A Hardware–Software Blueprint for Flexible Deep Learning Specialization

Thierry Moreau
University of Washington

Tianqi Chen
University of Washington

Luis Vega
University of Washington

Jared Roesch
University of Washington

Eddie Yan
University of Washington

Lianmin Zheng
Shanghai Jiao Tong University

Josh Fromm
University of Washington

Ziheng Jiang
University of Washington

Luis Ceze
University of Washington

Carlos Guestrin
University of Washington

Arvind Krishnamurthy
University of Washington

***Abstract*—This article describes the Versatile Tensor Accelerator (VTA), a programmable DL architecture designed to be extensible in the face of evolving workloads. VTA achieves "flexible specialization" via a parameterizable architecture, two-level Instruction Set Architecture (ISA), and a Just in Time (JIT) compiler.**

Hardware specialization is a powerful way to accelerate a known set of applications and workloads. Unfortunately, deep learning (DL) is anything but a static field, i.e., the machine learning (ML) community constantly changes the software they use to write models, the architecture of models themselves, the operators used by said models, and the data types they operate over.

Researchers have primarily focused on two approaches for accelerator designs, fixed function accelerators and programmable accelerators (also known as domain-specialized accelerators).

Digital Object Identifier 10.1109/MM.2019.2928962
Date of publication 16 July 2019; date of current version 10 September 2019.

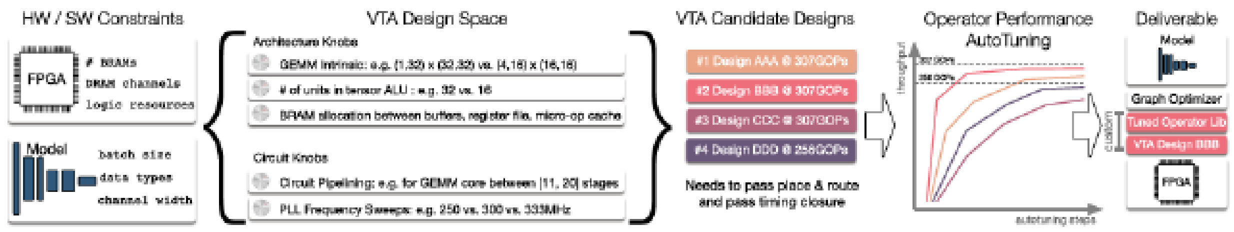

Figure 1. VTA provides flexibility with respect to hardware targets and DL models. This flow diagram shows the steps in adapting a given model to a hardware backend by exploring VTA hardware configurations and performing operator autotuning on the top hardware candidates. This process generates the binaries and hardware overlays necessary to deploy VTA in any DL framework.

Current solutions offer compelling peak performance, but they often fail to integrate into the evolving ML landscape.

Fixed-model accelerators are commonly spatially and statically laid out, offering attractive performance for certain kinds of workloads. Unfortunately, their static nature rules out the reuse of hardware resources, limiting support for larger or newer models.

In contrast, programmable accelerators[6] offer far more flexibility by leveraging Instruction Set Architecture (ISAs). Due to their programmable nature, achieving peak performance requires a competent DL compiler that can map many different workloads onto a fixed set of hardware intrinsics. Consequently, customizing the behavior of these accelerators, even when open-sourced, greatly depends on the availability of a transparent and modular software stack.

A central challenge of prior work is how to link innovations in specialization to rapidly changing ML applications. This challenge is not specific to computer architecture; it is present at all levels of the stack. An end-to-end approach requires integration of frameworks, systems, compilers, and architecture in order to execute state-of-the-art ML using hardware acceleration. Peak floating point operations per second (FLOPs) provide value only if a programmer can access them.

We present versatile tensor accelerator (VTA), an explicitly programmed accelerator paired with a capable Just in Time (JIT) compiler and runtime that can evolve in tandem with DL models without sacrificing the advantages of specialization. The VTA makes the following contributions.

- *A programmable accelerator design* that exposes a two-level programming interface, i.e., a high-level task ISA to allow explicit task scheduling by the compiler stack, and a low-level microcode ISA to provide software-defined operational flexibility. In addition, the VTA architecture is fully parameterizable, i.e., the hardware intrinsics, memories, and data types can be customized to adapt the hardware backend requirements.
- *An extensible runtime system* for heterogeneous execution that performs JIT compilation of microcoded kernels to provide operational flexibility. For example, the VTA runtime enables us extend the functionality of VTA's original computer-vision-centric design to support operators found in style transfer applications without requiring any hardware modifications.
- *A schedule autotuning platform* that optimizes data access and reuse in order to rapidly adapt to the changes to underlying hardware and to workload diversity.

We demonstrate VTA's flexibility by adapting different workloads for two edge class Field Programmable Gate Array (FPGAs). Figure 1 shows how to map a workload to FPGAs using the VTA accelerator and runtime. This process explores VTA hardware variants and performs software autotuning for each candidate design. The resulting design and customized software binaries can be easily integrated into a DL framework. Finally, we evaluate the full system, demonstrating VTA's ability to outperform edge Graphical Processing Unit (GPUs) using edge FPGAs on inference workloads.

VTA HARDWARE–SOFTWARE STACK OVERVIEW

Running an end-to-end workload on VTA requires a complete software stack that can map

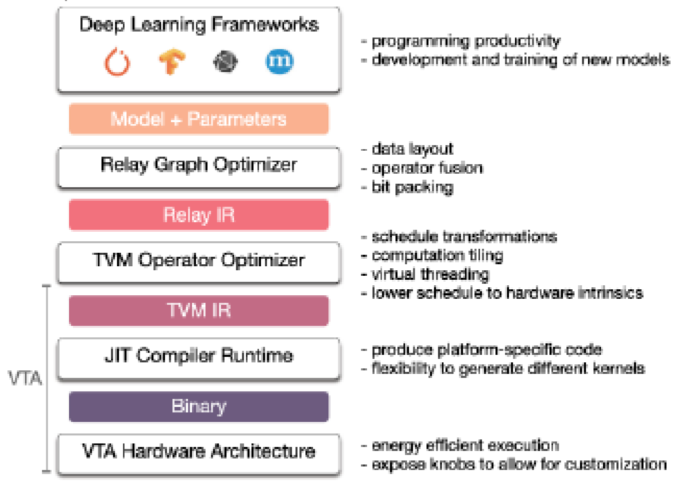
Deep Learning Frameworks
- programming productivity
- development and training of new models
Model + Parameters
Relay Graph Optimizer
- data layout
- operator fusion
- bit packing
Relay IR
TVM Operator Optimizer
- schedule transformations
- computation tiling
- virtual threading
- lower schedule to hardware intrinsics
TVM IR
JIT Compiler Runtime
- produce platform-specific code
- flexibility to generate different kernels
VTA
Binary
VTA Hardware Architecture
- energy efficient execution
- expose knobs to allow for customization

Figure 2. Overview of the software stack built for VTA. We leverage the Apache TVM compiler stack to target VTA.

high-level models down to the programming interface exposed by the VTA. We outline below and in Figure 2 the layers of the VTA system stack, which we built into the Apache Tensor Virtual Machine (TVM) DL compiler stack (https://tvm.ai/).

1) *Framework*: Frameworks let programmers easily express models in a declarative fashion and perform training at scale on standard datasets. Frameworks like TensorFlow, PyTorch, and MxNet have gained widespread adoption, allowing the community to easily share and deploy models. TVM's ability to ingest models from these popular frameworks enables the generic compilation from frameworks to VTA.
2) *Relay graph optimizer*: Relay[7] is TVM's high-level program representation. Relay generalizes the computation graphs used by prior frameworks and DL compilers into a full programming language. The Relay optimization pipeline performs generic optimizations, such as operator fusion and partial evaluation. Relay's design focuses on extensibility, a property we use to extend Relay with optimizations specific to VTA. To target VTA, we quantize inputs to match its low-precision data types, transform data layout, maximize data reuse, and transform input and weight data layouts to utilize VTA's tensor intrinsics.
3) *TVM operator optimizer*: TVM[3] automates the tedious process of scheduling workloads onto VTA accelerator variants. Scheduling is important for multiple reasons. First, it tiles the computation to maximize data reuse. Second, it inserts thread parallelism that VTA's runtime can translate into task-level pipeline parallelism. Third, it partitions operators into subcomputations, which can be mapped to high-level hardware intrinsics, such as bulk Direct Memory Access (DMA) load or General Matrix Multiply (GEMM). TVM incorporates AutoTVM,[4] an automated schedule optimizer, to guide our hardware candidate exploration search for the best VTA candidates given a workload.
4) *JIT compiler and runtime*: The runtime performs JIT compilation of accelerator binaries and manages heterogeneous execution between the CPU and VTA. The JIT compiler abstracts binary compatibility by introducing one level of indirection. We describe the compiler and runtime in more detail in "JIT Runtime System."
5) *Hardware architecture*: VTA is a parameterizable accelerator that speeds up computationally expensive portions of the DL compute graph. It is explicitly programmed by the compiler stack using a two-level programming interface. The architecture is parameterized by the size of the GEMM core, the shared memory (SRAM) shapes, and data type widths. A parameterized hardware architecture makes it possible to retarget the same design to devices with different hardware resources. We describe VTA in more details in the "Hardware Architecture" section.

VTA ARCHITECTURE AND JIT RUNTIME

A successful implementation of a flexible DL accelerator requires the codesign of hardware and software stacks. We next describe, at a high level, two components that we codesigned to achieve this goal: the VTA hardware accelerator architecture and the VTA JIT compiler and runtime.

Hardware Architecture

Figure 3 presents a high-level overview of the VTA hardware organization. VTA consists of four modules: `fetch`, `load`, `compute`, and `store`. Together, these modules define a *task pipeline*, which enables both high compute resource utilization on compute-bound workloads, and high memory bandwidth utilization on memory-bound workloads. These modules communicate over command queues and on-chip SRAMs, which act as unidirectional data channels. Accesses to these memories are synchronized via dependency queues to prevent data hazards, such as READ after WRITE and WRITE after READ. Finally, the multistage architecture (`load-compute-store`) can be used to build task pipelines of arbitrary depth as long as dependencies are properly managed.

Parameterizability: The VTA architecture is fully parameterizable, i.e., the shape of the GEMM tensor intrinsic can be modified to influence the utilization of hardware resources. Modifying the shape of the input, weight, and accumulator tensors that feed the GEMM unit directly affects how many arithmetic units to instantiate and how many SRAMs READ banks need to be exposed. In addition, each data type can be customized to a different integer precision, i.e., weight and input types can be 8 bits or fewer, whereas the accumulation type can be 32 bits or fewer. Control of integer precision enables us scale arithmetic density on the chip when resources are constrained.

Exposing task-level pipeline parallelism: Task-level pipeline parallelism (TLPP) is a vital VTA feature because it enables simultaneous use of compute and memory resources to maximize their utilization. TLPP is based on the paradigm of access-execute decoupling.[8] To extract TLPP, we partition tasks into two mutually exclusive execution contexts, so that concurrent load, compute, and store operations do not interfere with one another. Virtual threads[3] make this partitioning intuitive in TVM. To guarantee timely and correct execution of decoupled access-execute instruction streams, we encode dependency information into task instructions. This results in memory latency hiding for compute-bound workloads (e.g., 2-D convolutions).

Task level ISA: VTA supports a high-level task ISA that encodes multicycle compute and memory operations, including `LOAD`, `GEMM`, `ALU`, and `STORE` instructions. `LOAD` and `STORE` describe how data from Dynamic Random Access Memory (DRAM) is loaded and stored into on-chip SRAMs. Strided memory access is supported to load tensor tiles without modifying memory layout. `GEMM` and `ALU` instructions invoke micro-coded kernels based on micro-op instructions, which describe the data-access patterns over inputs, weights, and biases tensors that define a given DL operator.

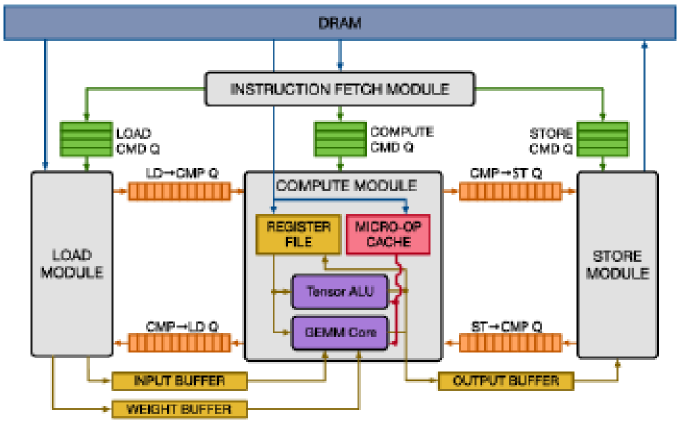

Figure 3. VTA hardware organization. VTA consists of four modules that communicate via queues and shared memories (SRAMs). This defines a task pipeline, which helps maximize compute resource utilization.

A simple execution pipeline in VTA follows.

- The `fetch` module loads task instructions from DRAM and dispatches them according to their type to the corresponding command queues connected to `load`, `compute`, and `store` modules.
- The `load` module loads input, weight, and bias tensor tiles from DRAM into on-chip memories.
- The `compute` module loads a microcoded kernel from DRAM into on-chip memory.
- The `compute` module executes the microcoded kernel to perform either a dense linear algebra computation via the GEMM core or a pairwise arithmetic operation via the Tensor ALU.
- The `store` module READS results processed by the `compute` module and WRITES them to DRAM.

Compute module: Two functional units perform operations on the register file, i.e., the tensor ALU and the GEMM core. The tensor ALU performs element-wise tensor operations, such as addition, activation, normalization, and pooling tasks. The GEMM core performs high-arithmetic-intensity matrix multiplication over input and weight tensors to implement common DL operators including 2-D convolutions or fully connected layers.

The GEMM core performs matrix multiply operations at a pipelined rate of one input-weight matrix multiplication per cycle. Its logic is implemented as parallel vector dot-product using reduction trees, but it can be substituted with other implementations, such as systolic arrays. The GEMM core defines a low-level tensor *hardware intrinsic* that is exposed to the TVM compiler stack. The TVM uses *tensorization*,[3] an automated approach to mapping DL operators, such as 2-D convolutions, down to fixed tensor hardware intrinsics.

Microcode ISA: The `compute` core READS instructions from the micro-op cache, which describes how to perform computation over data. These micro-ops provide no control flow. Therefore, instructions must be unrolled to express repeatable data access stencils. The two types of compute micro-ops are ALU and GEMM operations. To minimize the footprint of micro-op kernels in the on-chip SRAMs and avoid the need for control-flow instructions, the compute core executes micro-op sequences inside a two-level nested loop that computes the location of each tensor register via an affine function.

JIT Runtime System

VTA's JIT runtime enables the cooperative execution of DL workloads between a CPU host and the accelerator. Its design adheres to five objectives:

1) enable heterogeneous execution;
2) lower compiler design complexity;
3) overcome physical limitations;
4) reduce binary bloat;
5) enable future proofing.

Heterogeneous execution: One challenge of fixed-function accelerators is model evolution because they are generally built for specific models. The heterogeneous execution schedules operators into appropriate targets (e.g., CPUs or VTA) depending on their affinity for different types of computation; for instance, it is well known that the first convolutional layer in most CNNs have low arithmetic intensity, and therefore, execute efficiently on CPUs. Heterogeneous execution also provides a *fallback* mechanism for supporting emerging operators that are not yet supported by VTA.

Compiler design: By adding a level of indirection, JIT compilation eliminates the need to WRITE compiler code-generation backends, which can be tedious to maintain for different programmable accelerators. The JIT compiler exposes a high-level Application Programming Interface (API) to TVM to lower schedules onto abstracting away VTA variant-specific architectural details. This enables us extend the TVM compiler support we built for VTA to cover future variants of different shapes and sizes.

Physical limitations: The JIT runtime generates and manages microkernels on the fly. It controls when to load kernels from DRAM into the accelerator-limited micro-op cache. This eliminates micro-op memory physical limitations and enables us support large models, even if all microkernels for all layers do not fit in SRAM at once. It also lets us trade area used by the micro-op cache for other resources, such as data storage or compute units.

Binary bloat: Delaying microkernel generation to the JIT compilation stage minimizes binary bloat. Since VTA's architecture has limited support for control flow, microkernels must be unrolled, which can produce fairly large binaries. In addition, microkernel JIT compilation expresses binaries for heterogeneous execution in a single ISA, i.e., instead of shipping a hybrid binary, we ship only one CPU binary to perform accelerator binary JIT compilation at runtime.

Future proofing: Advancements in DL have described the prevalence of dynamic neural network workloads that incorporate control flow. Additionally, advances in systems show trends toward heterogeneous multiaccelerator systems and scale-out acceleration. Having a runtime that handles dynamic decisions across heterogeneous platforms will simplify the design of hardware accelerators like VTA, and make future model support mainly a software-related endeavor.

VTA HIERARCHICAL OPTIMIZATION

Hardware Exploration for Varying FPGA Sizes

One way to showcase VTA's architectural flexibility is to target different FPGA platforms. FPGAs are becoming increasingly accessible, with sub-$100 development boards, and accessible FPGA cloud computing instances.

Our VTA design offers multiple architectural customization parameters, as shown in Figure 1. Architectural knobs include GEMM hardware intrinsic shape, data types, the number of parallel arithmetic units in the tensor ALU, ALU operations, and Block Random Access Memory (BRAM) distribution between on-chip memories. Circuit knobs include PLL frequency and the degree of hardware pipelining to close timing at higher frequencies. These customization knobs define a hardware design space with hundreds to thousands of individual designs. This design space can be exhaustively explored to find the best candidate for a particular workload. We perform this exploration in a sequence of stratified steps. First, we use a simple FPGA resource model to prune infeasible VTA parameterizations. After pruning, each candidate hardware design is compiled, placed, and routed. We pick the best feasible design for each $\{fpga \times dtype \times batch\}$ combination, but our exploration typically returns a handful of promising candidates; the rest of the designs either yield low peak performance or fail placement, routing, or timing closure. For this final set of designs, we generate optimized schedules using operator autotuning,[4] and we use these schedules to obtain the workload's performance profile.

An analytical model of *peak performance* is used to initially filter hardware designs based on theoretical throughput and frequency assuming compute resources are 100% utilized. However, assuming 100% utilization of compute resources by a particular operator is often inaccurate. For example, depending on the workload mix, operators like `conv2d` with large window sizes may exhibit high arithmetic intensity (measured in Op/Byte). Such operations translate to high utilization and are, therefore, close to peak performance. Operators with low arithmetic intensity (e.g., `conv2d` with a window size of 1) are generally memory bandwidth constrained. For such operators, we use task-level pipeline parallelism to mitigate performance loss resulting from waiting on memory.

Schedule Exploration for Operator Autotuning

Schedule autotuning is the process by which an automated search algorithm attempts to optimize a given program or workload toward peak hardware performance. We perform autotuning by applying different memory tiling, loop transformations (e.g., splitting, reordering, and unrolling), vectorization/tensorization, and parallelization strategies.[4] We then use the TVM compiler to express schedule templates for each operator (e.g., `conv2d`, `conv2d_transpose`, `group_conv2d`, `fc`) we support in hardware. We use TVM's automated scheduling library to obtain schedules that maximize performance for a given combination of operator, tensor shape, and hardware parameterization.

We used the XGBoost[1] search algorithm to find the best schedules for each hardware variant in a limited number of trials. Each workload's layers were then tuned for each hardware candidate. Aggregate inference time was used to select the best VTA hardware variant for a given model.

It takes several hours to exhaustively tune a network on a single hardware variant. Given the large number of VTA hardware designs to test and model architectures to support, autotuning search quickly becomes intractable without careful design. Minimizing full-network autotuning time across multiple hardware candidates introduces a *hierarchical prioritization problem*. We approach this challenge by applying a hyperparameter optimization technique that is based on `SuccessiveHalving`.[5] Instead of choosing from hyperparameters that define a network architecture, we apply this technique to choose from VTA design candidates. We simultaneously inspect how the relative performance of each hardware design evolves for a given workload over each iteration of the optimization algorithm. Throughout optimization we use a round-robin policy to update latency estimates across all operators for each hardware design.

Full Network Optimization Case Study

Figure 4 shows an example of hierarchical optimization for the ResNet-18 workload based on hardware exploration and schedule exploration techniques described previously. We perform these optimizations over a set of VTA candidates generated using W8A8 (8-bit weights, 8-bit activations) data representations. We select eight promising hardware candidates and apply `SuccessiveHalving` to prune designs that do not appear promising. Similar to hyperparameter optimization for neural network training, this task is difficult since the relative performance differences between hardware designs may initially be small. After a moderate number of iterations, `SuccessiveHalving` is able to converge to the best candidate hardware design.

This case study showcases VTA's ability to quickly navigate a nontrivial space of accelerator configurations for a given workload. As accelerator configurations change, so does the software that programs them. This joint-optimization problem can be solved only with a flexible stack.

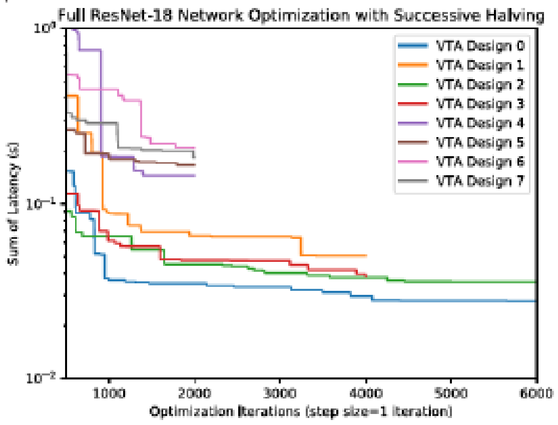

Figure 4. Example of hardware design exploration and schedule autotuning on a complete ResNet-18 inference workload run on Ultra96 FPGA. The exploration begins with promising VTA hardware variants and converges to the optimal hardware design while using a fraction of the optimization time required to exhaustively evaluate each hardware design.

EVALUATION

s the landscape of DL continues to evolve, it is important to support emerging models. We evaluate VTA's ability to support two recent model architectures beyond standard deep convolution nets. First, we evaluate `MobileNetG`, a variant of MobileNet that groups convolution channels by the vector factor of the VTA's GEMM core. Second, we evaluate `DCGAN`, a generative adversarial network model that is used for image-to-image translation and generation.

These models require nontrivial extensions to support new operators. `MobileNetG` requires support for grouped convolutions that exhibit block sparse patterns on channel groups. `DCGAN` requires support for 2-D convolution transpose, which has a spatial sparsity pattern. Accelerators must support these access patterns to avoid unnecessary computations and achieve maximum performance. The runtime can readily make use of schedules to generate microkernels that support these access patterns without changing the hardware.

We integrated VTA into Apache TVM and evaluate five DL models on two FPGA devices with different resource budgets. We import all models from MxNet[2] a DL framework used by Amazon. It is worth noting that Relay's model importers provide access to a wide variety of other front-ends, and VTA is not limited to MxNet.

Figure 5 compares performance across these models, showing VTA-accelerated execution versus highly optimized ARM CPU and GPU platforms that rely on industry-strength DL libraries, i.e., ARM ComputeLib (ARM CL) and TVM. The ARM Cortex-A9, ARM Cortex-A53, and Mali-T860 GPU are taken from the Pynq-Z1 (\$65), Ultra-96 (\$250), and the Firefly-RK3399 (\$200) development boards. For VTA hardware variants, we use an automated 8-bit integer scaling and translation pass from 32-bit floating-point (FP32) with negligible accuracy degradation. For our CPU baselines, we use the TVM autotuner to obtain FP32 CPU kernels that take advantage of NEON vectorization, multithreading and state-of-the-art scheduling tricks (spatial tiling, Winograd transform, etc.). For our GPU baseline, we use the ARM CL v18.03 and exploit 16-bit floating-point (FP16) library support. At the time of the evaluation, ARM CL lacked support for `conv2d` transpose for DCGANs. This motivates our flexible specialization approach to stay ahead of the curve while targeting unconventional workloads.

Figure 5 shows end-to-end results that can be discussed in two groups of comparable devices in terms of cost: first, VTA on the Pynq versus Cortex-A9 (sub-\$100) and second, VTA on Ultra96 versus Cortex-A53 and Mali-T860 GPU (\$200–\$250). First, VTA on the Pynq-Z1 outperforms the Cortex-A9 CPU by 3.0×, 4.4×, 5.3×, and 2.1× on `MobileNetG`, `ResNet-18`, `ResNet-34`, and `DCGAN`, respectively. Second, VTA on the Ultra-96 outperforms the Cortex-A53 by 2.5×, 4.7×, 6.0×, 3.8×, and 11.5× on `MobileNetG`, `ResNet-18`, `ResNet-34`, `ResNet-50`, and `DCGAN`,

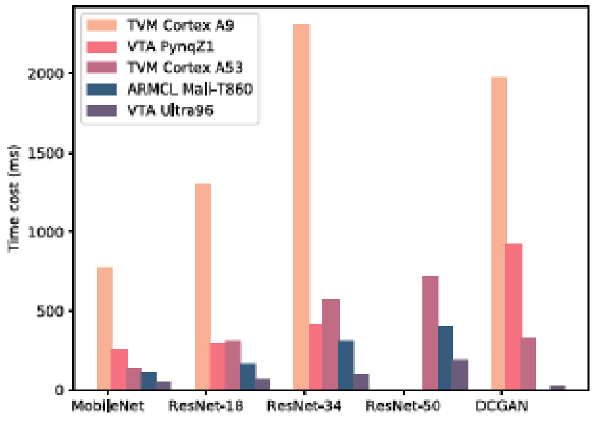

Figure 5. End to end performance evaluation over multiple CPU, GPU, and FPGA-equipped edge systems. For comparable systems, VTA provides a significant performance edge over conventional CPU and GPU-based inference.

respectively. In addition, VTA on the Ultra-96 outperforms the mobile-class Mali-T860 GPU by 2.1×, 2.5×, 3.2×, and 2.1× on `MobileNetG`, `ResNet-18`, `ResNet-34`, and `ResNet-50`, respectively.

Overall, VTA demonstrates its software-defined architectural flexibility, offering high performance while forming an evolutionary path forward for accelerating diverse workloads on various devices.

CONCLUSION

This article presented a hardware–software blueprint for "flexible specialization," i.e., the idea that efficiency gains from hardware specialization are not mutually exclusive with workload flexibility. We introduced VTA, a parameterizable DL architecture that is explicitly programmed via a two-level ISA. We codesigned the accelerator with a runtime system that JIT compiles microkernels to provide operational flexibility. Using this approach, we can support less conventional operators, such as convolution transpose and grouped convolutions without needing to make hardware changes. Our evaluation showed that the VTA effectively maps multiple workloads onto different FPGAs by leveraging off-the-shelf deep learning compilers to quickly integrate optimized software with specialized hardware. Finally, we demonstrated that a well-integrated hardware and software stack helps us perform full-stack optimization and exploration to automate model-to-gates compilation on FPGAs.

ACKNOWLEDGMENTS

The authors would like to thank members of Sampa and SAMPL groups at the Allen School for their feedback on the work and manuscript. This work was supported in part by the Google Ph.D. Fellowship for Tianqi Chen; in part by the National Science Foundation under Grants CCF-1518703, CNS-1614717, and CCF-1723352; in part by the Center for Resilient Infrastructures, Systems, and Processes, and Applications Driving Architectures (ADA), two of six centers in Joint University Microelectronics Program (JUMP); in part by a Semiconductor Research Corporation program sponsored by Defense Advanced Research Projects Agency (DARPA); and by the gifts from Xilinx, Intel (under the Computer Assisted Programming for Heterogeneous Architectures (CAPA) program), Oracle, Amazon, Qualcomm, NVIDIA Corporation, and other anonymous sources.

REFERENCES

1. T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 785–794.
2. T. Chen *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *Proc. Neural Inf. Process. Syst.*, *Workshop Mach. Learn. Syst.*, 2015.
3. T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 578–594.
4. T. Chen *et al.*, "Learning to optimize tensor programs," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, pp. 3389–3400, 2018.
5. K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2016, pp. 240–248.
6. N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th ACM Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
7. J. Roesch *et al.*, "Relay: A new ir for machine learning frameworks," in *Proc. 2nd ACM Int. Workshop Mach. Learn. Program. Lang.*, 2018, pp. 58–68.
8. J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. 9th Annu. Symp. Comput. Archit.*, 1982, pp. 112–119.

Thierry Moreau is a postdoctoral researcher at the University of Washington. He helps run the multidisciplinary Systems, Architectures and Programming Languages for Machine Learning Laboratory group on systems, architectures, machine learning, and programming languages for machine learning. His research interest focuses on building platforms and abstractions that make hardware accelerators easier to adapt and deploy as application trends evolve. He leads the VTA open source DL accelerator effort and serves as an Apache TVM PMC member. He has a BASc in computer engineering from the University of Toronto, and an MS and a PhD in computer science and engineering from the University of Washington. Contact him at: moreau@cs.washington.edu.

Tianqi Chen is working toward a PhD at the Paul G. Allen School of Computer Science and

Engineering Department, University of Washington, working on the intersection of machine learning and systems. He has led the creation of many important machine learning systems, including XGBoost, Apache MXNet, and Apache TVM. He is a recipient of Google PhD fellowship. Contact him at: tqchen@cs.washington.edu.

Luis Vega is working toward a PhD at the Paul G. Allen School of Computer Science and Engineering Department, University of Washington. His research interests include computer architecture, hardware accelerators for machine learning, and domain-specific hardware languages. He has an MS in electrical and computer engineering from the University of Kaiserslautern, Germany. Contact him at: vegaluis@cs.washington.edu.

Jared Roesch is working toward a PhD at the University of Washington, where he works on a variety of topics, including machine learning, programming languages, computer architecture, formal methods, and more. He has a BS in computer science from the University of California, Santa Barbara and an MS from the University of Washington. Contact him at: jroesch@cs.washington.edu.

Eddie Yan is working toward a PhD at the Paul G. Allen School of Computer Science and Engineering Department, University of Washington. His current research interest includes deep learning optimization with an eye toward automatic optimization techniques. He has a BS in electrical engineering from the University of California, Los Angeles. Contact him at: eqy@cs.washington.edu.

Lianmin Zheng is currently an undergraduate student at Shanghai Jiao Tong University. His research interest includes the intersection of machine learning and computer systems. He participated in TVM project during his internship at the University of Washington. Contact him at: lianminzheng@gmail.com.

Josh Fromm is working toward a PhD at the University of Washington, where he specializes in enabling deep learning on resource-constrained platforms through the development of novel architectures, approximating algorithms, and hardware aware scheduling. He has a BS in electrical engineering and computer science from the California Institute of Technology. Contact him at: jwfromm@uw.edu.

Ziheng Jiang is working toward a PhD at the University of Washington. His research interests include theories and practices of large-scale computer system and its intersection with machine learning, in particular deep-learning. He has a BS from Fudan University, where he was a member of Fudan NLP Lab. Contact him at: ziheng@cs.washington.edu.

Luis Ceze is currently a professor with the Paul G. Allen School of Computer Science and Engineering Department, University of Washington. His research interests focus on the intersection between computer architecture, programming languages, machine learning, and biology, currently focusing on end-to-end system optimizations for efficient machine learning and DNA-based data storage and computing. He codirects the Molecular Information Systems Laboratory, and the Systems, Architectures and Programming Languages for Machine Learning Laboratory. He has a BEng and an MEng from the University of São Paulo, Brazil, and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a Senior Member of IEEE and ACM. Contact him at: luisceze@cs.washington.edu.

Carlos Guestrin is currently the Amazon Professor with Machine Learning in Computer Science and Engineering Department, University of Washington. He codirects the Systems, Architectures and Programming Languages for Machine Learning Laboratory, an interdisciplinary ML research group addressing problems in the intersection between ML, systems, computer architecture, and programming languages. He also codirects the MODE Laboratory. He is also the Senior Director of Machine Learning and AI at Apple, where he runs the central ML team for the company, after the acquisition of Turi, Inc. (formerly GraphLab and Dato), a company he cofounded, which developed a platform for developers and data scientists to build and deploy intelligent applications. Contact him at: guestrin@cs.washington.edu.

Arvind Krishnamurthy is currently a professor with the Computer Science and Engineering Department, University of Washington. His research interests include all aspects of building practical and robust computer systems, currently focusing on to develop ways to dramatically improve the performance of applications deployed inside datacenters by integrating hardware innovations, rearchitecting across all layers of the software stack, and developing new algorithms for effective use of computing resources. He has a BTech from Indian Institute of Technology, Madras and a PhD in computer science from the University of California Berkeley. Contact him at: arvind@cs.washington.edu.