CFU Playground: Want a faster ML processor? Do it yourself!

Shvetank Prakash* Tim Callahan† Joseph Bushagour§ Colby Banbury*
Alan V. Green† Pete Warden† Tim Ansell† Vijay Janapa Reddi*
†*Google* §*Purdue University* **Harvard University*

***Abstract*—The rise of machine learning (ML) has necessitated the development of innovative processing engines. However, development of specialized hardware accelerators can incur enormous one-time engineering expenses that should be avoided in low-cost embedded ML systems. In addition, embedded systems have tight resource constraints that prevent them from affording the "full-blown" machine learning (ML) accelerators seen in many cloud environments. In embedded situations, a custom function unit (CFU) that is more lightweight is preferable. We offer CFU Playground, an open-source toolchain for accelerating embedded machine learning (ML) on FPGAs through the use of CFUs.**

I. Introduction

The Internet of Things (IoT) has transformed our lives by embedding computing into everyday objects at the edge. Machine learning (ML) is set to revolutionize the next generation of IoT devices as recent advancements have made it possible to deploy these powerful ML algorithms on commercial embedded systems [1]. Yet in many cases the computational burden of ML is still too large to run efficiently on commodity hardware, and specialized hardware is needed in order to meet an application's performance requirements.

However, building custom ASICs incurs large non-recurring engineering (NRE) costs that are generally not suitable for low-cost embedded ML systems that support a wide range of use cases. Moreover, the diversity in ML models and fast-changing nature of the field has made the reconfigurable nature of FPGA hardware an attractive alternative for ML acceleration and deployment [2]. The resource constraints of embedded systems also make "full-blown," discrete ML accelerators seen in cloud settings impractical for IoT devices.

II. CFU Playground

CFU Playground is an open-source toolchain to accelerate embedded ML on FPGAs using custom function units. "CFU" in CFU Playground stands for Custom Function Unit [3]: lightweight accelerator hardware that is tightly coupled into the pipeline of a CPU core, to add new custom function instructions that complement the CPU's standard functions (such as arithmetic or logic operations). The need for a CFU emerges in settings where (1) faster processing is desirable under (2) tight resource constraints. ML acceleration on microcontroller-class hardware [4] is a new area that combines both of these demands. CFU Playground is a collection of software, gateware, and hardware configured to make it easy to: make improvements in software by modifying source code, design custom function units, run embedded ML models, and benchmark and profile performance. An important aspect of CFU Playground is that it enables rapid iteration on processor improvements— multiple iterations per day!

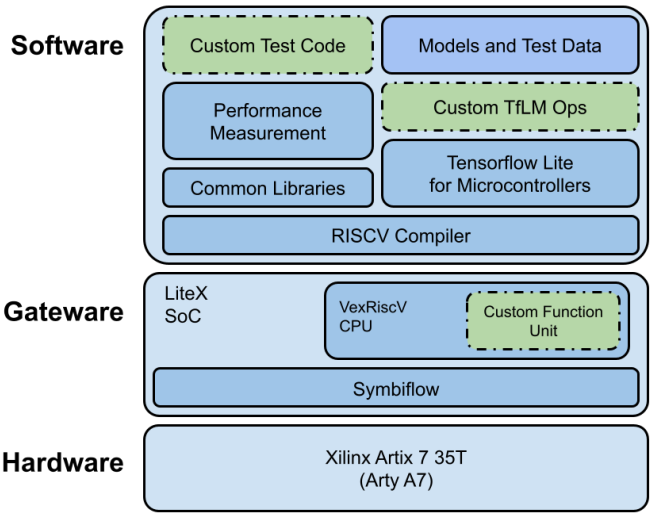

Fig. 1: CFU Playground is an open-source collection of software, gateware, and hardware that makes it easy to design custom function units and accelerate embedded machine learning on FPGAs via hardware and software.

The complete full-stack of CFU Playground presented in Figure 1 builds upon fully open sourced projects. VexRiscv [5] is the RISC-V soft CPU used that is configured with a CFU plugin. The CPU and CFU are placed inside a complete System On-Chip (SoC) built using the Litex framework [6]. In order to synthesize and generate the bitstream for the complete SoC, we use Symbiflow/F4PGA [7]. CFU Playground supports a variety of FPGAs boards such as Arty A7-35T/100T, iCE-Breaker, Fomu, OrangeCrab, and more. Simulation support is provided using Renode [8] and Verilator [9]. Finally, Tensorflow Lite for Microcontrollers [10] is the main software used for running ML on the soft CPU + CFU.

CFU Playground comes ready out-of-the-box, so you can start designing your own ML processor in minutes. Source code is available at www.github.com/google/CFU-Playground.

978-3-9819263-7-8/DATE23/© 2023 EDAA

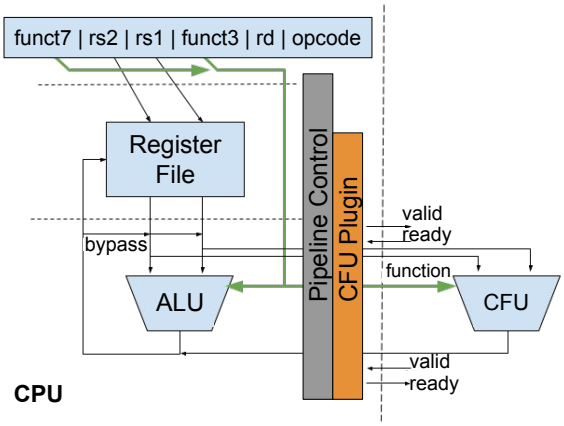

Fig. 2: Custom Function Unit (CFU) Architecture in CFU Playground. CFU instructions follow the RISC-V R-format pictured above. The register operands, function id, and result are exchanged via the interface. Handshaking between CPU and CFU is performed using valid and ready signals.

III. CFU Use Cases

The CFU architecture is flexible and allows for it to be used in many different ways. In this section, we outline a few of the common cases. Figure 2 depicts the the CFU interface which is referred to in the examples.

Combinational CFU: In the simplest CFU, there are no registers on the path from the operands (`rs1` and `rs2`) to the result (`rd`). The result is computed based on the desired function specified by `funct3` and the operands.

CFU with Latency: When the CFU computation critical path is longer than one clock cycle, the computation can be broken into stages. After one or more clock cycles the result will be ready, at which point the CFU asserts its `valid` signal. The latency from start to finish can be fixed or variable. The CPU will stall as needed until the result is produced and `valid` is asserted by the CFU. Of course, this means that a bug in your CFU may hang the system.

Split Phase: If the CFU operation runs for a long time, it might be useful to let the CPU carry on doing other work. You can do this with one instruction providing operands to start the operation (ignoring the immediately-returned result) and another instruction for retrieving the results later. If the CFU is not ready when the second instruction executes, the CPU stalls until it finishes. The first instruction can be thought of as a "fork" and the second instruction as a "join" of control flow.

Additional Cases: Some operations naturally have more than two operands or more than one result. In this case, multiple CFU instructions can be used to move operands and fetch results. For complex CFUs, it might make sense to have addressable storage or configuration registers. You can also have an instruction for polling the CFU to check if it is still working, and if it is not, have the CPU continue working on other stuff. Lastly, another common case is when you want to shove an indefinite amount of data into the CFU, and the CFU keeps a running accumulation or reduction (e.g., dot product). This is implemented as three cooperating instructions: one to initialize the state (i.e., zero out the accumulator), one to send the next set of data, and a final one to retrieve the result.

IV. Build Your Own ML Accelerator

In this section we briefly describe the process of designing your own ML accelerator in CFU Playground.

After cloning the CFU Playground repository and following the instructions to set up the environment, the user must make a project and select an ML model to accelerate. For the purposes of this example, we used MobileNet [11].

The first step after selecting a model is profiling portions of the source code to identify what to accelerate. From this, we found that about 75% of the time is spent inside Tensorflow Lite Micro's `CONV_2D` operation for this model.

We begin looking at software optimizations now. Performing some simple, model-specific optimizations for constant parameters and loop unrolling decreased the total number of cycles spent in inference by 36%.

Now we direct our attention to hardware. In the innermost loop of the `CONV_2D` operation we multiply and accumulate 8 bit quantified integers sequentially. This is wasteful since our registers are 32 bits wide. Thus, with a bespoke CFU we created an instruction that performs a SIMD multiply-and-accumulate operation in one or two cycles.

With just these simple software improvements and a tiny CFU, we decreased the total number of cycles taken by the innermost loop from 113 million down to just 22 million! Users can continue this iterative development process to make many more improvements.

References

[1] L. Andrade, A. Prost-Boucle, and F. Pétrot, "Overview of the state of the art in embedded machine learning," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1033–1038.

[2] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *ieee Access*, vol. 7, pp. 7823–7859, 2018.

[3] J. Gray, "Composable custom function unit specification." [Online]. Available: https://cfu.readthedocs.io/en/latest/

[4] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors Journal*, vol. 22, no. 22, pp. 21 362–21 390, 2022.

[5] C. Papon, "Vexriscv." [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[6] F. K. *et al.*, "Litex: an open-source soc builder and library based on migen python DSL," *CoRR*, 2020.

[7] K. E. Murray *et al.*, "Symbiflow and VPR: an open-source design flow for commercial and novel fpgas," *IEEE Micro*, vol. 40, no. 4, pp. 49–57, 2020.

[8] Antmicro, "Renode," 2018. [Online]. Available: https://renode.io/

[9] Veripool, "Verilator." [Online]. Available: https://veripool.org/verilator/

[10] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[11] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.