UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Optimizing RISC-V core for machine learning workloads

Author:
Volodymyr KUCHYNSKYY

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Volodymyr KUCHYNSKYY, declare that this thesis titled, "Optimizing RISC-V core for machine learning workloads" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Optimizing RISC-V core for machine learning workloads

by Volodymyr KUCHYNSKYY

Abstract

Machine learning has become widely used in many different applications. Specifically, machine learning models on embedded edge systems have been gaining popularity. Due to the high resource requirements of machine learning workloads and highly-constrained embedded systems, the idea of using custom hardware accelerators has become viable. Open-source CPU architectures such as RISC-V could be used for such purposes. Additionally, Field-Programmable Gate Arrays (FPGAs) offer a useful platform for running and prototyping custom hardware. In this thesis, we review the current state of machine learning acceleration hardware, optimize a MobileNetV1 model and describe a design process for prototyping hardware acceleration using CFU playground framework.

Acknowledgements

I am grateful to Oleg Farenyuk for supporting me throughout the whole process of writing this thesis. I am also grateful to Yevhen Korotkyi for helping me choose the topic and answering my questions. Special gratitute goes to the Ukrainian Catholic University and the APPS.UCU faculty. Without them, I would never imagine achieving what I achieved during the last four years. Finally, I am grateful to Oles Dobosevych for giving me an opportunity to write this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

CFU	**C**ustom **F**unction **U**nit
CPP	**C**overage **P**ath **P**lanning
CPU	**C**entral **P**rocessing **U**nit
DNN	**D**eep **N**eural **N**etworks
FPGA	**F**ield-**P**rogrammable **G**ate **A**rrays
GPGPU	**G**eneral **P**urpose **G**raphics **P**rocessing **U**nits
GUI	**G**raphical **U**ser **I**nterface
HLS	**H**igh-**L**evel **S**ynthesis
HPC	**H**igh **P**erformance **C**omputing
MAC	**M**ultiply-**A**dd **A**ccumulate
ML	**M**achine **L**earning
NN	**N**eural **N**etwork
NPU	**N**eural **P**rocessing **U**nit
SIMD	**S**ingle **I**nstruction-**M**ultiple **D**ata
SIMT	**S**inge **I**nstruction, **M**ultiple **T**hreads
TPU	**T**ensor **P**rocessing **U**nit

Dedicated to my father, who taught me a lot and helped me get to this point

Chapter 1

Introduction

Today, it is hard to imagine a world without machine learning. It has become a part of many people's daily lives without them knowing. Due to its rapid success in the industry in recent years, this area of research is popular and highly desired among large corporations, giving rise to famous projects like TensorFlow, OpenAI, responsible for the creation of GPT-3 models, and others. A common pattern among them is their open-source nature: the frameworks are available to anyone free of change, and researchers are encouraged to contribute. Many models are often also publicly available. All this is useful both for industry and other researchers, who do not have to reinvent the wheel and can use reliable, well-documented tools to train widely-accepted and proven models. This makes the process of prototyping a new model much faster.

However, machine learning workloads place high resource requirements on the system they are running on. Consumer-grade hardware is usually not enough. Some custom hardware is often hard to acquire, produce, or simulate, since it is proprietary, making it accessible only to a limited group of researchers and engineers. Alternatively, some ML researchers may lack knowledge about hardware architecture or use such hardware acceleration methods that are well-documented and wrapped in abstraction layers of popular frameworks. This creates a problem – research is done for some platforms mainly because of their popularity and not because they are somehow superior. In general, the field of open-source hardware acceleration from machine learning is currently not yet as developed, but there are some promising trends.

Edge computing and machine learning (ML) workloads on edge are gaining popularity. However, considering the limited resources of such embedded systems, hardware acceleration becomes crucial. Another trend is the popularization of the RISC-V central processing unit (CPU). This open-source CPU architecture is the first to become viable not only in academia but as a competing platform with ARM in many areas, including embedded and high-performance computing (HPC). This is beneficial for ML accelerator research since building upon such architecture would make accelerators more unified and better understood.

1.1 Goals

In this thesis, we set the following goals:

- Review current state of ML acceleration on hardware, which platforms are used and what workloads are executed.

- Use a particular framework, CFU playground, where CFU means Custom Function Unit, to create a prototype for an accelerator used to speed up a neural network (NN) model.
- Describe in detail the iterative design process used to optimize the model, which can be transferred and applied to other models.

1.2 Structure

- Chapter 2 reviews the field of ML accelerators, describes the differences between Cloud and Edge applications and gives an overview of RISC-V CPU.
- Chapter 3 describes the CFU playground framework and other tools used by it to develop, deploy and simulate accelerators.
- Chapter 4 explains the setting and the methods with which the results will be obtained, as well as brief description of model used.
- Chapter 5 presents the results of model optimization, including plots and detailed step-by-step iterative optimization process.
- Chapter 6 concludes the thesis.

Chapter 2

Related work

2.1 Cloud and Edge AI

In general, it is important to define a distinction between cloud and edge computing in the context of ML workloads. Cloud computing presumes that all computation is done on a remote system. This is a type of centralized computation where end devices only specify what computation they need, send requests, and then wait for the result. Such an approach has been popular for some time since it minimizes the computations done on the end devices. These end devices are usually embedded systems with limited resources, while Machine learning (ML) workloads, especially in the areas of Deep neural networks (DNN) such as Convolutional neural networks, require a lot of computational resources (Soro, 2021).

However, edge computing has become one of the current trends in applications of ML to embedded devices (Shi and Dustdar, 2016). It provides several benefits. In particular, cloud computing requires communication with the cloud. This, in turn, requires an internet connection to be available from the local network to the cloud host. Transmitting large data, such as images, might lead to high network strain. However, not all embedded devices have the necessary hardware. Moreover, internet connection is not always available or might be unreliable. Also, connections might be compromised by a third party, or end-users might have concerns for their privacy due to data storage on a remote system. Therefore, minimizing or removing the role of a centralized entity has some advantages.

Edge computing can rely on a centralized entity responsible for computation in some IoT settings. However, this device is part of a local network. Nevertheless, this requires embedded devices, which have direct access to data, to offload computation to the edge device, which requires a network connection and high throughput of network links.

TinyML paradigm addresses those issues by eliminating a centralized entity and performing all inference computations on the embedded devices. However, ML workloads have high resource requirements, while embedded devices have tight limitations on memory usage, power consumption and computational power (Soro, 2021).

2.2 Hardware accelerators ecosystem

Given the different hardware capabilities between embedded systems where TinyML can be applied and large distributed datacenter systems where cloud computing

takes place, we review the state of hardware accelerators and determine how suitable they would be for TinyML.

A survey of DNN accelerator architectures (Chen et al., 2020) gives a detailed overview of different approaches. Most importantly, all accelerators should target computational patterns that are the most prevalent in a set of models. Mostly these are matrix multiplication and convolution.

The authors mention the Neural processing unit (NPU), which uses on-chip NN to run a portion of a program using it instead of running it on the Central processing unit (CPU). In hardware, NPU consists of 8 processing engines, where each processing engine does the calculation of a neuron output. In order to use the NPU, the original code is replaced by NPU invocations. The NPU itself is coupled with the processor pipeline (Esmaeilzadeh et al., 2012). This allows NPU to be used for general-purpose computing. Something similar would make sense for TinyML due to the low latency of on-chip accelerators.

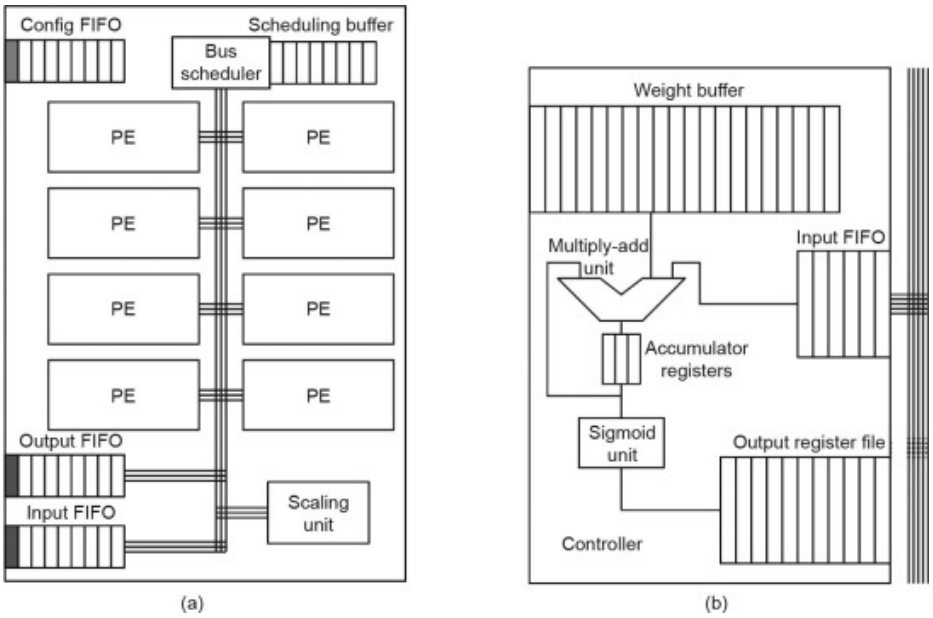

FIGURE 2.1: NPU Architecture (Esmaeilzadeh et al., 2012).

Another popular general-purpose computing approach is the usage of General-purpose graphics processing units (GPGPU). This is one of the earliest and still most widespread hardware accelerators for deep learning and DNNs (Mittal and Vaishay, 2019), both for training and inference. Their main advantage is parallelism: each GPU thread can work in parallel and perform identical and relatively simple operations. This pattern is known as Singe instruction, multiple threads (SIMT). This is useful because machine learning workflows have repetitive computation patterns that can be parallelized by broadcasting them to threads.

However, the authors also consider stand-alone domain-specific solutions. Compared with general-purpose solutions, such as GPUs, they offer higher productivity and better energy efficiency but require a deeper understanding of the target workload and are therefore harder to design.

The tensor processing unit (TPU) is such a type of accelerator mentioned in the survey. Tpu1 is built on the idea of a systolic array (Jouppi et al., 2017). Tpu1 is only

used for inference acceleration. Its successor, Tpu2, also uses a systolic array but adds a vector processing unit and works for training and inference.

Finally, Field-programmable gate arrays (FPGAs) can also be used to accelerate ML workloads. They can be used to synthesize the entire model on an FPGA. (Fahim et al., 2021) propose a tool called hls4ml, which translates a trained NN into a high-level synthesis (HLS) project that can be synthesized to FPGA flow.

2.3 RISC-V overview

RISC-V is an open-source project whose goal is to develop and provide a free basic instruction set architecture (ISA) based on a RISC load-store architecture. It was created at the University of California for educational purposes and research. The base ISA is very simple; it is similar to early RISC processors and has a minimal set of instructions. This is done by design so that most features will be expanded with either official standard ISA extensions or custom ISA extensions. There are 32-, 64- or 128-bit versions of this base ISA (Waterman and Asanovic, 2019).

2.4 ISA extensions

Since RISC-V has a very limited base ISA, most implementations are expected to use extensions. Standard ISA extensions are developed together with the base ISA, they must work with all base ISA variants and all other standard extensions. Extension usually is a set of extra instructions intended to add shared features. For example, the M extension adds instructions necessary for integer multiplication and division. Custom ISA extensions are similar in purpose, but are not accepted as standard extensions. Anyone can create their own custom extension. They are also not guaranteed to work properly with other extensions (Waterman and Asanovic, 2019). A few useful extensions for ML workflows:

- M extension – Standard Extension for Integer Multiplication and Division
- F, D and Q – Standard Extensions for single, double and quad precision floating-point operations
- C – Standard Extension for compressed instructions
- V – Standard Extension for vector operations, only recently ratified

2.5 CFU – Custom Function Unit

When using a RISC-V custom core for a domain-specific workload, especially on an embedded system with limited resources, it is possible to add hardware acceleration for specific functions by using a domain-specific RISC-V custom extension. Such extensions are application-specific and, therefore, most likely won't be accepted as standard extensions. However, unlike standard extensions, it is hard to reuse or combine custom extensions for different cores because they are not designed to be portable or compatible. The "RISC-V Composable Custom Extensions Specification" (Ansell, Callahan, and Gray, 2022), which is still in draft form, tries to resolve this by introducing a Custom function unit (CFU).

CFU is a small piece of hardware meant to accelerate a set of custom functions defined by the designer. In other words, it implements the custom interface that consists of a set of custom functions. Therefore, all custom extensions should be implemented as functions for the CFU. To invoke the CFU, custom instructions that follow the RISC-V R-format are added to the CPU. This format states that instruction has two 32-bit input operands from the register file and writes back one 32-bit result back to the register file. CFU can support internal state and multiple instructions but doesn't have access to the main memory or other registers from the register file.

A two-way handshake implements the CPU-CFU communication: first, the CPU issues a CFU request to compute a custom function sets ID (a 10-bit field that can be used as metadata inside the CFU function) and both operands. Then, CFU accepts a request, does computation and sends a CFU response, which updates the destination register. Formally, this is implemented by using two pairs of signals: *cmd_valid* and *cmd_ready* for the CPU, *rsp_valid* and *rsp_ready* ready for the CFU. The *valid* signal initiates communication. It is sent to indicate that a request or result is ready, and the *ready* signal is sent back as an acknowledgment to the initiator that the receiving side is ready. After this, the initiator sets request or result data. On (Figure 2.2) a

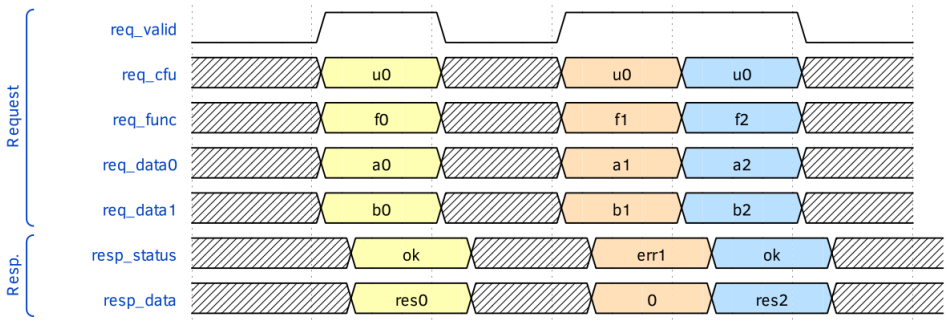

FIGURE 2.2: CFU timing diagram (Ansell, Callahan, and Gray, 2022)

timing diagram is shown. Here, *req_data* signals are the operands, *resp_data* is the result. Note that *req_func* specifies three different functions.

Chapter 3

Tools

3.1 CFU-playground

CFU playground is a framework that helps leverage the advantages of a full stack of open-source and configurable tools, both software and hardware, to quickly prototype and design new lightweight accelerators (Prakash et al., 2022). On the one hand, this allows the user to create a task-specific accelerator instead of a general-purpose one. This is beneficial for embedded systems with limited resources. But, at the same time, unlike models defined in hardware or on FPGAs, the system uses a von-Neumann architecture with a CPU and can be easily repurposed for a different model or different accelerator settings due to the use of CFU.

This framework provides a set of tools and scripts from them to create a SoC system with a modified RISC-V CPU and a CFU. This system is then either simulated or uploaded and run on an FPGA board. Along with the system, a pre-trained ML model for inference as well as software to run it is provided. Due to their open-source nature, many tools can be changed or replaced. There is also functionality for profiling and testing different CFU modifications in order to identify hotspots. In the framework project on GitHub, there is a template for custom CFU projects. It is also possible to add or modify a machine learning model. However, the project structure is currently not optimized for that.

The core idea of this framework is the usage of CFU to improve performance. ML models have significant optimization hotspots in their workloads. These are repetitive computations in their kernels, such as multiply-add accumulate in convolution or matrix-vector multiplication in NN fully connected layer computation. CFU makes it possible to significantly improve performance by hardware accelerating only these hotspots, while the rest of the inference code, including loops and setup code, will be executed as regular instructions by the CPU. Since the system is targeted toward FPGAs, this is especially beneficial because such computational workloads can be implemented in hardware primarily by combinatorial circuits, and they map well to FPGAs. For more complex CFUs, multiple computations per cycle are possible, such as with SIMD instructions, are possible due to the parallel nature of FPGA hardware. Moreover, since the CPU is also synthesized on the FPGA, the CFU can be easily integrated with the CPU, and the latency for communications between them will be low.

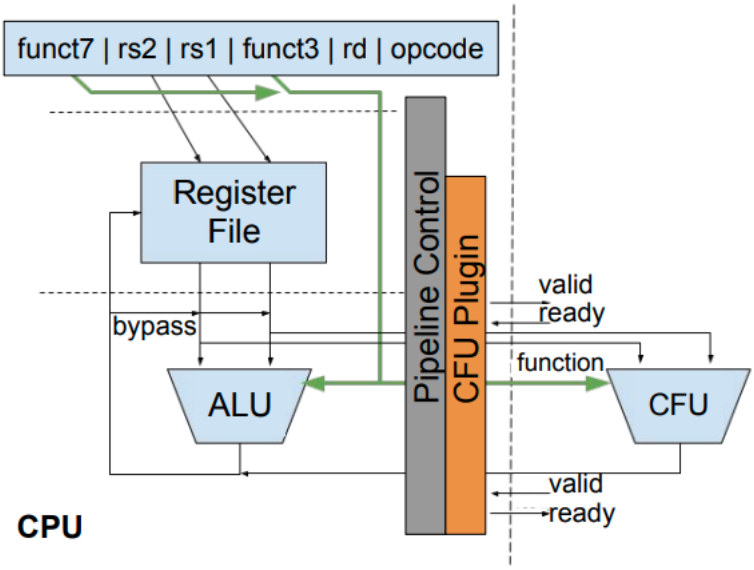

FIGURE 3.1: CPU to CFU iterface (Prakash et al., 2022).

3.2 CPU and SoC

3.2.1 Vexriscv core

The default RISC-V CPU core used in the CFU playground is Vexriscv (Papon, 2017), which is implemented in SpinalHDL and optimized for soft core CPUs. By default, it implements the RV32IM ISA and a 5 stage in-order pipeline (fetch, decode, execute, memory, write back). However, the main advantage of Vexriscv is its configurability. It is designed in such a way that nearly every CPU feature is a plugin. There is a set of standard plugins that come with this core. They can be added to a configuration file that specifies how the CPU should be built. Therefore, it is easy to add new instructions, pipeline stages, caches, etc. It is also possible to create and add custom plugins.

3.2.2 SoC creation

LiteX in a Python-based framework used to create FPGA soft cores and SoCs (Kermarrec et al., 2020). It supports different CPU architectures, including RISC-V. The SoC itself consists of the soft core CPU and other system components, such as buses, RAM drivers and other custom logic. It provides a library of common IP components that can be used in such SoCs, and custom components can also be added. Therefore, a CFU is also implemented as a custom component in the SoC. In order to generate FPGA bitstream, other tools, such as SymbiFlow, are used. Due to hardware differences, LiteX contains descriptions of most common FPGA boards.

3.2.3 Amaranth

Amaranth is a hardware description language (HDL) based on Python and implemented as a library (Kermarrec et al., 2020). Python was chosen since it is a well-known and easy-to-use language on a semantic level, providing useful abstraction.

In LiteX, all components are designed in Amaranth, including the custom CFU. On top of that, the CFU playground provides a python module that contains CFU abstractions: classes that implement the CPU to CFU communication or custom instruction addition and can be inherited by custom CFU instances. This approach simplifies the development workflow, avoids unnecessary code duplication and decreases the risk of an implementation bug.

3.2.4 Simulation

Renode is a development framework for simulating and debugging IoT and embedded systems. It can be used to assemble and simulate a SoC with CPU and peripherals. LiteX has scripts that convert SoC configuration into Renode scripts. Renode supports the Vexriscv CPU as well as its peripherals. CPU is simulated on the ISA level, while CFU simulation is cycle-accurate via Verilator. CPU performance in such a setup will differ from a hardware setup, but this works well if the goal is CFU prototyping and benchmarking in isolation.

3.3 Inference framework

TensorFlow Lite Micro (TFLM) is an inference framework for embedded systems (David et al., 2020). It is cross-platform and hardware-agnostic, with a possibility of platform-specific hardware optimizations from vendors. A subset (130 out of 1400) of all operations available in regular TensorFlow is implemented in TFML. Therefore, a pre-trained model must be provided in a special `.tflite` format, which can be generated from the standard TensorFlow model by a special exporter, which is available in the framework.

When running on an embedded device, the model, as well as the TFLM framework, must be uploaded onto it. At the core of the framework lies the interpreter, which loads the model data structure and parses and interprets it. In turn, the model contains information, like which operation to execute and where to get parameters. Other notable modules of TLFM include custom functions used for operations and an operator resolver that maps model-defined operations to these functions. When optimizing with CFU playground, usually, the implementations of these custom functions are changed to invoke new CFU commands.

Chapter 4

Development setup

4.1 General evaluation setup

Our intention is to optimize a TinyML model by designing a ML accelerator as a CFU, benchmarking model performance and iteratively improving it. In the process of doing so, we describe in detail how we arrive at certain solutions, their shortcomings and what are the alternatives. This iterative design process can be reused for different models or classes of models. All tests are conducted in Renode simulation. The general approach that we will take in this study is described below

1. Identify the most time-consuming operations.
2. Find the hotspot inside TFLM implementation of such operations.
3. Implement a basic widely-used optimization method in CFU. We start with SIMD.
4. Compare performance with unoptimized model.
5. Recognize potential limitations or further optimization opportunities and propose an improved CFU design.
6. Repeat until satisfied with the results.

4.2 MLPerf Tiny benchmarks

MLPerf Tiny is the first benchmarking suite and collection of pre-trained models for TinyML. It identifies 4 TinyML use-cases: keyword spotting, visual wake words, image classification and anomaly detection, with appropriate dataset and model architectures (Banbury et al., 2021). These applications are often used in real-world scenarios, like person recognition on security cameras, detection of audio wake words like “Hey, Google”, etc. The models are pre-trained and stored in *.tflite* format. Importantly, CFU playground framework also uses these models and already has built-in integration for these models, including the model itself, input values, tests, an option in the makefile to enable them and code in TFLM to load and run them.

We will focus our attention on the task of Visual wake words (VWW). The task is to detect the existence of an object of interest on an image. This is especially useful in always-on video feeds, like security cameras, where recording and streaming all footage is not desirable. Instead, triggering the system and starting streaming when an object of interest is present would be preferable. (Chowdhery et al., 2019) present a dataset for VWW, which is derived from the COCO dataset with binary labels on

whether a person is present on it. A MobileNetV1 CNN model is trained on this dataset, with 8-bit quantized values.

Chapter 5

Results

5.1 Visual wake words model

First, we begin by profiling the performance of the model in order to identify the hotspot. After running the model and collecting data about ticks per each individual calculation and its type from UART output, we aggregate them by operation type. see the results shown in Table 5.1.

Operation	Share of cycles
CONV_2D	74.1%
DEPTHWISE_CONV_2D	25.8%
AVERAGE_POOL	0.03%
RESHAPE	< 0.01%
FULLY_CONNECTED	< 0.01%
SOFTMAX	< 0.01%

TABLE 5.1: Distribution of cycles per operation

Here, the `CONV_2D` operation corresponding to 2-dimensional convolution is clearly the most time-consuming operation, taking up over 75% of all computational time. The second operation, `DEPTHWISE_CONV_2D`, is also a type of convolution computation. Together these two instructions take up almost all computational time.

We will focus on optimizing the `CONV_2D` instruction. In order to do that, we find the hotspot inside its TFLM implementation. In this case, it's the nested loop with multiplication and addition. Multiplication is necessary for the convolution formula, while addition is necessary due to quantization works.

The first optimization that we attempt is vectorization. There are only two simple operations here, and both are scalar and performed on 8-bit quantized values. Then, this product is aggregated into a 32-bit accumulator value. However, custom CFU instructions always have 32-bit input operands and 32-bit output. We can use this to pack four 8-bit values together in each operand, performing a Single instruction-multiple data (SIMD) addition and multiplication operations.

This requires some changes to the source code. Since operations are performed on four elements simultaneously, the loop will have to reflect that. Iteration is done by channel, and the number of channels may not always be divisible by 4. Therefore, as a special case, the first $n_{channel} \mod 4$ iterations we are done in a scalar fashion, and the rest will be vectorized.

```
1  const int scalar_iters = filter_input_depth % 4
2  const int vector_iters = filter_input_depth / 4;
3  //Scalar loop
4  int i_residual;
5  for (i_residual = 0; i_residual < scalar_iters; ++i_residual)
6  {
7      int32_t input_val = input_data[Offset(i_residual, /*...*/)];
8      int32_t filter_val = filter_data[Offset(i_residual, /*...*/)];
9      //accumulation
10     acc += filter_val * (input_val + input_offset);
11 }
12
13 for (int i_vector = 0; i_vector < vector_iters; i_vector += 1) {
14     uint32_t input_val = *(uint32_t *)(input_data +
15                         Offset(/*...*/, (4*i_vector + scalar_iters));
16     uint32_t filter_val = *(uint32_t *)(filter_data +
17                         Offset(/*...*/, (4*i_vector + scalar_iters));
18
19     //Warning: integer overflow may occur
20     uint32_t input_added = simd_add_op(CFU_OP_ANY_DEFAULT, //CFU_OP0
21                                        input_val,
22                                        TO_VECTOR(input_offset));
23
24     uint32_t result_low = simd_mul_op(CFU_OP_MUL_LOWER, //CFU_OP1
25                                       input_added,
26                                       filter_val);
27     uint32_t result_high = simd_mul_op(CFU_OP_MUL_UPPER, //CFU_OP1
28                                       input_added,
29                                       filter_val);
30
31     //accumulate all results
32     acc += simd_accum_op(CFU_OP_ANY_DEFAULT, res_low, res_high);
33 }
```

LISTING 1: Convolution pseudocode with SIMD add and multiply operations.

We added three new instructions, which are wrapped into C macros for convenience. Addition is used on line 20 in Listing 1. In order to add a scalar value to a vector, we duplicate the same 8-bit value four times in a 32-bit register and pass it as input. The result is a 32-bit vector of values as well. Multiplication is similar, but output needs extra explanation: given two 8-bit values, $x, y \in [-127, 127]$; $x, y \in \mathbb{Z}$, then their product, $x \cdot y \in [-16129, 16129]$, is a 16-bit value. But we are limited to a 32-bit output and therefore return the results in two portions: depending on the value of *funct7*, which is the first argument here used for purposes of control in the CFU instruction, we output either the lower 32-bits or the upper 32-bits. The computation here is done twice, which is redundant but necessary. Otherwise, we might only need the lower part if we intend to multiply only two values. Finally, we add another instruction that adds all values in both lower and upper parts and returns a scalar.

While such SIMD approach is conceptually simple, it has a few noticeable problems.

First, from it's usage here we see that we are quite constrained by the instruction encoding. We can only take two 32-bit operands and return a 32-bit value; we also can't access the CPU register file to use extra registers or access RAM. In turn, this creates problems with returning complex structures, as we saw with multiplications. Moreover, addition also has an unaddressed problem here, namely integer overflow. If the sum of two 8-bit values has to carry one, this bit is truncated, and it's impossible to return this extra carry bit because then the other sums wouldn't fit into the 32-bit return value. This means that the result of addition might be incorrect. It is possible to solve this by adding a 4-bit carry flag state register to the CFU since it allows to store persistent internal state. Then, both addition and multiplication would have access to it. Such change wouldn't affect performance because 1-bit register access can be done during the same cycle. Nevertheless, all these problems suggest that a different approach should be taken, with the idea of SIMD in mind.

Benchmarking our model, optimized using the SIMD approach, we found that compared to the default implementation, the model is now about **1.30** times faster. See column "SIMD Add/Mul" on Fig. 5.1 for comparison with other optimizations.

Our next approach is based on the previous, by acknowledging that its main problem was excessive data-passing: this created design problems with returning data and created unnecessary extra work for the program. Now let us consider replacing the entire accumulation step with one instruction. In such a case, it would have to take three inputs and perform addition and multiplication. Moreover, we can get the same aggregated value in the CFU as well, which would remove another addition in the loop. We will call such instruction multiply-add accumulate (MAC). It is still generic and can be used in other settings beyond convolution. However, the implementation of this instruction raises another question: how should the third input value be handled? In this case, we determine that input offset will be that third value because it is loop invariant. In general, two opposite approaches can be chosen:

- **General purpose approach**. Use *funct7* field with a unique value. In such case, the CFU would expect to receive the value of offset in one of the operands and save it internally. All following MAC instructions would use that value until further changes,
- **Model-specific approach**. Set a constant value inside the CFU, which will be used in all further instructions. This only works for models with identical `input_offset` values.

The second approach has advantages in simplicity and, possibly, performance, while the first is more versatile.

We also have to consider the changes to the TLFM convolution code. The loop conditions, step and data size will depend on whether scalar or SIMD MAC implementation is used. But the loop itself will only have one instruction related to the accumulation step. In this case, SIMD MAC operation is performed on input and filter values, with a constant offset being added inside. The accumulator is also stored in the CFU; its value is returned on each iteration. After the last iteration, the variable will be set to the result of accumulation. As we can see, compared to SIMD multiply and add, this implementation has less data-passing; only one 32-bit scalar value is returned. It is also important to add an option to reset the accumulator at the beginning of each convolution, see line 1 in Listing 2. In case of offset not being a constant in the CFU, we have to set it manually using another instruction on line 2 in listing 2, which has to be uncommented in a general-purpose approach to MAC design.

```
int32_t acc = simd_mac_op(CFU_OP_MAC_RESET, 0, 0);
//simd_mac_op(CFU_OP_ADD_SET, input_offset, 0);
/*...*/
for (int i = 0; i < iters; ++i)
{
    /*...*/
    acc = simd_mac_op(CFU_OP_MAC_ACC, input_val, filter_val};
}
```

LISTING 2: Convolution pseudocode with MAC operation.

After benchmarking both instructions on the visual wake words model, we see that scalar MAC yields a **1.15** speedup, while SIMD MAC yields a **1.55** speedup. In both cases, the offset was constant. See columns "Scalar MAC" and "SIMD MAC" on Fig. 5.1 for comparison with other optimizations.

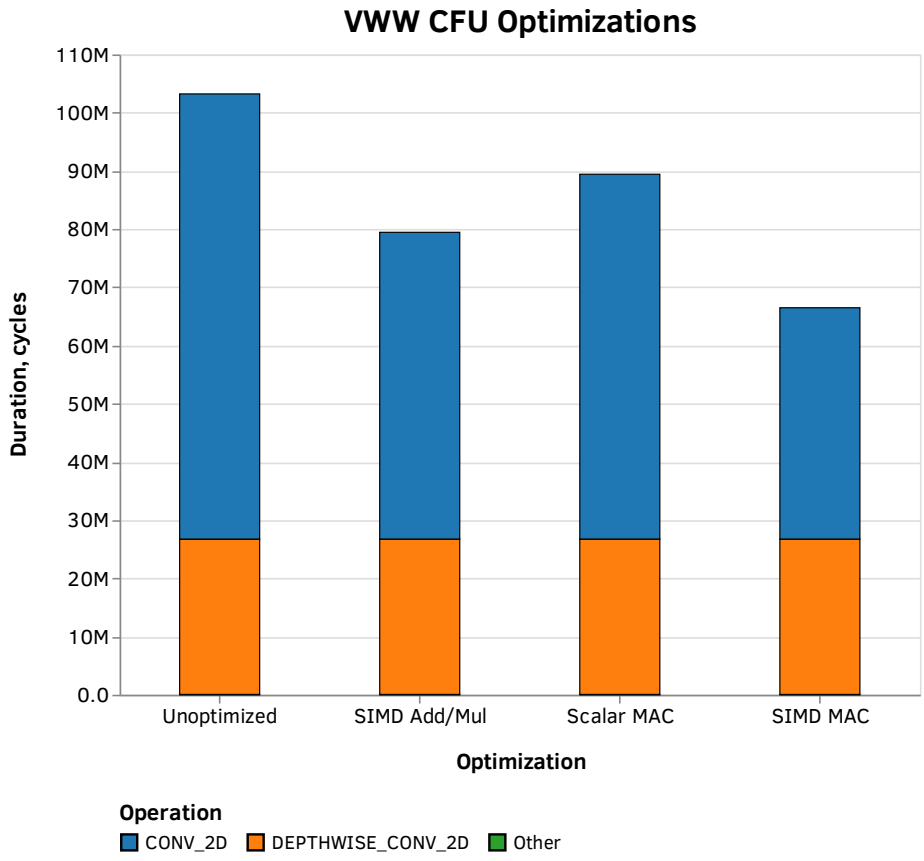

FIGURE 5.1: Amount of total clock cycles per CFU optimization

Chapter 6

Conclusions

6.1 Conclusions

In this work, we gave an overview of opportunities in hardware design for machine learning. In particular, we focused on extreme edge devices and TinyML models running on embedded systems with open-source hardware and software stacks. We used the CFU playground framework to incrementally optimize performance by prototyping multiple composable function units, which we benchmarked on a simulated RISC-V core using the TFLM framework. During the design phase, we also mention that a choice between flexibility and specialization has to be considered when designing an accelerator. This split is important for us in the context of CFU optimizations because a good CFU design for a specific workload consists of a list of such choices.

The general-purpose approach requires architectural improvement to the accelerator – by vectorizing, reducing data-passing, and increasing throughput, we can arrive at a complex ML accelerator with an optimized dataflow architecture. This idea is reflected in the Neural Processing Unit or other similar accelerators. Such an approach can be applied when run on more resource-friendly hardware with multi-model workflows.

Instead, the model-specific approach tries to improve performance by removing unnecessary abstractions or functionality in a resource-constrained workflow, specialized for a limited set of models or one specific model. This can be achieved by embedding constants into the CFU, using specific data representations to save space or optimizing larger, more specific operations. This idea is reflected in models where weights are hardcoded into the accelerator or calculating a particular convolution layer entirely on the CFU.

The code for CFU projects, as well as scripts and logs are located in a GitHub repository: https://github.com/Centurion256/ml-cfu-optimization

6.2 Future work

- Testing on a larger sample of models with different operators, in particular on fully connected MLP, where neuron computation can also be represented as MAC instruction.
- Illustrating the flexibility or specificity of a particular CFU optimization on multiple models with in differnet domains would better explain the choice between general-purpose and model-specific approaches.

- Experimenting with changes to the RISC-V core configuration and their influence on the performance of a CFU.
- Bringing the CFU playground to a physical system: either an FPGA board or ASIC. Benchmark performance, as well as memory usage and power efficiency. This is the ultimate goal of any experimentation done with CFU playground

Bibliography

Ansell, Tim, Tim Callahan, and Jan Gray (2022). *Draft Proposed RISC-V Composable Custom Extensions Specification*. URL: https://github.com/grayresearch/CFU/blob/main/spec/spec.pdf.

Banbury, Colby R. et al. (2021). "MLPerf Tiny Benchmark". In: *CoRR* abs/2106.07597. arXiv: 2106.07597. URL: https://arxiv.org/abs/2106.07597.

Chen, Yiran et al. (2020). "A Survey of Accelerator Architectures for Deep Neural Networks". In: *Engineering* 6.3, pp. 264–274. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2020.01.007. URL: https://www.sciencedirect.com/science/article/pii/S2095809919306356.

Chowdhery, Aakanksha et al. (2019). "Visual Wake Words Dataset". In: *CoRR* abs / 1906.05721. arXiv: 1906.05721. URL: http://arxiv.org/abs/1906.05721.

David, Robert et al. (2020). "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems". In: *CoRR* abs/2010.08678. arXiv: 2010.08678. URL: https://arxiv.org/abs/2010.08678.

Esmaeilzadeh, Hadi et al. (2012). "Neural Acceleration for General-Purpose Approximate Programs". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

Fahim, Farah et al. (2021). "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices". In: *CoRR* abs/2103.05579. arXiv: 2103.05579. URL: https://arxiv.org/abs/2103.05579.

Jouppi, Norman P et al. (2017). "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12.

Kermarrec, Florent et al. (2020). "LiteX: an open-source SoC builder and library based on Migen Python DSL". In: *CoRR* abs/2005.02506. arXiv: 2005.02506. URL: https://arxiv.org/abs/2005.02506.

Mittal, Sparsh and Shraiysh Vaishay (2019). "A survey of techniques for optimizing deep learning on GPUs". In: *Journal of Systems Architecture* 99, p. 101635.

Papon, Charles (2017). *VexRisv*. https://github.com/SpinalHDL/VexRiscv.

Prakash, Shvetank et al. (2022). "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (tinyML) Acceleration on FPGAs". In: *CoRR* abs/2201.01863. arXiv: 2201.01863. URL: https://arxiv.org/abs/2201.01863.

Shi, Weisong and Schahram Dustdar (2016). "The Promise of Edge Computing". In: *Computer* 49.5, pp. 78–81. DOI: 10.1109/MC.2016.145.

Soro, Stanislava (2021). "TinyML for Ubiquitous Edge AI". In: *CoRR* abs/2102.01255. arXiv: 2102.01255. URL: https://arxiv.org/abs/2102.01255.

Waterman, Andrew and Krste Asanovic (2019). *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.