Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis in Physics
submitted by

Mathis Kunold

born in Leimen (Germany)

2020

Towards Structured Data Exchange in Distributed Neuromorphic Systems

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

Abstract

As a step towards structured data exchange of different components used in neuromorphic systems, a serialization library called `bitsery` is tested on a HICANN-DLS v2 prototype setup designed by the Electronic Vision(s) Group during this thesis. This is done to see how it could potentially be used in future distributed neuromorphic systems, where multiple processor units perform different experiment control tasks.

To examine how the software performs under the constraints given by the ASIC, multiple tests will be conducted to ensure functional data exchange. These standalone tests are then concluded with the application of `bitsery` for parameter transfer in the NSEM (Neuromorphic Spike-based Expectation Maximization) experiment.

Contents

1 Introduction

Both to further our undestanding of brain by emulating physical processes happening in its cells, and as a different approach to the field of machine learning that has seen major advances and lots of practical uses in the past decade, neuromorphic hardware offers a lot of new possibilities.

Analog neural networks like the ones built into various systems designed by the Eletronic Vision(s) Group are able to capitalize on some of the same advantages the brain has over the conventional Von-Neumann-architecture that is basis of most modern computers. Such advantages are lower power consumption and increased robustness, meaning that if one of many artificial neurons on a system is defective, the others continue to work just fine, while if the Central Processing Unit of ones desktop computer breaks, the whole systems is no longer able to continue its operation.

That is not to say conventional computer architecture is simply inferior to the brain and systems that mimic its function in every way, to illustrate: what is $1750303 * 4614944$ and what did you have for lunch on the 6th of march in 2015? For all of its amazing feats, the human brain is not nearly as good at arithmetic and storing information as a conventional computer. To reap the benifits of both of those computing paradigms, the hybrid chips of the Electronic Vision(s) Group combine an analog neural network with an embedded processor called the PPU (Plasticity Processing Unit).

One of the central motivations for this development is the simulation of biological learning. Plasticity and thus learning in the brain is something that happens locally, between individual neurons or population of them. Adding the PPU as an extra element for experiment control closer to the analog core helps to decentralize the learning processes implemented in a simulation, thus allowing more flexibility and biological realism. With distributed experiment control arises the need for efficient communication between its components in order to set parameters for task and to extract data.

In the current setups the chip is connected to a *host computer* which is used to control the experiment via a *Field-Programmable Gate Array* (FPGA), in this context communication between PPU and host is the primary use case, but future setups will be more complex. While the HICANN-DLS (Hight Input Count Analog Neural Network - Digital Learning Systems) line of prototype chips features one PPU [1], the newer HICANN-X has two of them [2], and plans are being made to build setups with multiple of those HICANN-X chips.

An obstacle for communication between the host and the PPU lies in differences in their respective processor architecture. These differences effect how the processors read sections of memory, thus not allowing it to just copy a data structure from one systems memory to the other. In order to transmit data structures, they need to be reshaped into a one-dimensional format from which the recieving system can then reconstruct them

again. Translating data into such a format is called *serialization*, the subsequent reconstruction back into structures the processor can work with is known as *deserialization*. To achieve this, an open source C++ library called `bitsery` will be used to serialize data. Upon verifying that everything goes as expected the time required for the serialization and deserialization of different data structures can be measured to approximate the speed of those processes.

After verification and benchmarking, the next step is to utilize bitsery in actual experiment where the PPU has to perform its originally intended purpose of making calculations to adapt the synaptic weights in the neuromorphic part of the chip. Serialization will be performed together with that to both get parameters for the experiment onto the memory of the PPU as well as to send results back to the host for data analysis.

The experiment chosen for this test is the Neuromorphic Spike-based Expectation Maximization [3]. Note that the overall experiment does not change, only the method of setting parameters, hence the results should be the same as in previous iterations of the experiment.

2 Methods

2.1 Theoretical Background

2.1.1 The Leaky Integrate and Fire Neuron

To emulate the biophysics of a brain, the analog neural networks found on the chips consist of integrated circuits that approximate the theoretical behavior of neurons and synapses. While other modes of operation are possible, the one used here is based on the Leaky Integrate and Fire (LIF) model [4]. It is well suited for simulations since it requires only one differential equation. This equation describes the electric potential u on the cell membrane of a neuron.

$$C_m \frac{d}{dt} u = g_L(u - E_L) + I(t)$$

where C_m is the capacitance of the membrane, g_L the conductance of the leak current, E_L the leak potential and $I(t)$ an external input current. With the membrane time constant $\tau_m = \frac{C_m}{g_L}$ this equation can also be written as

$$\tau_m \frac{d}{dt} u = (u - E_L) + \frac{I(t)}{g_L}$$

In practice, this results in a behavior where the membrane potential exponentially decays to reach the leak potential if no external stimulus is given. The time constant of this exponential function is τ_m. If there is an external current, it will force the membrane potential in its direction. Upon reaching the threshold V_{thres}, the neuron spikes and its membrane potential forced to the value V_{reset}, where it is held for the refractory time τ_{ref}, after which it will again follow the dynamics outlined in the differential equations above.

2.1.2 Current based Synapses

The dynamics of the synapses connecting these neurons are described by the Current Based (CuBa) model. Just as the neuron models, this is a property of the neuromorphic hardware as the synapses are also physically represented by analog circuits on the chip. In this model, the input current $I(t)$ is divided into

$$I(t) = I^{ext} + I^{syn}$$

where I^{ext} is a arbitrary outside current, I^{syn} can be well described by looking at the neurons connected and their spiking behavior. Assuming a set of neurons i is connected

with synaptic weights w_i and send sets of spikes at the times t_i^s, then the synaptic input current to the neuron we are observing is

$$I^{syn}(t) = \sum_i \sum_s w_i \epsilon(t - t_i^s)$$

In the case of an exponential synapse, the synaptic kernel ϵ is an exponential decay starting at t_i^s with the synaptic time constant τ_{syn}. Using this the expression above becomes

$$I^{syn}(t) = \sum_i \sum_s w_i exp\left(-\frac{(t - t_i^s)}{\tau_{syn}}\right) \Theta(t - t_i^s)$$

When looking at networks of interconnected neurons, the synaptic weights are denoted in a matrix $(w_{i,k})_{i,k}$ where one entry is the weight associated with the connection from neuron i to neuron k.

2.1.3 Spike-based Expectation Maximization

The **Expectation Maximization** (EM) algorithm is an iterative method to find a local maximum likelihood approximation of a statistical model with latent variables [5]. Models for biological neural networks are often not deterministic to reflect high complexity and stochatic background present in the brain, therefore statistical methods like this can be useful for such models.

In the following we assume probability distributions of the shape

$$p(\boldsymbol{z}, \boldsymbol{y}|\theta)$$

where $\boldsymbol{y}$ are observables, $\boldsymbol{z}$ the latent variables and θ is the vector of parameters. One iteration t of the EM algorithm consists of two steps: first the expectation step (E-step) where from a distribution with with a set of parameters $\theta^{(t)}$ a function

$$Q(\theta|\theta^{(t)} = \langle lnp((y)|(z)|\theta)\rangle_{(z)|(x),\theta^{(t)}}$$

is calculated. The parameter set that maximizes this function is then chosen in what is called the maximization step (M-step) as to be used in the next iteration.

$$\theta^{(t+1)} = argmax_\theta Q(\theta, \theta^{(t)})$$

When repeating these steps, the distributions using the new parameters will converge towards a maximum likelihood approximation for the model.

In the NSEM experiment, a neural network is modeled as a *Boltzmann machine* where each neuron has a binary random variable asigned to it with the value 1 when firing and 0 when inactive. The parameters of the probability distribution are the synptic weights. The details of the implementation of NSEM can be found in [3].

2.2 Hardware

The type of chip used in all subsequent tests and experiments is the HICANN-DLS (Hight Input Count Analog Neural Network - Digital Learning System) prototype version 2. While not the newest hybrid chip available, it is overall very stable and has been tried and tested before. Its latest successor is the HICANN-X, which includes a non-spiking operating mode similar to a chip called HAGEN (Heidelberg AnaloG Evolvable Neural Network) and a second PPU on top of the features already present in HICANN-DLS chips. The X in the name refers to the HAGEN eXtension [2]. Since the scalar section of the PPU is not subject to any changes, so much the existing software framework made for HICANN-DLS as well as the software examined throughout this thesis can also be used on HICANN-X.

2.2.1 Analog Neural Network

In the neuromorphic section of the HICANN-DLS v2, neurons and synapses are physically emulated by microscopic analog integrated circuits where electrical elemts are used to represent their biological counterparts, such as leak conductance and membrane capacity. The chip features 32 neurons and 32 rows of current based synapses, with one synapse for every neuron each row, making for a total of 1024. A row of synapses can be set to inhibitory or exitatory. When a spike is routed to a neuron, the signal is first sent to one or more synapse rows, where it is passed on to the correct synpse based on a 6 bit address. The weight of each synapse is also encoded in 6 bits, when a spike is routed to the correct synpse, it will generate a current pulse based on its weight. This pulse will then travel to the neuron at the bottom of the synapse column where it serves as input, depending on which the neuron located there may or may not output a spike, if it does the whole process of a spike being routed to the connected synapses can begin once more if further connections for the signal to be routed to exist.

Each of the synapse rows is connected to the CADC (Correlation Analog to Digital Converter), where both a causal and an anti-causal correlation measurements are taken and digitized with a resolution of 8 bits. For both the causal and anti-causal correlation measurements there are 32 channels respectively, onto which the 32 synapse rows are multiplexed. The rates at which neurons output spikes are counted digitally by 10 bit rate counters.

The circuits of the analog neural network do not have the same membrane capacity and leak conductance as the biological neurons they represent. Since the membrane time constant τ_m depends on those two parameters, the artificial neurons are accelerated by an order of $O(10^3)$ compared to biological neurons.

2.2.2 Plasticity Processing Unit

The Plasticity Processing Unit (PPU) is what makes the HICANN-DLS a hybrid of neuromorphic and von Neumann architecture. It is a custom designed processor using the Power Instruction Set Architecture with a general purpose scalar part as well as a

vector unit that is able to process either 8 × 16 bit or 16 × 8 bit vectors. Through the vector unit, the PPU is able to process more data in parallel, which is very useful for its inteded purpose: making computations to update synaptic weights. These have the form of matrices, so it is more efficient to process them in parallel compared to making calculations with a single scalar arithmatic unit. Since it is connected to the neuromorphic region of the chip, the PPU can be used to configure and read out networks running there. With this, the chip is able to emulate synaptic plasticity and thus learning while having a higher degree of independance from the host.

It has a 16 kB memory and uses a 32 bit Power Instruction Set Architecture (Power ISA) [6]. While experiments that do calculations via the host make use of Python, software for the PPU has to be written in a lower level programming language to use the smaller memory efficiently. Most of the software stack available for it uses C++, although C and assembler also are possible options [7]. Programs written for the PPU can be compiled with a customized version of the GNU Compiler Collection (GCC).

A quarter of the 16 kB memory is by default reserved from standard usage. This memory section is called the **mailbox**, writing data to this mailbox is the standard way of communication for the PPU so far.

2.3 Properties of Data

Most of the available computers today are based on the von-Neumann-architecture, which means they consist of a central processing unit (CPU) that is connected to random access memory (RAM) as well as input and output devices. Instructions and data are stored in the memory, the CPU executes these instructions and writes results to the memory. The input allows for interactions with the program flow or the data, the output devices represent the results of computations in some form. While this is the underlying principle of almost all modern computers, not every CPU works exactly the same way. The options for machine code instructions available makes up the *instruction set* and the size of its *registers* which hold pieces of data while they are being processed dictates what the natural units of data, simply called *words* are for a given processor architecture.

In the context of the experimental setup consisting of a host computer and the PPU on the chip, it is important to know that the host has a x86-64 CPU, which means it has words of the size 64 bits and uses the x86 instruction set. As mentioned in 2.2.2, the PPU uses the Power ISA with a wordsize of 32 bits. With these differences in architecture arise differences in the way data is read and written by processors, therefore serialization is needed for a device agnostic transfer of data structures.

2.3.1 Endianness

Endianness is a term coined by Cohen 1980 [8] used to describe how the bytes in the binary representation of data are ordered. In a computer system, endianness is tied to the processor architecture since the CPU has to interface with the memory to read the binary data stored there.

Under **Little Endian**, the individual bytes that make up a number are ordered in a way that the least significant byte stands at the first position, the second least significant byte at the second position and so on, similar to the European style of writing dates, where the day comes first, then the month, and the year last. When applied to a number however, this system is counterintuitive sice numbers are usually written the other way around. As an example, the number 439041101_{10}, which is $1A2B3C4D_{16}$ in hexadecimal will be represented in this way:

```
0x1a2b3c4d == | 4d | 3c | 2b | 1a |
```

Little Endian is for example used by the very widespread x86-64 instruction set architecture to which the host of the experiment setup belongs, along with many modern consumer machines.

On the other hand, **Big Endian** orders bytes the other way around, with the most significant byte first. For a human reader this is much more intuitive when applied to numbers since it is the order one would write them down in, the bytes of the example from above will be represented as

```
0x1a2b3c4d == | 1a | 2b | 3c | 4d |
```

The Power ISA instruction set of the PPU uses **Big Endian** [6], which is the primary reason why data structures cannot just be copied from one system to the other.

Alignment and Padding

On top of the issue of endianness, the other difference between the host computer and the PPU lies in their respective word size. When interfacing with the memory, code that writes and reads data **aligned** to the boundaries of the systems *words* is more effecient than it would be without **alignment**. A *word* in this context is fundamental data type that can very depending on the instruction set architecture of a processor [9]. As an example lets look a data structure made of three 8 bit elements and one 16 bit element on a 32 bit system that uses big endian just for the sake of better human readability:

```
struct example_structure
{
        uint8_t example_char_array[3] = {0x1a, 0x2b, 0x3c};
        uint16_t example_short = 0xffff;
};
```

Since the sum of its members sizes totals at 40 bits, it is too large to fit into one 32 bit word. If it was stored as an uninterrupted string of bytes, the 16 bit element would have to be split in half between the first and the second word:

```
1A 2B 3C FF FF 00 00 00
|--word1--| |--word2--|
```

Since the second word is not filled, the last three bytes are not used, here this is represented by writing zeros, on actual memory those are just ignored and not overwritten with zeros, hence they could hold bytes from previous allocations. These bytes get ignored when data from the structure is read. If it was now necessary to access the 16 bit number from this structure, the CPU would have to load two memory words and puzzle the number back together. In most application cases this would be considered a waste of computational resources, which is why the data structure will by default altered during the compilation of the program be written into the memory as follows:

```
1A 2B 3C 00 FF FF 00 00
|--word1--| |--word2--|
```

In this way, the 16 bit number is not split between two words and can therefore be accessed with less expenditure. The zeros to fill up the first word behave like the ignored bytes that fill up the second word, they serve no other use then to assure that the relevant data is in **alignment** with the word boundaries. Bytes that are used in such a way are called **padding**. If the same example structure would be used on a 64 bit architecture, there would be no need for **padding** between the 8 bit and the 16 bit elements since they fit in one word, instead they would be directly next to each other. In the case of host and PPU, this creates an additional obstacle since the host uses 64 bit words while the PPU has a word size of 32 bits.

Further information on the details related to memory and how it is accessed can be found in [10].

2.4 Experiment Control

2.4.1 Basic Usage

All experimental setups are nodes of the groups cluster that is managed with `Slurm` [11], which is used to schedule jobs. As a way to manage dependencies, jobs are run encapsulated on a virtual environment called containers with `singularity` [12]. To build software a tool called `waf` [13] is used.

2.4.2 Software Framework

For software running on the host there is a hardware abstraction layer called `haldls` [14]. It provides things like a classes that represents an entire PPU memory as well as blocks or individual words. Hardware coordinates are provided by `halco` [15]. Software for the PPU makes use of `libnux` [16] for math functions, macros and functions for the vector unit and memory management as well as ways to access neurons and synapses from the PPU. `libnux` also contains functions to write individual integers or strings onto the mailbox, but this method of communication is limited when dealing with more complex data structures.

2.4.3 Host

To control the chip from the outside, a host computer interfaces with it via an FPGA. This host is a node of the groups cluster and can therefore be accessed and given tasks via `Slurm`. While the cluster is not homogeneous and the architectures of individual host machines can differ, they all feature CPUs using the x86-64 instruction set.

2.4.4 Executing programs on the PPU

To control the PPU from the outside, it is possible for the host to access its memory via the FPGA. On the hostside an object of the type `PPUMemory` provided by `haldls` is initialized and the precompiled binary file containing the PPU program is loaded into it. This memory can be left as is or individual sections could be altered, for example data could be written into the mailbox this way before even executing the program. The memory object can then be loaded onto the actual memory of the PPU where the program is then executed. It is also possible to send data to the PPU during runtime, for example during the NSEM experiment parameters are loaded in this way instead of being written onto the memory object before transfer. In addition to the mailbox writing functions provided by `libnux`, the memory of the PPU can also copied by the host, this again creates a memory object, by knowing where which data is, results of experiments can be accessed this way.

The entry-point function of a PPU program is called `int start(void)`, other functions can be defined or included from headers to then be invoked while `start` is running. Upon finishing, `start` returns an integer to a fixed address. When the PPU memory is later transferred to the host, the address of the return code can be accessed with `haldls`.

2.5 Serialization

2.5.1 Concept and Formats

As mentioned in 1, in its most basic sense the word **serialization** means to take a data structure and turn it into a one-dimensional format from which the original data can then be restored again via the process of **deserialization**. Ways of doing this can be diveded into two categories, **binary** and **text based**. Serialization is used both to store data and to transmit it, in the context of enabling more structured communication the transfer aspect is the more important one. Binary serialization formats perform better in terms of processor usage, while text based formats are easier to read from a human perspective than a stream of ones and zeros. One example for a text based serialization format is `JSON` (JavaScript Object Notation), since it is derived from the JavaScript programming language it is often used by Web APIs [17].

2.5.2 Bitsery

`bitsery` is a header only C++ library for binary serialization developed by Mindaugas Vinkelis and is available on GitHub under the MIT license [18]. Among other things it

allows the user to configure endianness, which is exactly what is needed for communication between host and PPU. In addition to that, the design choice to make it binary only instead of including other formats of serialization such as `JSON` allows it to run very efficient on little memory which is a strong concern when working within the limits of the PPU.

2.5.3 Usage and Function

Aside from including the relevant headers, for a structure to be serialized with `bitsery` it needs to have a function called `serialize`. This function needs a templated argument, written with `S` as the typename and `s` as the argument itself in examples given in the documentation, but this is an arbitrary choice. This template will then work with the class `Serializer` for serialization and `Deserializer` for deserialization respectively, those classes are defined in their own header files and automatically included by `bitsery/bitsery.h`.

```
struct test_structure
{
        uint8_t a;
        uint64_t b;
        uint16_t c[3];
};

template<typename S>
void serialize(S& s, test_structure& o)
{
        s.value1b(o.a);
        s.value8b(o.b);
        s.container2b(o.c);
}
```

The functions used by the `Serializer` and `Deserializer` classes should be used as follows:

Function	Use case
`value`	Fundamental types
`valueXb`	Fundamental types, X being 1, 2, 4 or 8bytes
`container`	Container of fundamental types
`containerXb`	Container with specified element size
`text`	Similar to container, but optimized for text
`textXb`	For a normal string use X = 1
`object`	For nesting structures

Both the `container` and `text` functions will need a maximum size specified if the containers passed into them can be resized. If they have a fixed size, this size will be used as maximum. Similarly, the byte size specification variants for all of the functions

above will serve as template parameters for their unspecified counterparts, e.g. `value4b` will invoke `value<4>`. When using `object` to serialize a structure nested into the main one, it should have its own `serialize` function.

In addition to the `serialize` function, is is also necessary to define a `Buffer`, which will then hold the serialized data. The `Buffer` is a container of `uint8_t` with a fixed size. This will be used together with adapter classes for input and output that have templates specified with `Buffer`.

```
using namespace bitsery;
using Buffer = std::array<uint8_t, 200>;
using OutputAdapter = OutputBufferAdapter<Buffer>;
using InputAdapter = InputBufferAdapter<Buffer>;

Buffer buffer;
```

With all of those requirements met, the easiest way to serialize and deserialize is to use the `quickSerialization` and `quickDeserialization` functions with the `OutputAdapter` and `InputAdapter` as template parameters. The first function takes the `Buffer` and the structure as arguments, puts the serialized structure into the `buffer` and return the size written, the latter needs as arguments the `buffer`, the size of the serialized structure on the `buffer` and a result structure into which the function will write the serialized data. `quickDeserialization` returns a tuple containing a boolean that indicates wether or not the deserialization was successful and a code to specify the error in the case one occurs.

```
test_structure example = {2, 3, {4, 5, 6}};
test_structure result;

auto written_size = quickSerialization<OutputAdapter>(
        buffer, data
        );
auto state = quickDeserialization<InputAdapter>(
        {buffer.begin(), written_size}, result
        );
```

The example above does of course happen on the same processor, in our use case one of those steps would happen on the host, the other on the PPU. To account for the difference in endianness of the two systems, the adapters need another template parameter in form of an endianness configuration struct. To not concern the limited ressources of the PPU with this extra step, this configuration has been happening on the hosts side for all of the test conducted. Having the option to decide on which side of a data transfer this step is taken was an important part of the library choice.

```
struct ppu_endianness_config
{
```

```
        using namespace bitsery
        static constexpr EndiannessType Endianness
                = EndiannessType::BigEndian;
        static constexpr bool CheckDataErrors = true;
        static constexpr bool CheckAdapterErrors = true;
};
```

After defining the `Buffer`, the adapters using this custom configuration would then be initialized as

```
using OutputAdapter =
        OutputBufferAdapter<Buffer, ppu_endianness_config>;
using InputAdapter =
        InputBufferAdapter<Buffer, ppu_endianness_config>;
```

With this configuration in place, the host will serialize to and deserialize from a form that can be used without further work on a big endian system like the PPU.

3 Results

3.1 Verification

Before taking any quantitative measurements concerning `bitsery` or using it in the context of a more complex experiment, it had to be verified that serialization and deserialization work as intended on the hardware. The test make use of the Google Test (`gtest`) library for C++, where test functions will report failure or success based depending on wether or not specified conditions are met after the test function has been executed.

The least intricate of those test was to serialize and deserialize data structures only on the host, since it uses a very common CPU architecture which in turn means programs written for it can be compiled with the standard version of the GCC compiler, after that serialization and deserialization could be tested on the PPU, the final test case would involve both devices transmitting serialized data.

All of these tests would involve easier, homogenous structures with properly aligned members, but also heterogenous structures that would purposefully invoke padding. Nesting structures were also tested at a later stage.

As expected, the test to verify `bitsery` on the experiment host succeeded without issues, which paved the way for tests involving the PPU.

Using the PPU, the test was similar to the one done on the host; serialization and deserialization would both take place on the PPU now. Similar to the test before, some structures would cause the compiler to pad them were examined. Instead of using `gtest`, success or failure would be econded in the return code of the `start` function.

To run this test program on the PPU, the precompiled binary file would be loaded via a host program. After a small waiting period ensuring the PPU would have completed its program, the memory would be loaded back to the hosts memory where the return code can then be checked to see if the test succeeded.

Just like on the host, the test on the PPU also succeeded.

With `bitsery` being confirmed to work on both of the relevant components of the setup, the next step was to actually test serialization and deserialization in the context of data exchange between the host and the PPU. For this purpose, the binary file would be loaded into a PPU memory object on the host again, but before loading it onto the PPU, the host program would alter is by writing a `Buffer` containing a serialized structure into the mailbox region of the memory object.

When injecting extra data to a PPU program in this way, an extra step is necessary. When a binary file is transferred from the host it is formatted by reversing the byte order of each 32 bit section (i.e. PPU memory word). Any data injected into the binary file will

undergo the same formatting, counteracting this is not difficult since it is an involution; cancelling itself out when applied twice. For this purpose the following function is used:

```
template<typename T, size_t size>
void htonl_on_buffer(std::array<T, size>& buffer)
{
        uint32_t* ptr = reinterpret_cast<uint32_t*>(buffer);
        for(int i = 0;
                i < size_to_wordcount(size);
                i++) {
                *ptr = htonl(*ptr);
                ptr++;
        }
}
```

where `size_to_wordcount` returns the amount of 32 bit words a data structure with size `size` would take up. The function `htonl` takes a 32 bit value as argument and changes its byte order, it is included via the `arpa/inet.h` header [19]. With this extra operation applied to the `Buffer` and using the extra endianness configuration shown at the end of 2.5.3, data that has been serialized on the host and transferred with the binary file can be successfully deserialized on the PPU.

With a successful transfer of serialized data from the host to the PPU, the last step to verify that `bitsery` works as intended on the hardware setup is to try a tranfer in the other direction. With the same correction for the word-wise byte reordering as above, this also succeeds.

3.2 Benchmarks

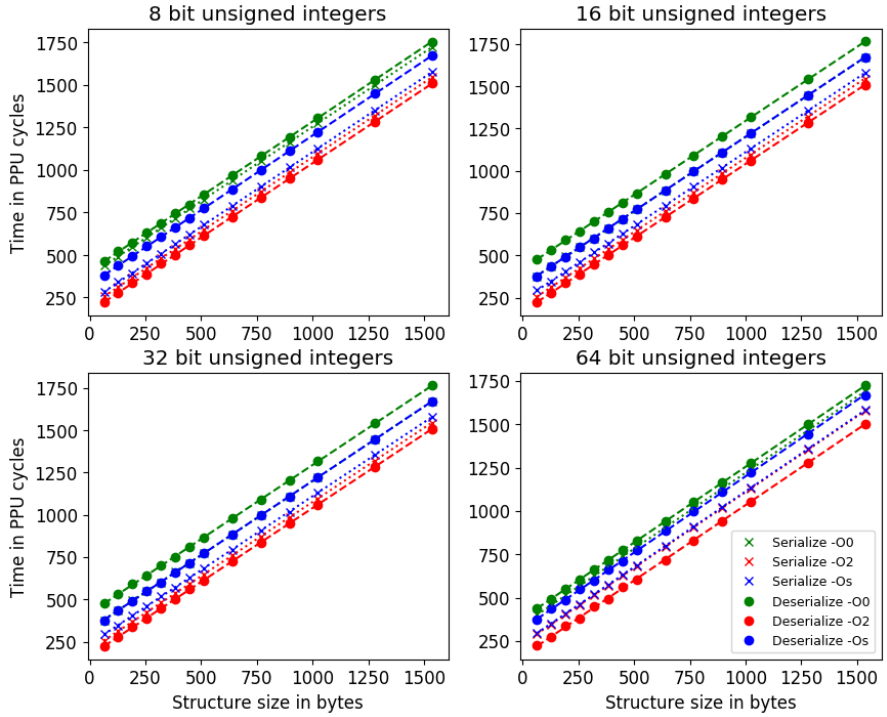
3.2.1 Large Homogenous Structures

After verifying serialization and deserialization work as inteded on both devices of the experimental setup, the next step was to quantify their performance on the PPU. To give a rough speed estimate under favorable conditions, the first series of test used data structures only containing an array of unsigned integers with varying sizes, for example:

```
struct test_structure
{
        uint16_t arr[128];
};
```

The serialization functions to such a structures just needs one `container` function with no specified maximum size since the array is fixed. All of the structures tested had sizes ranging from 64 to 1536 bytes, with a version for unsigned integers of 8, 16, 32 and 64 bits. Those test were generated from a template file via a python script. In addition to that, each of those tests was compiled with different three compiler optimization

Figure 3.1: Serialization and deserialization times for large homogeneous structures



options. `-O0` is the default with no optimization, instead offering lower compile times which is useful for debugging, `-O2` which is optimized for faster run times and `-Os` to create smaller binaries, both of the latter two options come at the expense of longer compile times.

To measure the time needed for both processes on the PPU `libnux/time.h` is used. Instead of conventional units for time, the function it features return their measurement in number of PPU cycles. Both the PPU and FPGA operate on the same clock rate of 100 MHz, 100 clock cycles per microsecond. The time is measured both before and after each serialization and deserialization, then the difference is calculated. The time differences of both the serialization and deserialization are then serialized themselves and later deserialized and saved in a file by the host. An extra measure taken here is that all variables involved in the processes are afterwards passed into a `do_not_optimize_away` to prevent the compiler from omitting steps in the process. This may seem strange, but accessing for example the entries of the result structure after the deserialization outside of the time measurement has had an effect on these measurements. In a fixed configuration, the time measurements are completely deterministic, giving the exact same number of cycles for a given structure over multiple runs.

Shown in 3.1 are the serialization and deserialization times for each unsigned integer size and compiler option. The dashed lines between those measurements represent linear fits that assumened

$$t = a * s + b$$

with t being the time, s the size of the structures, a the number of cycles needed for one byte of data and b the overhead. While the processes themselves are deterministic, the error of a fit through data points can be given by the squared diagonal entries of its covariance matrix. In this case however, the correlation with linear growth was very strong, thus the errors of the fit are negligibly small. With the lines being almost parallel, the parameter a corresponding to their gradients is almost equal for all of these measurements, with compiler options and the difference of serialization and deserialization only affecting the overhead b, visible in the graphs as the distance between the lines.

The mean of all those fits puts the parameter a at the value

$$a = (0,8745 \pm 0.0001)\frac{PPUcycles}{byte}$$

For the different values obtained via linear fit, the errors for a given by the covariance matrix are several oders of magnitude smaller, in the range of 10^{-12}, and are therefore negligible. The standard deviation of the different values for a is $\sigma_a = 0.002$. With 24 values, the error of the mean is therefore

$$\delta_a = \frac{\sigma_a}{24} = 9.2 * 10^{-5} \approx 1 * 10^{-4}$$

This result should be taken with a grain of salt since the structures examined are very simple, more complex ones will not behave exactly in this linear speed to size relationas their complexity generates additional overhead. What this result shows is that under ideal conditions, serialization and deserialization with `bitsery` can achieve high speeds on our hardware, with the aforementioned clock rate of 100 MHz that means less than 100 nanoseconds per byte in this case case.

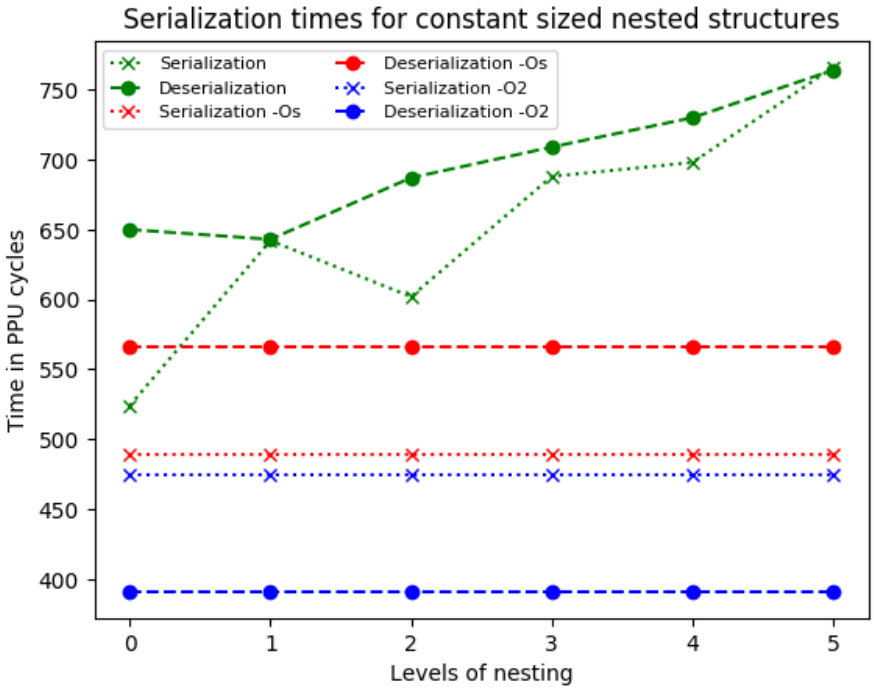
3.2.2 Nested Structures

In actual use cases, structures rarely consist of flat, homogeneous arrays only. To see how `bitsery` would perform with added depth, nested structures where tested. Implementation and measurement and done similar to the ones in 3.2.1. The measurements method for during this test is identical to the ones 3.2.1, only the shape of the structures is different.

The first test case would have had a base structure used as the only member for the first nested one, then the first nested structure as only content within the second one and so on, up to five levels of nesting above the base structure. In this first test case, no extra data was added per nesting step, so each of those structures had the same size as the one it is derived from. This base structure had the form:

```
struct base_struct
{
        uint8_t base_a = 0xff;
```

Figure 3.2: Serialization and deserialization times for constant sized nested structures




```
        uint64_t base_b = 0x1a2b00003c4d;
        uint16_t base_c = 0x5678;
        uint32_t base_d = 0x1234abcd;
};
```

Each consecutive nested structure for this test thus had the following shape:

```
struct nested_const_N // 1 < N <= 5
{
        nested_const_N-1 content;
        // in case N=1: base_struct content;
};
```

As shown in figure 3.2, both variants of compiler optimization resulted in no extra time being needed for deeper nesting, compared to the debug option which grew with extra depth. Since the nested structures scored the exact same results as the flat base structure with higher optimization, it could be that the compiler flattened them when no extra data is added.

For the next test, the structures would be homogeneously increased in size with each additional level of nesting. *Level 0* for this test was the same base structure used for the previous test, with each extra nesting step the previous structure would be used as content along with a fixed piece of extra data. In one measurement, the extra content consisted of a single `uint32_t`, the second measurement instead added an array of five 64 bit unsigned integers with each nesting step, which resulted in structures of the shapes:

```
struct nested_const_N // 1 < N <= 5
{
        // for the 4 byte growth
        uint32_t extra_content_N;

        // for the 40 byte growth
        uint64_t extra_content_N[5];

        // both of those growths using
        nested_const_N-1 content;
        // in case N=1: base_struct content;
};
```

An interesting observation to make here is that in the second measurement shown in figure 3.3, the times for both processes are not significantly higher than the ones in the first one, even though the growth for the second one was 10 times larger. This shows serves to illustrate that using containers instead of single values can speed the process up significantly.

Furthermore, a set of flat structures containing the same data as their nested counterparts where used for measurements in the same way to see how nesting effects the performance. The resulting serialization and deserialization times for `-O0` and `-O2` can be seen in figure 3.4. Not pictured is `-Os`, which is similar to `-O0`, only slightly faster. When comparing the plots for the two compiler option we can see that for `-O2` that the deserialization times get very similar for both nested and flat structures.

Figure 3.3: Serialization and deserialization times for homogeneously growing nested structures

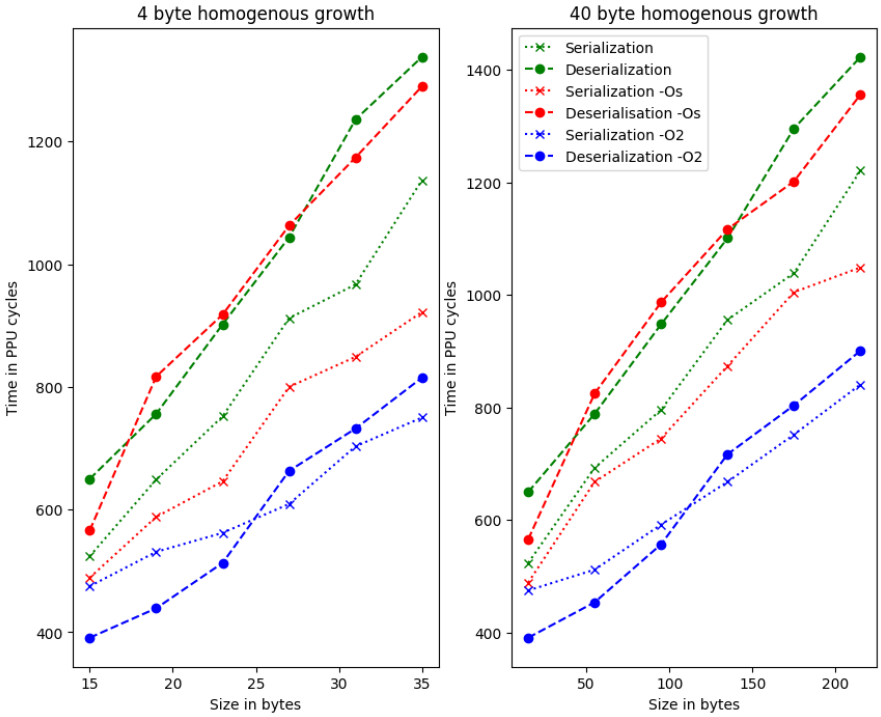


Figure 3.4: Serialization and deserialization times for homogeneously growing nested structures and their flat counterparts

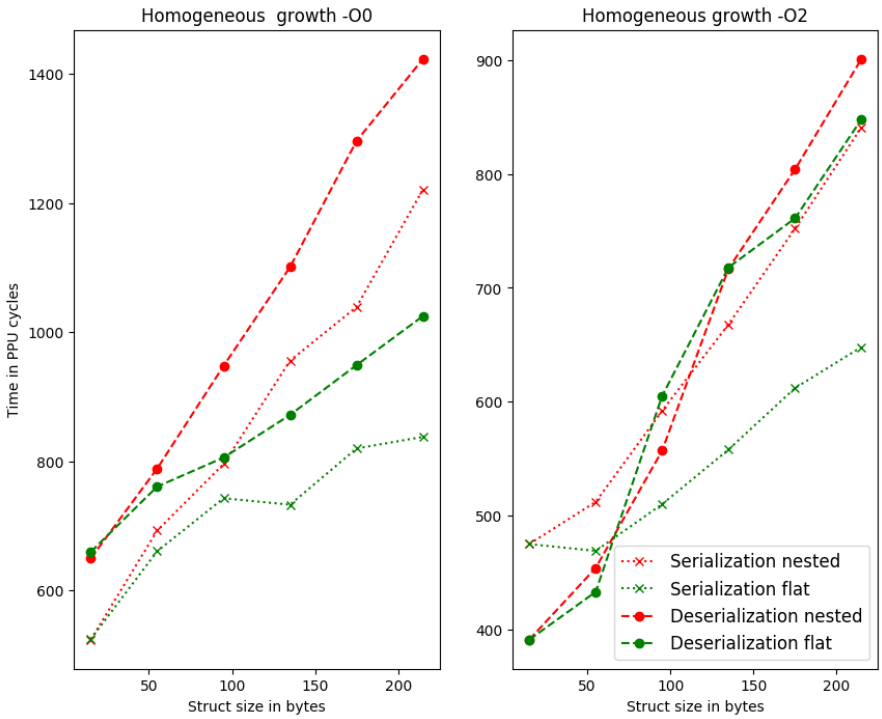
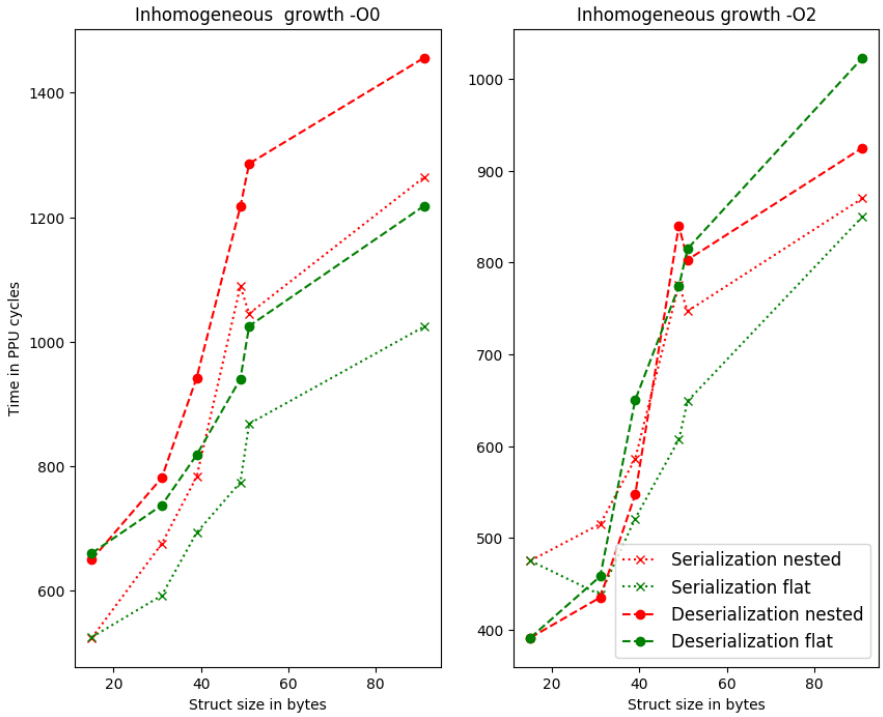


Figure 3.5: Serialization and deserialization times for inhomogeneously growing nested structures and their flat counterparts



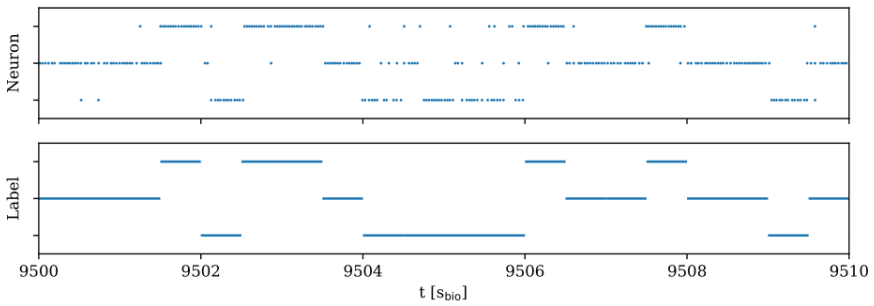
The structures chosen for the final test grow inhomogeneously. Starting with the same base structure as with the previous two tests, does not add the same content with each nesting level. The extra content added varies between the four usnigned integer types and arrays containing them. Just like for the homogeneous structures, these have flat counterparts containing the same data. The comparison between flat and nested for one of the measurements can be seen in figure 3.5. To produce the measurements shown in figure 3.5 the following pieces of extra data where added with each nesting step.

Nesting level	Added data
1	`uint32_t extra_content_1[4]`
2	`uint64_t extra_content_2`
3	`uint8_t extra_content_3[10]`
4	`uint16_t extra_content_4`
5	`uint64_t extra_content_5[5]`

Just like with the homogeneous measurements, the plot for the `-O2` shows little difference between nested and flat deserialization speeds, here it even goes to the extend that the nested structures developes an advantage over the flat one towards the end. This could be observed with multiple structure configurations. The prominent dip of the nested curve between the 4th and 5t data point is however unique to this specific

measurement, with different added contents such a behaviour could not be reproduced.

Figure 3.6: Raster plot comparing the spikes generated by the neurons with the input, parameters serialized with `bitsery`



3.3 Using bitsery for NSEM

The part of the NSEM experiment where `bitsery` was used to serialize parameters is the one described in section 3.7 of [3]. In this experiment, three neurons recieve a 5 by 5 pixel image as input, each pixel here is one synapse connected to the neuron.

Since the neurons of the analog neural network are accelerated compared to biological ones (see 2.2.1), in the case of this experiment by a factor of 975, the times and frequences that follow are meant as biological equivalents, not the actual values on hardware. Hence a measurement time of $10000 s_{bio}$ does not mean the experiment takes almost three hours, but it simulates this time in a biological network while taking only about $10s$ on the hardware.

The different pictures used as input are horizontal bars at different heights. Pixels that are part of a bar recieve a spike train of $70 Hz_{bio}$ while the background pixels get an input of $10 Hz_{bio}$. Every $0.5 s_{bio}$ a picture is randomly selected and presented to the network. Of the parameters given to the experiment, three where in this run serialized with `bitsery`, those being the periods for homeostasis and SEM as well as the timeframe of the experiment. The spikes generated by the three neurons after $9500 s_{bio}$ can be seen in a raster plot in figure 3.6. Those spikes mostly overlap with the given inputs since each neuron reacts to one of the images, thus spiking at a much higher rate than the other two. This is the same behaviour that can be observed without `bitsery`, see [3] and [7].

4 Discussion

Over the course of this thesis, serialization with `bitsery` on the HICANN-DLS v2 prototype chip has been done successfully in a variety of test scenarios and even a scientific use case.

Starting with the verification of serialization and deserialization on the hardware it could be seen that the light weight design of the library allows it to run successfully within the constraints of the PPU. The subsequent test of data transfer between the host computer and the HICANN-DLS v2 has shown that `bitsery` provides the ability for device agnostic exchange of structured data with configurable endianness that is needed for our use case.

The subsequent benchmarking with large array structures not only shows that it is possible to handle large amounts of data with `bitsery`, but also shows linear behaviour of serialization and deserialization times with structure size and gives an estimate of how fast the serialization and deserialization can be under ideal conditions.

Nesting data structures within each other has not lead to dramatic drops in performance, especially with the `-O2` compiler option the difference is often negligible. The two different measurements for homogeneous growth of nested structures also show how well arrays work with `bitsery`, since the increase in serialization time between is very little even though the growths differ by a factor of 10. For future uses it might be advantageous to remember this.

The counterintuitive measurements of faster deserialization times for nested structures over flat ones in the test for inhomogeneously growing nested structures as well as the strange dip that can be seen in Figure 3.5 could be caused by the fact that in these structures, containers and single values both are added during the growth steps. This is however purely speculative, another hypothesis regarding this behaviour was that the serialization function of a nested structure has to call less functions from the `Serializer` class, but this is unlikely. Analyzing the binaries with `objdump` has shown that the nested structures make calls to the `Serializer` and `Deserializer` more often than the flat ones.

Using `bitsery` with the NSEM experiment has reproduced the previous results and can therefore be considered a success. The fact that only three parameters where serialized is mostly due to time constraints; serializing even more of them would require more work on the experiment code and in some cases minor changes on the software architecture, but based on the good results so far there should not be any fundamental issues. It shows that `bitsery` can be used in a more complex setting where interaction with the analog neural network core takes place.

5 Outlook

Looking at the results so far, `bitsery` seems suitable for future applications in hybrid neuromorphic systems.

Since PPU code should be portable from HICANN-DLS to HICANN-X with few to no issues future tests can be moved to this new platform instead. Here the interesting new challenge would be to transfer data between the two PPUs on the chip. Further down the line the hybrid design is planned to be integrated on wafer scale [2], in such a setup data exchange for decentralized experiment control would be even more beneficial.

When using `bitsery` for more experiments, it would be advantageous to have an API that allows to send and recieve structured data with the different experiment control units. With such software, it would be possible to stop using the mailbox for communication, thus freeing up the 4 kB currently reserved for it and allowing more complex PPU programs.

With such improvements to data exchange, setups with multiple hybrid chips could have increasing flexibility in the way experiments can be controlled. This would allow for large decentralized simulation of parts of the human brain with increasing biological realism. For example in a configuration where multiple PPUs handle plasticity for populations of neurons in a larger simulation, changes in parameters communicated between them could be used to simulate the influence chemicals like dopamine have on learning.

Bibliography

[1] S. A. Aamir, Y. Stradmann, P. Müller, C. Pehle, A. Hartel, A. Grübl, J. Schemmel, and K. Meier. An accelerated lif neuronal network array for a large-scale mixed-signal neuromorphic architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12):4299–4312, 2018.

[2] Johannes Schemmel, Sebastian Billaudelle, Phillip Dauer, and Johannes Weis. Accelerated analog neuromorphic computing, 2020.

[3] Philipp Spilger. Spike-based expectation maximization on the hicann-dlsv2 neuromorphic chip. Bachelorarbeit, Universität Heidelberg, 11 2018.

[4] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[5] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[6] Power ISA™ version 2.06 revision b. Technical report, IBM Corporation, 2010.

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

[8] D. Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[9] Intel® 64 and ia-32 architecturessoftware developer's manualvolume 1:basic architecture. Technical report, Intel Corporation, 2019.

[10] Ulrich Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[11] Slurm workload manager.

[12] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PLoS ONE 12(5)*, 2017.

[13] Thomas Nagy. *The Waf Book*. 2010.

[14] haldls repository on github.

[15] halco repository on github.

[16] libnux repository on github.

[17] Douglas Crockfort. Introducing json.

[18] Mindaugas Vinkelis. bitsery repository on github.

[19] Byte order conversion.

Appendix

Used Hardware Setup

Component	Identifier
Board Name	Fantasio
Chip ID	30
FPGA ID	B291656

Used Software

Software item	Commit ID or other identifier
Singularity container	`/containers/stable/2020-03-27_1.img`
`ppu-toolchain`	`ppu-toolchain/2020-03-17-1`
`waf`	`waf/2020-03-23-1`
`libnux`	Ie099a28e0157f9430c57b4f128eb410717d781b8 (CS 10181)
`haldls`	I03406fb2cefc4a4250f8a8a627bcbfc8b0c0d2c8 (CS 8335)
`model-hw-nsem`	I22e0a777f5a82cf5ced243e0947c3c47ee465b2e (CS 10032)
`bitsery`	501d60f67d7bb4bc824b5c58101ca0a6eaa5c7b6

Acknowledgments / Danksagung

Ich möchte mich bedanken bei:

Dr. JohannesSchemmel für die Betreuung meiner Arbeit

Eric und Philipp dafür, dass sie mich eingearbeitet und mir viele neue Dinge beigebracht haben, und für die viele konstruktive Kritik zu Code und der geschriebenen Arbeit

Sebastian Schmitt für die Vorlesung "Brain Inspired Computing" im letzten Sommersemester, die dazu ermutigt hat, meinen Bachelor in dieser Arbeitsgruppe zu schreiben

Der gesamten Electronic Vision(s) Gruppe für die freundliche Arbeitsatmosphäre

Meinen Eltern und meinem Bruder ihre Unterstützung zu Hause

Furthermore I would like to thank Mindaugas Vinkelis for this great serialization library he made, without which this thesis would have looked very different.

This work would not have been possible without the founding the Eletronic Vision(s) Group recieves from the Human Brain Project (HBP) since this is what enables the development of new systems and prototypes.

Erklärung

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