Flexible Low-Precision Machine Learning on Reconfigurable Hardware

Christopher De Sa and Adrian Sampson, Cornell University
Gates Hall, Cornell University, Ithaca NY 14853-7501 `cdesa@cs.cornell.edu` `asampson@cs.cornell.edu`

As the efficiency returns from Moore's law wane, reconfigurable hardware accelerators are emerging as a specialized, highly efficient alternative to general-purpose CPU and GPU computing. Field-programmable gate arrays (FPGAs) offer particularly compelling gains for machine learning workloads because of their customizable data movement fabrics and customizable numerical precision [1]. Current approaches to exploiting FPGAs for deep learning fall into two categories: (1) fixed-function accelerators for generic primitives like matrix multiplication and convolutions, and (2) DNN-to-Verilog compilers that generate completely custom accelerators for each model. Neither extreme is ideal. The former approach suffers from a "one-size-fit-all" design approach that cannot customize the underlying fabric for evolving model demands. The latter requires re-engineering and hours- or days-long RTL synthesis for each new model or training strategy. A practical solution would claim the middle ground between fixed-function design and total customization.

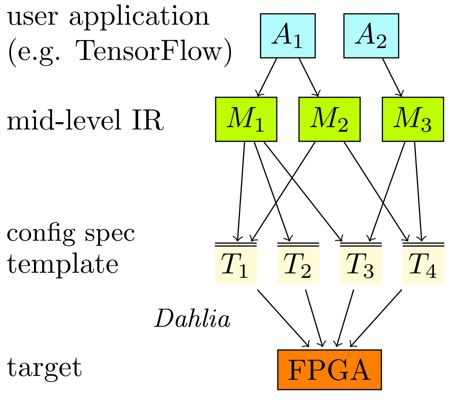

Figure 1: Accelerator *templates* interpose between high-level machine learning frameworks and low-level FPGA implementations.

We propose an end-to-end approach to reconfigurable acceleration of ML training that balances specialization with flexibility. The key idea is to introduce a new intermediate level of abstraction between ML frameworks and the FPGA fabric: a range of semi-specialized overlay *templates* that can support a constrained set of numerical-precision-aware training implementations. A template implements a set of functional units, memories, and communication channels that can be assembled to implement a specific training workload. We will define a complete, formal semantics for a new domain-specific language (DSL) for specifying templates and their configurable parameters. Importantly, templates have already undergone the expensive place-and-route phase of FPGA design, which allows templates to be flexible: they can be configured for a given training run without reprogramming the underlying FPGA.

Figure 1 shows how templates integrate into the end-to-end machine learning stack. The advantages of this new template abstraction level include:

- *An interface between ML engineers and hardware engineers.* Templates provide a common specification that (a) hardware engineers can use to implement new hardware-level capabilities and (b) machine learning engineers can rely on to map new models and training algorithms onto hardware. On the high-level side, a template will both (a) effectively define a template-specific "language" that can be used to configure that template and (b) describe the function, including estimated performance, of the configured template—i.e. the operational semantics of that language. On the low-level side, a template will include a design for an FPGA that can be combined with a configuration to produce a full working design.
- *Quick reconfiguration without FPGA synthesis.* Templates evolve slowly—they can support a range of models and training workloads without incurring the extremely long synthesis latencies required to generate complete FPGA configurations. This will allow application developers to target FPGAs without substantial overhead.
- *Easy addition of new templates.* The system described in Figure 1 will use a mid-level IR that is automatically mapped onto whatever available template would achieve the best performance. This will allow the system to easily scale to add new templates as needed or as hardware capabilities change.
- *Preparation for post-FPGA hardware.* While the current proposal concretely targets FPGA-based acceleration, CGRAs and other more specialized architectures [2] offer advantages over traditional FPGAs. Templates provide an opportunity to abstract over the specific hardware substrate: the system that maps the mid-level IR to the template configuration space does not need to know the details of the hardware that lies beyond the template configuration spec.

The rest of this proposal describes our approach to implementing each level in this end-to-end flow.

Specifying accelerator templates. A template is an abstract description of a hardware accelerator architecture. It describes the design's execution units, memories, interconnection networks, off-chip memory interfaces, and configurable parameters. A given template provides a set of parallel execution contexts equipped with SIMD arrays of functional units along with primitives for moving data according to the communication topology. The template specifies the available ranges of numerical precision settings available in each execution unit and memory module. To implement a specific training workload, an ML framework generates code to map to each execution context. This code is responsible for managing the on- and off-chip movement of data.

As part of this work, we will design a DSL for specifying templates. Programs in the DSL describe the space of valid programs that can run on the accelerator template and their performance characteristics—for example, the latencies of memory banks and the bandwidth capacity of each link in the interconnection network. A training workload implementation consists of a *configuration spec* that targets a specific template. We will implement a functional simulator that takes an accelerator template in our DSL and a corresponding configuration spec and predicts the output and resource consumption of a hardware implementation. The simulator will let software engineers experiment with template programming even when hardware implementations are not yet available.

Mapping low-precision training to templates. Training and deploying ML models in low precision confers many systems advantages, as the resulting computations use fewer bits and so require less time, power, and memory. Configuring a system to use low-precision arithmetic properly is particularly important on FPGAs, since the fine-grained reconfigurability of FPGAs allows them to support many different precision levels natively—and previous work has produced substantial speedups by using low-precision arithmetic in fixed-function DNN inference accelerators on FPGAs [3]. To exploit low-precision hardware primitives, we need infrastructure to map training workloads onto the accelerator templates described above. We plan to extend an existing machine learning framework, PyTorch [4], to automatically generate high-performance code to implement training for a given model. While standard 32-bit floating-point representations make training staightforward, smaller bit widths or fixed-point representations can significantly increase hardware efficiency. But due to the quantization error, novel training techniques are required.

PI De Sa brings expertise in low-precision arithmetic for ML models [1, 5, 6]. He has also developed a framework, QPyTorch [7], for simulating low-precision training in PyTorch, which will provide the basis for the numerical-precision-aware extensions we propose to develop. We propose to realize these techniques in an extension to PyTorch that will automatically transform an off-the-shelf model implementation to use low-precision arithmetic. Using this tool, developers will be able to design and train models that faithfully simulate execution on hardware with low-precision arithmetic, even without access to that hardware. These tools will ease the transition to execution on real hardware.

Implementing templates on FPGAs. A central component of our infrastructure will be a toolchain for implementing and verifying templates on FPGAs. We need a way to instantiate generic computational elements and compose them into pipelines that implement a given template. PI Sampson brings ongoing work on a new programming language that enables agile development of highly parameterized accelerator designs [8]. This current toolchain, called Dahlia, combines some advantages of low-level RTL design with high-level synthesis (HLS) tools without inheriting the frustrating unpredictability pitfalls of the latter. Dahlia also makes it possible to specify reusable, abstract *families* of accelerator elements and to compose them quickly into complete designs. The end result is a programming model that combines high-level algorithmic specification with low-level control over hardware resources.

Building on the Dahlia programming language and compiler toolchain, we will implement a set of libraries and tools to provide a complete design framework for implementing accelerator templates on FPGAs. The libraries will make it easy to assemble and test new accelerator architectures on real FPGA hardware. We also plan to provide a set of validation tools that compare these Dahlia-based FPGA implementations against our software simulator to help find and fix implementation bugs.

References.

[1] C. De Sa, et al. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ISCA*. 2017.

[2] R. Prabhakar, et al. Plasticine: A reconfigurable architecture for parallel paterns. In *ISCA*. 2017.

[3] J. Fowers, et al. A configurable cloud-scale DNN processor for real-time AI. In *ISCA*. 2018.

[4] A. Paszke, et al. PyTorch. `https://github.com/pytorch/pytorch`, 2017.

[5] R. Zhao, et al. Improving neural network quantization without retraining using outlier channel splitting. In *ICML*. 2019.

[6] G. Yang, et al. SWALP: Stochastic weight averaging in low precision training. In *ICML*. 2019.

[7] T. Zhang, et al. Low-precision arithmetic simulation in PyTorch. `https://pypi.org/project/qtorch/`. Accessed: 2019-06-11.

[8] R. Nigam, et al. A typed language for safe high-level synthesis. `https://github.com/cucapra/dahlia`. Accessed: 2019-06-11.