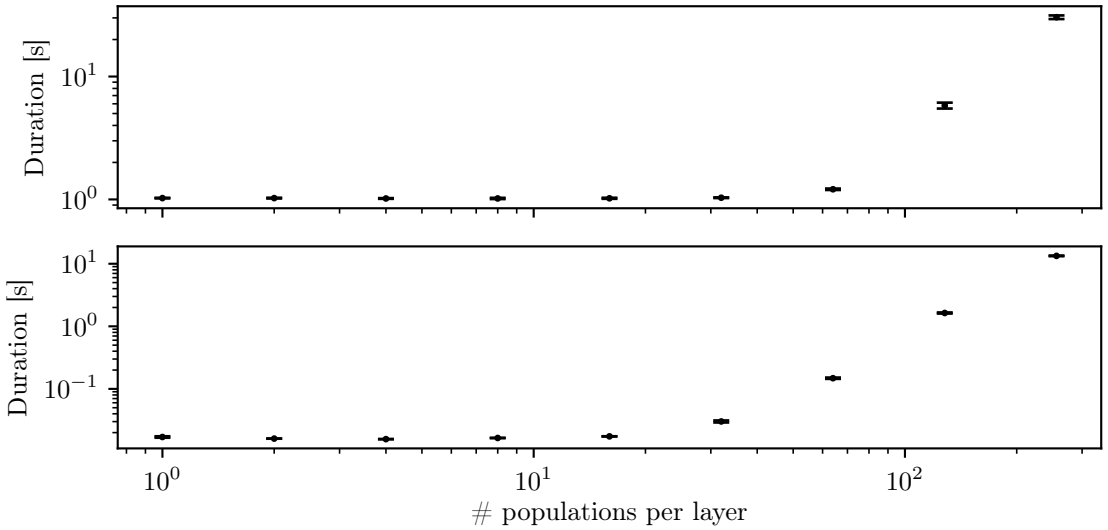


Figure 5.16: Time expenditure of mapping and routing (top) and conversion of its result to a signal-flow graph hardware representation (bottom) for a feed-forward network with varying population granularity projecting 256 external sources onto 256 internal neuron circuits. Until 64 populations per layer, the mapping and routing time expenditure stays constant at 1.0 s. The conversion of the mapping result stays at constant time expenditure until 16 populations per layer with ≈19 ms. Above these thresholds, polynomial increase in time duration is measured, which can be explained by the linear growth in populations and quadratic growth in projections and that this granularity is directly replicated in the signal-flow graph representation.

algorithm requires 1.0 s at least, while the conversion to the signal-flow graph representation uses 19 ms. The conversion therefore is two orders of magnitude faster than the mapping algorithm and stays below its time expenditure even for the maximal number of populations and projections of 256 per layer or 16 384 respectively.

5.4 PyNN

The PyNN front end for spiking neural networks developed in section 4.3 with signal-flow graph back end is to be evaluated against the former Python-based implementation. The abstract network description and mapping and routing developed in section 4.1.4 is already evaluated independently in section 5.3.4 and used here. Therefore, in this section, we focus on the adapter performance to the PyNN front end.

For this we use a soft-winner-take-all network, which was implemented and characterized in [13] for the Python-based front end development. It consists of a ring of neurons, which are connected to their neighbors with decreasing strength depending on the distance. They are each connected to an inhibitory pool of neurons as well. Moreover, they are subject to external excitatory Poisson sources. This network structure is not representative, but it being the only real-world application example accessible at the time is weighted more important than generality in the following evaluation. It consists of 50 populations of size 1 in the ring, 5 external Poisson source populations and an inhibitory pool population of size 10. In total, 61 population and 1790 projections resemble the network under test.

As described in section 4.3, the complete interaction with the back end happens during the `run()` function call. Therefore, we limit the evaluation of runtime performance to this function call in the following. Performance evaluation of the user-facing API outside this function has been evaluated already in [13]. Table 5.1 shows the time expenditure for a single execution of the experiment averaged over 12 executions. We observe a total time expenditure of 490(30) ms for the signal-flow graph implementation and 683(8) ms for the Python-only implementation. The largest portion is spent in transforming the routing result to the signal-flow graph representation with 179(6) ms. Comparing to fig. 5.16, this behavior is expected, but occurrence for a real-world experiment suggests it as entry-point for optimization.

5.5 PyTorch extension

The PyTorch extension for support of the BrainScaleS-2 hardware as accelerator's main element are the operations on tensor data. Its implementation is described in section 4.2. The forward pass is based on subgraph generators, cf. section 4.1.3, of the signal-flow graph representation layer and uses the just-in-time executor, cf. section 4.1.2. Their performance is evaluated in section 5.3. New elements in this extension therefore are the conversion

label	time [ms]
total	683(8)
initialization	80(3)
run	1.8(1)
build run program	70(2)
routing	371(8)
static configuration	98(1)
remainder	62

label	time [ms]
total	490(30)
initialization	58(2)
run	59(4)
routing	13.6(1)
build signal-flow graph	179(6)
build abstract network	21.8(1)
static configuration	88(2)
remainder	70

Table 5.1: Time expenditure of the `run()` invocation using the abstract network and signal-flow graph hardware representation (right), cf. section 4.1.4, and the former Python implementation using the lower level software, cf. section 3.1.3 directly (left). Durations are measured like described in listing 1 using the `time.time()` functionality. We observe a decrease in total time from 683(8) ms to 490(30) ms, which is a reduction of 28 %. The conversion from PyNN populations and projections requires 21.8(1) ms, which is only 4.4 % of total time. The routing in conjunction with generation of the signal-flow graph consumes 193(6) ms, where 71 % are spent in creation of the signal-flow graph. In contrast, in the Python implementation the routing consumes 371(8) ms, which is twice the requirement of the other.

of PyTorch tensor data to and from limited-precision integer data for the hardware, cf. section 4.2.2 and the removal of these conversions by tracing multiple operations from the extension for fused execution (and deployment), cf. section 4.2.7. In the following, both features are investigated.

5.5.1 Tensor data conversion for hardware

For execution on the hardware, floating point tensor data is to be transformed to and from hardware-compatible types. The design decisions are explained in section 4.2.2. For optimal performance, these conversions shall pose minimal overhead, since their execution is required to be serial to the actual hardware execution. We therefore investigate the conversion performance of three mostly used types, 8 bit signed output activation values, 5 bit unsigned input activation values and 6 bit weights values plus a bit for the sign in isolated micro-benchmarks. Conversion duration is measured for two-dimensional data, which relates to value and batch dimension for activations and weight matrices. We measure the time expenditure for conversions of randomized data in fig. 5.17. This shall mimic real-world data. We measure a conversion rate of 276 MHz from 8 bit signed values, 162 MHz to unsigned 5 bit values and 155 MHz to weight values. The last two are expected to be convertible at a slower rate, because they include range checks, while the first one is a direct cast. Comparing these

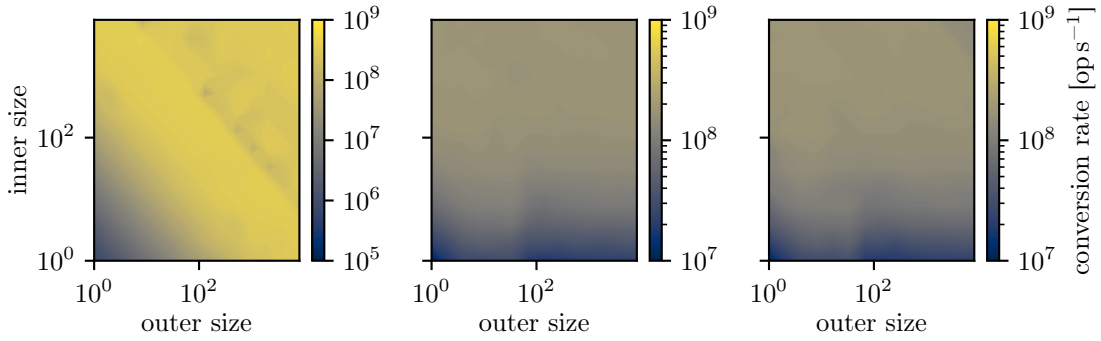


Figure 5.17: Time expenditure of data conversion between PyTorch floating point tensors and hardware-compatible ranged integer values. Conversion is performed for two-dimensional data of shape between 1×1 and 8192×8192 with random element values. Left: Conversion of signed 8 bit (neuron membrane readout result) values to PyTorch tensor data; Middle: Conversion of PyTorch tensor data to unsigned 5 bit (input activation); Right: Conversion of PyTorch tensor data to 6 bit plus sign values (weight). Conversion of the signed 8 bit data is fastest with a median rate of 276 MHz. The unsigned 5 bit data conversion has a median rate of 162 MHz, while the weight conversion is achieved with a median rate of 155 MHz. We observe a performance drop for small inner dimension. This is because the inner dimension memory location of the hardware data is guaranteed to be contiguous, while the outer dimension is not. The left figure shows a performance drop at around 512 kB combined data (e.g. around a shape of 128×512), which coincides with the (single-core) level-2 cache size of the used processor. All measurements are taken on a machine with an AMD Epyc 7402P processor.

findings to the multiply-accumulate performance from section 5.3 of $710\,\mathrm{Mop\,s^{-1}}$, they seem slow. However, in a matrix multiplication, a single input conversion is necessary for matrix-width-many single-value multiply-accumulate operations same as a single output conversion is necessary for matrix-height-many single-value multiply-accumulate operations. This means, the conversion overhead is expected to be small for reasonably large matrices with width and height of similar order of magnitude. This effect is demonstrated in section 5.5.2, where we remove intermediate conversions in a sequence of operations.

5.5.2 Tracing operations for inference

As seen in section 5.5.1, data conversion of operations requires a possibly significant amount of time especially for operations with small dimensionality. In part to mitigate this, in section 4.2.7, we develop the ability to trace a sequence of operations from the PyTorch extension into a single operation, where all intermediate (identity) type conversions are removed and the number of function invocations from Python is reduced to one. To evaluate this described effect, we use an artificial $MRMR...$ sequence of square matrix multiplications M and rectified linear unit operations R (and divisions to convert the positive membrane potential value range of $[0, 127]$ to the input activation range of $[0, 31]$) of varying shape. Figure 5.18 shows the relative performance of execution of the traced sequence of operations compared to using single PyTorch extension operations. We observe a speed-up of $1.9(9)\,\%$

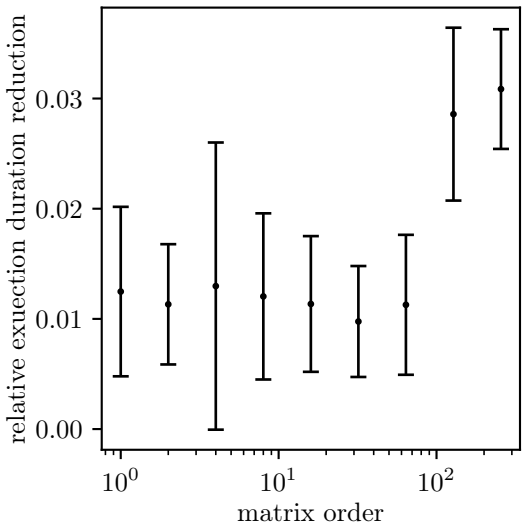


Figure 5.18: Time expenditure measurement of executing an alternating matrix multiplication ReLU sequence of 10 repeats for varying matrix order. The batch size is fixed at 8192. We observe a small speed-up of $1.9(9)\,\%$. While this implies little performance gain by tracing, in reverse it shows, that the type conversions pose negligible overhead.

when using tracing compared to a sequence of single operations.

5.6 Competition — Application on ECG trace classification

A large period of time of this thesis overlapped with a competition, at which the group participated. Its objective was energy efficient classification of atrial fibrillation in electrocardiogram data. Section 5.6.1 gives a more detailed description of the objective. The Author's part in this group project was the development of software abstraction for both the process of training a suitable model and efficient standalone inference of a trained model. In section 5.6.2 we introduce the used model. A performance evaluation is conducted in sections 5.6.3 and 5.6.4.

5.6.1 Objective

The competition is called *Energieeffizientes KI-System* (German for *Energy efficient AI system*) and is organized by the German Federal Ministry of Education and Research [8]. Its objective is promotion of energy efficient solutions to problems solvable by using artificial neural networks. A considered application are edge-devices like wearables. The task chosen is classification of electrocardiogram data for the heart disease atrial fibrillation, which is a form of abnormal irregular heart rhythm [77]. For model development, the group was provided with a data set of 16 000 two-channel time-series recordings of 120 s length each, where half of the samples are sick and the other half is healthy. Evaluation is performed via energy measurements of a test setup, which shall be able to classify unknown test data while finding at least 90 % of sick recordings and wrongly suggesting at most 20 % of healthy recordings to be sick. The proposal by the group is called HdBioAI (*Hardware Demonstration Biologisch Inspirierter Informationsverarbeitungssysteme Optimiert für Analoge Inferenz*, german for *biologically inspired information processing system for analog inference*).

5.6.2 Model

Development of a model is lead by Arne Emmel as part of his Master thesis [22]. The final proposal uses the non-spiking mode of operation of the BrainScaleS-2 hardware with a convolutional artificial neural network in combination with data-reduction preprocessing on the FPGA. The complete model is depicted in fig. 5.19.

5.6.3 Training

Training is performed using PyTorch and the extension developed in section 4.2. We don't use the operations, cf. section 4.2.1, directly, but in form of per-operation layers derived from `torch.nn.Layer` implemented in [22]. They allow specification of a model without direct execution in form of a PyTorch `Model`. Listing 28 shows the model specification. This can then be directly used for training using the forward- and backward-pass of the operations, cf. section 4.2.1. To evaluate the performance during training, we measure the

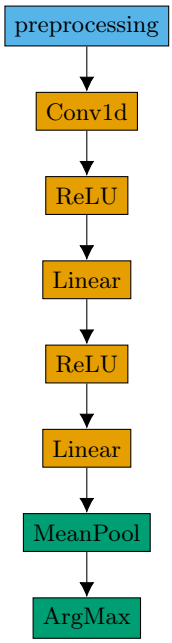


Figure 5.19: Logical description of the developed model. Preprocessing is used to generate input for a one-dimensional convolution for feature detection. Following, two linear layers accompanied by a rectified linear unit are used as classifier. Their result is determined via mean-pooling followed by finding the index with maximal activation to determine the binary healthy or sick result. The preprocessing's implementation is developed by Joscha Ilmberger and is constituted by selecting one of the two channels, running-difference and boundary-pooling followed by element-wise linear scaling. It shall reduce a longer portion of the recording to fewer samples to be fed into the analog neural network core and improve signal-to-noise ratio. The network is designed such that it fits entirely on a single chip instance. This allows complete classification of one recording after another without the need to reconfigure weight matrices. In total, 65 486 multiply-accumulate operations are performed per recording, which is 99.9 % of a full chip instance.

```
class Model:
    def __init__(self, ...):
        self.features = torch.nn.Sequential(
            hxtorch.nn.HDBioAIConv1d(...),
            hxtorch.nn.ConvertingReLU(...),
        )
        self.classifier = torch.nn.Sequential(
            hxtorch.nn.Linear(...),
            hxtorch.nn.ConvertingReLU(...),
            hxtorch.nn.Linear(...),
        )

    def forward(x):
        x = self.features(x)
        x = self.classifier(x)
        x = hxtorch.meanpool(x, ...)
        x = hxtorch.argmax(x, ...)
        return x
```

Listing 28: Model, cf. fig. 5.19, described using the PyTorch extension and Layers. The parameters to the layers are left out for readability. The convolutional layer incorporates the preprocessing, because the latter is only available as fused operation with a analog multiply-accumulate operation. The `ConvertingReLU` is a rectified linear unit with rescaling 7 bit unsigned membrane potentials after a rectified linear unit to 5 bit unsigned input activation values.

runtime during training. Figure 5.20 visualizes the results. We observe a total runtime of

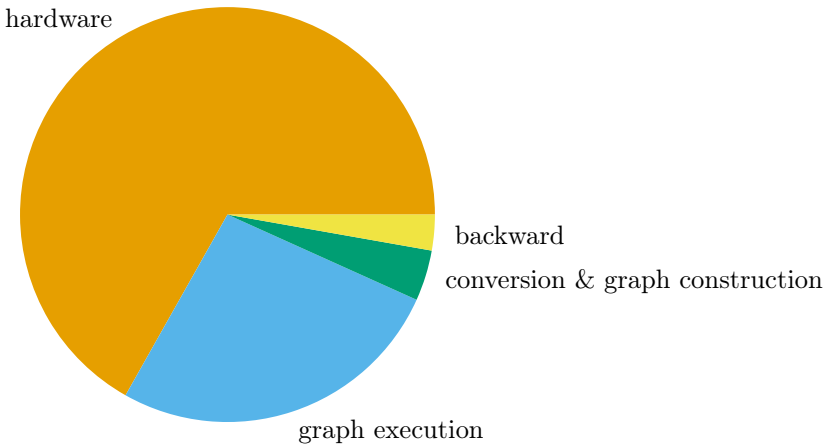


Figure 5.20: Time expenditure distribution during training of the competition model using minibatches of 1000 recordings. The `torch.optim.Adam` optimizer is used in conjunction with `torch.nn.functional.cross_entropy` as loss function. Each recording requires 895(17) µs of time, of which 24.9(32) µs are spent in the backward pass. This is only 2.8 % of total runtime. In the forward pass, 598.0(51) µs are spent with execution on the hardware, 237(10) µs are required around for execution of the graph and conversion and graph construction require 35(20) µs. In total, 66.8 % of time are spent with execution on the hardware.

895(17) µs per recording for training with a minibatch of 1000. 66.8 % of this time are spent with execution on the hardware, the backward pass requires only 2.8 % of the total time. This implies, that training using the PyTorch extension poses negligible overhead. Knowing the number of multiply accumulate operations of 65 486 per recording, this results in a rate of these operations of 73 Mop s^{-1}. It is only a fraction of what is measured for large square weight matrices in fig. 5.11. However, this is expected due to the single operations here being smaller than a full chip (because the whole model fits onto a single chip), which results in increased overhead of the static configuration.

5.6.4 Inference

In contrast to training, the inference shall be performed as standalone as possible, since the whole system's energy consumption is measured. The evaluation protocol of the competition requires separation of load of data from a removable storage medium, classification and write-back of acquired result. The compiler for standalone execution, cf. (second part of) section 4.1.2, has been developed alongside these requirements and is used for this task.

label	time [ms]
initialization	47.4
load	28600
execution	158
store	45.9
postprocess store	2.2

Table 5.2: Time expenditures during inference of the competition model using a batch-size of 500 recordings. Execution is performed using compiled programs from the compiler for standalone execution, cf. (second part of) section 4.1.2. Classification is performed in 158 ms. In the context of the competition's evaluation, this is the only relevant measurement. This is equivalent to 316 µs per recording or 207 $\mathrm{Mop\,s^{-1}}$ when only counting the multiply-accumulate operations. Compared to the training, the total time expenditure of 57.7 ms per recording is highly increased. This is explained in that here, all input data is to be transported to the hardware, whereas in training, the preprocessing is performed on the host computer instead of the FPGA, vastly decreasing the amount of data to be sent to the hardware.

It operates on a single signal-flow graph, which is constructed from the PyTorch-based model using the tracing of operations described in section 4.2.7. As described in fig. 4.10, separation of initialization, input-load, execution and result-store is implemented by this compiler. The model used, cf. section 5.6.2, fits completely on a single chip without the need for reconfiguration due to overlap in the configuration. This additionally allows reordering these parts of the execution, such that a single static initialization is performed, then all input load operations can be grouped, the execution can be optimized for data-locality and merged into one execution and lastly all store operations can be performed, cf. fig. 4.12. The competition requires classification of data in minibatches of 500 recordings. Table 5.2 shows measured runtime for such an inference run. We measure a classification duration of 316 µs per ECG recording, which is equivalent to 207 $\mathrm{Mop\,s^{-1}}$ when counting only multiply-accumulate operations. This is comparable in the order of magnitude to the measurement for a single multiply accumulate operation in fig. 5.13, where 490 $\mathrm{Mop\,s^{-1}}$ are measured. A decrease is expected here due to the surrounding digital operations like ReLU or addition of bias constants and the preprocessing performed on the FPGA. When looking only at the chip (and ignoring the FPGA and its memory) as the primary element performing the classification and estimating its power consumption with being 300 mW [11] constant, each classification requires 95 µJ. The complete setup (chip, FPGA, controlling host computer) used for the competition requires approx. 5 W, leading to 1.58 mJ per classification.

5.7 Software organization

The source code of all developments presented in this thesis is available via Git repositories. It is split into three main repositories, grenade for the signal-flow graph representation, pynn-brainscales for the PyNN front end and hxtorch for the PyTorch extension. As described in section 3.5.2, the group uses code-review prior to application of changes to the production software state. The production state is available publicly via GitHub [17, 21, 20].

At the time of writing, part of the developed software is still in review and therefore only available privately (via `gerrit.bioai.eu`). In grenade, the compiler for standalone execution, the subgraph-insertables and the abstract network notation are in review. For the PyTorch extension in hxtorch, the special operations necessary for the competition are in review. For the PyNN front end, using the signal-flow graph representation as back end is in review (in part because it depends on the abstract network notation).

6 Discussion

In this thesis, a signal-flow graph based representation of experiments on the BrainScaleS-2 neuromorphic hardware is successfully developed and implemented. Two execution models, a just-in-time execution model and standalone execution model is developed. The latter features separation of compilation and execution and uses the embedded general purpose processors as experiment controller. To increase abstraction, generators for subgraphs are used for representation of reusable reoccurring parts of experiments. Especially spiking experiments are shown to benefit from abstract specification using populations and projections. Given such an abstraction, the process of mapping and routing is inserted as a black box algorithm resulting in a signal-flow graph hardware representation.

Initially targeting the non-spiking mode of operation, a PyTorch extension is developed providing hardware support at the level of single independent operations. The formerly developed signal-flow graph representation and just-in-time execution are used to implement its back end. Serialization of sequential execution of operations is implemented. This allows combined execution and export for deployment, e.g. for standalone execution. Rounding up, prerequisites for integration of spiking experiments into the PyTorch extension are investigated.

For spiking experiments, a back end to PyNN using the abstract population and projection-based network description is developed and implemented and choices concerning the PyNN front end for BrainScaleS-2 are described. The back end leads to a one-to-one relation between PyNN populations and projections and abstract hardware network description populations and projections.

For evaluation of the developed software, performance measurements are being focused. Program runtime (or its inverse as rate of operations) is used as primary metric to investigate introduced overhead. A combination of artificial benchmarks and real-world experiments is used to highlight different aspects of the developments.

First, a baseline is established by characterization of the already existing software to control the hardware. The transport speed via the $1\,\mathrm{Gbit\,s^{-1}}$ Ethernet connection between host computer and FPGA is shown to be the limiting element. However, encoding and decoding of configuration on the host computer is within one order of magnitude and therefore contributes significantly to the achieved information transfer. This can be explained by that encoding and decoding contains multiple stages of (often unaligned) bit-formatting.

Additionally, a software mock of the hardware for non-spiking experiments is implemented

to allow for exploration of full-stack performance for hardware execution speed alterations to existing physical prototypes. A peak performance of 8 ns per executed instruction is reached, which is equivalent to $8\,\mathrm{Gbit\,s^{-1}}$ wire speed or 125 MHz and coincides with the maximal transport bandwidth of the current chip version.

For the signal-flow graph representation, expected build-up time and peak memory consumption is verified. The just-in-time execution is evaluated for both spiking and non-spiking experiments. The achieved maximal spike loop-back rate is decreased by a factor of approximately two compared to a baseline measurement. However, also the baseline measurement does not coincide with the expectation and requires further investigation. Non-spiking analog matrix multiplications reach a rate of $710\,\mathrm{Mop\,s^{-1}}$ using the just-in-time execution for a single physical hardware setup. Comparing to the theoretical data transport limit constructed in section 3.1.4 of $4\,\mathrm{Gop\,s^{-1}}$, the achieved rate is approximately a factor of 5.6 smaller. Together with the high hardware utilization this suggests, that the majority of time is spent on the actual operation instead of data transport. With concurrent execution of simulated mock setups with $8\,\mathrm{Gbit\,s^{-1}}$ wire speed and a host-computer hardware concurrency of 48, a peak operation rate of $23.7\,\mathrm{Gop\,s^{-1}}$ is achieved. This demonstrates scalability for upcoming multi-chip setup developments. Comparing to the theoretical data transport limit constructed in section 3.1.4 of $4\,\mathrm{Gop\,s^{-1}}$ for a single simulated setup connected via $1\,\mathrm{Gbit\,s^{-1}}$, the achieved rate of $2.8\,\mathrm{Gop\,s^{-1}}$ is slightly decreased. For the simulated $8\,\mathrm{Gbit\,s^{-1}}$ connection, the difference of the theoretical limit of $32\,\mathrm{Gop\,s^{-1}}$ and the achieved rate of $11\,\mathrm{Gop\,s^{-1}}$ for a single setup grows. This suggests, that at these rates, overhead from en- and decoding and compilation of playback sequences from the graph becomes dominant. Compilation for standalone execution only supports non-spiking experiments with the current hardware. A peak performance of $490\,\mathrm{Mop\,s^{-1}}$ is achieved for matrix multiplication fully controlled by the on-chip general purpose processors. Like the signal-flow graph hardware representation, the abstraction network construction is verified against runtime and memory consumption expectations.

For evaluation of the PyNN front end, a formerly developed experiment implementing a soft winner-take-all spiking neural network is used [13]. A decrease in runtime by 28 % is observed for replacing the former Python-only routing implementation using lower-level configuration and control software directly by the developed abstract network description. This is explained by less Python to `C++` transitions and the possibility for more efficient compilation of the `C++`-based implementation of the mapping and routing algorithm. This advantage is expected to grow further for multiple interconnected chips.

For the PyTorch front end, the overhead introduced by data conversions between PyTorch tensors and hardware value types is evaluated. No significant overhead is observed. When removing the overhead by serialization and fused execution of a sequence of operations, a decrease in runtime of less than 3 % is observed. This shows, that the PyTorch extension

yields an efficient adapter to the BrainScaleS-2 hardware.

Finally, the developed techniques are applied for the non-spiking mode of operation, namely the PyTorch front end, serialization and standalone execution via compilation to a problem requiring the classification of electrocardiogram recordings energy efficiently. The used model, developed in [22], fits entirely on a single chip without need for reconfiguration. Training the model with the PyTorch extension shows, that the majority of time is spent on execution on the hardware (including data transfer), the implemented backward pass for gradient calculation only requires 2.8 % of the runtime. $73\,\mathrm{Mop\,s^{-1}}$ are reached during training, when counting only multiply-accumulate operations. For inference, standalone execution with separated compilation is used. Here, its true potential is visualized by reaching $207\,\mathrm{Mop\,s^{-1}}$, which is 2.8 times the training's speed. Moreover, energy consumption is highly reduced compared to including a standard host computer during the experiment for intermediate data handling and digital operations. When taking into account only the chip's expenditures, 95 µJ per classification are achieved. With the complete setup, 1.58 mJ are achieved. Taking into account a recording's length of 120 s, the complete setup could be operated continuously for approximately 5.5 yr on a `CR2032` button cell (assuming 3 V nominal voltage and 200 mAh capacity).

To put the achieved performance for the non-spiking mode of operation into perspective, it is compared against the performance of a digital accelerator, Google's Edge-TPU [45]. Its operations are deemed comparable in that they are 8 bit fixed-point integer operations. The peak performance (of the m.2 package) is specified with $4\,\mathrm{Top\,s^{-1}}$ and $2\,\mathrm{Top\,W^{-1}}$. When comparing this performance to BrainScaleS-2 (we use performance per power of the chip only in order to accommodate different power envelopes), achieved performance ranges around four orders of magnitude lower with a factor of 3.5×10^{-4}. While the exact order of magnitude is debatable due to lack of comparing measurements using the same problem or model, the general statement remains. As discussed above, the majority of time is spent on the actual execution on the hardware. However, comparing to the theoretical limit set via data transport with a rate of $8\,\mathrm{Gbit\,s^{-1}}$, calculated in section 3.1.4 to be $32\,\mathrm{Gop\,s^{-1}}$, and its simulated measurement of $11\,\mathrm{Gop\,s^{-1}}$, it becomes evident, that architectural (hardware and following software) changes are required for reducing the remaining discrepancy. In chapter 7, propositions are made following this direction.

Understanding of newly developed software often benefits from comparison to existing approaches targeting similar problems for working out strengths and weaknesses of the new approach and ease its accessibility to a wider audience. Therefore, we compare the developed abstraction for BrainScaleS-2 to the existing operating system for the predecessor hardware BrainScaleS-1, described in detail in [51]. For the arguments to be made, performing a vast reduction, the hardware can be seen as being comparable except multiple chips allowing interconnections and the lack of embedded general purpose processors and a non-spiking

mode of operation. Coming from the user-side, PyNN is available as front end similarly. In contrast to the implementation performed here, the PyNN API is made available directly via Python wrapping of `C++` implementations instead of the API being formulated in Python and using a `C++`-based back end. The new approach eases alignment to and reuse of the upstream API and parts of its implementation. For BrainScaleS-1, a complete calibration (based on database-lookup and parameter interpolation) exists, allowing specification of parameters via natural units, together with facilities to mask part of the resources as unavailable due to e.g. malfunctioning or non-matching parameter ranges. While this does not yet exist for BrainScaleS-2, its integration is anticipated. The mapping and routing on BrainScaleS-1 requires different (stronger) constraints to be taken into account than on BrainScaleS-2 currently. However, its front end, called `Biograph`, is similar to the now developed abstract network representation (except featuring natural units as explained before). The largest difference is in the mapping and routing result. In contrast to the newly developed unified signal-flow graph representation of the hardware configuration, it is composed of multiple partial graph representations and lower-level configuration container and coordinate pairs directly. Therefore, signal-flow is known only implicitly and dependencies can only be retrieved by going through parts of the mapping again.

7 Outlook

The developed software abstraction forms a basis for further features and optimization. In the following, the embedding of such enhancements is discussed keeping in mind current and future hardware architectures.

First, solutions to improve the non-spiking performance are proposed. As observed, the time spent on the actual computation is dominant in the current prototype. Ways to mitigate this are already discussed in detail in [74]. Here, the gap still present through the software and data transport is focused. The data transport is almost exclusively defined via the lower-level software layers and their container en- and decoding. When revisiting eq. (3.1) and table 3.1, the obvious point of optimization is the efficiency of packing of data being transported. Currently, a packing-efficiency of 10 % to the chip and 50 % from the chip is achieved. When both ways are optimal (i.e. $d_{\text{in}} = 5\,\text{bit}$, $d_{\text{out}} = 8\,\text{bit}$, $d_{\text{trigger}} = 0\,\text{bit}$), the achievable performance improvement is a factor of four compared to the current implementation (then limited by transport of membrane traces from the chip). This would surpass the current hardware design goal of one MAC operation per micro second (which would result in $64\,\text{Gop}\,\text{s}^{-1}$ with signed weights). Since the achievable rate is not reached for the simulation of $1\,\text{Gbit}\,\text{s}^{-1}$ and even more of $8\,\text{Gbit}\,\text{s}^{-1}$ transfer rate, the software overhead for the types to be transported is to be optimized. Section 5.2 shows, that the en- and decoding in the lower-level software layers contributes significantly. It is expected, that more optimal packing of data results in improved software performance as well. Therefore, optimization of the data packing in conjunction with in-detail measurement of individual contributions of the software layers specifically for the used data types is proposed. Equation (3.1) suggests another point of optimization, namely the height and width of the synapse matrix. When for example both are doubled, the rate of operations (limited by data transport) doubles. However, implications for the power requirement and analog behavior as well as digital line length remain to be investigated.

The performance of spiking experiments, namely loop back with high rate was observed to be significantly lower including software overhead than the rate during the realtime execution solely on the hardware. To investigate the cause of this lack of performance, individual throughput measurements at the different software layers are proposed. In addition to physical hardware, the developed software simulation could be used as sink for spikes of configurable performance and could be enhanced to provide spike loop back. Using this, the spike throughput performance of the software could be evaluated also taking into account

bandwidth alterations of future hardware developments.

The current execution model for realtime time evolution is limited to a single chip. In the future, multi-chip setups are planned, which shall support exchange of event data during realtime operation. Therefore, synchronization between the chips is required. Additionally, the signal-flow between chips will be configurable. The signal-flow graph already supports multiple independent realtime executions distributed over a collection of physical chips. The required alteration for support of synchronized chips is straight-forward. Instead of realtime executions belonging to a single physical chip, a collection is to be linked together. All coordinates referring to entities on that realtime execution gain chip selection as dimension. A realtime execution in this new graph therefore is simply executed on a larger setup consisting of multiple chips and realtime executions are allowed to contain a heterogeneous number of interconnected chips. Figure 7.1 visualizes this concept. It is to be noted, that

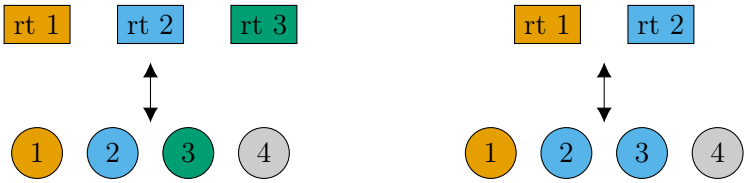


Figure 7.1: Realtime executions for multi-chip support (right) vs. current implementation supporting individual single chips (left). Realtime executions are identified via color and displayed on top, the single chips are displayed at the bottom. Left: Each realtime execution is identified with a single chip. Right: Multiple chips are identified for the same realtime execution, leading to synchronization.

the actual implementation of synchronization remains undefined, as it is subject to support by the FPGA(s) and alterations in low-level software are expected. Spiking experiments on multiple synchronized chips with inter-chip realtime communication requires routing also for these inter-chip connections. The mapping and routing therefore is to be extended to allow placement onto multiple chips under (yet undefined) inter-chip connectivity constraints. It is expected to become more similar to the current BrainScaleS-1 mapping and routing, where multiple chips are already supported (with possibly different constraints).

For spiking experiments in PyTorch, only prerequisites are investigated in this thesis. In contrast to classical operations, spiking networks allow recurrence intrinsically. Therefore, eager execution is not feasible and some form of separation of network construction and execution is required. A builder-pattern like present in PyNN is proposed, allowing construction of a network from populations (or layers) and projections. The benefit of PyTorch is, that individual parts of the network can be accompanied by custom backward models, which promises seamless integration for training. It remains to be evaluated, whether integration into an existing library for spiking neural networks in PyTorch (however without support for similar hardware) like BindsNET [35] is feasible and beneficial.

The realtime execution implemented currently is basic in that it only allows temporal placement of spike events. Timed query of state variables, e.g. via ADC-based readout is another application, where temporal placement is required. Such temporal placement can be realized by (virtual) trigger signals. These signals can then naturally be integrated into the existing signal-flow graph. Trigger sources become vertices. In contrast to other input data, generator-based trigger signals, e.g. a periodic signal, will be more common, requiring thought about their interface. Additionally, trigger targets might be highly experiment-specific. Therefore, integration of a plugin system would allow experiment-specific hook-in of timed operations into the signal-flow graph. The challenge there is development of a generalized interface to retain correctness of the signal-flow, for example such a generalization requires hand-over of callables for post-processing of recorded data.

Closely linked to the arbitrary time evolution described above is integration of plasticity. Currently, plasticity, i.e. alteration of network parameters like connectivity or neuron parameters, is only possible as outer loop. For example training with the PyTorch extension resembles such an outer loop in that alterations are performed outside the hardware execution. However, the BrainScaleS-2 platform is geared towards enabling local learning via its embedded general purpose processors. Plasticity algorithms are highly experiment specific. Integration as plugin system for flexibility is suggested. This needs support for just-in-time compilation or at least linkage of programs for the plasticity processors. Furthermore, precise timing is required, which can be supplied by trigger signal sources as proposed above. For integration of representing plasticity as part of the signal-flow graph, two propositions are envisioned. On the one hand, it could be integrated as annotation at existing vertices. For example an algorithm to alter synaptic weights could be supplied by an annotation at a synapse matrix vertex. On the other hand, the algorithm's signal-flow could be integrated as well into the graph. For example an algorithm requiring neuron membrane potentials and altering synaptic weights could lead to a subgraph like visualized in fig. 7.2.

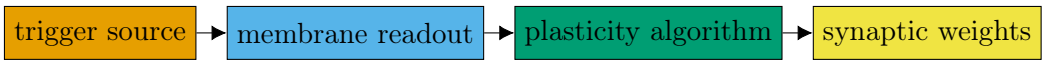


Figure 7.2: Signal-flow graph representation of an exemplary plasticity algorithm. The algorithm is data-driven. A triggered read-out is fed into the algorithm, which as result of its computation leads to alteration of (part of) the synapse matrix.

Picking up the idea of just-in-time compilation from integration of plasticity above, this concept is useful in general. Currently, the compilation for standalone execution, cf. second part of section 4.1.2, implements parameterized execution on the embedded processors via an interpreter approach with dispatch at runtime. On the one hand this limits the achievable performance for tight loop in comparison to compiled control flow and inhibits optimization involving multiple operations. On the other hand, it requires all possible operations to be

present in the program to allow runtime selection. Both problems can be solved by integration of just-in-time compilation of the control flow extracted from the signal-flow graph. For this integration, two tasks exist: First, the PPU cross-compiler is to be integrated into the graph compiler and temporarily created programs are to be loaded during the experiment. Completing, generation of source code from the control flow and operations extracted from the graph is to be solved. Once control flow deviates from sequential operations, this becomes algorithmically challenging. Additionally, user-definable operations are desired, e.g. for injection of custom plasticity algorithms. For this, the interface is to be defined. As first approach, direct supplication of source code is suggested, for which however also the interface is to be defined in order to be compatible to its surrounding, i.e. input and output data flow and hardware observables and parameter access. Furthermore, this allows injection of multi-level optimization along the same lines as XLA [34], MLIR [44] or GLOW [59]. An example for such an optimization, which is infeasible with the interpreter approach, is changing the iteration order of element-wise operations for improved locality via fusing.

To the other end of interfaces, the PyTorch extension hxtorch would greatly benefit from further integration beyond single operations. Access to the compute-graph, also already during training, would allow multi-operation optimization on a signal-flow graph hardware representation containing multiple high-level operations. Additionally, such back-to-back integrated representation allows for improved data locality. In the standard operation mode of PyTorch, operations are invoked eagerly without build-up of an underlying graph representation to access. However, PyTorch features support for tracing operations via their JIT intermediate representation [23]. This tracing is expressed via annotations in Python. It results in a graph of operations, which is accompanied by a framework allowing supplication of rules for optimization, e.g. replacement of sequences by fused operations. Operations residing in PyTorch extensions like hxtorch are easily addable into this framework by dynamic registration. It is therefore proposed to investigate an adaptation of the hardware signal-flow graph representation to this compute graph directly to gain intrinsic knowledge of the inter-dependencies of operations.

Finally, the developed concepts are implemented specifically targeting BrainScaleS-2. However, especially the signal-flow graph representation and handling of connectivity constraints is expected to be applicable more widely. For example, future hardware will change in that a different set of vertex and edge types becomes necessary, but the surrounding is expected to remain comparable. Similarly, the execution and compilation will most certainly change both in the back end and the execution model. However, parts like extraction of static configuration could be made reusable. Therefore, it is proposed to investigate generalizability of parts of the developed framework. It is expected, that the current implementation benefits from such effort as well and might lead to simplification of integration of above-proposed plugin systems for timed placement and plasticity algorithms.

8 Acknowledgments (Danksagungen)

Ich möchte danken:

- Johannes Schemmel für die Betreuung meiner Arbeit.
- Meinen Eltern für ein sicheres Zuhause und ihre Gesellschaft in einer einsamen Zeit.
- Eric für seinen Wissensschatz, die sehr gute Zusammenarbeit beim Paper-Schreiben, unzählige Diskussionen und alles außenrum.
- Christian für die vielen Diskussionen und dafür, ein gutes Tischgegenüber zu sein.
- Yannik dafür, immer wieder einen neuen Blickwinkel in Diskussionen zu beleuchten und für die professionelle Leitung des Wettbewerbsteams.
- Johannes Weis für die Diskussionen über den nicht-spikenden Modus, Kalibration und die Hardware sowie hxtorch.
- Arne für die Diskussionen um hxtorch.
- Milena für die gute Zusammenarbeit bei PyNN.
- Vitali für seinen FPGA Support und ein immer offenes Ohr für Wünsche.
- Joscha für die Diskussionen um den Vektor-Generator und den erstklassigen FPGA Support für den Wettbewerb.
- Dem Wettbewerbsteam für die gute Zusammenarbeit, und die täglichen Morgenrunden, die einem trotz räumlicher Ferne das Gefühl des Miteinander erhalten haben.
- Christian, Eric und Yannik für das gründliche Korrekturlesen dieser Arbeit.
- Christian, Oliver, Joscha und Hartmut für die Pandemie-Spielabende.
- Der Electronic Vision(s) Gruppe für eine tolle (Arbeits-)Atmosphäre und dem Container für das regelmäßige gute Grillen (als das noch möglich war), der Versorgung mit reichlich Mandarinen und das freundschaftliche Miteinander.

The work carried out in this Master's Thesis used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 785907 and 945539 (Human Brain Project, HBP) for SGA2 and SGA3 funding, from the BMBF (16ES1127), and from the Lautenschläger-Forschungspreis 2018 for Karlheinz Meier.

9 References

[1] S. A. Aamir, Y. Stradmann, P. Müller, C. Pehle, A. Hartel, A. Grübl, J. Schemmel, and K. Meier. "An Accelerated LIF Neuronal Network Array for a Large-Scale Mixed-Signal Neuromorphic Architecture". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12 (2018), pp. 4299–4312. ISSN: 1549-8328. DOI: 10.1109/TCSI.2018.2840718.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[3] Andrew W. Appel. "SSA is Functional Programming". In: *SIGPLAN Not.* 33.4 (1998), pp. 17–20. ISSN: 0362-1340. DOI: 10.1145/278283.278285.

[4] Valentina Armenise. "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery". In: *Proceedings of the Third International Workshop on Release Engineering*. RELENG '15. Florence, Italy: IEEE Press, 2015, pp. 24–27. DOI: 10.1109/RELENG.2015.19.

[5] Jørgen Bang-Jensen and Gregory Z. Gutin, eds. *Classes of Directed Graphs*. SpringerLink : Bücher. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-71840-8. DOI: 10.1007/978-3-319-71840-8.

[6] S. Billaudelle, Y. Stradmann, K. Schreiber, B. Cramer, A. Baumbach, D. Dold, J. Göltz, A. F. Kungl, T. C. Wunderlich, A. Hartel, E. Müller, O. Breitwieser, C. Mauch, M. Kleider, A. Grübl, D. Stöckel, C. Pehle, A. Heimbrecht, P. Spilger, G. Kiene, V. Karasenko, W. Senn, M. A. Petrovici, J. Schemmel, and K. Meier. "Versatile Emulation of Spiking Neural Networks on an Accelerated Neuromorphic Substrate". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180741.

[7] Boost.Graph. *Version 1.74.0 Website*. http://www.boost.org/doc/libs/1_74_0/libs/graph. 2020.

[8] Bundesministerium für Bildung und Forschung (BMBF). *Bekanntmachung: Richtlinie zur Förderung des Pilotinnovationswettbewerbs "Energieeffizientes KI-System"*. German. https://www.bmbf.de/foerderungen/bekanntmachung-2371.html. 2021.

[9] Clang.Format. *Version 12 Website*. https://clang.llvm.org/docs/ClangFormat.html. 2020.

[10] Clang.Tidy. *Version 12 Website*. https://clang.llvm.org/extra/clang-tidy/. 2020.

[11] Benjamin Cramer, Sebastian Billaudelle, Simeon Kanya, Aron Leibfried, Andreas Grübl, Vitali Karasenko, Christian Pehle, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Johannes Schemmel, and Friedemann Zenke. "Training spiking multi-layer networks with surrogate gradients on an analog neuromorphic substrate". In: *arXiv preprint* (2020). arXiv: 2006.07239 [cs.NE].

[12] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*. 2018. arXiv: 1801.08058 [cs.DC].

[13] Milena Czierlinski. "PyNN for BrainScaleS-2". Bachelorarbeit. Universität Heidelberg, 2020.

[14] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[15] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11 (2009). DOI: 3389/neuro.11.011.2008.

[16] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. Ortega Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinstry, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha. "TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years". In: *Computer* 52.5 (2019), pp. 20–29. DOI: 10.1109/MC.2019.2903009.

[17] Electronic Visions(s), Heidelberg University. *GRaph-based Experiment Notation And Data-flow Execution*. URL: https://github.com/electronicvisions/grenade.

[18] Electronic Visions(s), Heidelberg University. *halco*. URL: https://github.com/electronicvisions/halco.

[19] Electronic Visions(s), Heidelberg University. *hwdb*. URL: https://github.com/electronicvisions/hwdb.

[20] Electronic Visions(s), Heidelberg University. *hxtorch: PyTorch for BrainScaleS-2*. URL: https://github.com/electronicvisions/hxtorch.

[21] Electronic Visions(s), Heidelberg University. *PyNN for BrainScaleS-2*. URL: https://github.com/electronicvisions/pynn-brainscales.

[22] Arne Emmel. "Inference with Convolutional Neural Networks on Analog Neuromorphic Hardware". Master's Thesis. Universität Heidelberg, 2020.

[23] Facebook, Inc. *PyTorch JIT Overview*. URL: https://github.com/pytorch/pytorch/blob/master/torch/csrc/jit/OVERVIEW.md.

[24] Facebook, Inc. *PyTorch Sparse Documentation*. URL: https://pytorch.org/docs/stable/sparse.html.

[25] Simon Friedmann. "A New Approach to Learning in Neuromorphic Hardware". PhD thesis. Ruprecht-Karls-Universität Heidelberg, 2013. DOI: 10.11588/heidok.00015359.

[26] Simon Friedmann. *Omnibus On-Chip Bus*. forked from https://github.com/five-elephants/omnibus. 2015. URL: https://github.com/electronicvisions/omnibus.

[27] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *Software: Practice and Experience* 30.11 (2000), pp. 1203–1233. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

[28] *Gerrit Code Review*. https://www.gerritcodereview.com/. accessed March 11, 2020. 2020.

[29] Wulrfram Gerstner and Romain Brette. "Adaptive exponential integrate-and-fire model". In: *Scholarpedia* 4.6 (2009), p. 8427. DOI: 10.4249/scholarpedia.8427.

[30] gettimeofday. *Linux Programmer's Manual*. https://man7.org/linux/man-pages/man2/gettimeofday.2.html. 2019.

[31] M. Gewaltig and M. Diesmann. "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2.4 (2007). revision #130182, p. 1430. DOI: 10.4249/scholarpedia.1430.

[32] N. H. Goddard, M. Hucka, F. Howell, H. Cornelis, K. Shankar, and D. Beeman. "Towards NeuroML: model description methods for collaborative modelling in neuroscience." In: *Philos Trans R Soc Lond B Biol Sci* 356.1412 (2001), pp. 1209–28. DOI: 10.1098/rstb.2001.0910.

[33] Google. *Googletest Github repository*. https://github.com/google/googletest. 2020.

[34] Google Brain Team. *XLA Architecture*. URL: https://www.tensorflow.org/xla.

[35] Hananel Hazan, Daniel J. Saunders, Hassaan Khan, Devdhar Patel, Darpan T. Sanghavi, Hava T. Siegelmann, and Robert Kozma. "BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python". In: *Frontiers in Neuroinformatics* 12 (2018), p. 89. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00089.

[36] Horace He. "The State of Machine Learning Frameworks in 2019". In: *The Gradient* (2019). URL: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry.

[37] Stanford University Human-Centered Artificial Intelligence. *The AI Index Report 2019*. https://hai.stanford.edu/research/ai-index-2019. 2020.

[38] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. 2019. URL: https://github.com/pybind/pybind11.

[39] Sebastian Jeltsch. "A Scalable Workflow for a Configurable Neuromorphic Platform". PhD thesis. Universität Heidelberg, 2014. DOI: 10.11588/heidok.00017190.

[40] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17.

Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246.

[41] Vitali Karasenko. "Von Neumann bottlenecks in non-von Neumann computing architectures". PhD thesis. Ruprecht-Karls-Universität Heidelberg, 2020. DOI: 10.11588/heidok.00028691.

[42] Johann Klähn. *genpybind software v0.2.0*. 2020. DOI: 10.5281/zenodo.372674. URL: https://github.com/kljohann/genpybind.

[43] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (2017), pp. 1–20. DOI: 10.1371/journal.pone.0177459.

[44] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[45] Google LLC. *Coral M.2 Accelerator datasheet Version 1.5*. 2020. URL: https://coral.ai/docs/m2/datasheet/.

[46] Daniel Marjamäki. *Cppcheck Website*. http://cppcheck.sourceforge.net/. 2020.

[47] S. J. Mason. "Feedback Theory-Some Properties of Signal Flow Graphs". In: *Proceedings of the IRE* 41.9 (1953), pp. 1144–1156. DOI: 10.1109/JRPROC.1953.274449.

[48] C. A. Mead. "Neuromorphic Electronic Systems". In: *Proceedings of the IEEE* 78 (1990), pp. 1629–1636. DOI: 10.1109/5.58356.

[49] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[50] Eric Müller, Christian Mauch, Philipp Spilger, Oliver Julien Breitwieser, Johann Klähn, David Stöckel, Timo Wunderlich, and Johannes Schemmel. "Extending BrainScaleS OS for BrainScaleS-2". In: *arXiv preprint* (2020). arXiv: 2003.13750 [cs.NE].

[51] Eric Müller, Sebastian Schmitt, Christian Mauch, Hartmut Schmidt, José Montes, Joscha Ilmberger, Johann Klähn, Felix Passenberg, Christoph Koke, Mitja Kleider, Sebastian Jeltsch, Maurice Güttler, Dan Husmann, Sebastian Billaudelle, Paul Müller, Andreas Grübl, Jakob Kaiser, Jonas Weidner, Bernhard Vogginger, Johannes Partzsch, Christian Mayr, and Johannes Schemmel. "The Operating System of the Neuromorphic BrainScaleS-1 System". In: *arXiv preprint* (2020). arXiv: 2003.13749 [cs.NE].

[52] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[53] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[54] Christian Pehle and Jens Egholm Pedersen. *Norse - A deep learning library for spiking neural networks*. Version 0.0.5. 2021. DOI: 10.5281/zenodo.4422025.

[55] *A Python package for simulator-independent specification of neuronal network models*. [Online; accessed: 2014-04-29]. The NeuralEnsemble Initiative. 2014. URL: http://www.neuralensemble.org/PyNN.

[56] Python. *Unittest Documentation Website*. https://docs.python.org/3/library/unittest.html. 2020.

[57] Marco Rettig. "Characterizing the Event Interface of the HICANN-X". Bachelorarbeit. Universität Heidelberg, 2019.

[58] Johann C. Rocholl, Florent Xicluna, and Ian Lee. *Pycodestyle Website*. https://pycodestyle.pycqa.org/en/latest/. 2020.

[59] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. *Glow: Graph Lowering Compiler Techniques for Neural Networks*. 2019. arXiv: 1805.00907 [cs.PL].

[60] Bodo Rueckauer, Connor Bybee, Ralf Goettsche, Yashwardhan Singh, Joyesh Mishra, and Andreas Wild. *NxTF: An API and Compiler for Deep Spiking Neural Networks on Intel Loihi*. 2021. arXiv: 2101.04261 [cs.ET].

[61] Grigory Sapunov. *Hardware for Deep Learning. Part 4: ASIC*. https://blog.inten.to/hardware-for-deep-learning-part-4-asic-96a542fe6a81. 2021.

[62] J. Schemmel, D. Bruderle, K. Meier, and B. Ostendorf. "Modeling Synaptic Plasticity within Networks of Highly Accelerated I F Neurons". In: *2007 IEEE International Symposium on Circuits and Systems*. 2007, pp. 3367–3370. DOI: 10.1109/ISCAS.2007.378289.

[63] Johannes Schemmel, Sebastian Billaudelle, Philipp Dauer, and Johannes Weis. "Accelerated Analog Neuromorphic Computing". In: *arXiv preprint* (2020). arXiv: 2003.11996 [cs.NE].

[64] Sebastian Schmitt, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Güttler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, Vitali Karasenko, Mitja Kleider, Christoph Koke, Alexander Kononov, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Mihai A. Petrovici, Bernhard Vogginger, Stefan Schiefer, Stefan Scholze, Vasilis Thanasoulis, Johannes Schemmel, Robert Legenstein, Wolfgang Maass, Christian Mayr, and Karlheinz Meier. "Classification With Deep Neural Networks on an Accelerated Analog Neuromorphic System". In: *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks* (2017). DOI: 10.1109/IJCNN.2017.7966125.

[65] Philipp Spilger. "Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip". Bachelorarbeit. Universität Heidelberg, 2018.

[66] Philipp Spilger, Eric Müller, Arne Emmel, Aron Leibfried, Christian Mauch, Christian Pehle, Johannes Weis, Oliver Breitwieser, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, and Johannes Schemmel. "hxtorch: PyTorch for BrainScaleS-2". In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Ed. by Joao Gama, Sepideh Pashami, Albert Bifet, Moamar Sayed-Mouchawe, Holger Fröning, Franz Pernkopf, Gregor Schiele, and Michaela Blott. Cham: Springer International Publishing, 2020, pp. 189–200. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_14.

[67] Marcel Stimberg, Romain Brette, and Dan FM Goodman. "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8 (2019). Ed. by Frances K Skinner, e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314.

[68] Yannik Stradmann. "Verification and Commissioning of Mixed-Signal Neuromorphic Substrates". Master's Thesis. Ruprecht-Karls-Universität Heidelberg, 2019.

[69] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355.

[70] *The GNU Compiler Collection*. Website. Free Software Foundation Inc. 59 Temple Place Boston MA, USA. 59 Temple Place, Boston, MA, USA. URL: http://gcc.gnu.org/.

[71] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. "Trends in Data Locality Abstractions for HPC Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020. DOI: 10.1109/TPDS.2017.2703149.

[72] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (2011), pp. 22–30. ISSN: 1558-366X. DOI: 10.1109/MCSE.2011.37.

[73] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. 2019. ISBN: 978-1-4842-4397-8. DOI: 10.1007/978-1-4842-4398-5.

[74] Johannes Weis. "Inference with Artificial Neural Networks on Neuromorphic Hardware". Master's thesis. Universität Heidelberg, 2020.

[75] Johannes Weis, Philipp Spilger, Sebastian Billaudelle, Yannik Stradmann, Arne Emmel, Eric Müller, Oliver Breitwieser, Andreas Grübl, Joscha Ilmberger, Vitali Karasenko, Mitja Kleider, Christian Mauch, Korbinian Schreiber, and Johannes Schemmel. "Inference with Artificial Neural Networks on Analog Neuromorphic Hardware". In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Ed. by Joao Gama, Sepideh Pashami, Albert Bifet, Moamar Sayed-Mouchawe, Holger Fröning, Franz Pernkopf, Gregor Schiele, and Michaela Blott. Cham: Springer International Publishing, 2020, pp. 201–212. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_15.

[76] Timo Wunderlich, Akos F. Kungl, Eric Müller, Andreas Hartel, Yannik Stradmann, Syed Ahmed Aamir, Andreas Grübl, Arthur Heimbrecht, Korbinian Schreiber, David Stöckel, Christian Pehle, Sebastian Billaudelle, Gerd Kiene, Christian Mauch, Johannes Schemmel, Karlheinz Meier, and Mihai A. Petrovici. "Demonstrating Advantages of Neuromorphic Computation: A Pilot Study". In: *Frontiers in Neuroscience* 13 (2019), p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260.

[77] M. Zoni-Berisso, Fabrizio Lercari, Tiziana Carazza, and Stefano Domenicucci. "Epidemiology of atrial fibrillation: European perspective". In: *Clinical Epidemiology* 6 (2014), pp. 213–220. DOI: 10.2147/CLEP.S47385.

A Experiment environment

The experiments conducted require both software and hardware which is not yet integrated into the production state. For reproducibility, the used state is collected here. We use git hashes in combination with Gerrit change set numbers to identify the state in the used repositories.

The group uses singularity containers [43] for tracking of third-party software. The container used for all experiments is described in table A.1.

key	value
path	`/containers/stable/2020-12-15_2.img`
fingerprint	`842c6ee6-67dc-4a6c-96ba-577ed9feccb0`
app	dls

Table A.1: Singularity container used for all experiments.

Experiments are performed using two different host computer setups. One is chosen for its single-thread performance, while the other features increased hardware concurrency. Their specification is collected in table A.2.

property	host computer A	host computer B
CPU name	AMD Ryzen 3800X	AMD Epyc 7402p
hardware concurrency	16	48
RAM	64 GB	256 GB

Table A.2: Host computers used for experiments. Host computer A features good single-thread performance, while host computer B has a higher hardware concurrency.

In the following, the software- and FPGA-state used in experiments is described. Table A.3 shows the software state for the experiments conducted in sections 5.1 and 5.2. Table A.4 shows the software state for the experiments conducted in sections 5.3, 5.5 and 5.6 except section 5.3.4. Table A.5 shows the software state for the experiments conducted in sections 5.3.4 and 5.4. Table A.6 shows the FPGA state for the experiments conducted in sections 5.1, 5.2, 5.3.4 and 5.4. Table A.7 shows the FPGA state for the experiments conducted in sections 5.3, 5.5 and 5.6 except section 5.3.4.

repository	git hash	change set
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	
fisch	eb3f00d5f2870f15c4eda68a0817500356aae76b	13504
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	
halco	c4f6222499fb994e2ac10892959b23badfc202ad	
haldls	7637909796fc2fa5472db46358a35e42d801cea8	9792
hate	98595229dce410622c4d56d6ef86d2c306e74a03	13053
hwdb	7787934c0ee334393641179d4b1c4d7e5215988d	
hxcomm	62bb1c2586205d28d6e573bbcf6accd3525b10cd	10935
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	
lib-rcf	5b16326ae30ee08a322a6569887ca8bd2684c252	
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	
sctrltp	b5f825007b842f44f3e6401f00cf93387e5e3f3c	
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	

Table A.3: Software state used for measurements conducted in sections 5.1 and 5.2.

repository	git hash	change set
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	
fisch	ab9f98ad2dbd2386aa92fd79507e7fd31a46f348	13186
flange	2fb312fb4fc31634d3dbf74243c13a566b79810f	
grenade	0820e67ee9d5a53581772657fd23673684b22796	13631
grenade (left of fig. 5.13)	7621e4203f85ea91111a1484fb0e2ee3aee3fa04	13546
halco	54cfa14fc63dafce0d04d8f035bd08261e81231d	13435
haldls	47b9020dffed8e349c6dc06d1e0ddc4cfe111920	13185
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	
hwdb	13fadc068fc7bbaabef7ed678ef5237b423f00be	
hxcomm	ee9e4b76b23ac58ef85cc2d675a97ecff9bdccb5	12371
hxtorch	cf1a89656ef228570b5891f8471601dec45743d1	13572
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	
lib-rcf	2d1b221b2a9833c4e9a76d4e1df5004a7cb38785	
libnux	334d87b70febd0cd4568c9913bc08e0e59bdd287	13457
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	
model-hw-hdbioai	ae1fca864526e41c7267428a56703d09ad89c1a4	13469
pyublas	fb538e8c313a3f04d1a5b77200d192fece3ea901	
pywrap	550051ab0faad678e58cb456079b1ba45ad2230a	
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	
sctrltp	be58599f60a8652b0404bf3a5f7dd3a3b4d1c303	
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	

Table A.4: Software state used in measurements conducted in sections 5.3, 5.5 and 5.6 except section 5.3.4.

repository	git hash	change set
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	
fisch	c65f43a7aae6e0d4e22b5b7779d9bb05b1410b2e	
flange	e6bd35e67bfa56d4bf0dbfc6a0587610e36c5630	
grenade	286df0e1c93e6f5daa1b5b192150771c85360bc9	13639
halco	307aa73f4c4ffbbee734cb8c3752f9f584ecf260	
haldls	e8dbed6827987e4811ec69de15ab72358e04bc31	
hate	e2ae29a3caa52852dd89c9371fd6f6c8f2043d1f	
hwdb	64ac5b5bc495a3af602033e77f0d927d5fd99e67	
hxcomm	fafed1860d27869366d8f809fb3e5f677b911b5d	
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	
lib-rcf	aad007af401087a32e8ba387824239cbc5f1b222	
libnux	46422c744a89ab656fd9ade0ecd6da8f5703dcf8	
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	
pynn-brainscales	b409dc567735c2dc5b29d2737e0c698317e6c137	13596
pynn-brainscales (left of table 5.1)	b409dc567735c2dc5b29d2737e0c698317e6c137	12030
pyublas	fb538e8c313a3f04d1a5b77200d192fece3ea901	
pywrap	550051ab0faad678e58cb456079b1ba45ad2230a	
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	
sctrltp	be58599f60a8652b0404bf3a5f7dd3a3b4d1c303	
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	

Table A.5: Software state used in measurements conducted in sections 5.3.4 and 5.4.

repository	git hash	change set
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	
fisch	0665adf9b07df9617e14635e292eb19ae2e6e878	
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	
halco	c4f6222499fb994e2ac10892959b23badfc202ad	
haldls	83c02c82858854eee8ebdcd241a2c63ef35aee7a	
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	
hicann-dls-private	e0564a3349d46ec9babb19400ef3fcab1c82daa2	
hmf-fpga	dfa5395ca9681e4614c83780b7ec49d2a58f5252	
hmf-fpga-test	80b8fc93498557722344d1f164d95e84168b9a88	
hwdb	7787934c0ee334393641179d4b1c4d7e5215988d	
hxcomm	a01ba278fb4994463a9e539aaeeadb950f05256e	
hxfpga	1a99147a7967843ea79b22c9b03efe2ce0601b50	
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	
lib-rcf	5b16326ae30ee08a322a6569887ca8bd2684c252	
lib-vhdl-utils	59c07b9b0edb9248c64fdfef75ac9a373b061065	
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	
s2pp	78b205f8b7189c286a5932f2568aa08de30f6009	
sctrltp	b5f825007b842f44f3e6401f00cf93387e5e3f3c	
verilog-i2c	ad61cd1b90cb60d0776fbc2f4d8fe5f81f28c437	
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	

Table A.6: FPGA state used in measurements conducted in sections 5.1, 5.2, 5.3.4 and 5.4.

repository	git hash	change set
code-format	be6615c28aedac9e423c5bc0cb602379ad775b18	
fisch	e25c2d84ed207c1b46c815a256332811e5a9b6e5	12086
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	
halco	e0aad1527e6f61020097caf137b5d17a9a25b234	12015
haldls	f55cfe123d95ad805f6fa3b3b6df129452017ce6	12124
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	
hicann-dls-private	7d81e1ec2b4668147654dc4973f9089151975e00	13408
hmf-fpga	dfa5395ca9681e4614c83780b7ec49d2a58f5252	
hmf-fpga-test	80b8fc93498557722344d1f164d95e84168b9a88	
hwdb	7787934c0ee334393641179d4b1c4d7e5215988d	
hxcomm	a01ba278fb4994463a9e539aaeeadb950f05256e	
hxfpga	ed91ebaf004b6e6120d91d5e56b551fb9b93a43f	13410
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	
lib-extoll-utils	06235908a4cb703e1ffbc56548223e6cf31f1b05	
lib-rcf	5b16326ae30ee08a322a6569887ca8bd2684c252	
lib-vhdl-utils	59c07b9b0edb9248c64fdfef75ac9a373b061065	
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	
s2pp	78b205f8b7189c286a5932f2568aa08de30f6009	
sctrltp	b5f825007b842f44f3e6401f00cf93387e5e3f3c	
verilog-i2c	ad61cd1b90cb60d0776fbc2f4d8fe5f81f28c437	
visionary-rtl-utils	3032693a101a1d5054dce685b13542e12b3c5056	13194
visions-slurm	3777a9dc36a7067be3657ce06253efec32db260e	
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	

Table A.7: FPGA state used in measurements conducted in sections 5.3, 5.5 and 5.6 except section 5.3.4.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den (Datum) .....