Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Falk Leonard Ebert

born in Münster (Germany)

2021

Real-time Image Classification on Analog Neuromorphic Hardware

This Bachelor Thesis has been carried out by
Falk Leonard Ebert
at the Kirchhoff Institute for Physics in Heidelberg
under the supervision of
Dr. Johannes Schemmel

Real-time Image Classification on Analog Neuromorphic Hardware

As dedicated logic components for accelerating demanding workloads are becoming more ubiquitous in information technology, new hardware architectures are being explored for computation beyond the classical von-Neumann paradigm. This thesis demonstrates real-time classification of handwritten digits using the neuromorphic BrainScaleS-2 architecture. In contrast to many established accelerators for artificial neural networks which operate digitally, BrainScaleS-2 is facilitating its analog synapse array to perform vector-matrix-multiplication. Enabled by the portable and fully integrated BrainScaleS-2 mobile system, images are captured by a standard commercially available camera and processed locally. Using an artificial neural network trained on the MNIST data-set, the BrainScaleS-2 application-specific integrated circuit classifies images at 66.7 Hz with an average energy-consumption of 10.5 mJ per image. The model is trained in a state-of-the-art machine learning framework via an extension for the BrainScaleS-2 hardware, allowing the system to be adapted to many artificial intelligence workloads.

Bilderkennung in Echtzeit mit Analoger Neuromorpher Hardware

Dedizierte Logikkomponenten zur Beschleunigung aufwändiger Berechnungen werden in der Informationstechnologie immer allgegenwärtiger und inspirieren die Erforschung neuer Hardwarearchitekturen jenseits des klassischen von-Neumann-Paradigmas. Diese Arbeit demonstriert die Klassifikation von handgeschriebenen Ziffern in Echtzeit mit der neuromorphen BrainScaleS-2-Architektur. In Gegensatz zu vielen etablierten, digitalen Beschleunigern für künstliche neuronale Netze verwendet BrainScaleS-2 ein analoges Synapsen-Array zur Berechnung von Vektor-Matrix-Multiplikationen. Das tragbare und vollständig integrierte BrainScaleS-2-Mobilsystem ist in der Lage, Bilder von einer handelsüblichen Kamera zu erfassen und lokal zu verarbeiten. Mit Hilfe eines künstlichen neuronalen Netzwerks, das auf dem MNIST-Datensatz trainiert wurde, klassifiziert der BrainScaleS-2 Mikrochip Bilder mit einer Frequenz von 66.7 Hz bei einem durchschnittlichen Energieverbrauch von 10.5 mJ pro Bild. Das Modell wird in einem weit verbreiteten Framework für maschinelles Lernen über eine Erweiterung für BrainScaleS-2 trainiert, wodurch das System an viele Anwendungen der künstlichen Intelligenz angepasst werden kann.

Contents

1 Introduction

In the modern landscape of information technology and manufacturing processes, field-effect transistors are approaching physical limitations, slowing the performance gains of long-established general-purpose computing architectures (Theis and Wong, 2017). These general-purpose architectures are often supplemented by application-specific integrated circuits (ASICs), which make more efficient use of the computational power available *in silico* (Mittal, 2020). Research into new computational architectures specialized on specific workloads has increased in recent years. Especially for use in the field of artificial intelligence (AI), new accelerators designed for those workloads are presented at a very high rate (Chen et al., 2020; Reuther et al., 2020).

The increasing demand for intelligent control systems by for example the automotive industry (Lee et al., 2018) has fueled research into low latency inference devices for local artificial intelligence applications (Edge AI). Especially the high data rates encountered in tasks such as real-time video analysis benefit considerably from edge computing (Ananthanarayanan et al., 2017). The increasing use of AI with sensitive data also highlights the benefits of local processing (Zhao et al., 2018).

Inspired by neuroscientific research, neuromorphic computing tries to mimic the principles of the biological nervous system and has lead to the development of many new architectures (Thakur et al., 2018; Schuman et al., 2017). Understanding the dynamics of spiking neural networks and improving the energy-efficiency of their electrical imitations are main areas of research in this field. The BrainScaleS (BSS) neuromorphic architecture presented in Schemmel et al. (2010) is a mixed-signal ASIC capable of emulating spiking neural networks (Wunderlich et al., 2019; Billaudelle et al., 2019) using analog electrical circuits. It uses analog circuitry to emulate the spiking behavior as it is more energy-efficient (Joubert et al., 2012) while using digital components to propagate signals over larger distances. While originally developed for wafer-scale integration and emulation of large spiking neural networks (Schmitt et al., 2017), its successor BrainScaleS-2 (Schemmel et al., 2020) is additionally capable of accelerating artificial neural networks (ANNs) (Weis et al., 2020), which makes it applicable to a large set of workloads (Mnih et al., 2015). Recently it has also been used in the BrainScaleS mobile system, a portable low energy computing platform for Edge AI and neuromorphic research, which has been demonstrated by Stradmann et al. (2021).

Making use of the speed of the accelerated neuromorphic architecture of BSS-2, this thesis demonstrates real-time classification of handwritten digits from the MNIST dataset (LeCun et al., 2010). Enabled by the portable and fully integrated BSS-2 mobile

system, handwritten digits are captured by a camera and locally processed at interactive rates, demonstrating the applicability of the BSS-2 architecture to these emerging workloads. In contrast to many other processors specialized for embedded AI like Google's Edge TPU (Coral) which operate digitally, BSS-2 uses its analog core to perform vector-matrix multiplications, which has the potential to be more energy efficient (Yamaguchi et al., 2019).

2 Methods

This chapter provides an overview of the different techniques, frameworks and hardware systems used throughout this thesis. First, a general overview of the BrainScaleS-2 neuromorphic architecture and its operating principles is given. The supporting software framework and hardware abstraction layers that enable flexible usage of this versatile system are presented afterwards. This thesis uses a specialized hardware setup enabling standalone operation without the need for a dedicated host computer. Its system architecture and principal components are described and finally, the high-level software frameworks which enable integration with state-of-the-art machine learning workflows are described and explained in some detail to aid in understanding the following discussions.

2.1 BrainScaleS Neuromorphic Hardware

The HICANN-X-v2 (High Input Count Analog Neural Network) ASIC is the most recent implementation of the BSS-2 architecture and has been presented in Schemmel et al. (2020). The neuromorphic substrate is manufactured in a 65 nm complementary metal-oxide-semiconductor (CMOS) process and uses a combination of digital control logic and analog neuromorphic circuits. The digital part is used for event routing, communication with the host system and the on-chip plasticity processing unit (PPU), while the analog core comprises the synapses, neurons and their parameter storage. This mixed-signal approach plays to the strengths of both techniques while avoiding complicated workarounds to mitigate their respective shortcomings and results in a flexible and powerful architecture (Wunderlich et al., 2019).

2.1.1 Chip Architecture

To prevent degradation of the analog signals over long distances, the chip's analog components are split into four symmetric quadrants grouped pairwise into hemispheres. Each hemisphere contains a digital co-processor, the so-called plasticity processing unit (PPU) capable of executing arbitrary code and connected to the components in a highly parallel manner. These co-processors allow efficient on-chip learning and flexible usage of the system as demonstrated by Friedmann et al. (2017). Figure 2.1 shows the arrangement and size of the functional blocks within the chip while fig. 2.2 shows the layout of a single quadrant's analog components in more detail. In total, the BSS-2 chip contains 512 neurons and $512 \cdot 128 \cdot 2 = 2^{17}$ synapses.

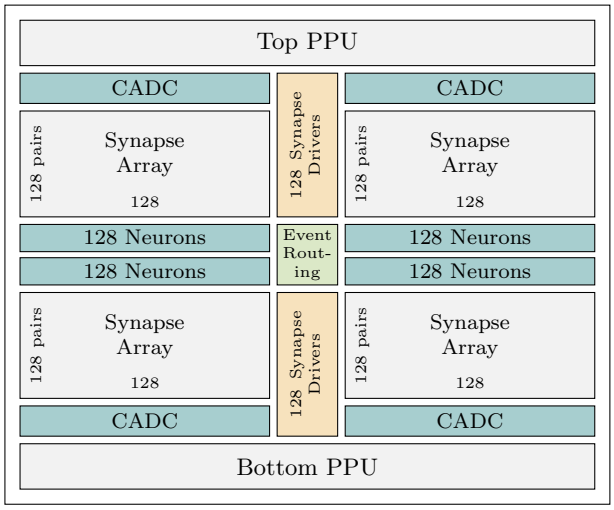

Figure 2.1: Dimensions and approximate layout of the BrainScaleS-2 ASIC's main components. External stimulus and neuron spikes are distributed to the synapse drivers or FPGA by the event routing logic.

As the neuromorphic hardware is intended to be used as an accelerator for emulating neural networks, it needs to interface with a host computer that provides configuration, external stimulus and records the experiment's results. This interface is provided by a controller based on a Field Programmable Gate Array (FPGA). It contains various supporting logic elements, which execute the experiment and record the results in the form of timestamped spike-trains or membrane-voltage recordings. High-speed low-voltage differential signaling (LVDS) links are used to connect the chip and the controller at high bandwidth. In addition to a low-latency transport protocol for spike-events, these links also transmit a reliable transaction-based communications protocol ("*omnibus*") for access to configuration parameters and generic memory.

Each quadrant of the chip consists of 128 analog neuron circuits implementing the Adaptive Exponential Integrate and Fire (AdEx) model presented in Aamir et al. (2018), which extends the Leaky Integrate and Fire (LIF) model (Lapicque, 1907). The synapses are arranged in an array and contain static memory for their parameters. They are connected directly to the PPUs using parallel data lines enabling quick and efficient configuration. Each neuron is connected vertically to a column of 128 signed synapse pairs making up the array. Two rows of unsigned synapses make up one signed synapse row. Each signed row is stimulated by one synapse driver circuit which implements short-term plasticity modulating the pre-synaptic pulse strength (Schemmel et al., 2007). When a neuron spikes, it sends a digital signal to the event routing logic whose configuration determines whether this event should be sent to another synapse driver implementing interconnected neurons or recorded on the controller.

To store parameters for the analog circuitry, each row of neurons is supplemented by a row of analog parameter storage that is kept synchronized to the quadrant's digital

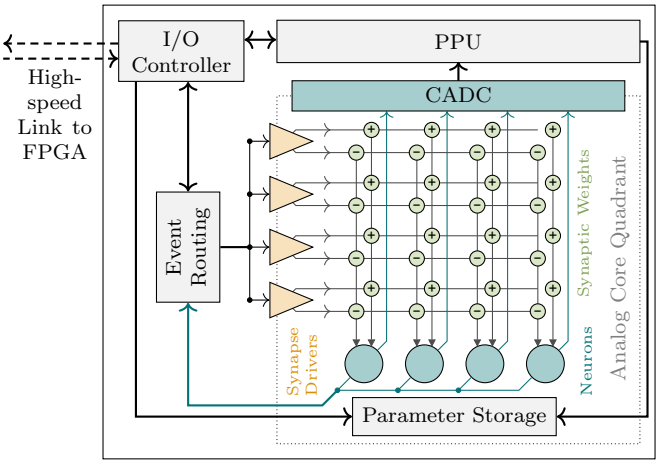

Figure 2.2: Schematic block diagram of a quadrant's relevant components are shown; their quantities are not representative. Each neuron is connected to a column of synapse pairs usually configured as inhibitory and excitatory respectively. The synapse drivers are driving two rows of unsigned synapses each.

memory block. This capacitive analog memory is presented in more detail in Hock et al. (2013). For fast parallel readout of the neuron membranes and sensing correlations in the firing patterns, each column is connected to a channel of the column analog to digital converted (CADC) controlled by the PPU.

Calibration

Since the manufacturing process for CMOS has inherent variability which is especially noticeable in analog circuits, not all synapses and neurons behave equally. To achieve predictable and consistent behavior, the AdEx neuron model incorporates many runtime-configurable parameters, which allow fine-grained tuning of the circuits' electrical behavior. This makes the BrainScaleS architecture applicable to different biologically inspired neuron models and also allows for novel uses of the analog neuromorphic substrate (Schemmel et al., 2017). Thus, the exact calibration targets depend on the intended use-case and are presented in more detail in Leibfried (2018) for spiking operation and in Weis et al. (2020) for acceleration of artificial neural networks.

HAGEN mode

Inspired by earlier hardware systems facilitating analog circuitry such as the Heidelberg AnaloG Evolvable Neural Network (HAGEN) (Schemmel et al., 2004), the BSS-2 hardware also implements non-spiking operation to accelerate vector-matrix multiplication. In this aptly named HAGEN mode, the neurons' membranes are used as integrators for the post-synaptic currents emanating from the synapse array. The synaptic weight matrix is used to encode the matrix, while the inputs generated by the synapse drivers correspond to the input vector. The synapse driver's short-term plasticity is used to scale

the pre-synaptic currents according to the input vector's entries. After passing through the synapse matrix, the post-synaptic currents are scaled according to the configured weight matrix and accumulate on the neuron membranes. Precisely timed readout of the resulting membrane potentials via the CADC yields the result of the multiply-accumulate (MAC) operation. To optimize this behavior, the neurons are calibrated for a slow decay towards their resting potential while the synaptic time constant is minimized to accelerate the integration.

During the experiment described in this thesis, the BSS-2 is used in this non-spiking HAGEN mode to accelerate vector-matrix multiplication on the analog neuron membranes. More details on the exact operating principles can be found in Weis et al. (2020).

2.1.2 Software Framework

Neuromorphic architectures and the interfacing with them pose many challenges in terms of software control and integration. Making efficient use of the computing resources is an integral part of the research into neuromorphic architectures (Titirsha et al., 2021) and providing powerful and user-friendly interfacing options is crucial for widespread adoption of this technology (Moradi and Indiveri, 2014). For the BrainScaleS-2 architecture, supporting hardware and software infrastructure is presented in Müller et al. (2020a,b) and continues to be extended to enable flexible and efficient research.

Since the BSS-2 architecture is capable of accelerating several different neuromorphic computing workloads, the software architecture aims to provide a useful level of abstraction that provides easy usage for common applications while preserving the flexibility required for diverse research. Using multiple layers of abstraction, some of which are briefly outlined below, the software stack provides the ability to integrate the hardware with high-level description languages allowing for user-friendly operation while also providing fine-grained control at lower layers if needed. This is only a general overview of the architecture to aid in understanding the following sections on software development. A comprehensive explanation of the architecture can be found in Müller et al. (2020b).

- **Hardware Coordinate System** The Hardware Abstraction Layer Coordinate System (*halco*, Electronic Vision(s) Group (2021a)) is used to address the large number of configuration parameters and other components on the system. It provides a comprehensive interface and reflects the symmetries inherent to the system.
- **FPGA Instruction Set** The FPGA Instruction Set Compiler (*fisch*, Electronic Vision(s) Group (2021b)) abstracts accesses to the systems components (identified by their coordinates) on the basis of so-called *containers* while providing basic validation. Writes to (and reads from) these containers are encoded by messages in the Universal Translator (UT messages) format (Karasenko, 2020). An experiment is completely described by a sequence of these UT messages. To execute the experiment, its messages are serialized into a playback-program, which is understood

by the FPGA.

- **Hardware Abstraction Layer** Building on the aforementioned abstraction layers, the Hardware Abstraction Layer (*haldls*, Electronic Vision(s) Group (2021c)) provides nested data structures that consolidate individual hardware into logical units and provide a higher-level interface (Müller et al., 2020b). This enables a more succinct description of experiments and allows the hardware to be easily configured without the need for expert knowledge on all component's inner workings.

This list is not exhaustive as it excludes for example the library implementing communication with the FPGA as well as the framework for programming the PPUs. There also exist higher-level abstraction layers enabling descriptive configuration of spiking neural networks such as PyNN (Davison et al., 2008).

2.2 BrainScaleS Mobile System

The BrainScaleS-2 ASIC is well suited for local artificial intelligence applications because of its low energy consumption. Stradmann et al. (2021) paired it with a small single-board computer with an integrated FPGA providing a versatile computing platform for inference on the edge (fig. 2.3). The direct interface between the computer's CPU and the FPGA reduces input and output latency and enables the use of local peripherals.

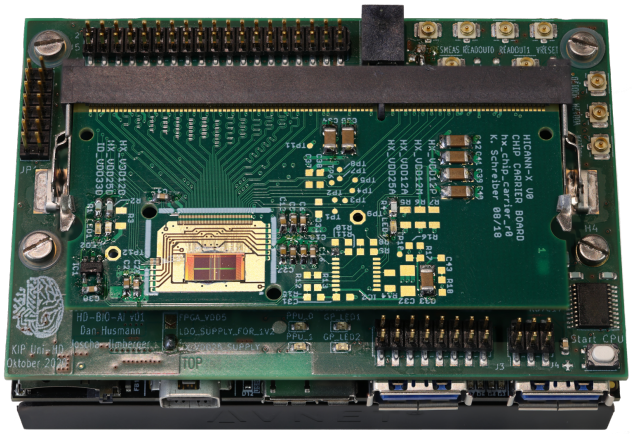

Figure 2.3: Picture of the BrainScales-2 mobile system. Components form bottom to top: Ultra96 development board (FPGA based system controller), ASIC Adapter Board, BSS-2 ASIC bonded to carrier board. Figure from Stradmann et al. (2021).

The BrainScaleS-2 mobile system consists of the following core components:

1. **Ultra96 development board (Avnet Inc., 2020)**
   General purpose development board built around a *Xilinx Zynq* Multi Processor System on a Chip (MPSoC), which provides a quad core 64bit ARM processor and

programmable logic fabric. Peripherals can be connected over USB and interfaced with via standard Linux drivers.

2. **ASIC Adapter Board (ASICAB)**
   This adapter board is compatible with the expansion interface of the Ultra96 and provides power regulation, references and general IO circuitry for the BSS-2 ASIC.

3. **HICANN-X Carrier Board**
   Printed circuit board (PCB) to which the BrainScaleS-2 analog neuromorphic accelerator is bonded. It also contains passive filtering components and a small integrated circuit (IC) for digitally identifying the chip.

The CPU on the development board runs a Linux operating system and is capable of running the control software. This eliminates the network latency and bandwidth limitations present in classical host-based BrainScaleS configurations. For the purpose of this thesis, a regular consumer webcam is connected to the setup via USB and controlled using the Video for Linux API (Dirks et al.).

To train the system in conjunction with a distributed computing cluster, a communications protocol for interfacing over the network ("*quiggeldy*") has been developed in Breitwieser (2021). The playback-program is serialized on the host computer and then sent to the mobile system over the network, which executes this program and returns the results the same way.

2.3 System and FPGA Architecture

The FPGA-based system controller's main function is to control the bidirectional data transfer between the user's application and the chip. It receives the encoded playback-program from the application level and decodes it for pipelined execution. An in-depth description of the FPGA architecture can be found in Rettig (2019).

This section introduces the architectural differences of the BrainScaleS-2 mobile system. In contrast to the network-based BSS setups, this system tightly couples the FPGA and CPU via an Advanced eXtensible Interface (AXI) main communication bus, which is integrated into the MPSoC. Figure 2.4 shows a functional block diagram of the system's components and logic modules of the FPGA fabric, that are relevant to the work in this thesis.

To run an experiment, the software framework produces a playback program consisting of UT messages as described in section 2.1.2. These messages encode the experiment's static configuration and input data such as neuron events. The playback program is then loaded into the FPGA's playback buffer via the internal AXI interconnect. After being triggered, the executor begins the execution of the experiment. It performs configuration of the system components via omnibus transactions and generates neuron events which are transmitted to the BSS-2 ASIC via five high-speed links. During the execution of the program, all results are recorded by the executor and written into the trace buffer.

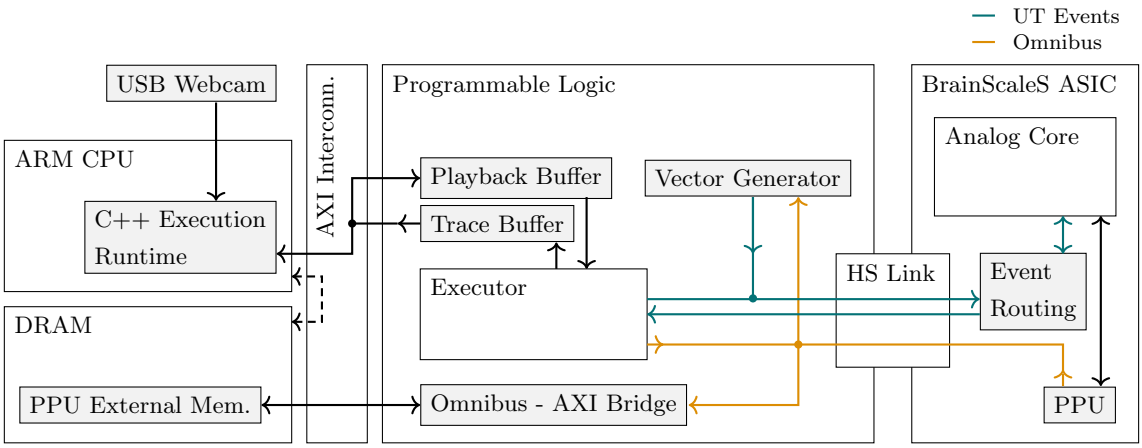

Figure 2.4: BrainScaleS Mobile System - functional block diagram. The arrow heads denote data flow direction and bus controllers.

This buffer is then read by the control software and decoded to be available to the user's experiment.

The PPU on BSS-2 can also execute arbitrary code and has access to the same on- and off-chip resources as the executor. In addition, it supports more efficient access to the analog core via *Same Instruction Multiple Data* (SIMD) operations. It can for example be used to quickly read out the neuron membrane potentials via the column digital-analog-converter (CADC) and has greater bandwidth connectivity to the synram. To quickly store partial results and other transient data which is not recorded to the trace buffer, a portion of the system's DRAM is reserved as the PPU's external memory. Its AXI interface is connected to the omnibus infrastructure via a bridge.

2.3.1 Vector Generator

Since the chip is used in HAGEN mode, the input vector of a vector-matrix multiplication has to be converted into its corresponding spike sequence. To reduce the computational load of this operation, Stradmann et al. (2021) introduced the vector generator, a real-time logic module written in SystemVerilog. This module serves as a spike-event source for the analog core and supports pipelined pre-fetching of data from either a first-in-first-out (FIFO) queue or (with an additional DMA controller) directly from system memory. This forgoes the need to encode the input data as spike events in the playback program and allows it to be transmitted as binary data instead. The vector generator also supports configuration of many hardware-specific parameters such as the number of re-sends for each vector entry and the delay between them. In order to achieve accurate timing, it provides a real-time synchronization interface to the PPU via omnibus. Additionally, it makes use of the synapse drivers' label-bit (see section 2.1.1) providing a mechanism to apply a mask to each spike event, which is honored by the synapses. Placing another weight matrix on the vertically adjacent synapses and configuring them

to the appropriate mask, allows a matrix with an input dimension of up to 256 to be processed side-by-side in one operation. This comes at the cost of using twice the number of neurons as integrators and needing to sum the activations of each neuron pair afterwards.

2.4 PyTorch Extension

PyTorch (Paszke et al., 2019) is one of the most popular frameworks for researching artificial neural networks (ANN) (He, 2019). It provides APIs for modelling neural networks and also supports automatic differentiation which greatly improves the training process. HxTorch, an extension for PyTorch, adds support for the BrainScaleS hardware and has been presented in Spilger et al. (2020); Electronic Vision(s) Group (2021d). It contains common operations for matrix multiplication where the forward pass can be executed on BSS-2. It also implements the backward pass which is executed on the host system's processor and a hardware emulation mode ("mock mode"), which can be used to speed up training. A typical training process where the BSS hardware is emulated for the forward pass is outlined in fig. 2.5.

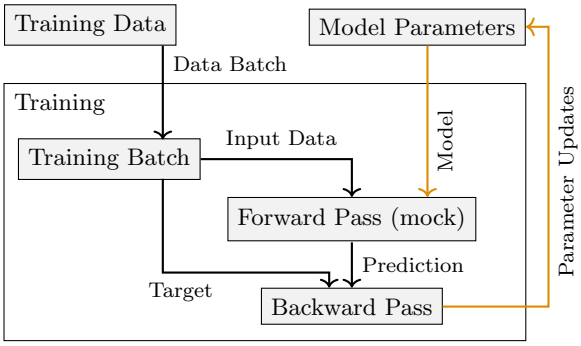

Figure 2.5: Training with HxTorch in mock mode

This training process is typically repeated with the hardware-based forward pass for a few epochs in order to compensate for fixed-pattern noise on the hardware. While calibrating the chip can reduce these fixed-pattern deviations, their remains are not adequately accounted for by the mock mode's Gaussian noise. Training with hardware in the loop can improve an MNIST model's accuracy on test data by about 6 % (Weis et al., 2020).

2.5 Graph Based Experiment Notation and Execution

HxTorch's forward pass consists of several operations - one for each layer or unit in the model. These operations are translated into graph representations describing the processing steps which need to be taken on the hardware. Building these representations

and executing them on the hardware is handled by the *Graph Based Experiment Notation and Execution* (grenade) which is covered in more detail in Spilger (2021a) and can be found online at Electronic Vision(s) Group (2021e).

This representation as a directed signal-flow graph has several advantages over the naive approach of a flat, sequential configuration. It represents operations as a collection of vertices, which are connected by directional edges representing the computational operations and their interdependence respectively. While being a more intuitive representation it can also express the digital operations (such as read, store and additions) in between the various hardware execution steps. This allows experiments to be more naturally designed and reasoned about since the notation follows the natural data flow through the network. It also has some technical benefits, since the same vertices can be processed differently depending on the execution environment allowing for greater flexibility and optimization for application-specific constraints.

2.5.1 Execution Instances

One central aspect of the technical execution details is the placement of the graph's vertices on so-called *execution instances*. They represent a real-time execution step and its physical and temporal placement on the hardware. Edges between vertices of the same execution instance signify immediate, volatile data dependencies like events being sent to the synapse drivers or the neuron's membrane potentials after an operation. Edges between different execution instances on the other hand can only transfer digitized data stored in memory on either of the PPUs or the FPGA. This grouping of sub-graphs into execution instances is depicted in fig. 2.6.

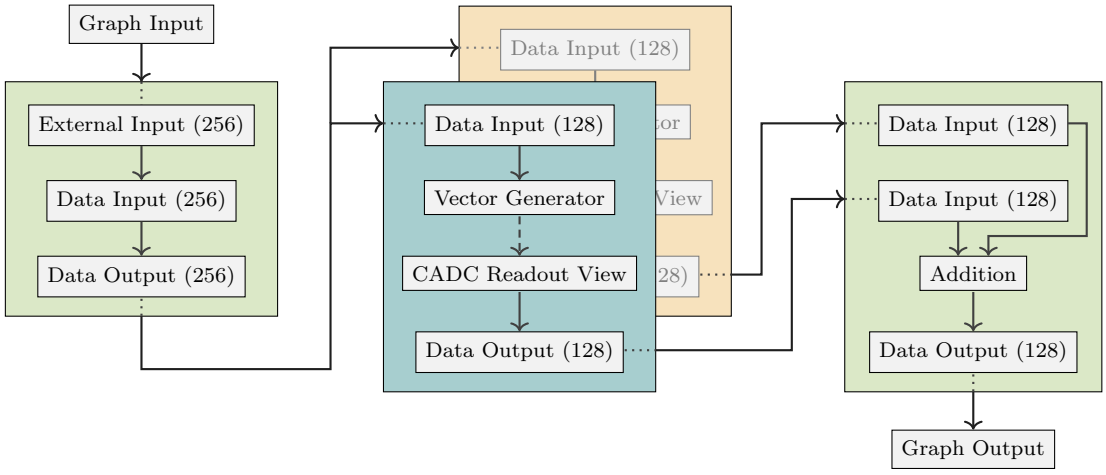

Figure 2.6: Execution instance assignment depicted for the example of a (simplified) 256x128 matrix multiplication. Different execution instances are indicated by colored boxes around sub-graphs. The matrix multiplication is split into two parts, since only 128 inputs can be processed at once due to physical limitations of the chip.

After placing the vertices on execution instances, their sub-graphs are traversed and the static configuration is extracted from each vertex. These configuration parameters are bundled into the instance's configuration playback program. The execution of the real-time operations depends on the backed and is described in the following paragraphs.

2.5.2 Just-in-Time (JIT) Execution

The JIT compiler and executor make use of the host computer for most of the digital calculations and orchestration of the graph. The execution instances are transformed into a graph as shown in fig. 2.6 which is traversed in order and executed one instance at a time. For each instance, its static configuration is applied to the chip, after which the inputs to the instance are pre-processed on the CPU of the host. They are then executed in real-time and the results are subsequently fetched from the hardware and also post-processed on the host. Thus, the configuration parameters which are relevant for a particular execution instance are always applied right before the operation takes place, which enables the same hardware components to be used with different configuration across multiple instances.

2.5.3 PPU Mastered Execution

In contrast to the previously described method, the PPU mastered executor makes use of the SMID CPU embedded in the BrainScaleS-2 hardware for executing the real-time portion of an execution instance. The PPU is fully programmable and is connected to the system's omnibus. Its vector unit and direct access to the CADC enable a more efficient readout of the neuron membrane potentials after a multiply-accumulate (MAC) operation. Instead of transferring the results of every execution instance to the host for processing, the on-chip processor can perform these digital operations, which in principle allows execution instances to be merged. Optionally, the instance's static configurations can be merged as well, which allows the real-time part of the execution to be completely independent of the host (see fig. 2.7). However, since each execution instance contains a static configuration program that also need to be merged together, the graphs can only use a single static configuration for each physical hardware component throughout the program. This limits the size of networks that can be executed in this manner and also necessitates spatially efficient mapping of vertices onto the hardware coordinates to prevent conflicts in their static configuration.

Merging the execution instances and building the execution program requires an additional compilation step. Initially, the graph of execution instances is traversed to generate the initial configuration and execution commands for each instance. Afterwards, the initial configurations and the execution commands are concatenated. In practice, the execution program is split into three separate playback-programs, which handle loading the input data to the FPGA, triggering PPU mastered execution and retrieval of the results respectively. Spilger (2021a) describes the graph execution in more detail.

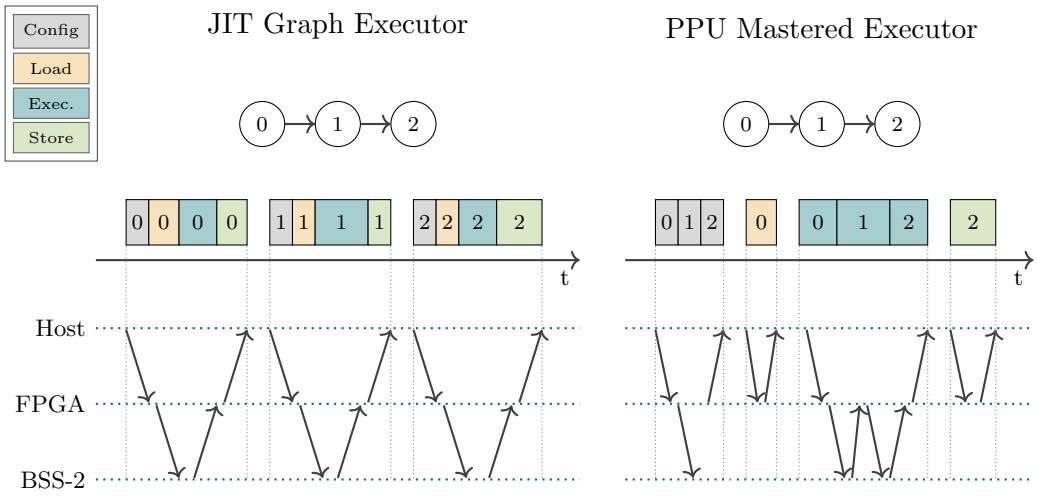

Figure 2.7: Comparison of just-in-time execution and PPU mastered execution, time axis not to scale. From top to bottom: Execution instance graph, schematic execution steps and data flow between system components. For the JIT executor, the processing steps on the host in between different instances are not shown.

Following the argument of Stradmann et al. (2021), the PPU mastered execution flow is used in this thesis. Since the goal is inference at interactive rates and MNIST does not require a very large model, also the static configurations will be merged, restricting the model to fit completely on one set of hardware components, i.e. one chip.

2.6 Modelling Workflow

This section describes the workflow for designing, training and executing a model in the framework described above. Defining the model adheres to the usual PyTorch workflow as described in section 2.4 with the only difference being the use of the HxTorch functions in the model definition. Their API is designed to be familiar while also providing control over hardware-specific parameters where necessary. Initial training of the model is done with the emulated forward pass which does not interact with the grenade hardware layers. When the model has converged to a suitable accuracy, retraining with hardware in the loop can be used to further improve the accuracy as mentioned previously.

When executing the forward pass on the hardware, the HxTorch functions are executed using grenade's JIT executor. Each function is implemented by a *computable* representing the sub-graph of hardware operations necessary to accomplish this computation. These sub-graphs are executed one at a time during the forward pass, which results in many round-trips to the hardware per inference batch. In addition to training on the hardware, the inference can also be traced into a sequence of computables, which comprise the complete inference graph and model parameters. The traced sequence can be serialized and parsed, which allows it to be easily copied into the BSS-2 mobile system

for deployment.

For execution, the model's traced sequence can be transformed into a playback program using the PPU mastered compiler. This initial compilation step only needs to happen once and since the static configurations can be merged as well, the playback programs for initialization and configuration also only need to be executed once. Performing inference on a single batch of input-data consists of three steps. Firstly, the input data is transformed into a load program and run on the hardware. The already compiled execution program is then executed after which the store program returns the model's prediction.

2.7 Webcam Image Capture

Capturing a frame from the camera hardware requires interfacing with the camera's Linux driver. The Video for Linux API (Dirks et al.) provides a well-supported abstraction layer and is included in the default configuration of most Linux kernels. To capture frames from the camera, it needs to be initialized through the API by specifying the capture parameters, which include the desired data format and resolution. For efficient and performant frame capture, a memory-mapped ring buffer is chosen, which operates by directly mapping parts of the camera's hardware frame buffer into system memory. The application can request buffers to be dequeued, which allows access to the captured frame. After processing, the buffer is enqueued again to be filled by the camera once another frame is available.

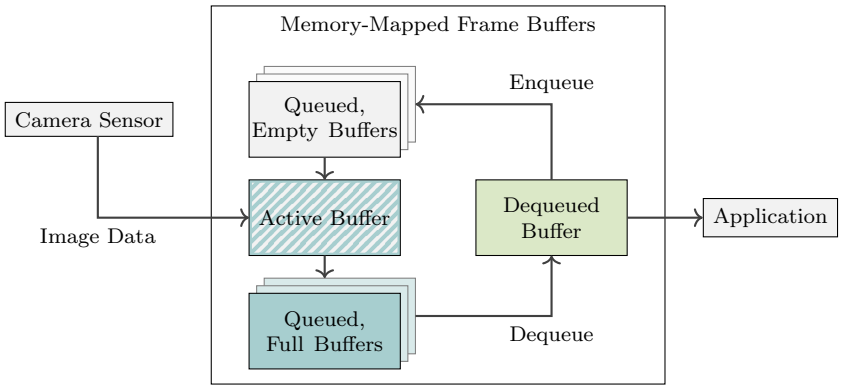

Figure 2.8: Schematic flow diagram of ring-buffer based frame capture using a USB webcam.

The maximum number of frame buffers is limited by the size of the camera's hardware frame buffer and should be chosen as small as possible to decrease latency since the buffers are always filled sequentially. However, it needs to be ensured, that at least one buffer is always available for the camera to fill; otherwise, frames would be dropped. Assuming that the processing of a frame is consistently faster than the camera can deliver new frames, two buffers are an ideal configuration for low-latency video capture. For this

thesis, processing a frame only entails creating a monochrome copy of the buffer. This is very simple since the camera supplies frames as YUYV-encoded data, which consists of alternating chroma (U, V) and luminance (Y) values. Copying the luminance values into a new buffer yields a 8 bit per pixel monochrome copy of the frame. Since the images will be further downscaled to 28 px × 28 px resolution, the frames are requested at the lowest resolution supported by the camera (176 px × 144 px).

3 Implementation

This section presents the implementation details for various software components involved in the real-time image classification. First, an introduction to the neural network and the data pre-processing is given. Motivated by the model's topology, the implementation of the hardware operations used in the inference is presented with a focus on optimizations for runtime performance. Finally, the processing steps making up the execution loop are shown and the resulting data flow is discussed.

3.1 MNIST Model

As described in Deng (2012), the MNIST data-set is a widely used benchmark in basic image recognition. The data-set consists of 28 px×28 px monochrome images of single digits ranging from 0 to 9 and their respective labels. As can be seen in Weis et al. (2020) and Lecun et al. (2000), even very shallow network topologies consisting only of fully-connected linear layers can achieve good accuracy of up to 98 percent on the test data.

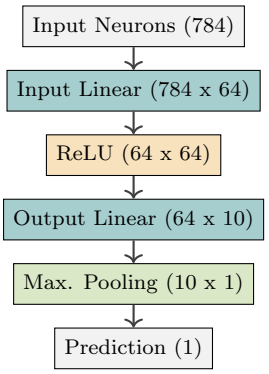

Figure 3.1: MNIST model topology. The linear layers are fully connected and dimensions are indicated in brackets.

Since using the PPU mastered execution limits the model's size (see section 2.5.3), the following network topology was chosen. The input image data is squashed into a one-dimensional tensor of length $28 \cdot 28 = 784$. This input data is then fed through a fully connected layer onto 64 hidden neurons. After applying a Rectified Linear Unit (ReLU), the hidden neurons are fed into another fully connected layer which maps them onto the 10 output neurons. A max-pooling operation then selects the output neuron with

the largest activation as the model's prediction. A visual representation of this model is shown in fig. 3.1 and an implementation is shown in listing 3.1.

Due to size constraints of the hardware, this is a rather minimal model which forgoes processing steps like convolutional layers which are typically found in machine learning, especially with image data. It should be noted, however, that despite its small size, this model can process the full MNIST input data without any prior downsampling or other size-reduction techniques.

```
class ModelMNISTSmall(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.fc_1 = torch.nn.Linear(28 * 28, 64, bias = False)
        self.fc_2 = torch.nn.Linear(64, 10, bias = False)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = self.fc_1(x)
        x = self.relu(x)
        x = self.fc_2(x)
        return x
```

Listing 3.1: Pure PyTorch implementation of the small MNIST model.

3.2 Image Preprocessing

In order to use images captured by a webcam as inputs to the model, they need to be preprocessed to match the training data. This consists of downsampling and cropping the images since virtually no webcams support capturing in arbitrary resolutions and aspect ratios. The method applied here is average downsampling, for which every pixel in the input image is mapped non-injectively onto a downscaled output image and averaged with all other pixels mapped onto the same downscaled coordinate. Parts of the input image, which do not fit into the desired aspect ratio, are discarded.

In addition to the dimensions of the image, special care also needs to be taken to match the training data regarding the distribution of pixel values. Specifically, the MNIST data is formatted such that the pixels which make up the digit have values close to 255 while the background is uniformly close to values of 0. The camera captures images of black digits on a white background and the downsampled images are distributed very differently, as fig. 3.2 clearly shows.

In the following, input pixels $p_i \in I$ denote the values of individual pixels of the captured and downsampled images. The input pixels are firstly converted to floating-point values according to eq. (3.1). Afterwards (eq. (3.2)), pixels are divided by a constant threshold s and restricted to a maximum value of 1 to clip pixels that are brighter than this threshold. This image is then inverted by subtracting the pixel values from 1 and scaled back up to the range $[0, 31] \subset \mathbb{Z}$ corresponding to 5 bit accuracy supported by the vector

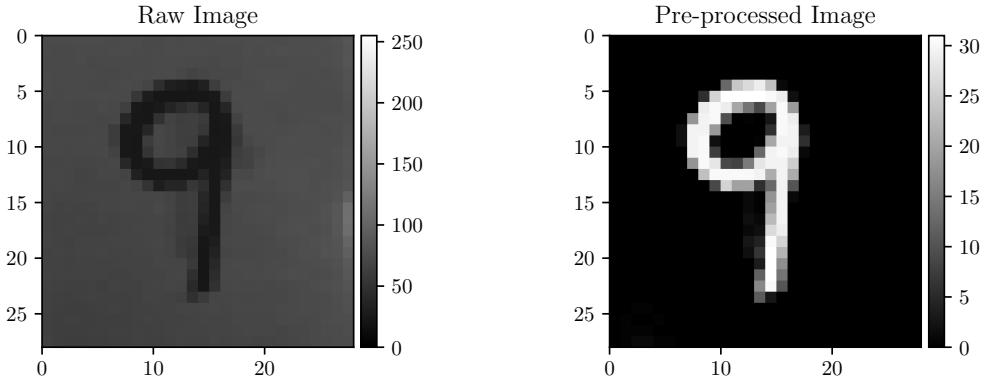

Figure 3.2: fLTR: Raw downsampled image captured by the camera, image after pre-processing.

generator (eq. (3.3)). A white threshold of $s = 0.4$ yields good results for the camera and lighting conditions encountered in testing. The code in listing 3.2 implements this pre-processing algorithm in C++.

$$p_{i,\text{rel}} = \frac{p_i - \min\left(p_i \in I\right)}{\max\left(p_i \in I\right) - \min\left(p_i \in I\right)} \tag{3.1}$$

$$p_{i,\text{clip}} = \min\left(1, \frac{p_{i,\text{rel}}}{s}\right) \tag{3.2}$$

$$p_{i,\text{out}} = 31 \cdot \left(1 - p_{i,\text{clip}}\right) \tag{3.3}$$

```
void preprocess_image (unsigned char* data,
                       size_t const width,
                       size_t const height)
{
    static const float white_threshold = 0.45;
    unsigned char min = 255, max = 0;
    for (size_t i = 0; i < width * height; i++) {
        min = data[i] < min ? data[i] : min;
        max = data[i] > max ? data[i] : max;
    }

    const float min_max_diff = max - min;
    for (size_t i = 0; i < width * height; i++) {
        if (min_max_diff == 0) {
            data[i] = 0;
        } else {
            const float normalized = (data[i] - min) / min_max_diff;
            const float scaled = normalized / white_threshold;
            data[i] = 31 * (1 - (scaled > 1 ? 1 : scaled));
        }
    }
}
```

Listing 3.2: Camera input pre-processing implemented in C++

3.3 MAC Operation

Although the previously presented model is in theory sufficiently small to completely fit on one chip with no reconfiguration, the large input dimension of the first linear layer presents a challenge. Referring to section 2.1.1, the chip can only process 128 inputs simultaneously in one analog matrix multiplication due to the limited number of synapse drivers per column. Processing the 784 inputs of the first linear layer would require $\lceil 784 / 128 \rceil = 7$ separate operations. Making use of the label-bits supported by the vector generator (see section 2.3.1), the number of operations can be reduced to $\lceil 784 / 256 \rceil = 4$. However, the use of label-bits needs twice the number of output neurons so that, while it increases the execution speed, it does not allow larger MACs to be placed on the chip. The absolute size limit for single-chip vector-matrix multiplication stems from the number of weights available in the synapse matrix, which is $512 \cdot 128 \cdot 2 = 2^{17}$ for signed matrices (512 neurons, 128 rows of signed synapse pairs per hemisphere and 2 hemispheres). The required execution time scales discretely and linearly with the input dimension, as it needs to be executed in chunks of 256 entries.

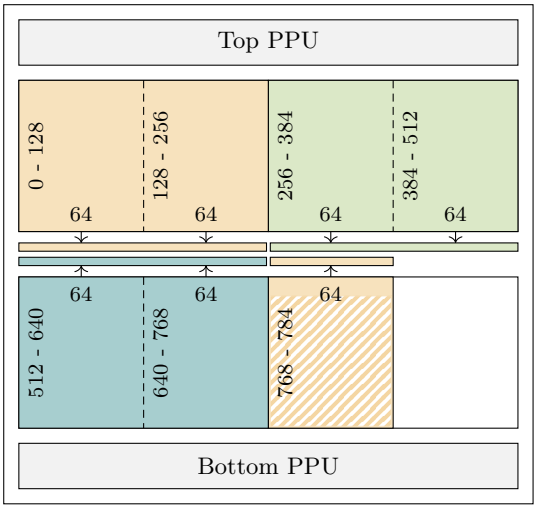

Figure 3.3: Placement of the 784 x 64 MAC on the chip. Two adjacent blocks are combined into one hardware operation through the use of label-bits as indicated by the colors. The last operation uses only part of the synapse rows.

This spatial and temporal splitting of the MAC operation allows the long input to be processed on the chip without any reconfiguration in between runs while still leaving space for up to a 128 x 64 MAC. Due to its capability of filling a whole chip with one MAC, this operation is referred to as `MACSingleChip`. To execute this operation on the hardware, the execution graph connecting the various steps needs to be built.

After determining the placement of sub-operations on the physical chip, the graph input needs to be stored in memory so that it is available for all following execution instances. For each partial operation, the corresponding slice of the input data is loaded and a `MACSmall` computable is inserted to perform the partial vector-matrix multiplication.

The computable exposes parameters, which are used to assign it to a hemisphere and place it via offsets for synapse columns and rows. In fig. 3.4, the construction of the complete operation is outlined. Each `MACSmall` computable in a sub-operation corresponds to a colored pair of operations in fig. 3.3.

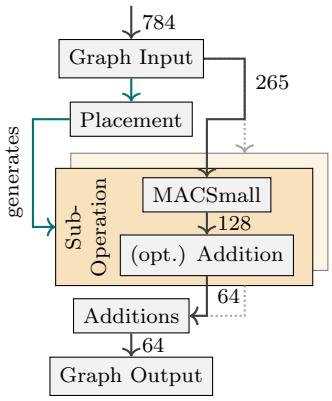

Figure 3.4: Schematic graph construction for `MACSingleChip` operations.

A computable's sub-graph then loads the corresponding input slice into the vector generator, configures the synram for the multiplication and stores the output of the neurons. If this sub-operation uses label-bits, its output is split in the middle and digitally summed to obtain the sub-operations final result with the correct output dimension. Finally, the results of all computables are summed together, which yields the final result of the MAC operation.

Mapping of vertices onto the execution instances is very similar to the exemplary operation in fig. 2.6. The first instance stores the graph's external input for later use. Each `MACSmall` operation is mapped onto a separate instance, which begins with loading the input data slice, then performs the vector-matrix multiplication and finally reads back and stores the result. In order to simplify the crossing of hemispheres, the additions are placed onto separate instances as well, so that their results are stored in memory.

It should also be noted that execution instances are assigned to a hemisphere on the chip, which allows simultaneous execution of two instances when using PPU mastered execution. For the `MACSmall` operations, the hemisphere is explicitly specified by the placement. For all other vertices in the graph, the hemisphere-placement is deduced from the vertex's source edge (defaulting to the top PPU), which reduces data transfer between PPUs. When branches of the graph meet, a synchronization barrier is inserted into the PPUs' command queues to ensure that all data is available before continuing the graph execution. This simultaneous execution allows the complete 784 x 64 MAC to be executed with just two analog full-chip operations.

3.3.1 HxTorch Integration

As already discussed in section 2.4, integrating with the machine-learning framework requires the implementation of at least a backward-pass for this operation. For increased training performance, a mock mode for the forward-pass is also crucial. However, since the MAC operation outlined above behaves just like standard matrix multiplication, the mock mode and backward-pass are no different from the already existing implementations for MACs on the hardware, even though those are executed differently on

hardware. The mock mode's forward-pass quantizes the operation's inputs and weights and performs matrix multiplication. The result is then scaled according to the expected hardware gain, combined with Gaussian noise and finally clamped to the hardware's digital range.

HxTorch already provides analogues to PyTorch's linear layers, which provide useful abstraction from the primitive computation elements. Listing 3.3 shows how the native PyTorch layers can be easily substituted with the HxTorch variants. In order to use the `MACSingleChip` and `MACSmall`, the layer's internal matrix multiplication are reassigned to the specific variants. Line 3 in listing 3.3 specifies the placement of the second layer on the chip's bottom hemisphere with column 192 as its starting column, as is the first one that is unoccupied following the large input layer (see fig. 3.3).

```
1  def mnist_matmul_out (x: torch.Tensor, other: torch.Tensor, ..., mock: bool):
2      return hxtorch.mac_small(x, other, labels = None, mock = mock, ...,
3                               hemisphere = 1, start_row = 0, start_column = 192)
4
5  class ModelMNISTSmall(torch.nn.Module):
6      def __init__(self, mock: bool):
7          super().__init__()
8          self.mock = mock
9          if (mock):
10             hxtorch.init(hxtorch.MockParameter(noise_std = 1.6, gain = 0.0018))
11         else:
12             hxtorch.init()
13
14         self.fc_1 = hxtorch.nn.Linear(28 * 28, 64, ..., mock = mock)
15         self.fc_2 = hxtorch.nn.Linear(64, 10, ..., mock = mock)
16         self.relu = hxtorch.nn.ConvertingReLU(mock = mock);
17
18         # use specific matmul operations for efficient hardware execution
19         self.fc_1._matmul = hxtorch.mac_single_chip
20         self.fc_2._matmul = mnist_matmul_out
21
22     def forward(self, x):
23         x = x.view(-1, 28 * 28)
24         x = self.fc_1(x)
25         x = self.relu(x)
26         x = self.fc_2(x)
27         return x
28
29     def serialize_to_grenade(self, destination_file: Path):
30         dummy_inputs = torch.zeros((1, 28 * 28), dtype = torch.float)
31         hxtorch.no_init()
32
33         tracer = hxtorch.InferenceTracer(str(destination_file.resolve()))
34         tracer.start()
35         hxtorch.argmax(self(dummy_inputs), dim = 1, keepdim = True)
36         tracer.stop()
37         hxtorch.release()
```

Listing 3.3: HxTorch implementation of the MNIST model. Some parameters have been omitted for brevity. The linear layer's internal matrix multiplications are replaced with variants with better runtime-efficiency.

Another difference from the plain PyTorch model (listing 3.1) is the use of a *Converting ReLU*. In the hardware-based forward-pass, this ReLU scales its result to 5 bit accuracy, so that it can be directly fed into the next layer while in mock-mode it just applies the linear rectifier. Section 2.6 describes the training procedure. To trace the model's execution graph after training, the serialization method in listing 3.3 is executed on a trained model using hardware execution. For tracing, a vector of dummy inputs is fed into the model while the *InferenceTracer* logs all instantiations of computables used in the model's execution. This sequence of computables and their parameters is then serialized into a binary file.

3.4 Experiment Execution

This section describes in detail the steps which are taken at runtime to execute the model with live image data input. The program (*runner*) is implemented in C++ for increased performance and native interfacing with both the V4L2 API and grenade.

The runner is invoked with file paths to a setup calibration, the traced model sequence and an output file to which timing information is logged during the execution. Firstly, a connection to the webcam is established and the ring-buffer is allocated. Then, the model sequence is loaded and converted into an execution graph to which the calibration is prepended. This graph is then transformed into playback-programs via grenade's PPU mastered sequence-graph-compiler as outlined in section 2.5.3. Afterwards, a connection to the BSS-2 hardware is established and the initialization program is executed, which applies all static configuration to the chip, including the model's weights.

Following the initialization procedure, the program enters its main loop, which gracefully exits when receiving a `SIGINT` interrupt signal. In the main loop, an image is captured at the webcam's minimum resolution and converted into a monochrome image which is then downsampled to 28 x 28 px. This image is then pre-processed to obtain the model's input data. The compiled graph's load-step then compiles input data into a playback-program, which is executed on the hardware followed by the execution program. After the on-chip operations are complete, the store program is executed to obtain the model's prediction. Listing 3.4 shows the execution steps for a single frame after image acquisition and pre-processing.

```
CamdigitsRunner::output_data_t CamdigitsRunner::run_inference(
    input_data_t const& data)
{
    // Compile load-program
    std::vector<input_data_t> inputs(1, data);
    auto copy_data_program = compiled_graph->load(inputs).done();

    // Execute playback-programs
    stadls::vx::v2::run(connection, copy_data_program);
    stadls::vx::v2::run(connection, run_inference_program.value());
    stadls::vx::v2::run(connection, copy_results_program.value());

    // Extract result
    auto const result_variant = compiled_graph->process_store();
    auto const result = std::get<std::vector<std::vector<output_data_t>>>(
        result_variant);

    return result.at(0).at(0);
}
```

Listing 3.4: Inference of a single pre-processed frame on the hardware. Profiling and validation has been omitted for brevity.

To make quantitative statements on the runner's performance and to identify potential bottlenecks, it is necessary to measure the runtime of certain parts of the inference loop. This can be achieved very simply by noting the time just before and after the interesting section and calculating their difference as shown in listing 3.5.

```
using namespace std::chrono;
auto const before = high_resolution_clock::now();

// Section, for which the runtime will be measured
runner.run_inference(input_data);

auto const after = high_resolution_clock::now();
auto const duration_us = duration_cast<microseconds>(after - before).count();
```

Listing 3.5: Measuring the runtime of a code section.

However, special care should be taken to avoid creating noticeable overhead with the measurement itself, since retrieving the current time and calculating the difference also requires some CPU cycles. A way to avoid this is choosing sufficiently coarse sections of code so that the measurement overhead becomes negligible with respect to the section's runtime.

3.5 Data Flow

Analyzing the path data takes through the system's components during inference provides insight into potential performance bottlenecks as well. In contrast to measuring the runtime of certain operations during a fixed task as presented in the previous section, a more general analysis allows extrapolation to previously unseen usage patterns.

During the initialization phase, the data flow resembles traditional usage of the Brain-

ScaleS system. The PPU program and containers for the hardware parameters are packed into a playback-program, which is uploaded to the FPGA's playback buffer through the BSS-2 mobile system's AXI interface (see fig. 2.4). The executor is then triggered, processes the playback-program and, upon completion, signals this to the framework as described by the authors of Müller et al. (2020b).

The real-time part of the program follows a slightly different pattern since it is optimized for high throughput. When running directly on the BSS-2 mobile system's ARM CPU, the execution runtime can directly write data to the portion of DRAM, which is used as the PPU's external memory. This significantly simplifies the data path for loading data onto the hardware, as no containers need to be compiled into a playback-program which is then sequentially executed on the FPGA. As described in section 2.2, the BSS-2 mobile system can also be used over the network via *quiggeldy* with an external host computer running the experiment. In this case, no direct access to the DRAM is possible and the runtime falls back to container-based data transfer.

The data path from external memory into the analog core using the vector generator has already been outlined in section 2.3.1. It is possible to transfer data directly and asynchronously from memory into the vector generator as has been demonstrated in Stradmann et al. (2021). However, that requires the FPGA to provide a direct memory access (DMA) stream for each hemisphere of the chip and the integration of this data flow is not in scope for this thesis. Instead, each PPU reads the data from external memory and writes it into its vector generator's FIFO, effectively replacing the DMA engine.

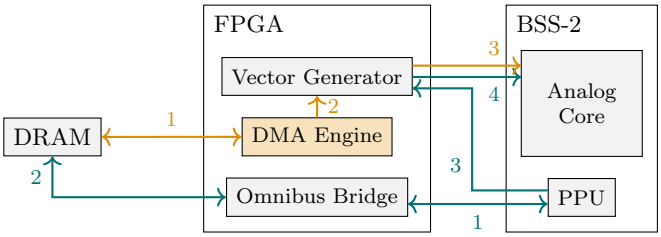

Figure 3.5: Data flow during the real-time-portion of the execution. Blue arrows show the data flow for PPU-based access to the vector generator, whereas orange ones show DMA-based operation (not used in this thesis).

The DMA-based operation offers a performance advantage, particularly for big chunks for input data, since the PPU is not involved in the load process. But for the relatively small data chunks (784 bytes per frame) processed in the MNIST model, this inefficiency is not critical.

4 Results

In this chapter, the results will be presented with a focus on runtime performance starting with a demonstration of the newly implemented MAC operation. Then, the MNIST model's accuracy in different execution environments is presented. Finally, results of profiling the inference loop are shown.

4.1 MAC Operation

Results from testing the `MACSingleChip` operation presented in section 3.3 are discussed in more detail in this section focusing on two aspects. Firstly it is important, that it correctly executes the vector-matrix multiplication for input sizes ≤ 256. Secondly, it must yield correct results for inputs that are split across multiple partial MACs which verifies the digital addition of the partial results.

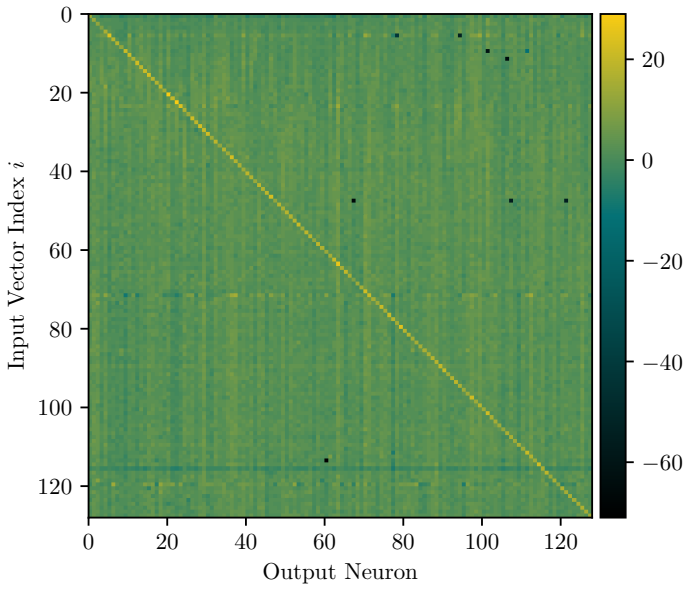

Figure 4.1: 128 x 128 vector-matrix multiplication using the `MACSingleChip` operation. Results for 128 input vectors $(\vec{v_i})_j = (31 \cdot \delta_{i,j})_j$ and a weight-matrix $W_{i,j} = 63 \cdot \delta_{i,j}$. While these dimensions do not use any special placement on the chip, this shows expected results. Note the artifacts for some neurons in some batches.

For verifying the vector-matrix multiplication, it is executed with a fixed weight-matrix for a set of input vectors (fig. 4.1). For example, a simple diagonal test writes a diagonal 20 x 10 weight-matrix to the chip and then sends 20 input vectors of length 20, each with one ascending element at maximum activation and all others at zero activation. This sweep of the input vector verifies the correct functionality of each synapse driver and output neuron used in the MAC operation but only the synapses on the diagonal are tested here. Since the input vectors are converted into spike-events by the vector generator, this test also verifies the correct mapping of vector entries to spike labels, which is illustrated in fig. 4.2.

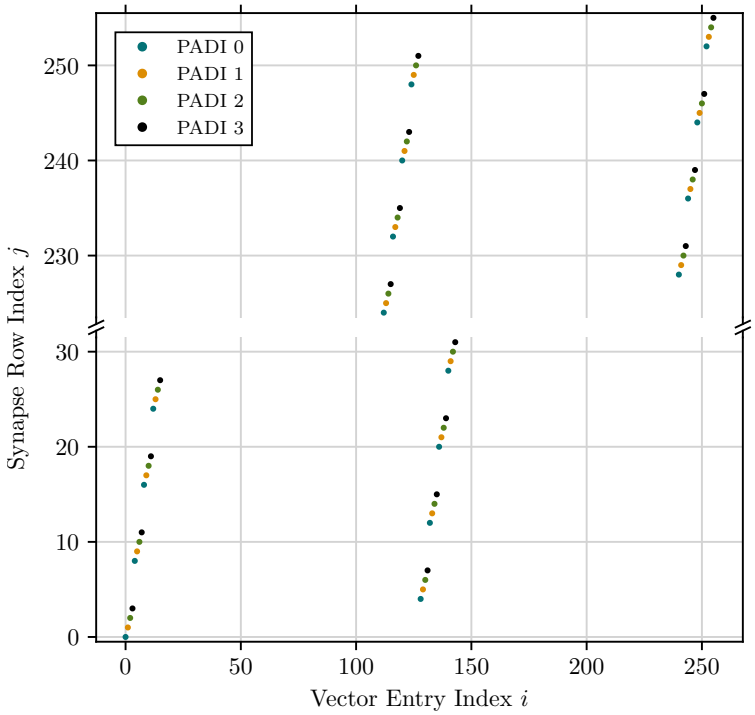

Figure 4.2: Vector generator look-up-table (LUT) entries for an input vector of length 256 using label-bits. Each PADI bus controls 32 synapse drivers, serving two synapse rows each. The driver's index on the PADI is given by $j_{\text{padi}} = \lfloor j \,/\, 4 \rfloor$. The last bit of j_{padi} is used to address two different sets of synapses (each set contains an inhibitory and an excitatory synapse) from the same synapse driver.

Since the `MACSingleChip` operation is designed to be very performant in PPU mastered mode, its performance is also analyzed. Table 4.1 lists execution times for different input dimensions and fixed output dimension of 64. For runs using the JIT executor, the graph is recompiled with new input data during each execution since this is representative of its typical usage as a backend for HxTorch. In the PPU mastered mode, the initialization comprising graph construction, compilation and chip configuration is not measured.

Executor	Input dim.	Partial MACs	Graph Execution [ms]	Exec. on Chip [ms]
JIT	64	1	235.23 ± 3.77	11.80 ± 0.05
	96	1	236.11 ± 2.80	11.82 ± 0.08
	128	1	235.71 ± 3.15	11.81 ± 0.05
	192	1 with labels	267.04 ± 5.63	11.94 ± 0.06
	256	1 with labels	245.37 ± 3.73	11.89 ± 0.05
	512	2 with labels	479.93 ± 8.80	23.78 ± 0.08
	768	3 with labels	486.35 ± 7.82	24.09 ± 0.09
	784	3 with labels + 1	496.58 ± 7.17	24.37 ± 0.12
	1024	4 with labels	502.33 ± 8.22	24.46 ± 0.27
PPU	64	1	4.65 ± 0.34	0.58 ± 0.04
	96	1	4.48 ± 0.14	0.59 ± 0.04
	128	1	4.97 ± 0.43	0.59 ± 0.04
	192	1 with labels	5.86 ± 0.34	0.68 ± 0.04
	256	1 with labels	5.45 ± 0.53	0.68 ± 0.04
	512	2 with labels	6.37 ± 0.47	0.84 ± 0.04
	768	3 with labels	7.30 ± 0.53	0.97 ± 0.04
	784	3 with labels + 1	7.64 ± 0.63	0.96 ± 0.06
	1024	4 with labels	8.44 ± 0.46	1.02 ± 0.05

Table 4.1: Comparison of graph execution times for one `MACSingleChip` operation in JIT and PPU mastered executors with statistics from 100 runs. The output dimension is always 64. Partial MACs describes the number of partial MACs placed on the synram. The graph execution time comprises one execution of the graph with fresh, random data and excludes initialization in PPU mastered mode. The execution time on chip is the cumulative hardware execution time of all playback programs executed.

Here, each execution consists of compiling the load-program with new data and executing it, followed by the execution- and store-program. This is also representative of the operation's usage in the runner.

A significant reduction in execution time for the duration on-chip and the whole graph can be observed when executing via the PPU in comparison to the JIT executor. It can also be observed that with a constant output dimension the execution time only depends on the number of partial MAC operations. An operation using label-bits causes a slight increase in execution time in comparison to the same operation without labels. This is to be expected since the vector generator generates double the input events. This shows, that this input generation is not a significant contributor to the overall execution duration but rather the memory operations and post-processing the membrane potentials.

Finally, the results of the `MACSingleChip` operation are also examined for correctness on larger input dimensions. In fig. 4.3, a vector-matrix multiplication for a 1024 x 64

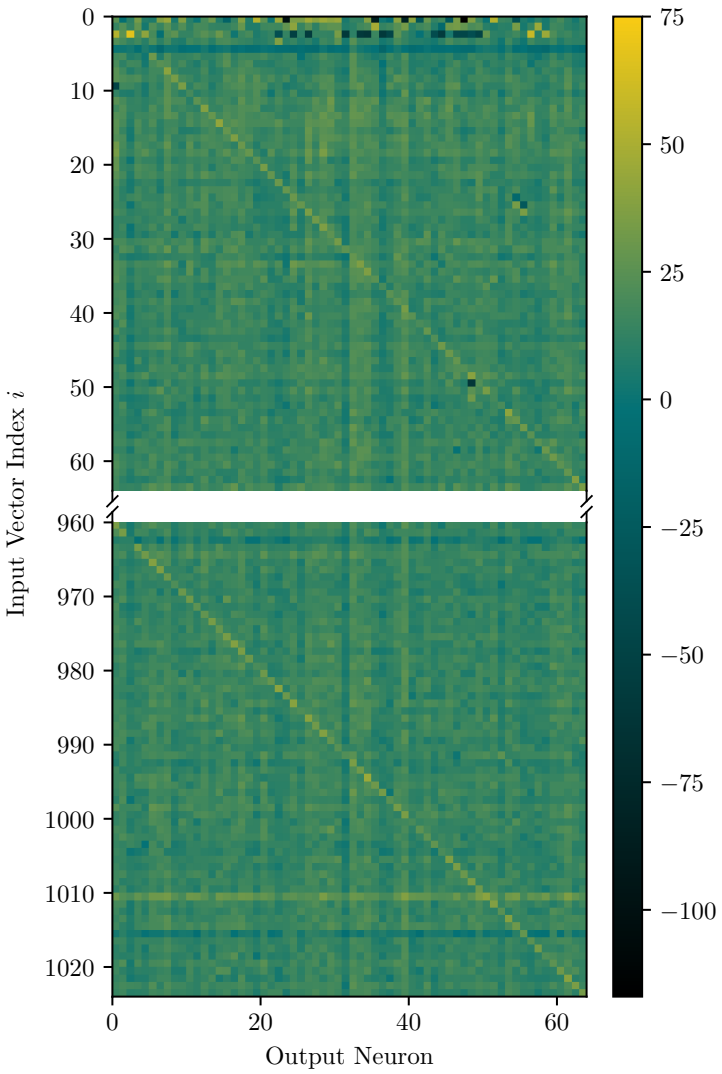

Figure 4.3: Vector-matrix multiplication for a 1024 x 64 matrix $W_{i,j} = 63 \cdot \delta_{(i \bmod 64),\, j}$ using the `MACSingleChip` operation. Results for $i \in [65, 959]$ have been omitted for brevity. Strong artifacts can be noticed on beginning of partial MACs as well as a higher overall noise floor.

matrix is executed in the same way as outlined above. Since the results of this operation are accumulated on 7 sets of 64 neurons (fig. 3.3), which are digitally added afterwards, the overall noise floor is increased in comparison to the single MAC in fig. 4.1, which is to be expected. The result shows the repeating diagonal throughout the full height of the input vector, which confirms that the operation correctly executes the partial MACs and accumulates their results. However, in addition to occasional neurons with strong negative membrane potential, strong artifacts can be seen for the first 5 rows of each new partial MAC ($i \bmod 256 \leq 5$). Possible causes for this are discussed in chapter 5.

4.2 MNIST Results

In this section, the accuracy and training performance of the neural network model is presented. Different implementations are compared in training speed and efficiency as well as in their accuracy on test data. For all implementations, the *Adam Optimizer* [51] is used with a learning rate of $l_r = 10^{-3}$ and a numerical stability factor of $\epsilon = 10^{-7}$. Training is conducted with a batch size of 100 and *CrossEntropyLoss* [52] on the model's prediction against the true labels.

For comparing the training speed of the different implementations, they are executed on the same 8-core CPU (AMD Ryzen 4700u) using 8 threads.

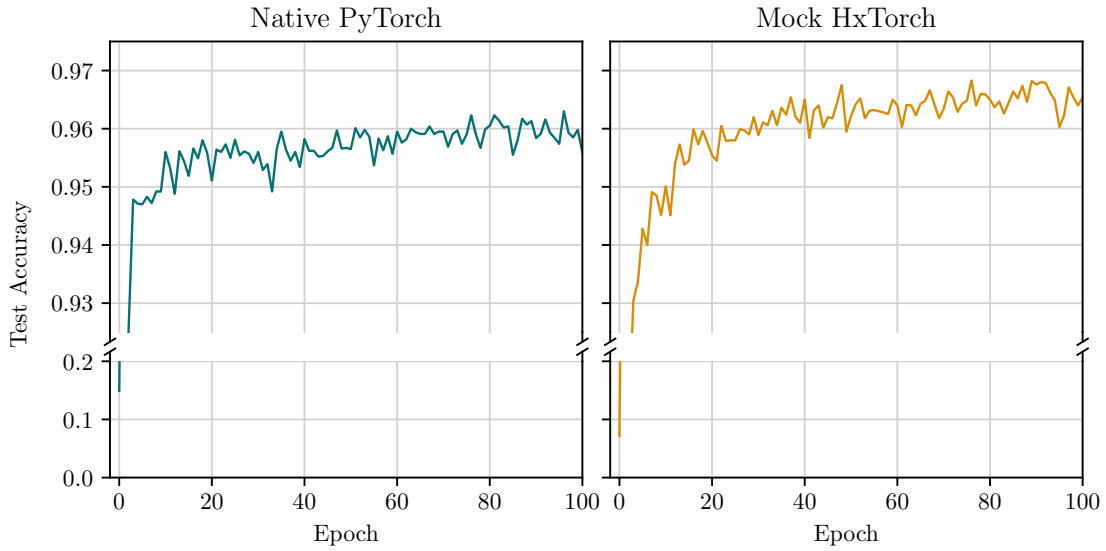

Figure 4.4: Comparison of the model's accuracy on test data during training for native PyTorch and HxTorch running in mock-mode. The initial accuracy before training varies due to the random initialization of weights.

Native PyTorch

The model's implementation using only native PyTorch components presented in listing 3.1 is trained on the MNIST data-set. Its accuracy on test data reaches 95.9 % after about 80 epochs that take an average of 2.87 s. During training, the CPU is fully utilized while running at 3.9 GHz on all cores.

HxTorch in Mock-mode

Using the hardware emulating forward-pass, the model presented in listing 3.3 is trained in the same way. The accuracy on test data is slightly better than the native PyTorch implementation and reaches 96.5 % after around 70 epochs. However, the epochs take considerably longer to compute, averaging 34.89 s while not fully leveraging the available

CPU performance. The training is utilizing only one core to 100 % at 4.2 GHz and three more to ∼60 % at 1.9 GHz.

HxTorch with Hardware in the Loop

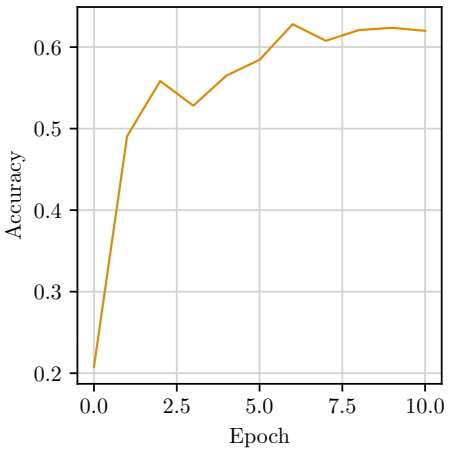

Figure 4.5: Re-training of the HxTorch model on BSS-2 for 10 epochs. Note the low accuracy before the first epoch.

Evaluating the mock-trained model with the forward-pass on BSS-2 yields an accuracy of just 20 % which points to a strong mismatch between the previous emulation and the hardware itself. Re-training the model on hardware for 10 epochs increases its accuracy substantially to 62 % on test data. This on-chip training is limited by the hardware's speed and results in only light CPU utilization. While this performance is well below the expected level for the hardware, it shows the potential for re-training to adapt to slight inaccuracies of the analog substrate.

Further investigation points to a problem with the analog calibration (section 2.1.1) of the setup, whose results are shown in fig. 4.6. If we assume the model to predict the label randomly and the MNIST test-set to be balanced, the expected accuracy of this random model is $P(\hat{y} = i \,|\, y = i) = 10\,\%$. This shows that the trained model is able to make some correct predictions even with non-ideal analog accuracy.

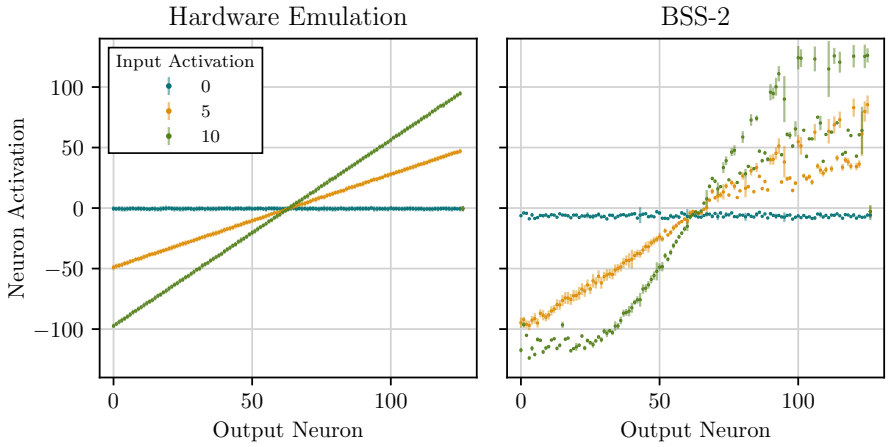

Figure 4.6: Linearity plot of a 128 × 128 matrix multiplication for emulation and hardware execution. The weight matrix is increasing linearly from −63 to +63 from left to right. Output neurons 64 through 128 show strong irregularities.

4.3 Runner Performance

In this section, the performance of the whole inference loop is presented. This includes image acquisition, downsampling, pre-processing and inference on the hardware. The measurements of the step's execution time are performed as described in listing 3.5.

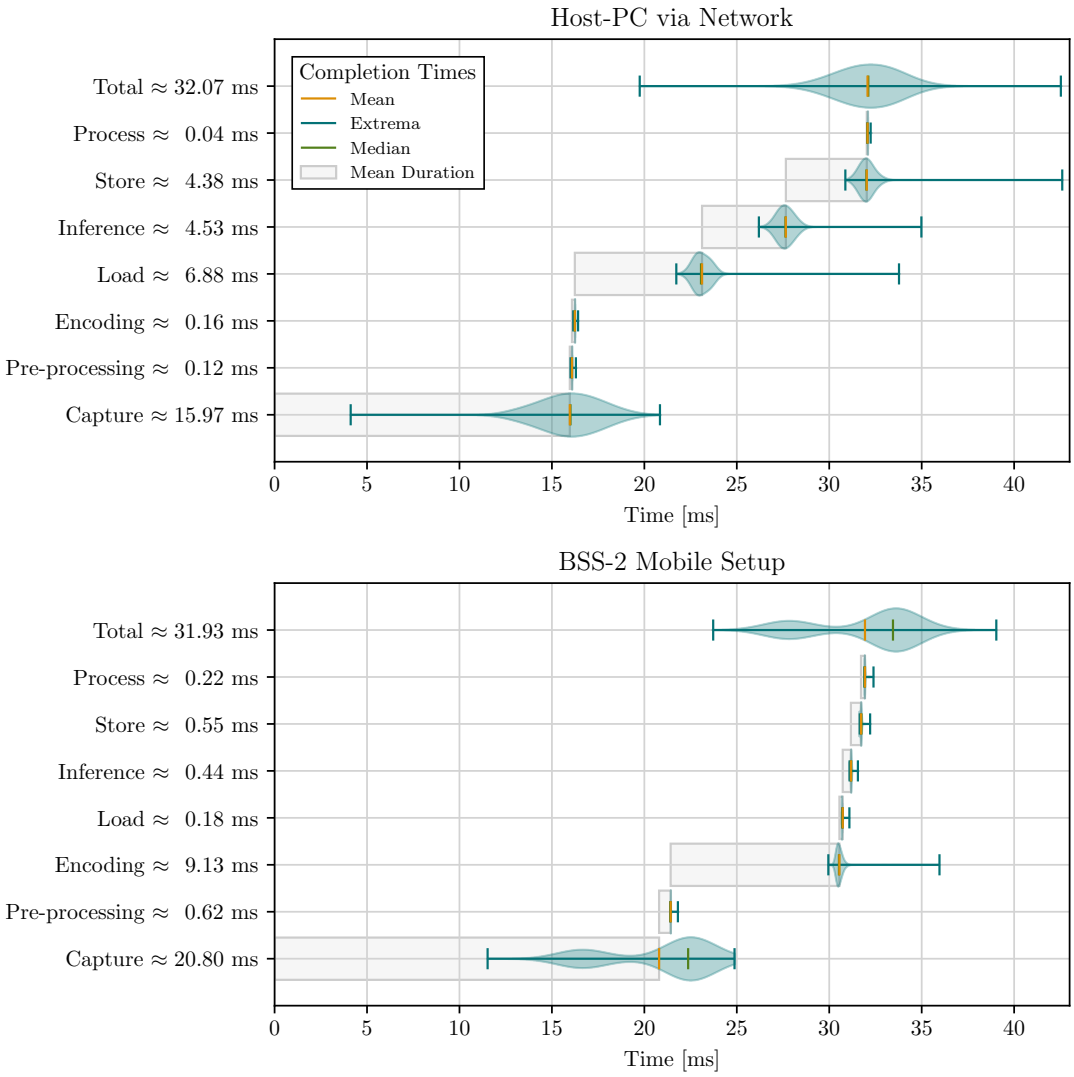

Figure 4.7: Runtime of inference steps with statistics from more than 4000 frames. The violin plots show only the variations in duration of their respective step. Both execution environments use the PPU mastered executor. Note the larger extrema for execution via the network, which rarely reach up to 44 ms for a complete frame.

Figure 4.7 compares the runtime of the PPU mastered inference loop between execution locally on the BSS-2 mobile system and remotely over the network via quiggeldy on

a host computer. Both variants saturate the webcam's 30 FPS, which correspond to 33.3 ms per frame. Since the total inference of a frame takes less than that, the *Capture* step blocks until a new frame is available.

The BSS-2 mobile system has a significantly slower CPU, which is especially noticeable in steps that require computation, such as *Pre-processing* and *Encoding*. The hardware execution steps (*Load*, *Inference* and *Store*) however execute very quickly and more than make up that. The *Load* step is especially quick since the data is not transferred in the playback-program and instead written directly into the FPGA's memory during *Encoding*. Because of this tight coupling of FPGA and CPU, the step's durations don't vary much except for *Capture*, whose distribution shows two distinct accumulations on either side of the mean duration. This points to an intermittent, small delay when accessing the camera's frame buffer, which explains the slower peak. Since the processing of that frame is therefore delayed, the next frame will be available just as much quicker, causing the second, faster peak.

Using a regular host computer and interfacing with the setup via a network interface introduces greater round-trip times for the hardware execution steps. However, the much more powerful CPU is able to quickly pre-process and encode the image data. Spikes in network latency occur sporadically and can delay the processing of some frames, which in turn makes the next frame available more quickly. This is reflected in the distributions of hardware execution steps and frame capture. Noteworthy is also the comparatively long duration for executing the *Load* program, which is most likely due to a larger playback-program being transmitted and executed. This hypothesis is also supported by the measurements in table 4.2 since the load program containing the image data takes twice as long to execute on the chip. Still, this mode of execution is limited by the webcam's frame-rate and not by the inference on BSS-2 hardware.

Host	Load [µs]	Inference [µs]	Store [µs]	Total [µs]
BSS-2 Mobile System	82.3 ± 30.1	421.3 ± 24.8	422.0 ± 40.9	925.6 ± 56.7
x86 Host Computer	191.4 ± 4.5	413.3 ± 10.1	362.1 ± 39.10	966.9 ± 40.6

Table 4.2: Comparison of on-chip execution times for the inference steps from fig. 4.7 for different hosts. Note the significantly quicker *Load* operation for the mobile system.

It should also be noted, that these results are in agreement with table 4.1, which measures the sum of *Encoding* through *Process* as the total graph execution time. Comparing the measurements of on-chip execution times from table 4.2 with previous results shows, that a 784×64 MAC takes 0.96 ms on-chip, while the whole model takes 0.97 ms. This might be surprising, considering the complete model also computes a ReLu, a second small MAC and max-pooling. However, the *Store* operations return the complete MAC result in the isolated test, whereas the model's output is a single number, which makes

up for this difference.

A detailed analysis of the JIT executor's performance with the whole model is not conducted, since previous measurements already rule it out for real-time inference. For example, the 784×64 MAC alone takes around 500 ms for graph-execution and even the on-chip execution time of 24 ms is close to the frame-time limit of 33.3 ms.

5 Discussion and Outlook

In this chapter, the previous results are discussed further and analyzed critically. This thesis presents new hardware operations to enable real-time image recognition on the BrainScaleS-2 mobile system.

To achieve real-time inference, the capabilities and constraints of the analog, neuromorphic substrate are analyzed. Subsequently, a machine learning model capable of classifying the images with good accuracy is developed. Due to the limited number of model-weights available on the chip without reconfiguration, the model's size is reduced to satisfy this constraint. Building on previous techniques, namely the graph-based PPU mastered execution, a new scheme for executing large analog vector-matrix multiplications is presented. This newly developed MAC makes use of efficient placement of partial operations to forego the need for re-configuring the chip's synapse matrix during inference while also enabling simultaneous execution on both hemispheres of the chip. This MAC operation is subsequently validated, which verifies its principal functionality. Its performance is further examined for different input dimensions and compared between the JIT executor and the PPU mastered execution mode. Finally, the real-time inference loop including image capture, pre-processing and execution on the hardware is presented and profiled.

5.1 MAC Operation

As already mentioned in section 4.1, the MAC operation does not perform as expected with regard to the accuracy of its results. This is very obvious in fig. 4.6 and affects only certain portions of the chip. The problem persists across different hardware setups ruling out hardware-related issues indicating a problem with the software. Since the relative timing of event input and neuron readout has to be very precise, this anomaly could be caused by the new operation introducing previously unseen latency to certain steps. However, the existing MAC operations designed for smaller matrices show the same behavior and the calibration routine which does not use the graph-based execution runtime also produces unusual results. This suggests an issue in the underlying hardware abstraction layers. Further investigation of the changes to these lower software layers is ongoing and qualitative analysis of the new MAC operation suggests that it is in principle capable of accurate operation.

5.2 Performance and Data Flow

The performance of the real-time image inference loop is presented in section 4.1. Its runtime performance is shown for JIT execution and in the PPU mastered mode. In the latter, an on-chip runtime of just 1.02 ms for a 1024×64 MAC is observed, which is ideal for runtime-critical applications such as live image recognition. Furthermore, image acquisition and pre-processing steps are outlined and the complete inference loop is profiled. Those results show that the inference loop is capable of processing one frame in under 15 ms on the BSS-2 mobile system, which saturates the webcam with its 30 FPS and theoretically enables frame-rates exceeding 60 FPS. The data also shows that real-time inference is possible with an external host computer that interfaces with the BSS-2 mobile system over the network. While the distribution of runtime with regard to the processing steps deviates from the completely local inference, the complete processing is only slightly slower and sufficient for real-time operation.

Potential for further improvement lies with the transfer of image data to the chip since it is currently processed by the graph-based executor framework and makes up a large portion of the processing time. Since the system is already using an FPGA-based controller, the captured frames could be fetched from memory and pre-processed using dedicated logic. This approach combines well with the DMA-based data flow outlined in fig. 3.5. Better performance is expected for this mode of operating the vector generator since it forgoes the need to copy the input data with the PPU. However, this would require multi-channel DMA to operate both hemisphere's vector generators simultaneously and implementation of the pre-processing in register-transfer-level (RTL) logic.

Future work could also develop a completely PPU mastered inference loop. Quite a substantial performance gain is expected since it eliminates the *Load* and *Store* steps from the inference loop and allows the CPU to pre-process frames in parallel to the on-chip inference. Additionally, the overhead introduced by decoding playback-programs and encoding trace data as well as their transfer can be avoided.

5.3 Local Execution

The presented hardware operations allow for standalone operation of the BSS-2 mobile system with a larger set of models. Its average energy consumption of 5.6 W for the complete system and 700 mW for the BSS-2 ASIC as measured by Stradmann et al. (2021) makes it well suited for use in edge computation devices or in densely packed data-centers. With classification taking 15 ms as demonstrated, the energy required to process one image is $E_{\text{System}} = 5.6\,\text{W} \cdot 15\,\text{ms} = 84.0\,\text{mJ}$ and $E_{\text{ASIC}} = 10.5\,\text{mJ}$ respectively. This power can easily be provided by a battery for portable applications like quad-copters or rovers, which opens up possibilities for interesting research projects.

5.4 Limitations

The main limitation of this approach to real-time inference is its constraint to models that do not exceed the size of the chip. Larger models require reconfiguration of the weight matrix. Grenade currently only implements access to the synapse weights via the narrow slow control data path, where updating the complete synapse array requires around 5 ms as measured by Weis et al. (2020). However, the PPU's vector unit is connected to the synapses in a highly parallel manner and can write one complete row of weights in 20 clock-cycles as measured by Spilger (2021b). Assuming immediate access to the new configuration data (possible with pre-fetching) and a frequency of 250 MHz for the PPU, this corresponds to 80 ns. Thus, a complete reconfiguration of the synapse array consisting of 256 unsigned rows per hemisphere is possible in just 20.5 µs. Integration of this update mechanism into the software abstraction layers has great potential for future work on larger models.

6 Acknowledgements

The work carried out in this report used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP), from the BMBF (16ES1127), and from the Lautenschläger-Forschungspreis 2018 for Karlheinz Meier.

Bibliography

Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

Sparsh Mittal. A survey of fpga-based accelerators for convolutional neural networks. *Neural Computing and Applications*, 32(4):1109–1139, Feb 2020. ISSN 1433-3058. URL https://doi.org/10.1007/s00521-018-3761-1.

Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. A survey of accelerator architectures for deep neural networks. *Engineering*, 6(3):264–274, 2020. ISSN 2095-8099. URL https://www.sciencedirect.com/science/article/pii/S2095809919306356.

Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2020.

Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. Techology trend of edge ai. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–2, 2018.

Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.

Jianxin Zhao, Richard Mortier, Jon Crowcroft, and Liang Wang. Privacy-preserving machine learning based data analytics on edge devices. In *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, AIES '18, page 341–346, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360128. URL https://doi.org/10.1145/3278721.3278778.

Chetan Singh Thakur, Jamal Molin, Gert Cauwenberghs, Giacomo Indiveri, Kundan Kumar, Ning Qiao, Johannes Schemmel, Runchun Wang, Elisabetta Chicca, Jennifer Olson Hasler, Jae sun Seo, Shimeng Yu, Yu Cao, André van Schaik, and Ralph Etienne-Cummings. Large-scale neuromorphic spiking array processors: A quest to mimic the brain, 2018, 1805.08932.

Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A survey of neuromorphic computing and neural networks in hardware, 2017, 1705.06963.

J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS"10)*, pages 1947–1950, 2010.

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. *Frontiers in Neuroscience*, 13:260, 2019. ISSN 1662-453X.

Sebastian Billaudelle, Yannik Stradmann, Korbinian Schreiber, Benjamin Cramer, Andreas Baumbach, Dominik Dold, Julian Göltz, Akos F. Kungl, Timo C. Wunderlich, Andreas Hartel, Eric Müller, Oliver Breitwieser, Christian Mauch, Mitja Kleider, Andreas Grübl, David Stöckel, Christian Pehle, Arthur Heimbrecht, Philipp Spilger, Gerd Kiene, Vitali Karasenko, Walter Senn, Mihai A. Petrovici, Johannes Schemmel, and Karlheinz Meier. Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate, 2019, 1912.12980.

A. Joubert, B. Belhadj, O. Temam, and R. Héliot. Hardware spiking neurons design: Analog or digital? In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5, 2012.

Sebastian Schmitt, Johann Klaehn, Guillaume Bellec, Andreas Gruebl, Maurice Guettler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Vitali Karasenko, Mitja Kleider, Christoph Koke, Christian Mauch, Eric Mueller, Paul Mueller, Johannes Partzsch, Mihai A. Petrovici, Stefan Schiefer, Stefan Scholze, Bernhard Vogginger, Robert Legenstein, Wolfgang Maass, Christian Mayr, Johannes Schemmel, and Karlheinz Meier. Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system. *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks*, 2017. URL `http://ieeexplore.ieee.org/document/7966125/`.

Johannes Schemmel, Sebastian Billaudelle, Phillip Dauer, and Johannes Weis. Accelerated analog neuromorphic computing, 2020, 2003.11996.

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*, volume 1325, pages 201–212. Springer International Publishing, 2020.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen

King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. ISSN 1476-4687. URL `https://doi.org/10.1038/nature14236`.

Yannik Stradmann, Sebastian Billaudelle, Oliver Breitwieser, Falk Leonard Ebert, Arne Emmel, Dan Husmann, Joscha Ilmberger, Eric Müller, Philipp Spilger, Johannes Weis, and Johannes Schemmel. Demonstrating analog inference on the brainscales-2 mobile system, 2021, 2103.15960.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Coral. Coral mini pcie accelerator. `https://coral.ai/docs/mini-pcie/datasheet/`. [Online; accessed 24/06/2021].

Masatoshi Yamaguchi, Goki Iwamoto, Hakaru Tamukoh, and Takashi Morie. An energy-efficient time-domain analog vlsi neural network processor based on a pulse-width modulation approach, 2019, 1902.07707.

Simon Friedmann, Johannes Schemmel, Andreas Grübl, Andreas Hartel, Matthias Hock, and Karlheinz Meier. Demonstrating hybrid learning in a flexible neuromorphic hardware system. *IEEE Transactions on Biomedical Circuits and Systems*, 11(1):128–142, 2017.

Syed Aamir, Paul Müller, Gerd Kiene, Laura Kriener, Yannik Stradmann, Johannes Schemmel, and Karlheinz Meier. A mixed-signal structured adex neuron for accelerated neuromorphic cores, 04 2018.

Louis Lapicque. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen.*, 9:620–635, 1907.

Johannes Schemmel, Daniel Bruderle, Karlheinz Meier, and Boris Ostendorf. Modeling synaptic plasticity within networks of highly accelerated i f neurons. In *2007 IEEE International Symposium on Circuits and Systems*, pages 3367–3370, 2007.

Matthias Hock, Andreas Hartel, Johannes Schemmel, and Karlheinz Meier. An analog dynamic memory array for neuromorphic hardware. In *2013 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, 2013.

Johannes Schemmel, Laura Kriener, Paul Müller, and Karlheinz Meier. An accelerated analog neuromorphic hardware system emulating nmda- and calcium-based non-linear dendrites, 2017, 1703.07286.

Aron Leibfried. On-chip calibration of analog neuromorphic circuits. Bachelor's thesis, Universität Heidelberg, 2018.

Johannes Schemmel, Steffen Hohmann, Karlheinz Meier, and Felix Schürmann. A mixed-mode analog neural network using current-steering synapses: Special issue on current mode circuit techniques. *Analog Integrated Circuits and Signal Processing*, 38, 02 2004.

Twisha Titirsha, Shihao Song, Adarsha Balaji, and Anup Das. On the role of system software in energy management of neuromorphic computing. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 124–132, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384049. URL `https://doi.org/10.1145/3457388.3458664`.

Saber Moradi and Giacomo Indiveri. An event-based neural network architecture with an asynchronous programmable synaptic memory. *IEEE Transactions on Biomedical Circuits and Systems*, 8(1):98–107, 2014.

Eric Müller, Sebastian Schmitt, Christian Mauch, Sebastian Billaudelle, Andreas Grübl, Maurice Güttler, Dan Husmann, Joscha Ilmberger, Sebastian Jeltsch, Jakob Kaiser, Johann Klähn, Mitja Kleider, Christoph Koke, José Montes, Paul Müller, Johannes Partzsch, Felix Passenberg, Hartmut Schmidt, Bernhard Vogginger, Jonas Weidner, Christian Mayr, and Johannes Schemmel. The operating system of the neuromorphic brainscales-1 system, 2020a, 2003.13749.

Eric Müller, Christian Mauch, Philipp Spilger, Oliver Julien Breitwieser, Johann Klähn, David Stöckel, Timo Wunderlich, and Johannes Schemmel. Extending brainscales os for brainscales-2, 2020b, 2003.13750.

Electronic Vision(s) Group. Coordinates for hicann-based and hicann-dls-based neuromorphic systems. `https://github.com/electronicvisions/halco`, 2021a. [Online; accessed 25/06/2021].

Electronic Vision(s) Group. Fpga instruction set compiler for hicann. `https://github.com/electronicvisions/fisch`, 2021b. [Online; accessed 25/06/2021].

Vitali Karasenko. *Von Neumann bottlenecks in non-von Neumann computing architectures*. PhD thesis, Universität Heidelberg, 2020.

Electronic Vision(s) Group. Hardware abstraction layer (and stateful encapsulation) for the hicann-dls. `https://github.com/electronicvisions/haldls`, 2021c. [Online; accessed 25/06/2021].

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. *Front. Neuroinform.*, 2, 2008.

Avnet Inc. Ultra96-v2 - xilinx zynq mpsoc development board. `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/`, 2020. [Online; accessed 25/06/2021].

Bill Dirks, Michael H. Schimek, Hans Verkuil, Martin Rubli, Andy Walls, Muralidharan Karicheri, Mauro Carvalho Chehab, Pawel Osciak, Sakari Ailus, and Antti Palosaari. Video for linux api version 2 (v4l2 api) specification. `https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html`. [Online; accessed 07/06/2021].

Oliver Breitwieser. *[unpublished doctoral dissertation]*. PhD thesis, Universität Heidelberg, 2021.

Marco Rettig. Characterizing the event interface of the hicann-x. Bachelor's thesis, Universität Heidelberg, 2019.

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 H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

Horace He. The state of machine learning frameworks in 2019. *The Gradient*, 2019.

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 – perceptrons on analog neuromorphic hardware, 2020, 2006.13138.

Electronic Vision(s) Group. Pytorch for brainscales-2. `https://github.com/electronicvisions/hxtorch`, 2021d. [Online; accessed 25/06/2021].

Philipp Spilger. From neural network descriptions to neuromorphic hardware — a signal-flow graph compiler approach. Master's thesis, Universität Heidelberg, 2021a.

Electronic Vision(s) Group. Graph-based experiment notation and data-flow execution. `https://github.com/electronicvisions/grenade`, 2021e. [Online; accessed 25/06/2021].

Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Yann Lecun, Larry Jackel, Corinna Cortes, John Denker, Harris Drucker, Isabelle Guyon, Urs Muller, Eduard Sackinger, Patrice Simard, and Vladimir Vapnik. Learning algorithms for classification: A comparison on handwritten digit recognition. *The Statistical Mechanics Perspective*, 07 2000.

Torch Contributors. Adam - pytorch 1.9.0 documentation. `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`, 2019a. [Online; accessed 18/06/2021].

Torch Contributors. Crossentropyloss - pytorch 1.9.0 documentation. `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`, 2019b. [Online; accessed 18/06/2021].

Phillip Spilger. Private communication, 2021b.

Appendix

Software Versions

Table 6.1 lists the repositories used throughout this thesis and their respective version (identified by the `git` commit-hash). The `model-hw-camdigits` repository contains the model implementation and the runner program, which is also interfacing with the camera. Additional instructions for configuring the BrainScaleS-2 mobile system can be found in the documentation of the `model-hw-hdbioai` repository.

All software was run inside the container built on June 16th 2021.

Repository	Commit-Hash
hate	`c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1`
model-hw-camdigits	`0d02c3e22ba3d010da45b921c4796ab027aae630`
hxcomm	`19bdf2a67352ecca1f6616b5a414ea3a3f7e4862`
grenade	`30205b7e54da7f943d18b6b9e9d97d87bfc2a668`
logger	`bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd`
code-format	`5d55a9952d4b6400fa5b2baeff9be546e45bf76d`
haldls	`cccab5f722d0515c925cf465d4b380332e70911e`
calix	`9de767d8508e1e39c2ac785c95cea04ac4863ef0`
hxtorch	`2e20bd1780824dac08917ed61641d35c4a7336a2`
sctrltp	`59a991f6d85ceaf81dbcf8958969a724958aba3c`
rant	`4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d`
hwdb	`9355f93596fcff2ab05973a63b62fb87b1bf6671`
visions-slurm	`5e7ea560235b068fc12f26e3f0d002d415f76cf9`
flange	`fcde2aafe69805487789ca0b1a8a245caf5fb8ed`
lib-rcf	`5b16326ae30ee08a322a6569887ca8bd2684c252`
halco	`5410b82a0f7a2e732913204a974023577c80850e`
libnux	`9a47fe6daf7298697c650029f72d81035df37197`
fisch	`5c59c0902ba2e4c4c9edf50b6d6b31ab00ef6700`
ztl	`d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6`
pywrap	`83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb`
lib-boost-patches	`2d7e07d4e74827c42d9e1a51f8d180af9907f7cb`
waf	`816d5bc48ba2abc4ac22f2b44d94d322bf992b9c`

Table 6.1: Version control commit-hashes for the repositories used in this thesis.

Erklärung

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

Heidelberg, den 30.06.2021,