(Please Build) an Accelerator Zoo

Katie Lim
katielim@cs.washington.edu
University of Washington

Matthew Giordano
mgiordan@cs.washington.edu
University of Washington

Jonathan Balkind
jbalkind@ucsb.edu
UC Santa Barbara

ABSTRACT

Building accelerator tooling and frameworks runs into a chicken and egg problem: it is hard to build accelerators without this infrastructure, but it is difficult to evaluate the usefulness of infrastructure without concrete, end-to-end examples using an application accelerator. This paper makes the argument for an "accelerator zoo" to break this cycle, lays out guidelines for accelerators that would be good for the zoo, and poses some open questions for the implementation of this framework.

1 INTRODUCTION

Alongside the proliferation of accelerators, work into better frameworks and infrastructure to support accelerators and ease deployment has also grown [6–8, 11]. When one builds a shiny new infrastructure prototype, the next step is convincing people it is good. This means carefully selecting accelerator examples to showcase both the quantitative performance of the framework as well as unmeasurable qualitative features such as generality. This is a difficult space to navigate with many trade-offs. For example, there is typically a compromise between the complexity of the accelerator and the cost of integrating it into your framework. If the accelerator for the end-to-end evaluation is perceived as too simple, it is considered a toy example and there are doubts about its ability to support realistic applications. However, if the accelerated application is complex, and the accelerator does not perform well, one then needs to performance debug the application as well as the framework lest the results can become "distracting" with the focus turning to the accelerator performance rather than the framework.

This problem is not unique to hardware, but is exacerbated by the nascent nature of the field. In the software world, new systems software infrastructure projects will typically integrate with a well-known open-source project such as Redis [2], RocksDB [4], or memcached [1]. These projects are complex, reasonably optimized, and readily available. What should we do in the world of hardware where open example accelerators are few and far between? How can we avoid our case studies being labelled as "toys" without putting in enough work to warrant a second paper?

We advocate for building an accelerator zoo, a collection of portable, high-quality accelerators ready to be used within accelerator infrastructure. This is akin to the model zoos in machine learning which are often provided to help jumpstart projects. In the rest of this paper, we first discuss the current workflow and pain points that we have observed when trying to construct end-to-end experiments for various projects. We then describe guidelines around "zoo-friendly" accelerators that prioritize reusability. Finally, we propose some open questions around realizing this zoo.

LATTE '24, April 28, 2024, San Diego, CA, USA

2 CURRENT STATE OF AFFAIRS

Imagine setting out to include an accelerator for evaluation. The first step is finding an accelerator. So where does one look right now? How can one find a new accelerator in the wild?

One way to start is to search sites like GitHub or OpenCores. This turns up a variety of different examples that vary vastly in code quality and complexity, ranging from a college student's class project to industrial-strength projects like Ariane/CVA6 [12] or NVDLA [10]. Finding one that might be workable involves wading through search results, trying to assess both code quality and how to use the accelerator. This typically involves reading the documentation, if there is any, and definitely reading code. This vetting has high overhead in terms of time and once an accelerator has been selected it still needs to be integrated.

Integration is a highly ad-hoc, one-off process specific to every accelerator that involves both figuring out how to connect it at the hardware level as well as what it expects when executing a computation. Sometimes an accelerator may contain enough documentation to understand how to connect signals to its hardware interface. But the vast majority of the time, one ends up reading code. If one is lucky, there is a good quality testbench. If not, it is necessary to dive into the HDL of the application logic itself. During the integration, one can find out that the accelerator does not actually perform as expected or even run correctly. The options then are either to try to debug the code or circle back to step one and search again.

This process is clearly time-consuming. A significant amount of work already goes into building the research prototype for the idea itself and having to unearth suitable accelerators makes the process of evaluation onerous. Ultimately this hinders progress of the development of tools and frameworks that would make building accelerators easier.

3 ACCELERATOR GUIDELINES

Our hope is that an accelerator zoo can curate a variety of accelerators ranging from traditional hardware computational atoms, such as FFTs, to complex implementations of traditionally software applications such as a key-value store. The goal of these guidelines is to set out how to make an accelerator friendly to portability to maximize the number of scenarios it can be used in. This may occasionally requiring prioritizing portability over performance or area. This may be at odds with the goals of certain accelerator implementations. In the same way that certain animals are not zoo-friendly because they require specialized conditions, certain accelerators may not be zoo-friendly.

While certain pre-existing frameworks, such as OpenCL, can provide solutions to some of these problems, it can be burdensome or simply not make sense to support these frameworks, so building too closely to these frameworks can inadvertently make portability more challenging.

3.1 Data Interface

The data interface to the accelerator is the most basic place to start. If inputs cannot be fed to the accelerator, there will be no computation.

For composability and generality, latency insensitive interfaces are a must. Supporting latency sensitive accelerators requires the whole framework to be engineered specifically to support a particular implementation, which restricts generality. There are a number of protocols that provide this kind of interface, such as AXI-stream, AXI-lite, Avalon, ect.. We do not intend to propose use of a certain protocol, because it is generally possible to transduce between protocols. Furthermore, even adhering to one protocol does not guarantee compatibility between interfaces. For example, AXI-stream's `tkeep` signal can permit non-contiguous valid data on a bus [3]. This means that two designs both with an AXI-stream interface can be incompatible with each other depending on how they treat the `tkeep` bits. Thus, accelerators should strive to keep their latency-insensitive interfaces as generic as possible.

Accelerators generally consume data either as an input stream or from data at rest stored in a memory of some sort, such as a cache or scratchpad. While support for scratchpads should generally be straightforward, caches typically requires more thought since the expectation is that caches are coherent, which places more requirements on the overall system. If an accelerator is designed to expect a cache-coherent system, it should be designed such that the cache can be separated from the accelerator, and the accelerator should not rely on specific timing from the cache.

3.2 I/O Usage

While certain ideas can be evaluated purely through simulation, others need to be prototyped on FPGAs or even taped out. In this case, we have to consider the I/O used by the accelerator. FPGAs have vastly different I/O capabilities ranging from speed of I/O, such as 1 Gbit Ethernet vs 100 Gbit Ethernet, to types of I/O, such as the presence or absence of DRAM or the use of PCIe (or AXI) to connect to a hosting CPU or directly to host memory.

Accelerators should be agnostic to the I/O whenever possible. For example, DRAM buffers should be replaceable with large BRAMs. While this may vastly change performance characteristics or supported input size, it enables designs to be used in a variety of different scenarios where certain I/O may not be available. It can also ease experimentation with new I/O types, such as replacing DRAM with some form of non-volatile storage or changing from DRAM accessible via DMA over PCIe to DRAM accessible using CXL.mem.

3.3 Configuration and Management

Alongside data operations, accelerators often have control interfaces that allow setting certain runtime parameters that modify their function, such as input dimensions for image processing or the key for encryption. There are a myriad of methods to provide this interface.

The method depends on how the accelerator is integrated into the larger system. A tightly-coupled accelerator might use configuration registers written by the core's pipeline. Meanwhile, an accelerator connected over AXI/NoC/PCIe might use memory-mapped I/O where the configuration registers are assigned addresses. These mechanisms come with a variety of constraints and result in the need for cores to understand the (non-)cacheability or side effects of the requisite MMIO accesses. This information must be thoroughly documented to make the accelerators suitable for zookeeping. Some accelerators may also expect interrupts. However, if interrupts are supported, polling should also be supported, because not every framework will have support for interrupts.

3.4 Supporting Software

Hosting an accelerator for operation can be complex. Many accelerators operate in tandem with a driver to prepare data for them [5, 9] or perform other operations, such as interacting with a host operating system. This software should be designed keeping in mind that the communication layer may change. For example, for a CPU-accelerator system on a single node, the hardware interconnect itself may be different. Zynq systems use an AXI bus, but there are many PCIe FPGA cards, and CXL.cache devices are becoming available. Particularly pertinent to a research oriented setting, the interconnect could be none of these and could instead be a new proposed interconnect under development.

Through standard software engineering principles of abstraction, what the software is doing for the accelerator should be separated from how the communication happens to allow for isolation of communication-medium specific code. This will hopefully reduce the amount of effort necessary to port an accelerator's supporting software to a new architecture.

4 OPEN QUESTIONS

Should there be a preferred HDL? While it would be preferable to use a newer HDL for readability and productivity, we want to avoid running into a situation where accelerators are suddenly unusable due to a new version of the language, require large software stacks to be generated, or depend very heavily on a certain version of an HDL. The only real requirement is that the HDL should be able to generate either Verilog or VHDL, so it can be fed to standard hardware tool chains. Ideally this code should also be human readable.

What framework can be or needs to be provided by the zoo itself? The models in model zoos are often trained to be used within a certain ML framework (e.g. TensorFlow or Pytorch). Our use case is a little different in that we are trying to build the zoo to allow for experimentation across frameworks, both existing and those that have yet to be developed. Should the zoo try to provide some software driver and FPGA or ASIC infrastructure that could be used across all accelerators?

Are there current accelerators meeting this requirement? This is more of a solicitation than a question, but it would be ideal to leverage preexisting accelerators as much as possible and either augment or make minimal modifications to make them more suitable. We have a small set, but they are typically smaller examples which some reviewers have considered "toy" examples.

How do we incentivize this work? Zookeeping the collected accelerators will require both curation and upkeep. How do we incentivise people to want to take up this responsibility?

REFERENCES

[1] [n. d.]. *Memcached.* Retrieved March 8, 2024 from https://github.com/memcached/memcached

[2] [n. d.]. *Redis.* Retrieved March 8, 2024 from https://github.com/redis/redis

[3] ARM. [n. d.]. *AMBA AXI-Stream Protocol Specification.* Retrieved March 8, 2024 from https://developer.arm.com/documentation/ihi0051/latest/

[4] Facebook. [n. d.]. *RocksDB.* Retrieved March 8, 2024 from https://github.com/facebook/rocksdb

[5] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[6] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[7] Joshua Landgraf, Matthew Giordano, Esther Yoon, and Christopher J. Rossbach. 2023. Reconfigurable Virtual Memory for FPGA-Driven I/O. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3582016.3582048

[8] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Xiaohe Cheng, Yanqiang Liu, Abel Mulugeta Eneyew, Zhengwei Qi, and Baris Kasikci. 2020. A Hypervisor for Shared-Memory FPGA Platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 827–844. https://doi.org/10.1145/3373376.3378482

[9] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: An Open Hardware-Software Stack for Deep Learning. *CoRR* abs/1807.04188 (2018). arXiv:1807.04188 http://arxiv.org/abs/1807.04188

[10] NVIDIA. [n. d.]. NVIDIA Deep Learning Accelerator. http://nvdla.org/.

[11] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. 2023. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/3582016.3582059

[12] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114 https://github.com/openhwgroup/cva6.