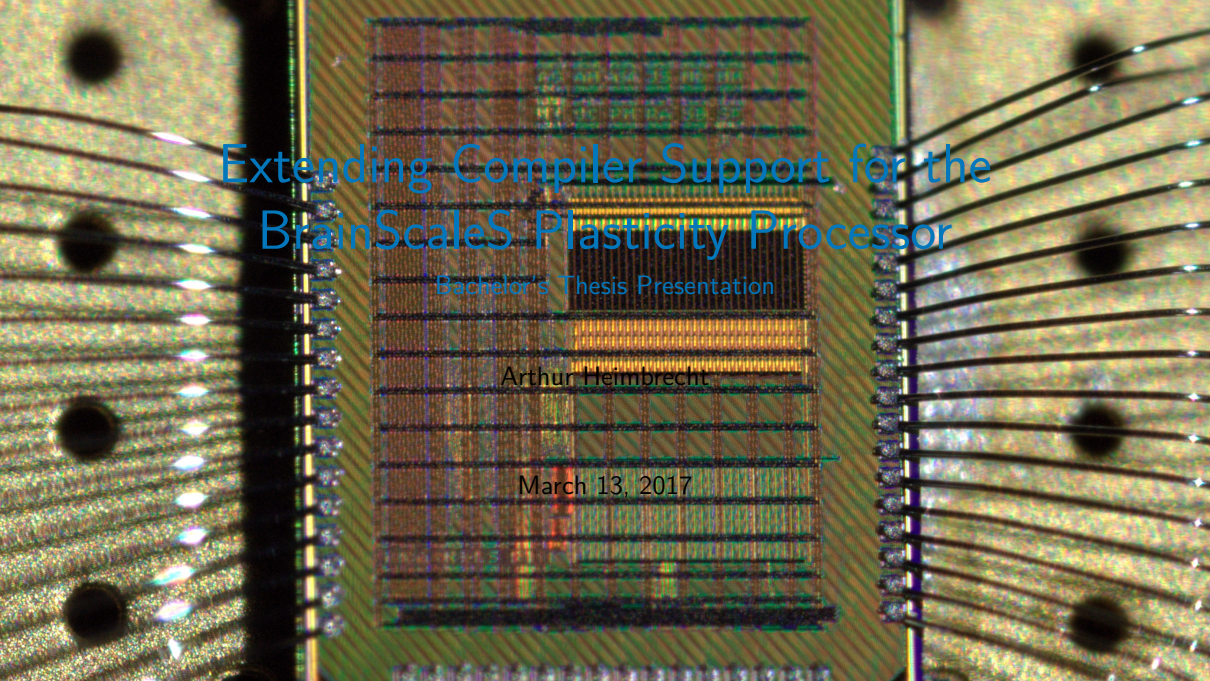


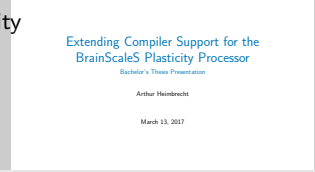
Extending Compiler Support for the BrainScaleS Plasticity Processor

Bachelor's Thesis Presentation

Arthur Heimbrecht

March 13, 2017

Extending Compiler Support for the BrainScaleS Plasticity Processor



Welcome everybody to this talk about my Bachelor thesis. As this is a quite technical talk, feel free to ask questions at any time. The topic of my thesis was Extending Compiler Support for the BrainScaleS Plasticity Processor.

Contents

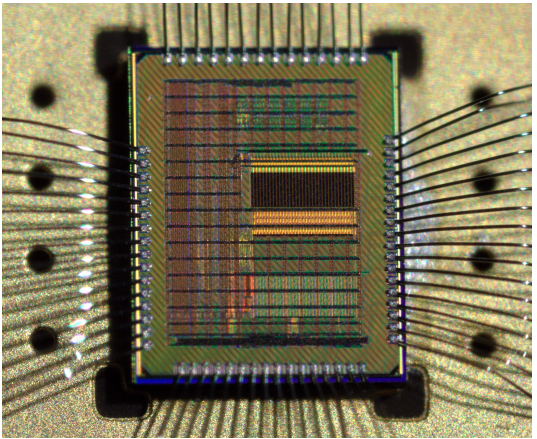
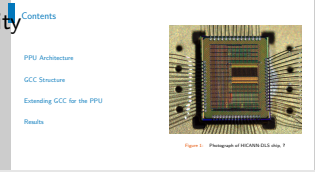


Figure 1: Photograph of HICANN-DLS chip, **?**



as the title hints there are two main components to this talk, which are the plasticity processing unit (PPU) and Compilers. I will briefly talk about both of these and their applications. Afterwards I will explain, what I did during my thesis, which is of course followed by a short presentation of the results.
But first I should explain, what the PPU is.

HICANN-DLS

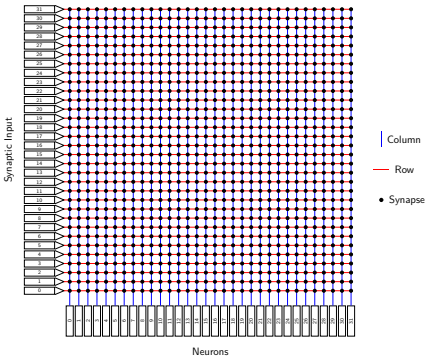


Figure 2: Schematic Representation of the Synapse Array on HICANN-DLS

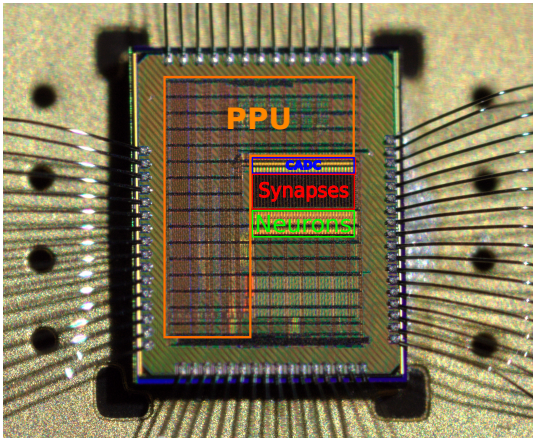
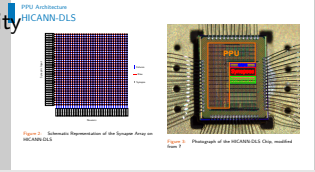


Figure 3: Photograph of the HICANN-DLS Chip, modified from **?**



I am assuming that everybody here probably has heard about the HICANN-DLS. In short it is a chip that emulates neural networks through neurons and synapses that are implemented directly on the chip. What is special about the DLS, is the addition of a PPU to the HICANN. It is capable of performing simple computation and allows for on-line plasticity directly on the chip.
As you can see to the right I have added a picture of the HICANN chip and marked the physical compartments. The PPU actually takes up a large space on the chip but before I am going to talk about the PPU, I will talk about the other parts briefly First we have the neurons. There are 32 neurons on the HICANN-DLS and each neuron is a circuit that receives some input signal which can cause the neurons to spike. These signals come from the so called synapse array. It connects 32 pre-synaptic inputs to all 32 neurons. This gives a total of 1024 synapses on each HICANN.
Each synapse is realized through a circuit that takes the input signal and modifies it with a synpatic weight, that is saved in each synapse. There are also other properties to each synapse, but for this presentation it is enough to focus on the synaptic weights. Each synaptic weight is 6 bits wide, which is equivalent to numbers between 0 and 63 or a fixed-point number with an accuracy of 2^{-6}. <bild whiteboard> Normally data segments are 8 bits wide, which is a byte, and the superfluous bits are used for calibration of the synapse.
The pre-synaptic inputs are handled by an FPGA, that takes care of spike routing.
But there is also a correlation analog digital converter, that receives also signals of synapses if the forward signals to neurons. Through the CADC one can read out correlation that can be used for plasticity.
The PPU is this remaining large section of the HICANN chip. This mostly is an existing architecture, that is called POWER architecture, which was extended with a custom vector extension which we will call s2pp.
In general it is good to keep this in mind during this talk: The processing unit in HICANN DLS is called the PPU, the architecture with the vector extension is called nux architecture and the vector extension will be called s2pp.

Plasticity Processing Unit

"Common" von-Neumann architecture

Machine Instructions

Arithmetic Logic Unit (ALU)

$\text{latency}(\text{register}) \ll \text{latency}(\text{memory})$

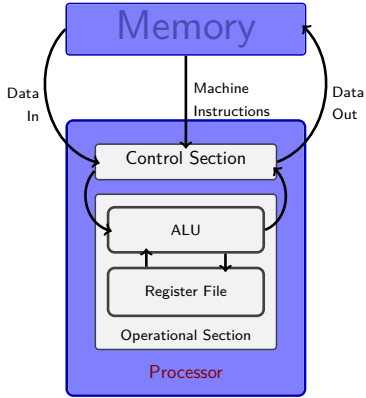


Figure 4: Schematic of General Purpose Processor with von-Neumann Architecture

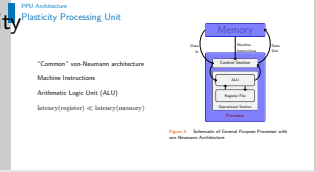


Figure 3: Schematic of General Purpose Processor with von Neumann Architecture

But before I shall go into more detail, I have to talk about processors in general. As you may know, next to all computing nowadays is done by processors which are often described as CPUs. The processors are normally built according to the von-Neumann architecture, which combines programs and data in the same memory. The processor fetches instructions from the memory and analyzes them to decide what to do. Most instructions get passed to the ALU that performs any sort of computation in the processor. To do this, the ALU has access to so called registers. These registers can hold small amounts of data but offer a very low access time. The ALU uses these registers for computation and saves results there as well. Data is then written separately into memory or new values for registers are loaded from memory. This usually takes a significant amount of time longer than access to registers and thus, programs try to use registers as much as possible.

PPU Architecture

Based on POWER architecture

Vector Extension
→ Parallelization
→ Performance

Weight Updates

Access to Synaptic Array

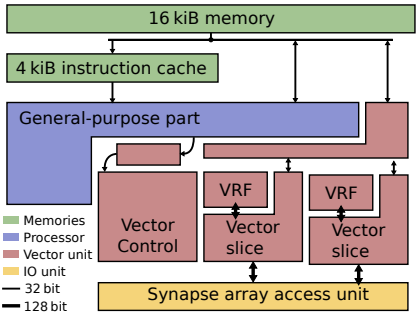
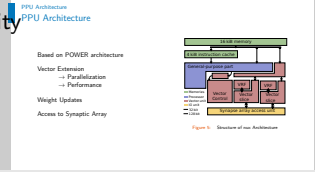


Figure 5: Structure of nux Architecture

└PPU Architecture



Instead of the ALU, some instructions can be passed to other units that can perform special instructions. These units may also use different registers.
This is the case for the PPU. It includes a special vector extension that includes vector registers as well. Using vector registers is favorable for operations that use have to be applied to multiple values and can be done in parallel.
This is the strength of the PPU as is allows to compute weight updates of up to 16 synapses at the same time, which increases performance significantly. For this reason the PPU is also connected to the synaptic array and is able to read out and write synaptic weights. Although it induces a problem because the vector unit uses special instructions that are unique to the nux architecture.

Compiler Structure

Program $\xrightarrow{\text{compile}}$ Executable

Compiling Stages

Compiler $\rightarrow$ Assembly

Back-End is target-dependent

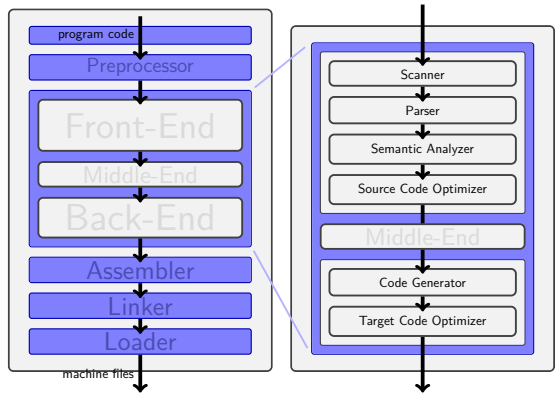
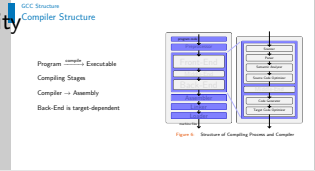


Figure 6: Structure of Compiling Process and Compiler



This is of special significance when using a compiler. But why? I am quite sure that all of you have used compilers just recently and generally know what they do. And most will probably say that a compiler compiles code, or to be more specific it converts a program into a file that the computer can run. This is of course right and is follows the definition of a compiler, but there is much more to it.
What we usually call compiler consists of several stages that follow different purposes and like the preprocessor, that takes care of define instructions and substituting macros. The actual compiler will only take that preprocessed code and create an assembly file from this. Assembly is a language that consists of machine instructions which have specific names and represents pretty much the lowest level of coding a human would do. From there on the assembly file gets converted into a pure machine code, hence no names only numbers, file by the assembler and then is combined with other files and gets assigned a memory location by the linker and loader.
But since I focused on the compiler in this thesis, I should also give an overview of the internal structure of a compiler. As you can see, this time there are different phases in the compiler and these similarly ordered like compiling stages. But we only need to focus on the last two stages which are part of the back-end.
The back-end is mainly responsible for creating the actual machine code which was previously processed by the front-end and the middle-end. It then also optimizes the code it generated, but this is machine-dependent. In general a back-end is always machine-dependent as the assembly language is different for different processor architectures. One architecture, that most of you might know is the ARM architecture that is at the heart of every smartphone and also SpiNNaker.

The PPU though, has a POWER architecture which is also called rs/6000.

GCC for the PPU

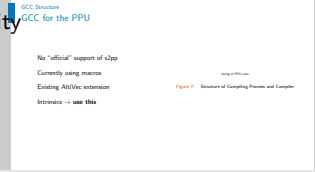
No “official” support of s2pp

Currently using macros

Existing AltiVec extension

Intrinsics → **use this**

Figure 7: Structure of Compiling Process and Compiler



now there exist different compilers and the one which is used by this group is the GCC. GCC is a compiler that is wide-spread especially at academic institutions. it is capable of compiling for a large variety of languages and also most processor architectures there are. Unfortunately this does not include the nux architecture. Although the basic architecture of nux is supported, there is no official support of the vector extension.
This means that users can code in C for example but every time they need to use the vector extension, it is necessary to handle assembly code. There have been efforts to make this easier, but still it mostly looks like this when programming for the PPU. <bild zeigen> Although there ARE ways to program vector extensions that look more common.
One such extension is the AltiVec extension to the POWER architecture. So in a way this is a competing vector extension to s2pp and programming for AltiVec usually looks like this.
it uses so called intrinsics for vector processing. An intrinsic basically works like a function, the same way addition is also a function, only does it map on a specific machine instruction that is invoked when using an intrinsic.

And if we compare these two, I think it becomes clear that this is what we want.

Main Work of this Thesis

No “official” support of s2pp

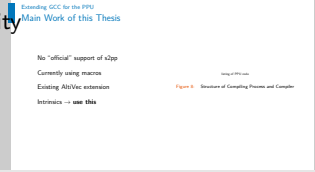
Currently using macros

Existing AltiVec extension

Intrinsics → **use this**

listing of PPU code

Figure 8: Structure of Compiling Process and Compiler



And this is what I did throughout my Bachelor's thesis. I added support for the nux architecture by extending the GCC back-end for POWER. And as simple as sounds it was quite a challenge, since there was no documentation but comments and general information on how to create a back-end.
Luckily there is already the AltiVec extension which I mentioned earlier and it provided a good guideline for me to find relevant sections in code.
Ultimately I had to go through roughly 50.000 lines of code and it needed about 3.000 additional lines to add support for the s2pp vector extension.
Now I want to show you the results of this and also give an outlook, of what might be possible in the future.

New Features

`-mcpu=nux` target flag

`vector` attribute

s2pp vector intrinsics

Inline assembly

Supports optimization

listing of PPU code

Figure 9: Structure of Compiling Process and Compiler

Let's start with generating code for the PPU. It simply takes one target flag and a header file, to create code for the complete nux architecture. <bild> Also there is a vector type which can be used to represent vectors as variables and it works exactly like other existing vector attributes. <extend picture> Most importantly, users can now use intrinsics for simple vector instructions and use the vector variables instead of arbitrary macros. <extend picture> Furthermore, there is inline assembly support which allows for simpler low level coding whenever it is needed. <extend picture> And all of this also supports optimization, that allows for efficient programs on the PPU, without particular knowledge by the user. <beispiel mit und ohne optimiereung>

Outlook

Further testing for bugs

Existing applications with `libnux`

Extending test coverage

Debugging support?

New tool for development!

listing of PPU code

Figure 10: Structure of Compiling Process and Compiler

Of course I have to mention that this is kind of a beta phase and that there are bugs that can occur. But the more people will use this compiler, the more bugs can be fixed and this will become an even more handy tool, when programming for the PPU.
Nonetheless there already exist programs and tests that make use of the compiler extension. This for example is the program David Stöckel used for his criticality tests, which he presented a few weeks ago. You can see the use of a testing framework which David established, called libnux. This is also a great help when programming for the PPU since it is available.
And this is one of a few tests which later shall extend the exiting testing scenarios on the PPU.
And there is more of what might be possible in the future with compiler support of nux. For once it will be easier to create high-level tests for the PPU and extending the current test coverage is something I would like to do in the future. Also it might be possible to realize GDB support for the PPU, which would allow for extended debugging of PPU software but this is something that is not yet on the agenda.
All in all I want to encourage as many of you as possible to use the new compiler, as it will become better over time as more users develop with it. And I hope, that I was able to give a comprehensive introduction to the compiler and some fundamentals of processor architecture.
If by any means some of you are interested in learning more about these topics I recommend the following books, which helped me to get into the subject and which I also used for references.