RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Arthur Heimbrecht

Bachelor Thesis

HD-KIP-TODOTODOTODO

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy

University of Heidelberg

Bachelor Thesis

in Physics

submitted by

Arthur Heimbrecht

born in Speyer

TODO 2123

Bachelor Thesis

This Bachelor Thesis has been carried out by Arthur Heimbrecht at the

Kirchhoff Institute for Physics

Ruprecht-Karls-Universität Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Bachelor Thesis

As part of the Human Brain Project, BrainScaleS is a unique project on many levels. This includes a processor solely used by the HICANN-DLS, which manages synaptic weights for every neuron built into one of the many wafers. To accelerate the speed at which this so called plasticity processor unit (PPU) computes all synaptic weights of every neuron used, the processor has an extended instruction set architecture (ISA) that supports vector registers and single input multiple data (SIMD). This report deals with the task of adding built-in functions to an existing backend of GCC, specifically the one used by the PPU, in order to extend the already implemented set of functions according to the users needs.

Contents

1 Introduction

The goal of this thesis is the extension of an existing back-end of GCC to a point where it supports the existing architecure of the PPU a.k.a. nux.

2 Basics neural networks

Neural networks build the main application of the Hicann DLS system. This short chapter is meant to give an overview over neural networks and synaptic weights.

On a very abstract level neurons in the brain resemble nodes of a network. As in a network neurons are interconnected through dendrites, synapses and axons which can be of different strength. Also a neuron is either spiking, meaning it is activated and sends this information to connected neurons or resting meaning it is not activated. In case a neuron is spiking, it send this information through its axon to other other neurons that are connected to the axon by synapses. These synapses can work quite differently but have in common that there is a certain weight associated to them, which we will call synaptic weight. This is equal to a gain with which the signal is either amplified or damped . The signal is then passed through the dendrite of the post-synaptic neuron to the soma where all incoming signals are integrated. If the integrated signals reach a certain threshold the neuron spikes and then sends a signal itself to other neurons.

is this word right?

With all these physiological parts there are only two important parts we need to take a look at in order to copy the function of a neural network: the neuron and the synapses. Basically we assume that all neurons are connected to each other through a synaptic weight. If two neurons are actually not meant to be connected, the synaptic weight simply is set to zero for these neurons to be disconnected. Now if we display the all neurons inputs and outputs in a 2D plain we get an array of synapses, which is equivalent to a weight matrix.

The synapses in the Hicann DLS laid out similarly hence its name “synaptic array".

3 The HICANN DLS system

This thesis mainly focuses on an essential part of the HICANN-DLS system. HICANN-DLS stands for Digital Learning System. For this reason it is important to understand the basics behind this and what the PPU is meant to do.

what does hicann stand for?

First of all

4 Processor basics

add reference

cite freidmann dissertation

Next to all processors used these days are built upon the so called von-Neumann architecture . Though the main goal of this group is to provide an alternative analogue architecture that is inspired by nature, there are advantages to the classic model of processors which are needed at some point. The main advantage of digital systems over analogue systems such as the human brain, is the ability to do calculations at much higher speeds. For this reason "normal" processors are responsible for handling experiment data as well as setting up different parts of the experiment. One such task is applying learning rules to the synapses during or in between experiments which can either be done by hand or with the help of the aforementioned PPU. The second option is especially valuable when updating synaptic weights during an experiment as the PPU does this much faster than a system which interacts from the outside. This is important for achieving experimental speeds that are 10^4 times faster than their biological counterparts.

Therefore the PPU is one of many von-Neumann processors in this world and follows the same basic concepts. It is important to understand these concepts as they build the foundation to this report!

ALU FPU memory memory controller clockcycle pipeline

5 Compiler Structure

This part explains the structure of a compiler only on a very shallow level. Therefore the reader is encouraged to read further literature as the knowledge of a compilers structure helps a lot in understanding the way a buillt-in functions works and what problems might occur.

As already hinted by the abstract, a compiler consists of a front-end and a back-end, but also a third part that is the middle-end. These three parts sit on top of each other with the front-end on top and the back-end at the bottom and pass down the program as it is translated and optimized or compiled. But communication between the parts does not go only one way and changes that are made in the back-end affect the front-end as well! The first part of the compilation process is the translation of code which is written in some programming language into a so called Immediate Representation (IR) that looks the same for every front-end language and usually is never seen by the user. Any supported programming language (C, C++, Java. . .) is implemented in its own front-end that defines how the language is translated into IR. After that the IR is send to the middle-end, which generally optimizes the IR and then passes the code to the back-end. The back-end first executes further optimizations that are target-specific followed by allocatiing registers and handling relative memory. Finally the code is translated into the assembly language that is supported by the target. After the code is compiled and emmited as an object file it is also linked, which means combining different objectfiles and and assigning absolute memory addresses to tehm. At last the binary file emmited by the linker is loaded into the memory of the processor and then can be executed.

Most Compilers that are used nowadays are built of three basic components which handle different steps in the process of converting human-readable programming language to machine-readable machine code. As does the GNU Compiler collection (GCC) which also can be seen as made of three main parts. The so called front-end, middle-end and back-end. All three parts work more or less independently from each other and communicate over a compiler-specific „language“, which is described a the Intermediate Representation (IR). It is typically never seen by the user and exists for a fact in many different forms [reference] one of which is Register Transfer Language (RTL), which is the lowest-level IR used by GCC. It is the most interesting IR when working on a back-end and will get more attention later in this report. But in before that we need to understand the structure an functioning of a compiler in general.

The first mentioned Front-end resembles the main interaction point between the human programmer and the machine. Front-ends are usually divided by their respective programming languages such as C, C++, Java. . . and have the main task of converting any programming language into unified IR, that can be passed to the middle-end in GIMPLE or GENERIC language. Therefore no matter which language you prefer, in an ideal case code, that is syntactically identical, should not differ after it is processed by the front-end. This is due to the goal of compilers such as GCC and LLVM to support as many languages and machines as reasonably possible while offering the equally good optimization and saving themselves overall work. It obvious that a single compiler for every combination of language and machine would simply not be practical, especially as the optimization taking place in the middle-end follows the same rules for pretty much any architecture. The middle-end main task is basically this sort of optimization, and makes for the main difference between compilers as most compilers offer the same range of front-ends and back-ends. (Part about the optimizations taking place in the middle-end). After all optimizations are through, the middle-end passes IR in form of RTL to the back-end. As you can see the middle-end rarely needs to be modified except for fundamental changes in the compilers architecture such as new kinds of optimizations and „multiple memory handling“ (also Harvard-Architecture (vielleicht)) The back-end is responsible for the final steps of the compilation process as it translates the general RTL IR into specific Assembly commands. It uses some sort of table of available assembly instructions, that is provided, and finds the best fitting instructions. GCC for example uses a Lisp-like language (is this RTL?) that uses something called insns. These combine different properties with the Assembly commands like an equivalent set of actions that are executed, the operands and the constraints it must satisfy. These will be further explained later. The back-end also contains the the code which implements processor-specific built-in functions. Depending on the Compiler architecture the back-end it finally emits IR to the assembler which emits the machine code in assembly or emits assembly itself. Then finally the linker links the assembly code of all program files together and substitutes the offset addresses with absolut addresses to generate the final machine code. reload register spilling register handling endianess wordsize

different parts of an instruction, mention: opcodes, asm instruction, operation, operand, insn, IR, builtin function/intrinsic

6 Insn-Coding

The PPU, which was designed by Simon Friedmann , is a custom processor, that is based on the Power Instruction Set Architecture (PowerISA), which was developed by IBM since 1990. Specifically the PPU uses POWER7 which is a successor of the original POWER architecture and was released in 2010.

PPU paper

mention vector pipeline mention processor cycles.

The PPU's distinct feature is its vector unit or vector extension (VE) that allows for Single Input Multiple Data (SIMD) operations. It is literally an extension to the main processors (MP) architecture, since there are very few ways these both can interact. most importantly all vector instructions that are intended for the VE must pass the MP first. The MP goes through the hole program step by step and passes any vector instruction to the VE whenever it occurs. These instructions then go into the so called vector pipeline that holds all vector instructions. The VE itself then executes all instructions in this pipeline as they occur but and allows for parallel in-order-execution.

The VE was added due to the need for fast handling and writing of the synaptic weights into the array of synaptic values on the HICANN. Hence the vector unit was equipped with an extra bus that connects to the mentioned synapse array. The basic processor does not have access to this bus and therefore must use to VE in order to communicate with the synapse array. On the other side both, the basic processor and the VE are connected to the same memory controller that manages 16 kiB of memory. This memory is used for data as well as the program itself and thus must be handled with some care as programs must not be larger than these 16 kiB. Having the VE and basic processor to share this memory controller without any caching can cause problems at times. An easy example is the sequential storing of a vector in memory followed by a loading instruction that requests the very same address of the vector in order to load one of its elements. This would lead to both instructions being in the wrong order as the vector pipeline delays the vector store instructions way to the memory controller by a few cycles ultimately reaching the memory controller after the load instruction which is passed on directly. But this not only happens for the VE but also can be caused by the compiler as it optimizes memory access mostly for performance. This may result in unintended reordering of memory operations which then leads to errors. Because of this there is the possibility to prohibit such reordering by introducing a memory barrier. A memory barrier basically is a line of code that that splits the remaining code into code that occurs before the barrier and code that occurs after the barrier. It prohibits the compiler from mixing instructions from different sides to the barrier due to reordering. Typically such a memory barrier is called like `asm (:::memory)`, which the compiler recognizes as a memory barrier. On a processor level there exists also a memory barrier that would typically not be called by a user. It's called syncing and on PowerPC the instruction `sync` is used for this. Because the processor itself delays certain instructions to optimize performance (as part of its out-of-order architecture) it may happen that a dependency between two instructions is not detected. This would result in the very same problem as described earlier. Therefore the processor can be instructed to wait until all store and load instructions are executed. The memory controller then sends a signal to the processor that it is all done.

It is fairly obvious that only the second kind of memory barrier helps us in or cause to avoid wrong ordering in the memory controller.Luckily both the VE and the basic processor use the same memory controller which makes `sync` apply in both cases. Also since all vector instructions must go though the main processor first, any hold executing instructions on the MP affects the VE the same way.

Besides the VE and the MP, the memory also provides access to the FPGA through a different interface in order to allow for external access to the memory. This is needed for writing programs into the memory as well as getting results during or after execution. This also allows for communication during runtime of the PPU. Specifically the vector extension allows for either use of 8 element vectors with element being halfword (1 halfword = 2 bytes) sized or 16 element vectors with each element byte sized Thus every vector is 16 bytes or 128 Bits long. This is also the size of each vector register that is available, 32 in total, in contrast to 32 general purpose registers with 32 bit each. Also there is an accumulator featured as part of the vector extension which is used in every arithmetic operation and a condition register which holds 3 bits, that determine which condition applies, for every half byte or nibble, making 96 bit in total. The PPU also features 4 KiB of memory as well as access to the synapse array of the HICANN which holds up to 32x32, thus 1024 different values. To handle the vector unit the instruction set was extended by xz new instructions that partly share their opcodes with existing AltiVec instructions.

check this

add special regs of normal prcessro part, CC, LR, etc.

count number of vector istructions.

theoretical gain in speed when using vector unit compared to normal instructions especially with single load and store. include PPU clock frequencies for each part in calculation.

7 Discussion

Discussion...

8 Outlook

Outlook...

Appendix

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

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

Heidelberg, February 9, 2017

....
(signature)