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 Neural Networks and Implementation in Hicann-DLS

what does hicann stand for?

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.

2.1 Basics of 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 we assume that 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 some 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.

2.2 Implementation in Hicann-DLS

The Hicann-DLS system tries to implement this structure as closely as possible in order to simulate physiological processes in such networks. At its core it therefore has a so called “synaptic array" that connects 32 neurons which are located on a single ship. Each neuron as an output as well as an input which are connected in a way that all outputs

are aligned along one axis and the inputs are collected rectangular to that axis. This gives a total of 1024 synapses which are aligned in an array, hence its name. Inside the neurons the individual input signal is evaluated in regard to a threshold and other parameters which decide whether the neuron is spiking or not. If the neuron is spiking it sends out an output signal to all connected synapses to the output line. Each synapse then multiplies the signal it receives with its weight and sends the result to the input of a neuron. All signals sent by synapses to an input are integrated to a resulting input signal which reaches a neuron. This is not done continuously but discretely and the output signals of all neurons are sent out at the same time. The output signal of each neuron is also sent to an analogue digital converter (ADC) in order to analyze the data in digital form.

The Hicann-DLS system is also equipped with a processing unit that includes a vector extension and some memory for it to operate on. This is the plasticity processing unit (PPU) which is also connected to the synapse array and thus can read and write synaptic weights.

The whole chip itself is also connected to a field programmable gate array (FPGA) that is connected to the synapses as well as the memory of the PPU. In general the PPU is meant to simulate plasticity of the synapses during experiments while the FPGA should be used to initially set up an experiment and record data. Therefore the PPU can also read out spiking times of neurons as they are needed for plasticity. This is realized through the same bus that connects the PPU to the synapse array and virtually the spiking times are saved in the synapse array where they can be loaded by the PPU.

The synapses in the synapse array in detail consist of ....

CURRENT STATE OF PROGRAMMING ON HICANN -> MOTIVATION main feature analogue spiking PPU can read spiking times size and alignment of weights

3 The HICANN DLS system

4 Basic Processor Architecture and PPU specifications

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.

add reference

cite freidmann dissertation

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!

4.1 Plasticity Processing Unit

The PPU was designed by Simon Friedmann and 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. POWER is one of the major CPU architectures that are used nowadays and as such falls under the group of von-Neumann architectures.

PPU paper

4.2 Pipelining

4.3 Memory

Normally the memory of a von-Neumann machine contains both, the program and data. Therefore it is important, especially for larger scale systems, to keep track of memory usage. This is e.g. to prevent malicious programs from accessing memory which is used by other software. For this reason next to all CPUs include a memory management unit (MMU). It handles all memory access of the processor as it can provide a set of virtual memory addresses which itself then transforms into physical addresses. Most

modern MMUs also incorporate a cache that stores memory operations while others are handled and detects dependencies within this cache which it can resolve. This results in faster transfer of data as two or more instructions access the same memory which then is handled in the cache. The PPU though includes only a very simple MMU that does not cache memory instructions and also has matching virtual and physical addresses. The memory of the Hicann-DLS is 16 kiB in total (ranging from addresses `0x0000` to `0x4000`) and both the vector extension and the main processor use the same MMU.

use this:

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.

check this

MMU

4.4 Vector Extensions

Using the same principles as mentioned in the section about von-Neumann architectures, theses is also the possibility to create registers or whole architectures that make use of even more than the common 64-bit. This is mostly wanted for highly parallel processes such as graphic rendering or audio and video processing . But also early supercomputers such as the Cray-1 made use of vector processing to gain performance by operating on multiple values simultaneously through a single register. This could either be realized through a parallel architecture or more easily through pipelining the instruction on one vector over its elements. The latter one makes sense since there are typically no depen-

reference

reference

dencies between single elements in the same vector. Nowadays many of the common architectures support vector processing. A few examples of these are:

- x86 with SSE-series and AVX
- IA-32 with MMX
- AMD K6-2 with 3DNow!
- PowerPC with AltiVec and SPE

As mentioned these were mostly intended for speeding up tasks like adjusting the contrast of an image. There is also the possibility to vectorize loops in programming if there are no dependencies between loop cycles.

4.4.1 AltiVec Vector Extension

In our case we take a special interest in the AltiVec vector extension which developed by Apple, IBM and Motorola in the mid 1990's and is also known as Vector Media Extension (VMX) and Velocity Engine. The AltiVec extension provides the processor with a single-precision floating point and integer SIMD instruction set. The vector registers are 128-bit each and 32 in total. These can either hold sixteen 8-bit `char`s, eight 16-bit `short`s or four 32-bit `int`s or single precision `float`s, each signed and unsigned. Single elements of these vectors can only be accessed through memory because there is no instruction that combines scalar register with vector registers. Except for one type of instruction that "splats" the value of a scalar register into all elements of the vector register. The reason we take such an interest in this vector extension is that it resembles most characteristics of the PPU's vector extensiond is already implemented in the PowerPC back-end of GCC. There a few differences though:

http://www.nxp.com/assets/doc
manuals/ALTIVECPEM.pdf

First the PPU's VE uses a conditonal register (CR) to perform instructions only on those elements of a register, that meet the condition, which is specified by the user, while the AltiVec VE utilizes the CR which included in the PowerPC architecture. This results in not allowing selective operations on individual elements through the CR but allows for checking if all elements meet the condition in a single instruction. If element-wise selection is needed AltiVec offers this through vector masks.

The AltiVec VE has two register on its own though, which are the VCSR and VRSAVE registers. The Vector Status and Control Register (VSCR) is responsible for detecting saturation in vector operations and decides which floating point mode is used. The Vector Save/Restore Register (VRSAVE) assists applications and operation systems by indicating for each VR if it is currently used by a process and thus must be restored in case of an interrupt.

Both of these register are not available in the PPU's VE but would likely not be needed for simple arithmetic tasks as the PPU is meant to perform.

4.4.2 s2pp Vector Extension

Vector condition register Vector accumulator

use this:

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.

vector register file
ALU FPU memory memory controller = MMU clockcycle pipeline

5 Compiler Structure

As already hinted by the abstract, a compiler consists of a front-end, 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 the very top and the back-end at the bottom and pass down the program as it is translated and optimized or "compiled".

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 is 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)).

5.1 back-end

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

5.2 instrinsics

6 Extending the GCC Back-End

To allow the compiler to differ which insns we acutally want to use as a built-in. The rs/6000 back-end has a special file which contians macros for every builtin function that will be available. These macros will be loadad in the rs6000.c file and also handled there. But before this can happen we have to create a built-in macro for our newly created insn. WE must first differ between the different kinds of macros. the rs/6000 back-end already provides us with templates for differnt altivec built-ins. These templates basically help identifying the right macros later on and avoid naming conflicts. Next we have to differ between the number of input opernad our built-in function is going to have and the number in th etemplate name is meant to be equal to th enumber of input operands . all Macros expect output operand except for the X-names macros. which are special cases we will emphasize later. We now choose `BU_ALTIVEC_2` because our newest insn has to input operands and add the following lines:

the first argument is the name of the macro and will be used by the compiler internally. the user only gets interact with the second string which is the new built-ins name for the front end. (for altivec built-ins the tempalte adds `__builtin_altivec_` in front of the given name). Next the attribute for the builtn must be given. typically this `CONST` but for builtins that load or store in the memory this is `MEM` for example. The last argument is the name of the insn we decided on earlier.

You can also add an „overload“ macro for your builtin as this allows for type checking and reducing builtins for different vector modes to just on final builtin. here we first need a Macro name and second the function name we want to use later (this functon name is preceeded by `__builtin_vec_`)

After we fully implemented our builtin functions we will finalize the functions by defining their final names. There are two naming conventions we can follow and typically it is advised to follow both in order to offer the user a set of alternatives. The first naming convention is the nameing convention already used by the AltiVec extension which starts every builtin function with vec_ and chooses a general descriptive name which is derived from the used Assembly macro if possible. This is the most intresting case for our exemplary Altivec builtin function. because there is already a vec_ mul builltin we will choose the descriptive abbreviation vec_ smul for scalar multiplication. The other naming convention gets intresting when extending the backend with another vector extension. If there were an S2PP vector extension, all assembly macros would begin with fxv which is a good alternative for the vec_ prefix altivec uses for its builtins. then it is a good idea so offer all assembly macros with their already known names e.g. fxvaddbm would become fxv_ addbm. for more genral builtins that combine the halfword and byte macro it is recommended to leave out the letter h or b likewise. rendering fxv_-addm the builtin function that works with both V8HI and V16QI vectors.

7 Test cases

not all cases can be handles by the aforementioned method. for example a builtin function wihout a return type. in htis case we use the BU_ ALTIVEC_ X template thaat slightly differs from other tmeplates. . . . . .

asm tests user level tests compiler tests

8 Results

still compiles old code optimizes runs on PPU

9 Discussion

Discussion...

10 Outlook

Outlook...

Appendix

Notes

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 11, 2017

.....
(signature)