# RISC-V Coprocessor interfaces for the use in Convolutional Neural Network accelerators

B.Eng Maximilian Konstantin Morgalla
*Masters Student, Computer Science*
*University of Heidelberg*
Heidelberg, Germany
lq196@uni-heidelberg.de

***Abstract*—RISC-V is an open and free to use Instruction Set Architecture (ISA). The base instruction set allows for modular extensions and there are many standard extensions available. The ISA also allows the use of custom instructions on top of that. This paper describes how the RISC-V ISA can be enhanced with custom instructions. Coprocessor interfaces have been developed additionally, these allow custom instructions to be passed to and to be executed by a coprocessor concurrently. We focus on the Rocket Custom Coprocessor Interface (RoCC) in terms of coprocessor interfaces. HWPE is a Convolutional Neural Network (CNNs) accelerator which is implemented as a coprocessor and will serve as an example of how RISC-V can be extended by a coprocessor.**

***Index Terms*—RISC-V, RoCC, Rocket, Coprocessor, CNN**

## I. RISC-V - An Extendable ISA

### A. Introduction and Motivation

RISC-V is a free and open instruction set architecture (ISA), which has been developed at the University of California Berkeley (UCB). It has good software and tooling support, including the ISA simulator Spike [1], the GNU C Compiler gcc and Debugger gdb. The available tools allow the interactive editing of the machine code and manipulation of the CPU state in order to aid in the development of accelerators and cores based on RISC-V.

What started out as a project for researchers, has gained traction in industry. Several companies are developing cores for their own application specific needs based on the ISA laid out by the RISC-V specification. RISC-V is aimed at a variety of computational environments, ranging from a small low power embedded system, to a personal computer and even up to a supercomputer.

### B. Standard Extensions

The base RISC-V instruction set is available in wordlengths of 32, 64 and 128 bits. At this present time the 32 and 64 Bit versions of the integer ISA RV32I and RV64I have been specified and frozen. The 128 Bit version is still in development. With approximately 40 instructions the base instruction set is kept very lean.

The base instruction set is so minimal that many instructions which one would expect from a general purpose core are actually available through the standard extensions.

The extension of and the adaptability to different environments and its needs, is at the core of the RISC-V ISA. In order to enable the expansion of the ISA through standard extensions care has been taken to use the bits of the instruction encoding wisely and sparingly. In addition to this, there are bits which have been reserved for specific custom application specific instruction extensions.

TABLE I
Common RISC-V Standard Extensions

| Name | Description |
|---|---|
| M | Multiplication and Division |
| A | Atomic Instructions |
| F | Single-Precision Floating-Point |
| D | Double-Precision Floating-Point |
| **G** | General Purpose and shorthand for MAFD |

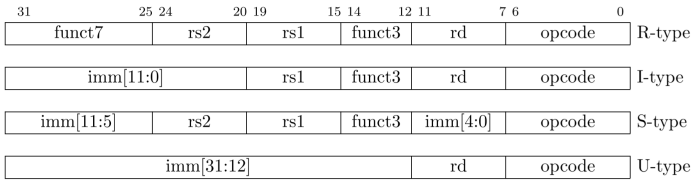



Fig. 1. RISC-V base instruction formats. Source: [2, Fig. 2.2]

### C. Custom Instructions

Fig. 1 shows the 4 basic instruction formats *R-type*, *I-type*, *S-type* and *U-type* as defined by the RISC-V base instruction set. All of these instruction formats have been designed in order to keep the decoding hardware as minimized as possible. For example the fields defined for the source registers *rs1* and *rs2*, as well as the destination registers *rd* are kept in the same place.

The custom instruction type, used by the Rocket Custom Coprocessor interface (RoCC), is based on the R-Type instruction format. The inner workings of the custom instruction format is detailed in chapter II-F on pg. 3.

All instruction formats share a 7 Bit opcode, which identifies the instruction. Furthermore, depending on the instruction type, additional fields can be used to pass arguments to or to encode information directly into the instruction, like constants. These fields are *funct7*, *fucnt3* and *imm* (immediate).

Fig. 2 shows the coding of the opcode to different instructions. The 4 rows show bits [6:5] while the columns show

| inst[4:2] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| inst[6:5] | | | | | | | | (> 32b) |
| 00 | LOAD | LOAD-FP | *custom-0* | MISC-MEM | OP-IMM | AUIPC | OP-IMM-32 | 48b |
| 01 | STORE | STORE-FP | *custom-1* | AMO | OP | LUI | OP-32 | 64b |
| 10 | MADD | MSUB | NMSUB | NMADD | OP-FP | *reserved* | *custom-2/rv128* | 48b |
| 11 | BRANCH | JALR | *reserved* | JAL | SYSTEM | *reserved* | *custom-3/rv128* | ≥ 80b |

Fig. 2. RISC-V opcode mapping. Source: [2, Table 19.1]

the coding for bits [4:2]. Bits [1:0] are not part of the table and are set to *0b11* for all instructions. This effectively makes RISC-V instructions 30 bits wide instead of 32 bits.

The 4 opcodes which can be used for custom instructions can be taken from Fig. 2. These are custom-0/1/2/3, of which custom-0 and custom-1 are available in all versions of the ISA. Custom-2 and custom-3 instructions, however, are reserved in RV128, but may still be used in RV32 and RV64.

As 2 or 4 custom instructions are most likely not enough to encode all the custom instructions, one would like to implement further bits in the instruction format can be used for the encoding purpose.

Fig. 1 shows the instruction coding of the R-Type instruction. It has 2 bitfields for encoding extra information, which is the 3 bit field *funct3* and the 7 bit field *funct7*. The *funct3* bitfield is suitable in order to specify if one of the 3 registers refers to a register in the core or in the accelerator. The special usage of the *funct3* bitfield in the RoCC custom instruction format is described in detail in chapter II-F on pg. 3. The *funct7* bitfield can then be used for the actual encoding of the instruction. A total of 512 (4*128) instructions can be encoded with this bitfield when all of the 4 custom instruction opcodes are utilized.

The communication with an accelerator depends on the implementation of the core vendor and the available interfaces will be explored in chapter II-A on pg. 2.

When a RISC-V core encounters a custom instruction, it will in principle hand it over to the appropriate accelerator for execution. However, RISC-V doesn't yet define an interface for the communication with the coprocessor.

## II. RISC-V - Coprocessor Interfaces

### A. What is a coprocessor?

Any circuit that is specialized in one or more tasks and is faster at completing it than the primary Central Processing Unit (CPU), can be referred to as a coprocessor or an accelerator. In general the coprocessor executes the task in parallel while the CPU can continue doing other tasks. These coprocessors can fulfill a variety of tasks.

In the 1980s Floating Point Units (FPUs) started out as external coprocessors in the form of the Intel 8087 floating point coprocessor. Later coprocessors moved to the same Integrated Circuit Die as the CPU and are included if the device may benefit from the specialized computing core. Modern examples of coprocessors include ones for the handling of sensor data in mobile devices like in the Apple M7 and M8 SOCs and circuits for artificial neural networks for vision and cognitive tasks.

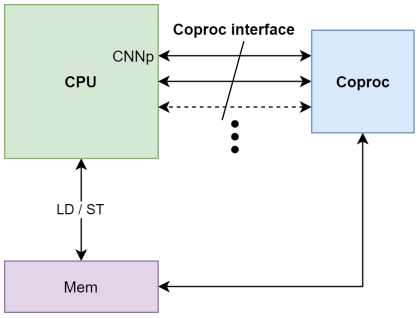



Fig. 3. Connecting a coprocessor to a CPU.

There are benefits of having these specialized computing cores being a separate entity and not a part of the core, as in a functional unit of the pipeline. As for example, in a multi core implementation not every core will need access to the specialized coprocessor simultaneously. This allows the integration of different coprocessors for different tasks which can then be shared between the cores to greatly increase performance in a multitude of specialized workloads.

### B. Coprocessor Interface Considerations

Fig 3 shows how a coprocessor can be connected to a CPU in principle. In a tightly coupled coprocessor system the most essential part of the coprocessor interface is the connection from the coprocessor to the CPU. The CPU has to be able to pass instructions to the coprocessor and the coprocessor needs to return the results. On top of the required exchange of instructions and their accompanying data also some basic status signals are needed for flow control. The *coproc* interface should introduce as little overhead as possible. This overhead may be in the form of latency and interruptions and stalls of the processor pipeline.

For some accelerators it may not be feasible to push all the required data over the *coproc* interface. This is due to the fact that the core has to first load the data into one of its registers before it can push it over to the coprocessor. A direct interface to the memory is needed for these coprocessors.

### C. Rocket Chip

The University of California Berkeley (UCB) develops a line of 64 bit processors based on RISC-V which are part of the SoC generator named Rocket Chip. The generator allows parameterization and a selection of different SoC variants. These include the superscalar out-of-order Boom [3] core and the Rocket [4, p. 4] core. Rocket is a 5-stage in-order scalar processor that implements the 64 Bit RV64G (MAFD) standard.

The cores and generator are implemented in Chisel (Creating Hardware in a Scala Embedded Language), which is a hardware description language embedded into Scala. Chips implemented in Chisel can either be transcompiled into a fast C++ model for cycle accurate simulation or a verilog model for a FPGA (Field-Programmable Gate Array) or an ASIC (Application-Specific Integrated Circuit) target.

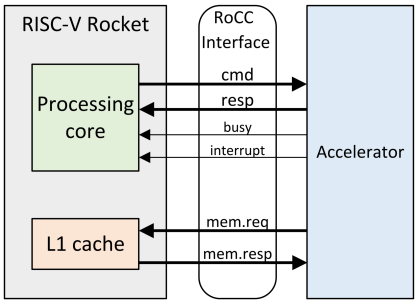



Fig. 4. A simplified view of the RoCC interface. Source: [5, Fig. 7]

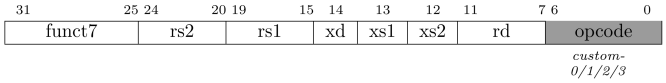



Fig. 5. RoCC instruction format. Source: [8, Fig. 2.3]

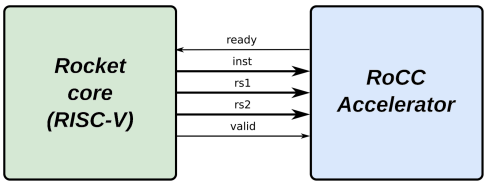



Fig. 6. RoCC cmd interface signals. Source: [8, Fig. 2.4]

### D. Rocket Custom Coprocessor Interface (RoCC)

Depending on how tightly a coprocessor is to be integrated into the core, Rocket Chip supports 3 different mechanisms of connection: [3, p. 4-5]

1) Extend the RISC-V ISA and attach the accelerator to the processor pipeline.
2) Attach the accelerator via the RoCC interface for a more decoupled solution.
3) A fully independent accelerator can be instantiated in its own Tile (a cache coherent container) and connected via TileLink (a protocol framework to describe cache coherence transactions and protocols) [3, p. 6].

The Rocket Custom Coprocessor Interface (RoCC) allows the connection of a coprocessor to a Rocket core and this will be explored in the following chapter.

### E. RoCC - Interface description

As can be seen in Fig. 4 the RoCC interface is divided into four sub-interfaces.

The 4 sub-interfaces of the RoCC interface are:

1) *Command Interface* in order to pass commands to the accelerator
2) *Response Interface* in order to receive results from the accelerator
3) *Memory Request Interface* allows the accelerator to make requests to the memory
4) *Memory Response Interface* allows the accelerator to receive responses from the memory

In addition to the signals of the 4 sub-interfaces, the RoCC interface also has a few status signals including a *busy*, an *interrupt* and an *exception* line which is not seen in Fig. 4, but present in the source code [6]. The *interrupt* line can be used by the accelerator in order to synchronize with the core and the *exception* line signals to the accelerator when the core has run into an exception. The *busy* signal is used to let the core know that the accelerator is currently busy doing work [7, p. 6].

The command and response interface are based on the decoupled interface available in chisel.This means they have buffers and forward and reverse flow control.

Forward flow control is in the form of a *valid* signal. When the *valid* signal is asserted, core B will process the data it receives from core A.

Reverse flow control is implemented as a *ready* signal. In the backwards direction, core B can set the *ready* signal to low in order to notify core A that it is currently unable to process data.

In the next sections we shall look at the four interface modules of the RoCC interface in more detail.

### F. RoCC - RoCC custom instruction format

Fig. 5 shows the custom RoCC instruction format. It is based on the RISC-V R-type format which can be seen in Fig. 1.

The bitfields *rs1*, *rs2*, *rd* are pointers to the 32 available general purpose registers in RV32. In the following we shall refer to them as *inst_rs1*, *inst_rs2* and *inst_rd*. These are the 5 bit register numbers/pointers. If the value of the register is meant, then we shall refer to them as *rs1*, *rs2* and *rd*.

As can be seen in Fig. 5, the *opcode* can take on the 4 custom encodings described in Fig 2.

The *funct3[14:12]* bitfield of the R-type instruction format has been split into *xd*, *xs1* and *xs2*. These fields are used to specify if the pointers *inst_rd*, *inst_rs1* and *inst_rs2* refer to a register in the core or in the accelerator. When set to *1* they refer to a register in the core and *0* means it is a register in the accelerator.

- **xd** bit is set when the core expects a result to be returned to the register pointed to by *inst_rd*. When set to *0* the core expects the instruction to have executed properly after sending it to the accelerator.
- **xs1/2** bit is set when the register pointed to by *inst_rs1/2* is a valid source register and its value should be transferred to the accelerator.

The behaviour of *xd*, *xs1* and *xs2* is described in the following description of the command and response interface.

### G. RoCC command (cmd) interface

Fig. 6 shows the cmd interface. It is used to pass instructions and the required data to the accelerator. The interface signals are described below.

- **inst** The entire 32-bit instruction is passed to the RoCC accelerator.
- **rs1/2** If *xs1/2* is set to *1* the value of the register pointed to by *inst_rs1/2* is passed to the accelerator.

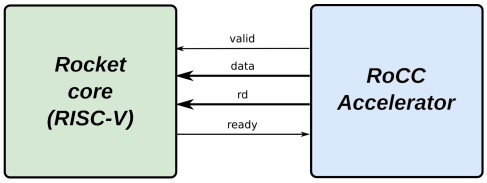



Fig. 7. RoCC rsp interface signals. Source: [8, Fig. 2.5]

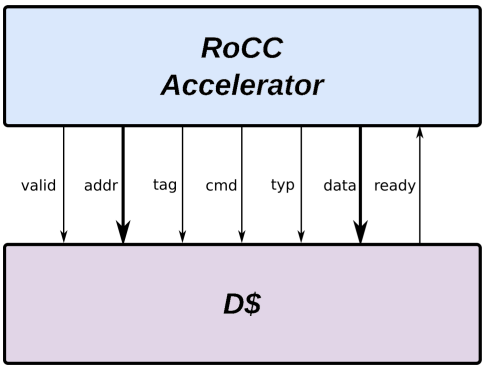



Fig. 8. RoCC mem req main signals. Source: [8, Fig. 2.6]

### H. RoCC response (rsp) interface

Fig. 7 shows the response interface on which results from instructions issued to the accelerator are returned.

- **data** The result data to be written into the register of the core.
- **rd** The 5 bit pointer to the register into which the result should be written. This has to be the same value as *inst_rd*, because the core has reserved this register until the result is returned.

### I. RoCC memory request and response interface

Fig. 8 shows the interface signals of the memory request interface. The signals of the request and response interfaces are the same and their direction is inverted. The only difference is that the *memory response interface* doesn't have a *ready* signal. This means that the responses of the memory interface have to be handled directly and the memory interface has to account for that. Responses might be received out of order, especially because the core and accelerator share the connection to the Level 1 Data Cache (L1-D$).

- **addr** a 32 bit memory address (RV32).
- **tag** 8-bit *tag* to uniquely identify memory requests, which can be served out of order.
- **cmd** 5-bit to select the operation of which the most important ones are load (*0*) and store (*1*).
- **typ** 3-bit to specify the width of the response data. (*0b000* = 8-bit, *0b001* = 16-bit, *0b010* = 32-bit, *0b011* = 64-bit)
- **data** 32 or 64 bit bus to transfer data in case of a store operation or receive data when a load is performed.

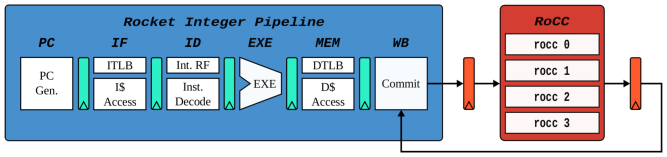



Fig. 9. Position of the RoCC interface in the pipeline. Source: [8, Fig. 2.11]

### J. RoCC - Extended interface

The *extended RoCC interface* includes some extensions in order to allow the development of more advanced coprocessors.

- **aUTL** *arbitrated Uncached Tile Link* provides a connection to the outer memory system. The Hwacha vector coprocessor uses this interface to fetch data directly from the L2 Cache. [9, p. 6-7]
- **FPU** is an interface used in order to connect to the *floating point unit*.
- **PTW** connects to the *Page Table Walker*, which is useful for coprocessors that want to do memory operations independently from the Translation Lookaside Buffer (TLB) of the core, as in a memory-to-memory copy coprocessor [10].

### K. RoCC - Limitations and Performance

This section serves as a brief summary of a latency study performed on the RoCC interface [8, p. 22-30,63-65]. The short version is that the RoCC interface is not suited for tightly coupled coprocessors, which is also mentioned in the technical report on the rocket chip generator [4, p. 4].

The RoCC interface has a high latency, which has several reasons. One of them is the position of the RoCC interface in the rocket processor pipeline as seen in Fig. 9. It is positioned after the Write Back (WB) stage. This has the advantage that only instructions that are guaranteed to be executed are sent to the coprocessor. In return the coprocessor doesn't need to support exceptions. However, the instruction that is sent to the coprocessor effectively occupies the Execute (EXE), Memory (MEM) and Write-Back (WB) of the rocket core, doing nothing but to prevent the rocket core from executing an instruction.

The *RoCC command router* [8, p. 24] also adds to the latency. It sits after the WB stage, buffers and routes the instructions to the correct coprocessor whilst adding one clock cycle of latency.

Retrieving results from the RoCC interface comes also with a penalty, as can be seen in Fig. 9, the results from the RoCC accelerator are fed back into the Write-Back stage. There is only one port, so either the core or the RoCC interface can write to the register file (RF) in a single clock cycle. The rocket core takes precendence and a response from the RoCC interface has to wait for an instruction that doesn't write to the RF. As a workaround a dummy instruction, which doesn't write to the register file, can be inserted in order to gain a free write-back slot and prevent a long pipeline stall.

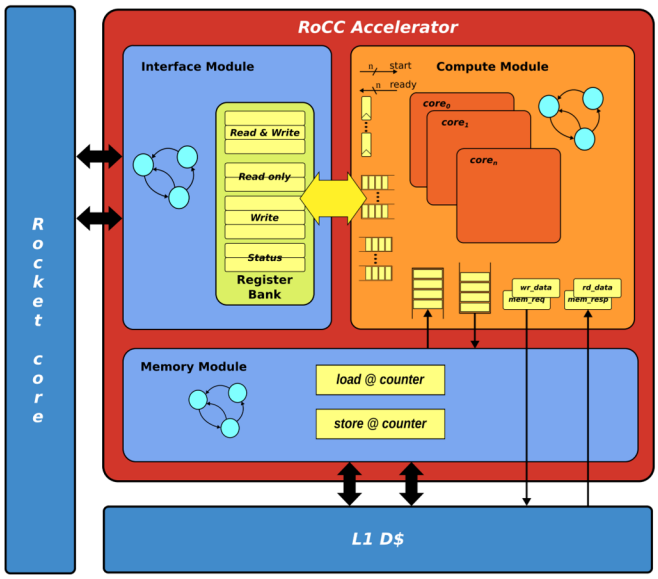



Fig. 10. Architecture of a general RoCC Accelerator. Source: [8, Fig. 5.4]

### L. RoCC - General accelerator architecture

Fig. 10 shows what a general architecture of a coprocessor connected via RoCC max look like. It consists of an *interface module* which handles communication with the CPU, a *compute module* which can be seen as the actual accelerator and the *memory module* handling memory transfers between the *compute module* and the L1-D$.

The *interface module* handles the *ready* and *busy* signals of the command and response RoCC sub-interfaces. It also contains the register bank with different types of registers. The configuration registers might be set to be read-only for the compute-module and write-only from the side of the interface-module, whilst the result registers should be writable from the compute side and read-only for the interface-module.

If one has a separate memory-module it allows the compute-module to be as simple as possible. It also doesn't have to deal with the fact that memory requests may be served out of order. A generic memory module could support frequently used memory access patterns, like streaming memory for encryption. Parallel access to the memory, using multiple compute cores, could also be handled by a simple arbitration mechanism, like round-robin or another mechanism, suited for the compute cores.

### M. RoCC - Conclusion

The RoCC interface exposes the L1-D$ to the accelerator ensuring the data in the cache is kept coherent. In order to obtain a higher bandwidth the extended RoCC interface may be used. Whilst the interface is not suited for tightly coupled accelerators the memory interface can be used to exchange larger amounts of data. The command interface can then be used in order to configure and to start the accelerator. While the RoCC interface signals are kept to a bare minimum, adding some more signals and moving the interface to an earlier stage of the pipeline would have allowed a higher throughput operation.

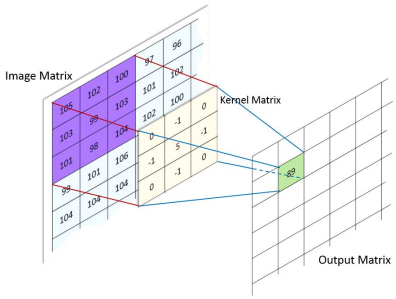



Fig. 11. Image Processing Convolution Operation. Source: [11]

## III. Convolutional Neural Networks

### A. CNNs - Introduction and Uses

Deep neural networks have multiple hidden layers between the input and output layer. A Convolutional Neural Network (CNN) is a type of deep neural network. CNNs are particularly well suited for image classification, but can also be applied to other types of data. In general any type of data which can be made to look like a picture (a grid of pixels) can be processed by a CNN.

### B. CNNs - Types of Layers

A CNN is made up of different types of layers. The different layers can be used multiple times and chained together to form networks of any complexity. Most of the research into CNNs is focused on the choice of parameters for the layers, the decision of how many layers to use and their order.

There are 4 main layer types in a CNN:

1) Convolutional Layer, which filters the input using convolution kernels.
2) ReLU Layer, this filters out negative values (Rectified Linear Units).
3) Pooling Layer, zooms out the image.
4) Fully Connected Layer, higher view and reasoning.

In the following sections we will briefly introduce these layers and how they are calculated.

### C. Convolutional Layer - Convolution Intro

Fig. 11 shows the convolution operation used in image processing. In convolution of an image, a filter is used to process a portion of the image. To calculate the result of the convolution, the filter is first flipped vertically and horizontally. After that every element of the image window is multiplied with the corresponding element in the filter. In the end the results from the multiplications are added up to form one result point in the output matrix. The calculation of the convolution requires a lot of operations (ops), f.e. a 3x3 filter requires 9 multiplications and 9 addition operations to calculate a single point in the output matrix. When the special processing of the borders of the image is ignored, an image with a size of

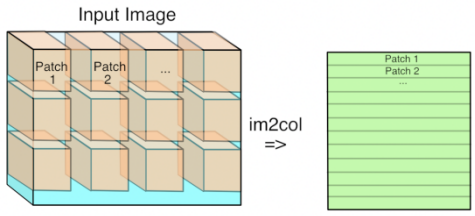



Fig. 12. Flattening of patches via im2col. Source: [12]

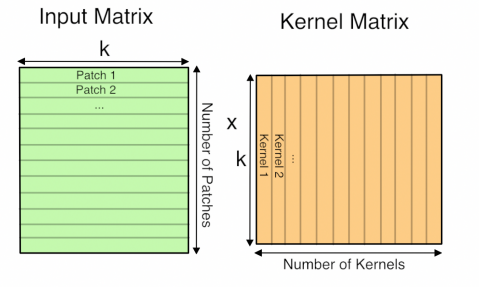



Fig. 13. After im2col convolution can be calculated with a matrix multiplication. Source: [12]

256x256 pixels already requires 1.18 million $(256 * 256 * 18)$ ops in order to convolve the entire image.

Depending on the size and the weights chosen, traditional image processing filters can perform many different operations. The most important filters are for edge detection and the smoothing and sharpening of an image.

### D. Convolutional Layer - Calculation Cost

The convolutional layer in a CNN takes the convolution task and increases the required amount of calculation by 2 more dimensions. The first increase in dimension is due to the support of convolution kernels with multiple channels, this makes the convolution kernels 3 dimensional. A convolutional layer will have many of these 3 dimensional filter kernels.

In turn the convolutional layer is the most compute intensive of the layers and requires about 90% of the processing time. When the processing time of the convolutional layers and fully connected layers are taken together, they require about 95% of the time. This major calculation expense makes the convolutional layer the prime target for the development of accelerators.

### E. Convolutional Layer - Optimization via im2col

Traditionally a convolution over an entire image is calculated by sliding a window across the image and calculating every point in succession. An image is usually stored row-wise as a matrix in memory, meaning that one row of pixels is stored after the other. When accessing the image in memory for the convolution task the rows are stored far apart and depending on the data type at least one memory access has to be made for each row. Ideally the image would be stored in a way that the memory accesses are contiguous to achieve the highest memory throughput.

Fig. 12 shows how the stencil operation can be turned into a matrix multiplication via *im2col*. *Im2col* writes the patches of the input image, which is divided into two dimensional patches for the convolution operation. This is done by flattening the patches and writing each of them into a line of the input matrix.

After patches of the input image have been flattened via *im2col*, the convolution can be calculated using a matrix multiplication as seen in Fig. 13. There are many highly optimized libraries for matrix multiplication and it can be computed in a highly parallel fashion.

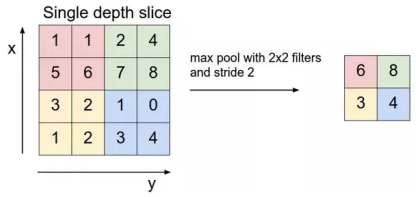



Fig. 14. Max pooling with a 2x2 kernel. Source: [12]

### F. ReLU Layer

The Rectified Linear Unit (ReLU) layer is an activation layer. Activation layers follow a convolutional layer. The ReLU layer applies the function $f(x) = max(0, x)$ to the feature map, this effectively removes negative values from the feature map. ReLu has proven itself as a fast way of training without loosing generalization accuracy [13].

### G. Pooling Layer

The pooling layer reduces the size of the feature map, effectively zooming it out. This makes the neural network more tolerant to slight variations of the features, like rotations and shifts.

Fig. 14 shows the most commonly used form of pooling which is max pooling with a kernel size of 2x2 and a stride of 2.

### H. Fully Connected Layer

In a fully connected (fc) layer all neurons receive all activations of the previous layer. In a CNN one or more *fc* layers are used to provide higher level reasoning.

To calculate a *fc* layer the *input vector* of the activations of the previous layer are multiplied by the weights that every neuron has learned. The matrix vector multiplication is shown

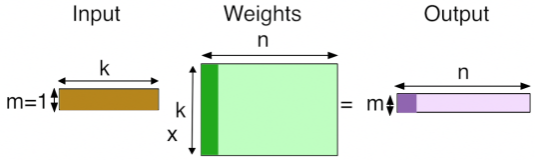



Fig. 15. Calculation of a fully connected layer via matrix vector multiplication. Source: [12]

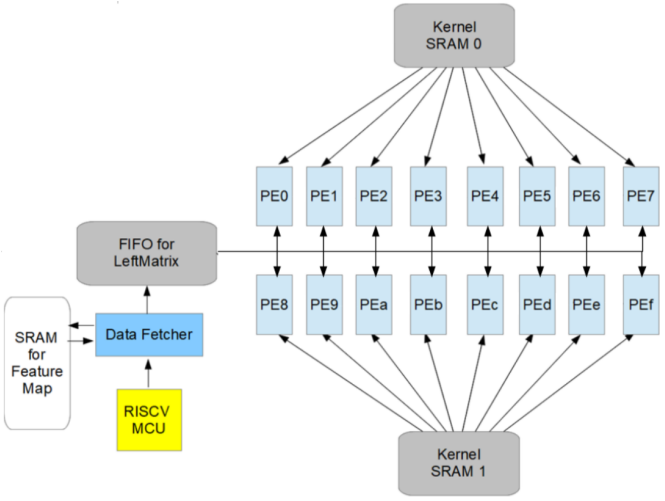



Fig. 16. Architecture diagram of HWPE. Source: [15, Fig. 3]

in Fig. 15. After the *input vector* has been multiplied by the *weight matrix*, a vector with the biases is added to the result. This step has been omitted in Fig. 15.

### I. CNN - Conclusion

This section summarizes the basic steps required to calculate inference of a convolutional neural network (CNN). The convolutional layer requires the most processing power and lends itself to the development of accelerators. To optimize the calculation of the convolutional layer it can be turned into a matrix multiplication via *im2col*. Both the convolutional layer and the fully connected layer require a matrix multiplication. An accelerator for a CNN can optimize this operation to speed up the calculation of a CNN greatly.

An accelerator that does this is detailed in the following section.

## IV. HWPE - Intro

*HWPE* is a CNN hardware accelerator for RISC-V. It uses the extension accelerator interface (EAI) of the Hummingbird E200 MCU [14]. The signals of the EAI interface are almost identical to those found in the *RoCC* interface.

Features of *HWPE*:

- Calculation of convolution layer and ReLU layer.
- Transformation of the convolution into a matrix multiplication via *im2col*.
- Kernel sizes can range from 3x3 up to 11x11.
- Supports different data types: Int8, Uint8, Ternary and Exp4 (Coding can be found in [15, p. 6])
- 16 dot-product operations on 64-bit (8 Int8, 16 Exp4, or 32 Ternary) operands per cycle. For Int8 this would result in 128 $(16*8)$ multiply and accumulate (mac) operations per cycle.

### A. HWPE - Architecture

Fig. 16 shows the architecture components of *HWPE*.

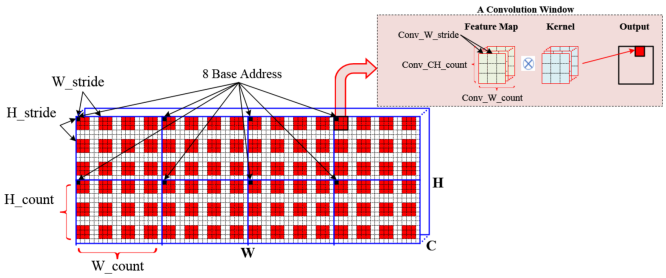



Fig. 17. The 8 convolution windows. Source: [15, Fig. 8]

| Instruction | funct | rd | xd | rs1 | xs1 | rs2 | xs2 |
|---|---|---|---|---|---|---|---|
| HWPEWriteFmapAddrReg | 1 | addr_idx | 0 | addr1 | 1 | addr2 | 1 |
| HWPEWriteCfgReg | 2 | —— | 0 | cfg | 1 | cfg | 1 |
| HWPEMatrixMac | 4 | —— | 0 | W/H_count | 1 | H/W_stride | 1 |
| HWPEWriteAccReg | 8 | AccReg_ID | 0 | M(idx) | 1 | PE_ID | 0 |
| HWPEReadAccReg | 16 | M(idx) | 1 | Acc_Reg_ID | 0 | PE_ID | 0 |
| HWPEReLUMemWriteAccReg | 32 | —— | 0 | M(addr) | 1 | AccReg_ID | 0 |
| HWPEReset | 64 | —— | 0 | —— | 0 | —— | 0 |

Fig. 18. Custom instruction table of HWPE. Source: [15, Table 2]

1) *PEs*: There are 16 Processing Elements (PEs), which are divided into two rows. These store a column of the input matrix (*LeftMatrix*) and a column of the convolution kernel within 64-bit registers. They also have eight 32-bit result registers which are called *AccRegs*.
2) *Kernel SRAMs*: The convolution kernel is rearranged and stored in two large SRAMS. One SRAM serves 8 *PEs*.
3) *Data Fetcher*: The data fetcher accepts commands from the MCU. It fetches and rearranges data from the Feature Map stored in the *Feature Map SRAM* and inserts it into the FIFO.
4) *Feature Map SRAM:* The SRAM for the feature map is divided into two parts, a left and right half. There are two memory interfaces to retrieve two values at the same time.

### B. HWPE - Instructions

Fig. 18 shows the 7 available custom *HWPE* instructions and their coding. The usage of the instructions is detailed below, all parameters can be found in the specification [15, p. 4-6]:

1) HWPEWriteFmapAddrReg: This is a configuration instruction to set the 8 initial points of the convolution windows as seen in Fig. 17.
2) HWPEWriteCfgReg: This is another configuration instruction to change settings like the size and dimension of the convolution kernel, data type and width of the convolution window.
3) HWPEMatrixMac: Starts the convolution computation. The two arguments configure the iterations needed within the convolution window and the strides.
4) HWPEWriteAccReg: Write a value from a general register of the MCU to a register of a PE. This can be used

to configure an initial value of the PE, which is equal to a bias.

5) HWPEReadAccReg: Read a register from a PE back to the MCU. Also contains a bit, that when set will continue the convolution after the registers are moved out.
6) HWPEReLUMemWriteAccReg: Execute ReLU operation, transfer the 32-bit *AccRegs* to 8-bit and store the result at the provided memory address. This operation requires 4 cycles.
7) HWPEReset: Synchronous reset of all flip flops except the *AccRegs* of the *PEs*, which can be reset using the *HWPEWriteAccReg* instruction.

### C. HWPE - Usage

To use *HWPE* it has to be configured by using the configuration instructions. When a round of convolution has completed an interrupt will be generated. The result is moved out and the task is resumed until the entire convolution task has completed.

A more detailed version follows:

1) Configure the eight base registers with the *HWPEWriteFmapAddrReg (1)* instruction. Configure the remaining parameters H_count, W_count, K_count, H_stride, W_stride, Kernel_size and Kernel_W_offset.
2) The convolution starts. The *DataFetcher* fetches a 64-bit row every clock cycle and inserts it into the FIFO.
3) Data is fetched, fed into the FIFO until 128 output points are calculated. At this point an interrupt is generated and execution of *HWPE* is halted.
4) The MCU moves the *AccRegs* either into its own general purpose register by using *HWPEReadAccReg (5)* or into memory with *HWPEReLUMemWriteAccReg (6)*.
5) The convolution continues until the entire plane has been covered, while interrupts are generated after every batch.
6) Now the convolution of the entire plane has been calculated for 16 convolution kernels. To calculate the next 16 kernels the base register of the convolution windows and the address of the next 16 convolution kernels in the kernel SRAM needs to be set.
7) A new convolution task can be started by going back to the first step.

## V. Conclusion and Further Work

The RISC-V ISA is a modular instruction set, that allows to be extended with custom instructions. This makes it a good candidate for the development and integration of coprocessors that can execute the custom instructions.

There aren't many coprocessor interfaces available for RISC-V. The most well documented interface is the Rocket Custom Coprocessor interface (RoCC) of the Rocket chip. While it suffers from some performance issues, particularly when retrieving results, it can be expanded by adding more status signals and moving it to an earlier stage in the processor pipeline.

*HWPE* is a simple CNN accelerator with 7 custom instructions for the Hummingbird E200 MCU. The custom instructions build upon the RISC-V instruction set. *HWPE* supports the most computationally demanding part of a CNN, which is the convolutional layer. The ReLU layer is the most commonly used activation layer and is also supported. In future work the integration of *HWPE* to a different coprocessor interface can be implemented and tested.

## References


[1] A. Waterman and Y. Lee, "Spike, a risc-v isa simulator." [Online]. Available: https://github.com/riscv/riscv-isa-sim/

[2] A. Waterman and K. Asanovic, "The risc-v instruction set manual-volume i: User-level isa-document version 2.2," RISC-V Foundation, Tech. Rep., 2017. [Online]. Available: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[3] C. Celio, D. A. Patterson, and K. Asanovi, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[4] K. Asanovi, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] S. Savas, Z. Ul-Abdin, and T. Nordstrm, "Designing domain-specific heterogeneous architectures from dataflow programs," *Computers*, vol. 7, p. 27, 04 2018.

[6] "Sourcecode for rocc interface (scala)." [Online]. Available: https://github.com/freechipsproject/rocket-chip/blob/master/src/main/scala/tile/LazyRoCC.scala

[7] C. Yarp, "Cs250 discussion 2 riscv, rocket, and rocc," slides from the Berkeley VLSI Systems Design Lab. [Online]. Available: https://inst.eecs.berkeley.edu/~cs250/sp16/disc/Disc02.pdf

[8] M. Poncino and D. Pala, "Design and programming of a coprocessor for a risc-v architecture," Master's thesis, Collegio di Ingegneria Informatica, del Cinema e Meccatronica, 2017. [Online]. Available: https://webthesis.biblio.polito.it/6589/1/tesi.pdf

[9] Y. Lee, A. Ou, C. Schmidt, S. Karandikar, H. Mao, and K. Asanovi, "The hwacha microarchitecture manual, version 3.8.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263, Dec 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-263.html

[10] H. Mao, "Hardware acceleration for memory to memory copies," Master's thesis, EECS Department, University of California, Berkeley, Jan 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-2.html

[11] machinelearninguru, "Image convolution," 2017, tutorial. [Online]. Available: http://machinelearninguru.com/computer_vision/basics/convolution/image_convolution_1.html

[12] P. Warden, "Why gemm is at the heart of deep learning," April 2015, pete Warden's Blog. [Online]. Available: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[14] "Hummingbird e200 github page." [Online]. Available: https://github.com/SI-RISCV/e200_opensource

[15] H. Chen and Q. Chen, "Cnn-hwpe spec," 09 2018, specification of HWPE CNN coprocessor. [Online]. Available: https://github.com/chenhaoc/cnnhwpe/blob/master/spec/CNN-HWPE_SPEC_EN.pdf