CFU Playground: Build your own ML Processor using Open Source

Tim Callahan, Tim Ansell
Google
Sunnyvale, California
{tcal,tansell}@google.com

Joseph Bushagour†
Purdue University
West Lafayette, Indiana
†Past intern at Google

Alan V. Green, David Lattimore, Dan Callaghan
Google
Sydney, Australia
{avg,dml,dcallagh}@google.com

***Abstract*—The CFU (Custom Function Unit) Playground allows you to design and build machine learning (ML) accelerators extending an FPGA-based RISC-V core, running on an FPGA board at your desk. Because the CPU is soft, it can be both tailored (e.g. cache sizes modified) and extended (new instructions added through the use of a Custom Function Unit). Push-button builds of the customized processor combined with a provided interactive software test/measurement harness allows for very rapid edit-compile-profile cycles (on the order of minutes) at each stage of accelerator development.**

A primary goal of the CFU Playground is to provide a delightful and fun experience to the developer, by, as far as possible, removing the burden of maintaining infrastructure such as configuring toolchains, writing test harnesses, building performance measurement jigs, or editing Makefiles. This rapid, lightweight framework lets the user realize a large performance benefit from a relatively small investment in creating customized hardware.

CFU Playground bundles together open source software (TensorFlow, GCC), open-source RTL generation IP and toolkits (LiteX, VexRiscv, Migen, nMigen), and open-source FPGA tools for synthesis, place, and route (yosys, nextpnr, vpr, etc.). By using open source for the entire stack, we give the user access to customize and co-optimize hardware and software, resulting in a specialized solution unencumbered by licensing restrictions and not tied to a particular FPGA or board.

CFU Playground is available under an Apache 2.0 license at https://github.com/google/CFU-Playground. Online documentation is available at https://cfu-playground.readthedocs.io.

***Index Terms*—FPGA, Machine Learning, Open Source EDA, Open Source Hardware**

I. Introduction

The genesis of the CFU Playground was a Google internal project that requires low power, high performance ML inferencing. For various reasons, the project's principals chose a small FPGA and Tensorflow Lite for Microcontrollers (TFLM) [1] as the processing platform. This then left the problem of how to measure and then improve the performance of the selected models on a soft RISC-V core on the FPGA.

In this work, we consider the ML model to be *fixed*, but the ML processor and the ML libraries to be *subject to optimization*. In short, we adapt the hardware and software to the specific model. Note that this is reversed from many research projects in the area of Neural Architecture Search [2], which strive to optimize the ML network given a fixed cost model of the target hardware (CPU, GPU, or TPU).

II. Customizing the Soft CPU

Because the starting point is a fully functional RISC-V CPU built using the FPGA fabric, TFLM and application software will run immediately, although it will likely not meet the design requirements of latency and power. We must exploit the fact that the CPU is not fixed, that instead we can tailor it and extend it, to achieve the desired performance.

If you assume that the CPU is provided as Verilog source code, then the prospect of editing the Verilog to attempt to improve performance would not be an attractive approach; it requires expertise, is time-consuming, and would require extensive verification to ensure that your modifications did not introduce errors.

Instead, we use the open source, highly configurable, generator-based VexRiscv [3] soft CPU, which allows us to generate exactly the CPU best suited to our needs. The number of pipeline stages, the branch prediction strategy, the cache sizes, and many other parameters are all configurable.

A. Custom Instructions

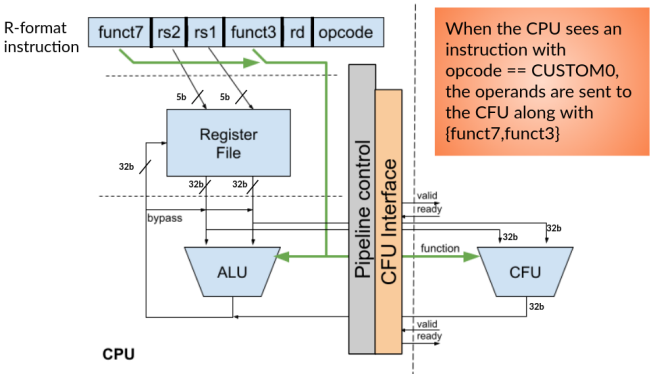

Fig. 1. CPU augmented with CFU (Custom Function Unit).

We then allow extensions to the RISC-V ISA by adding new instructions in the reserved CUSTOM0 opcode space. VexRiscv is configured to generate a fixed CFU interface, to which the user's CFU is attached (Figure 1). Any instruction with its opcode field specifying CUSTOM0 is sent to the CFU interface, along with two operands from the register file and

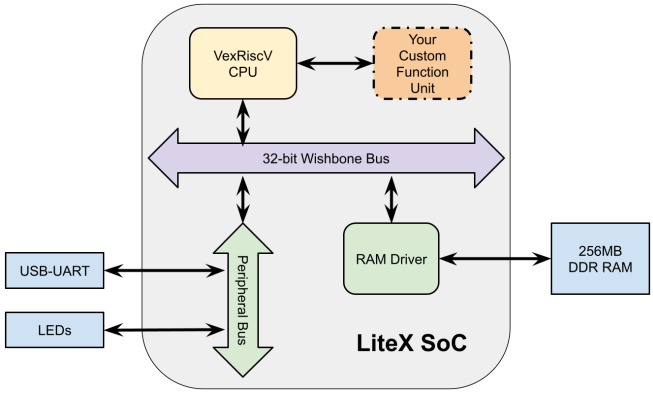

Fig. 2. Full SoC provided by LiteX. The user implements only the CFU. Digilent Arty A7-35T board illustrated.

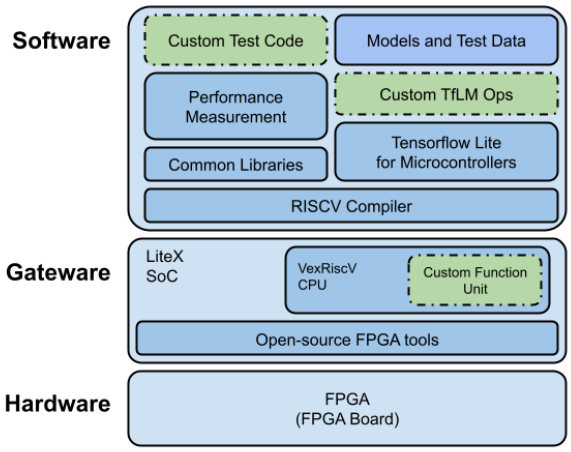

Fig. 3. CFU Playground Overview.

10 additional bits (funct7 + funct3) from the instruction to specify *which* custom instruction is to be executed.

The CFU can contain state. We assume an embedded computing context where we don't worry about context switches. The CFU does not have its own path to system memory. Any data into and out of the CFU must go through the CPU.

The CPU-CFU interface provides a very tight coupling between the CPU and the added custom functionality. During FPGA synthesis/place/route, the "interface" disappears, and the CFU essentially becomes part of the CPU pipeline.

The new instructions can be used from C or C++ using an inline assembly macro that we provide. No modification of the RISC-V GCC toolchain is needed. When a new custom instruction is added by modifying your CFU, it is instantly available.

III. The Rest of the Pieces

The CFU Playground includes everything needed to provide a useful environment for developing your tailored processor (Figure 3):

- A LiteX [4] SoC configuration (Figure 2). LiteX provides everything needed to turn a CPU into a complete functioning computer: it provides a system bus, on-chip RAM, and peripherals to connect to the outside world, such as UART over the USB connection.
- A framework for building new CFU implementations, using either nMigen [5] or plain Verilog.
- An implementation flow using either vendor tools (e.g. Vivado) or open source tools (see Figure 4).
- Multiple common tinyML models, including keyword spotting and person detection.
- An interactive menu-driven C program to run unit tests, perform TFLM inferencing, and measure performance.
- A software build system that allows a developer to override arbitrary files of C code, including TFLM code.
- Scripts to automate testing on the FPGA board.

IV. Development Cycle

To understand how the CFU Playground assists the developer, let's examine an example acceleration of the `pdti8` model (person detection 96x96 grayscale, int8 quantized).

A. Set up

The developer begins by cloning the git repository and following the provided instructions to install dependencies. They then copy the project template directory to make a new project. It is recommended that the developer commit the new project directory before any making any changes. Once the new project directory is established, the developer builds both the FPGA gateware and the C software with a simple `make` command. At this point, there is no CFU.

B. Set a baseline

The developer then profiles `pdti8` to understand baseline performance. TFLM's built-in profiling shows the name and running time of each TFLM operation as it executes.

`pdti8` uses 5 types of ops, but 70% of time is spent performing `CONV_2D`, 30% in `DEPTHWISE_CONV`, and negligible time elsewhere.

C. Simplify and Specialize

Given that it takes the bulk of execution time, CONV_2D seems the best place to start. The developer:

1) Adds `printfs` to print out parameters and identify constant parameters
2) Specializes C implementations to take advantage of those constants and remove unused code
3) Uses cycle counters to profile parts of the C code for the operation, identifying hot spots
4) Based on profiling adjusts C code to cache frequently used variables or unroll loops.

On CONV_2D, we are able to get 2x acceleration fairly quickly this way (software specialization and optimization).

FPGA	Open Source Tools	Example Supported Board(s)
Lattice ice40up5k	yosys, nextpnr-ice40, icepack	iCEBreaker, Fomu
Lattice ECP5	yosys, nextpnr-ecp5, ecppack	OrangeCrab, ULX3S
Lattice Nexus	yosys, nextpnr-nexus, prjoxide	Crosslink NX Eval Board
Xilinx XC7 series	yosys, vpr, fasm2bits (SymbiFlow)	Arty A7-35T

Fig. 4. Open source tools for supported FPGAs.

We also use this as an opportunity to deeply understand data flow through the kernel.

D. Identify opportunities for CFU acceleration

For an example, the `CONV_2D` kernel has the following computation in its innermost loop:

```
acc += filter_val *
            (input_val + input_offset);
```

This doesn't appear at first to be a candidate for a custom instruction because it requires 4 operands when considering `acc`'s previous value as an input as well. However, `input_offset` is loop-invariant, so it can be moved into the CFU before the loop nest is entered. `acc` as well can be stored in the CFU. We will then have four custom instructions: one to copy `input_offset` into the CFU; one to reset `acc` to zero; the main instruction with `filter_val` and `input_val` as operands, using the stored value of `input_offset` to compute the value added into `acc`; and finally an instruction to read the value of `acc` out of the CFU.

From here, making a 4x SIMD version of the CFU is a small step, giving another significant speedup.

A key point is that we are tailoring the CPU + CFU for just a single ML model. Thus, we don't need to build a general CFU that can accelerate *all* TensorFlow operators, or even all convolutions. We only need it to accelerate the operators and parameterizations that occur in our model.

V. Supported FPGAs and toolchains

Figure 4 shows supported FPGA families, which open source tools are used for each family, and a non-exhaustive list of example FPGA boards.

In some cases the user has a choice of using vendor or open source tools. Because CFU Playground uses the LiteX build system, whichever toolchain is the default for a particular board with LiteX is also the default for CFU Playground. For example, when targeting a board with the ice40up5k FPGA, the open source tools are default. But for Xilinx XC7-based boards, the vendor tool (Vivado) is default; open source tools can be chosen by specifying a configuration option.

CFU Playground *should* work with Intel FPGAs and in fact any vendor's FPGAs, as long as the FPGA and board are supported in LiteX. We simply haven't had a chance to test with any FPGAs except those listed in Figure 4. Also, other FPGAs would require the use of vendor tools.

VI. Benefits of Being Open Source

All IP and software used in CFU Playground is open sourced and licensed permissively – from the open RISC-V ISA that allows new instructions in the custom opcode space, to the VexRiscv soft core implementation and the LiteX system-on-chip IP, to the open source Symbiflow FPGA synthesis/place/route tools (vendor tools can be used instead if the user wishes), Renode and Verilator simulation environments, and TensorFlow Lite kernel libraries. This means that the CPU plus CFU plus kernel libraries that the user develops are not tied to any particular FPGA vendor; there are no licensing restrictions or fees; and there is no dependence on any black box proprietary tools. If project requirements change, the design can be easily moved to a different FPGA from a different vendor. Similarly in the case of a sourcing issue, the design can be moved to any similar available FPGA.

Another benefit of the fully open source stack all the way down to CPU RTL is the transparency that it provides. Since all parts can be inspected, you are not in the position of needing to place blind trust in either software or silicon providers. This is a core principle of the Betrusted system [6].

VII. Examples

A. MobileNet V2 Acceleration on Arty

Profiling of MobileNetV2 (MNV2) on the Arty board showed that 63% of the cycles were consumed by CONV_2D 1x1 layers, so our focus was accelerating that particular operator.

In the computation for a 1x1 convolution, for each x,y spatial coordinate, an input vector (the input tensor's column at x,y with length determined by the number of input channels) is multiplied by a matrix to produce an output vector (the output tensor's column at x,y with length equal to the number of output channels). The matrix contains the filter weights, has size input channels $\times$ output channels, and is the same for every x,y.

We started with a simple 4-way parallel multiply-accumulate CFU. From there, the CFU grew incrementally over many steps. The final incarnation of the CFU (Figure 5) stored the filter weights and current input vector internally, and had an internal sequencer to compute a complete output column from an input column. This allowed each value transferred into the CFU storage to be used multiple times. Also, the input and output vectors were double buffered: while one output vector was being computed, the previously-computed

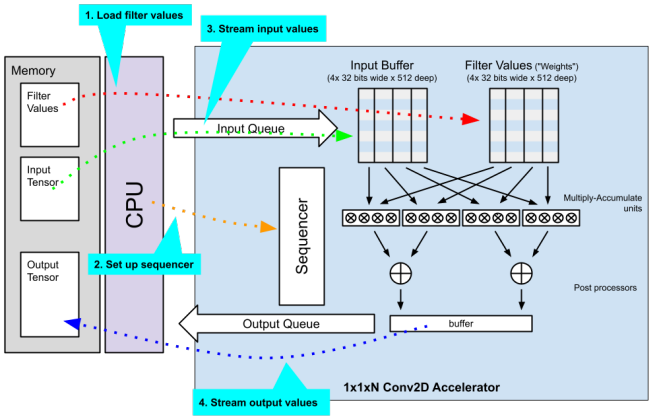

Fig. 5. Final CFU for MobileNet V2.

output vector was being copied out, and the next input vector was being copied in.

This MNV2-specialized CFU achieved a 55x speedup for CONV_2D over the software implementation, although it required fairly complex hardware design. We implemented this CFU using nMigen [5], an open source Python-based toolkit for RTL design, and this helped us to manage that complexity.

B. Keyword Spotting Acceleration on Fomu

The Fomu [7] FPGA board is roughly the size of a penny, and fits inside a USB slot. It combines an ICE40UP5k FPGA (with 5280 logic cells and 128kB of on-chip RAM) with a 2MB flash memory. At first look it seemed it would be impossible to fit CFU Playground on such limited resources.

With the limited FPGA resources, even the basic VexRiscv configuration did not fit. So we started with the "minimal" configuration, which has no caches, no hardware multiplier, and no branch prediction or bypassing. This fit, but was very slow.

Thus our first step was optimizing the VexRiscv configuration, taking out what wasn't needed and adding what gave the most benefit. After many iterations, we removed some error checking (e.g. for misaligned addresses) and some unneeded system reset functionality. This made room for a single-cycle hard multiplier (but not a hard divider) and small caches, with 500 logic cells left over for a CFU.

On the software side, the CFU Playground binary could not fit in 128kB, especially considering that much of this RAM is needed for working data. We modified the linker script to place the code (.text section) and read-only data (.rodata section – mostly weights from the ML models) into flash memory. However, reading code and data from flash is much slower than from RAM. Our solution was to move only the model weights and select subroutines into fast RAM memory.

While profiling the memory system, we realized that flash ROM accesses were slower than they should be. This pointed to some potential improvements in the SPI flash interface that we were able to implement.

Finally, we considered how to achieve further speed by adding custom instructions. The run time was dominated by convolution, and we found we had created sufficient room to add a 4-way parallel multiply-accumulate. This CFU was written in a subset of SystemVerilog that Yosys understands.

Depthwise convolution was the second runtime contributor. It has a different access pattern and therefore cannot use the 4-way multiply-accumulate that we built for convolution. Ideally, we could build separate CFU gateware for depthwise convolution, but there were no remaining resources to extend the CFU this way. After some examination we realized we could use just one lane of the 4-way multiply-accumulate CFU in depthwise convolution to achieve a modest, but still positive, speedup.

Overall, keyword spotting inference performance improved by a factor of 75x. The time for one inference was reduced from 2.5 minutes to 2 seconds. Only 3x of the speedup was directly attributable to the small CFU that was added. The other 25x was derived from optimizing the CPU configuration, software, memory accesses, and system interfaces/drivers. This illustrates the usefulness of CFU Playground's profile-analyze-improve loop for full-system optimization.

VIII. Updates and Status

- Done: We have completed the process of open-sourcing the repository.
- Done: We have generalized CFU Playground to support multiple FPGA families and boards, in particular, smaller lower-cost options.
- In progress: We are shifting to using Conda packaging to download all needed open source tools, rather than requiring the user to install them.
- Future: Make it easy to fine-tune the CPU configuration for each individual project. Currently the user can only choose from among a small number of pre-generated VexRiscv configurations.

References

[1] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems," 2020. [Online]. Available: https://arxiv.org/abs/2010.08678

[2] Wikipedia Authors. (2021) Neural architecture search. [Online]. Available: https://en.wikipedia.org/wiki/Neural_architecture_search

[3] C. Papon. (2021) VexRiscv: An FPGA friendly 32 bit RISC-V CPU implementation. [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[4] F. Kermarrec and Others. (2021) Litex wiki. [Online]. Available: https://github.com/enjoy-digital/litex/wiki

[5] whitequark. (2021) A refreshed Python toolbox for building complex digital hardware. [Online]. Available: https://github.com/nmigen/nmigen

[6] A. 'bunnie' Huang and Betrusted developers. (2021) Betrusted: A security enclave for humans. [Online]. Available: https://betrusted.io

[7] (2019) I'm Fomu, an FPGA in your USB port! [Online]. Available: https://fomu.im