©Copyright 2018
Thierry Moreau

Cross-Stack Co-Design for Efficient and Adaptable Hardware Acceleration

Thierry Moreau

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Luis Ceze, Chair

Ras Bodik

Arvind Krishnamurthy

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Cross-Stack Co-Design for
Efficient and Adaptable Hardware Acceleration

Thierry Moreau

Chair of the Supervisory Committee:
Professor Luis Ceze
Computer Science and Engineering

Hardware accelerators are becoming more critical than ever in scaling the capabilities of computer systems in a post-Dennard scaling computing landscape. As abstractions like ISAs and intermediate representations are shifting constantly, building capable software stacks that expose familiar programming interfaces poses a significant engineering challenge. In addition, the push for ever more efficient and cost-effective hardware has brought the need to expose quality-efficiency tradeoffs across the stack, particularly in hardware accelerators where computation and data movement dominates energy. The goal of this dissertation is to propose hardware and software techniques that work in concert to facilitate the integration of hardware accelerators in today's ever-evolving compute stack. Specifically, we look at co-design methodologies that (1) make it easy to program specialized accelerators, (2) allow for adaptability in the context of evolving workloads, and (3) expose quality-efficiency knobs across the stack to adapt to shifting user requirements. In **Chapter 1**, I discuss why specialization is critical to push the capabilities of modern systems, and identify challenges that remain in the way to provide efficient and adaptable specialization moving forward. In **Chapter 2**, I present SNNAP, a hardware design coupled with a familiar software API that approximately offloads diverse compute-intensive regions of code to a tightly coupled FPGA to deliver significant energy savings. This approach makes it much easier to target FPGAs for software programmers, as long as they can express quality bounds for their

target application. In **Chapter 3**, I present QAPPA an C/C++ compiler framework that can target quality programmable accelerators, i.e. accelerator designs that expose quality knobs in their ISA. The key of QAPPA is to translate application-level quality bounds into instruction-level quality settings via an auto-tuning process. In **Chapter 4**, I present the VTA hardware-software stack designed for extensible deep learning acceleration as data sets, models, and numerical representations evolve. VTA exposes a layered stack that offloads design complexity away from hardware: this makes updating the stack to support new models and operators a software-centric challenge. Finally, in **Chapter 5**, I discuss recent efforts outside of the research realm aimed at popularizing access and reproducibility of cross-stack pareto-Optimal design.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . viii

Chapter 1: Introduction . . . 1

Appendices . . . 10
- 1.A Background: A Taxonomy of General Purpose Approximate Computing Techniques 11
- 1.B Background: An Survey of Domain Specialized Deep Learning Accelerators . . . 21

Chapter 2: SNNAP: Approximately Mapping Diverse Regions of Code to a Single FPGA-Based Substrate via Neural Acceleration . . . 25
- 2.1 Introduction . . . 26
- 2.2 Programming . . . 27
- 2.3 Architecture Design for SNNAP . . . 31
- 2.4 Hardware Design for SNNAP . . . 35
- 2.5 Evaluation . . . 42
- 2.6 Related Work . . . 56
- 2.7 Conclusion . . . 57

Appendices . . . 58
- 2.A SNNAC: An Error-Tolerant Low-Voltage SRAM Neural Network Accelerator ASIC 58

Chapter 3: QAPPA: Quality Autotuner for Precision Programmable Hardware Accelerators . . . 63
- 3.1 Introduction . . . 64
- 3.2 QAPPA: A Quantization Autotuner . . . 66
- 3.3 PERFECT Application Study . . . 72

3.4 Dynamic Quantization Scaling 82
3.5 Approximation Study 89
3.6 Related Works 96
3.7 Conclusion. 97

Chapter 4: VTA: Flexible Architecture & Runtime Co-Design for Deep Learning Specialization 98
4.1 Introduction 99
4.2 Background: Anatomy of A Deep Learning System Stack 102
4.3 VTA Hardware Architecture 106
4.4 VTA JIT Runtime System 115
4.5 Evaluation 119
4.6 Related Work 128
4.7 Conclusion 130

Chapter 5: Conclusion 131
5.1 Accomplished Research Summary 131
5.2 Beyond Academic Research 133
5.3 Artifact Submissions Overview 140
5.4 Lessons Learned and Future Work 142
5.5 Concluding Remarks: An Outlook to the Future 143

Bibliography 145

LIST OF FIGURES

Figure Number Page

1.1 Analyzing compute intensive application kernels from the PERFECT benchmark suite [18] with ACCEPT [154] reveals the average of fraction dynamic instructions that are safe to approximate (i.e. an erroneous outcome would have no catastrophic side effects on the execution of the kernel [155]). Coincidentally, the same characteristics that make these applications prime targets for hardware specialization (no pointer chasing, low degrees of memory and control flow divergence) also makes them amenable to additional energy savings thanks to approximations. 3

1.2 Specialization can take different forms and for that reason constitutes an adoption challenge across the system stack. This high-level overview of specialization categorizes accelerators across two axes: (1) domain-specialization, i.e. how tailored an accelerator is to an application domain, and (2) quality-specialization, i.e. how tailored an accelerator is to a given use-case. This thesis dissertation explores how to make accelerators easier to program and extend as workloads evolve. SNNAP (Chapter 2) touches on targeting spatially programmable accelerators by approximately mapping diverse regions of code to a single hardware substrate, via an algorithmic transformation that uses neural networks to memoize the code target. QAPPA (Chapter 3) aims to make it easy to target quality programmable accelerators translate programmer-defined application quality constraints, down to ISA-level quality settings. Finally, VTA (Chapter 4) presents an domain-specialized software/hardware stack that make it easy to efficiently offload diverse deep learning workloads onto a modular deep learning architecture that can be tailored to specific application scenarios. VTA exploits layered intermediate representations to make the stack extensible to evolving deep learning workloads. 5

1.B.1 Salient characteristics of recently-proposed deep learning accelerators. 21

2.1 SNNAP system diagram. Each Processing Unit (PU) contains a chain of Processing Elements (PE) feeding into a sigmoid unit (SIG). 32

2.1 Detailed PU datapath. PEs are implemented on multiply–add logic and produce a stream of weighted sums from an input stream. The sums are sent to a sigmoid unit that approximates the activation function. 35

2.2 Implementing multi-layer perceptron neural networks with systolic arrays. 36

2.1 Performance and energy benefit of SNNAP acceleration over an all-CPU baseline execution of each benchmark. 45

2.2 Performance of neural acceleration as the number of PUs increase. 47

2.3 Impact of batching on speedup. 48
2.4 Resource Utilization for a 1-PU NPU containing 1 to 16 PEs. 49
2.5 Exploration of SNNAP static resource utilization. 50
2.6 Output of `sobel` for a 220x220 pixel image. 52
2.7 Performance and energy comparisons of HLS and SNNAP acceleration. 54
2.8 Resource-normalized throughput of the NPU and HLS accelerators. 55
2.A.1 **(left)** The fraction of total power dissipated by weight storage SRAMs, and **(right)** the fraction of total SRAM used to store fully-connected weights. On-chip weight storage accounts for a significant fraction of the total power dissipation in state-of-the-art DNN accelerators. Even for Conv-DNNs such as AlexNet, weight storage is dominated by fully-connected layers. 59
2.A.2 (left) MATIC [97] increases energy-efficiency by aggressively scaling supply voltages of on-chip weight SRAMs. (right) Compared to hardware paired with conventionally-trained neural network models, MATIC leverages an adaptive training process to recover from errors caused by voltage overscaling. 60
2.A.3 Architecture of the SNNAC DNN accelerator. The SNNAP design is tightly integrated with an OpenMSP430 micro-controller. 61
2.A.4 (a) Microphoto of a fabricated SNNAC test chip, and (b) summary of test chip characteristics. The baseline voltage, power, frequency, and energy efficiency are reported. 62
3.1 QAPPA Autotuner System Architecture. 65
3.1 Program annotation with APPROX type qualifier. Variables that are safe to approximated are annotated by the user. The compiler then infers the program instructions that can be approximated. 68
3.1 Dynamic instruction category mix of the PERFECT kernels. The approximable instructions are colored in shades of blue, and the precise instructions categories are colored in gray. 77
3.2 Aggregate bit-savings for 14 PERFECT kernels over a 20dB to 100dB SNR range. . 78
3.3 Bit-savings vs. SNR averaged over PERFECT kernels, for integer arithmetic, FP arithmetic, memory ops and math functions. 78
3.4 CDF of exponent value range of all floating-point variables in the PERFECT benchmark suite. 80
3.1 Quantization scaling mechanisms overview. (a) Default wide addition on wide adder. (b) Narrow addition on wide adder. (c) Wide addition on narrow adder (d) Narrow addition on narrow adder. 82

3.2 Energy vs. precision relationship for precision-scaled multiplier designs (32 bit baseline). 83
3.3 Simplified schematic of (a) bit-sliced adder and (b) bit-sliced multiplier. 84
3.4 Arithmetic energy reduction on the PERFECT benchmark at different bit slicing granularities and at different SNR targets (higher is better). 86
3.5 Ideal bandwidth reduction on PERFECT benchmark suite at different data packing granularities and at different SNR targets (higher is better). 87
3.6 Example of quantization-scalable pipeline: memory packing and unpacking mechanism used in Proteus [91] combined with operand narrowing used in Quora [186]. The input and output data can be loaded in its packed format to save memory bandwidth. 88
3.1 Bit-flip probabilities of each output bit for a single-precision floating point adder at voltage overscaling factors [0.8-1.0]. Sign and exponent bits are in blue, mantissa bits are in green/yellow. 90
3.2 PERFECT kernel SNR at voltage overscaling factors of 0.95, 0.90 and 0.84 corresponding to 10%, 20% and 30% energy savings. SNR is measured collected over 100 runs, values represent median SNR, and error bars represent min and max error. 91
3.3 Approximating the inverse kinematics kernel: (a) default DFG, (b) optimized fixed point DFG with PWP, (c) neural approximation DFG. Operations that read data from local SRAM are colored in gray. 93
3.4 Energy and storage comparison of quantized acceleration vs. neural acceleration on AxBench kernels (lower is better). 95
4.1 VTA provides flexibility with respect to hardware targets and deep learning 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 (section 4.5). This process generates the pieces necessary to deploy VTA in any deep learning framework (section 4.2). . . 99
4.1 Anatomy of a Modern Deep Learning System Stack. We provide Google's TensorFlow-XLA-TPU industrial stack as a reference point. VTA's architecture and runtime provide a reference implementation of a hardware design, and low-level code generator respectively. Operator-level schedule optimizations, and graph-level optimizations are handled by the upper layers of the software stack which VTA is integrated with. 103
4.1 The VTA hardware organization. VTA is composed of modules that communicate via queues and SRAMs. This enables task-level pipeline parallelism, which helps maximize compute resource utilization. 107

4.2 The VTA high-level instruction fields. LOAD and STORE instructions perform 2D strided DMA reads/writes between DRAM and SRAM. GEMM instructions are used to perform matrix multiplication and 2D convolutions while ALU instructions can perform a wide range of activation, normalization, and pooling tasks. . 108

4.3 Decoupled access-execute allows concurrent utilization of compute and memory resources in hardware: this uncovers task-level pipeline parallelism. 109

4.4 Inserting data dependences between instructions is essential to ensure execution correctness for a decoupled access-execute instruction stream. 110

4.5 Each module is connected to its consumer and producer via RAW and WAR dependence queues and unidirectional SRAM data channel. 111

4.6 The VTA GEMM core can perform one dense matrix multiplication over an input tensor and a weight tensor, and add the result into a register file tensor. The data addressing pattern is specified by a micro-coded sequence: this allows us to map very different deep learning operators onto a single hardware tensor intrinsic. . . 112

4.7 The VTA tensor ALU can implement tensor-tensor element wise operations, or tensor-scalar operations. 113

4.8 The `load` module can perform 2D DMA loads with a strided access pattern from DRAM to SRAM. In addition, it can insert 2D padding on the fly, which is useful when blocking 2D convolution. This means that VTA can tile 2D convolution inputs without paying the overhead of re-laying data out in DRAM to insert spatial padding around input and weight tiles. 115

4.1 Simple vector addition dataflow graph and corresponding low level calls into JIT runtime. A and B are stored in global memory (DRAM) and are copied via DMA into the register file (accumulator memory scope, a.k.a. register file). The vector add computes the results in the local register file, which are written back to DRAM via a DMA copy. 117

4.1 Schedule exploration with XGBoost for a single ResNet-18 layer on Ultra-96. Eight VTA design candidates with (2,16)x(16,16) and (8,8)x(8,8) GEMM intrinsic at W8A8 are considered. Layer is conv2d: IC=256, OC=256, H=W=14, KW=KH=3, stride=(1,1), padding=(0,0). 124

4.2 We use `SuccessiveHalving` for choosing among potential hardware designs on a complete ResNet-18 inference workload. Here, we begin with promising VTA hardware variants. `SuccessiveHalving` converges to the optimal hardware design while using a fraction of the optimization time required to exhaustively evaluate each design. 125

4.3 Throughput improvement on each ResNet-18 convolution layer versus integer precision of kernel weights (8-bit down to 2-bits) running on Pynq-Z1. 125

4.4 Improvement in compute throughput of ResNet-18 layers as we decrease the kernel weight accuracy of VTA designs variants, seen under their respective rooflines, on Pynz-Z1. 126
4.5 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. 127

5.1 Each of the students assignment submissions according to their efficiency (8k batch inference latency) and validation accuracy. The Pareto frontier is represented as a green dotted line. 135
5.2 We leverage the open Collective Knowledge workflow framework (CK) and the rigorous ACM artifact evaluation methodology (AE) to allow the community collaboratively explore quality vs. efficiency trade-offs for rapidly evolving workloads across diverse systems. 139
5.3 A live scoreboard can produce a scatterplot of system implementations across any two dimensions among accuracy, latency, throughput, batch size, price, model size, peak power, clock frequency. 140

LIST OF TABLES

Table Number Page

1.A.1 Taxonomy of approximate computing techniques. 12

2.1 Static PU scheduling of a 2−2−1 neural network. The naive schedule introduces pipeline stalls due to data dependencies. Evaluating two neural network invocations simultaneously by interlacing the layer evaluations can eliminate those stalls. 42

2.2 Applications used in our evaluation. The "NN Topology" column shows the number of neurons in each MLP layer. The "NN Config. Size" column reflects the size of the synaptic weights and microcode in bits. "Amdahl Speedup" is the hypothetical speedup for a system where the SNNAP invocation is instantaneous. . . . 43

2.3 Microarchitectural parameters for the Zynq platform, CPU, FPGA and NPU. . . . 44

2.1 Post-place-and-route FPGA utilization. 51

2.2 HLS-kernel specifics per benchmark: required engineering time (working days) to accelerate each benchmark in hardware using HLS, kernel clock, whether the design was pipelined, most-utilized FPGA resource utilization. 53

3.1 PERFECT kernels overview. "Annotations" refer to how many lines of code had to be altered with ACCEPT-style type annotations. "Static Approx. Insn." refers to the total number of instructions that were deemed safe to approximate by ACCEPT. "Dynamic Approx. Insn." refers to the percentage of overall instructions that are safe to approximate over the course of the program execution. "Approx. Runtime Overhead" refers to the slowdown experienced after approximate code injection by QAPPA over the original kernel. "Autotuner Steps" indicates the number of tuning steps taken to find a configuration that could not be approximated further without violating a 40dB quality target. "Autotuner Runtime" indicates how long it takes to tune each kernel as a multiple of its original runtime. 73

3.1 Bit-savings loss from using a empirical guarantee to statistical guarantee at 90% and 99% confidence. We vary the quality target at medium (20dB) a high (40dB) settings on the PA1 kernels. 81

4.1 Instruction stream management runtime functions 116

4.2 Compute micro-kernel generation functions 116

4.3 We target three classes of edge FPGAs, ranging different cost tiers, and that provide different amounts of resources. 119
4.4 We provide a sample of VTA configurations considered favorable that we compiled for Ultra-96/Pynq-Z1 and software constraints defined by data type, or batch size. The data shows promise in terms of adapting the design to (1) occupy available hardware resources, (2) successfully closing timing at high clock targets, (3) exposing different GEMM tensor intrinsics, and (4) scaling peak throughput when reducing precision. 120

ACKNOWLEDGMENTS

The author wishes to express his huge gratitude to his partner Michelle for her endless patience and support in spite of the long distance. The author also wishes to thank his parents, Jacques and Thérèse, his sister Nathalie, his uncle Jean-Claude and aunt Ursula for believing in him throughout those long years of studies. He would like to extend his thanks to his advisor Luis Ceze for this help and support throughout the years, and the members of the reading committee Ras Bodik, Arvind Krishnamurthy and Visvesh Sathe for their guidance, feedback, and time. Many of his graduate school colleagues have been a bottomless source of support through good times and hard times: he would like to thank the members of the Sampa and SAMPL lab, his CSE486 office mates, as well as all of the students from his cohort who embarked on this grad school adventure at the same time. The author would like to warmly thank Jacob Nelson and Tianqi Chen for being excellent research mentors. Last but not least, the author would like to thank Lindsay Michimoto for being a great source of help throughout the rough first weeks of his Ph.D.

DEDICATION

In memory of my good friend Michael, and my loving aunt Ursula.

Chapter 1
INTRODUCTION

"The computing scientist's main challenge is not to get confused by the complexities of his own making."

– Edsger W. Dijkstra

The faltering of Dennard scaling [58] announced the end of effort-free performance improvements. In other words the growth of processing efficiency, measured in throughput per Watts, would taper off from its exponential trajectory since the popularization of integrated circuits technology in the 60's. Traditionally, computer architects would manage to harness transistor scaling into building faster processors by enabling higher clock speeds with deeper pipelines, improved instructions-per-cycle (IPC) with smarter scheduling, and higher memory efficiency with better caches. But in a post-Dennard scaling landscape, architects would need to exploit more ambitious strategies to scale processor performance to and beyond the end of Moore's Law.

The introduction of *hardware specialization* has helped scale the capabilities of commodity systems from the datacenter [113, 90, 137, 94, 42] to edge devices [69, 56, 41]. By tailoring hardware to the characteristics of a stable application (e.g. type of data parallelism, amounts of data reuse, control flow divergence, memory access patterns etc.) hardware specialization can drastically minimize the number of Joules per operations required to complete a given computational task. Specialization can take different forms depending on the degree to which an accelerator is specialized to a domain (i.e. algorithm), or quality requirements (i.e. use case). Figure 1.2 shows an overview of the diversity of accelerator designs that have appeared over the last decades.

Domain-Specialization (a.k.a. algorithmic specialization) There exists a wide spectrum of hardware accelerators, each exhibiting different trade-offs between degree of specialization to a

given application domain, and allowable reuse across separate application domains:

- On one end of the spectrum, *fixed-function accelerators* (or domain-specialized accelerators [129]) are highly tuned to well understood and stable algorithms, and generally apply to pre-determined use-cases. The Anton supercomputer [163] is an example of a highly engineered ASIC chip tailored for calculating electrostatic and van der Waals forces in the context of molecular dynamics simulations. Every aspect of the Anton design down to the width of each arithmetic component was tailored for molecular dynamics. This makes the Anton supercomputer difficult to use across other applications, and across other use-cases that would dictate lower accuracy requirements to minimize energy.

- On another end of the spectrum, we have *spatially programmable accelerators*, which can take various forms, and accommodate for a varied set of applications that exhibit similar characteristics such as high degree of parallelism, high arithmetic intensity, regular memory access patterns and low amounts of control flow divergence. Spatially programmable accelerators most commonly take the form of FPGAs [137], which offer fine-grained control over how hardware resources are allocated and configured to offload compute intensive tasks. But the high degree of flexibility offered by FPGAs hurts their power efficiency [102]. CGRAs [32, 135] and dataflow processors [173, 28, 132] offer coarser degrees of hardware specialization to improve upon FPGA's relatively low power efficiency.

- In the middle of this specialization spectrum, we have *behavior-specialized accelerators* [129], which exploit program behavior across many application domains. Examples include accelerators for frequently executing short program traces such as BERET [75], XLOOPS [75], and DYSER [21]. These accelerators are typically highly integrated within a CPU pipeline to minimize the cost of offloading small bursts of computation. This dissertation won't tackle the question of programming this class of accelerators since research in that area is still in its infancy (in comparison, fixed function accelerators and FPGAs are already permeating the datacenter and popular edge devices).

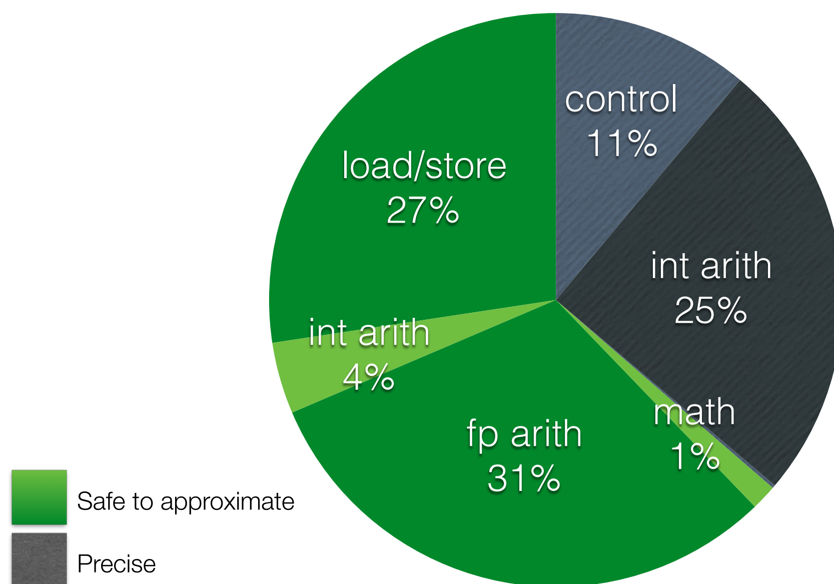


Figure 1.1: Analyzing compute intensive application kernels from the PERFECT benchmark suite [18] with ACCEPT [154] reveals the average of fraction dynamic instructions that are safe to approximate (i.e. an erroneous outcome would have no catastrophic side effects on the execution of the kenrel [155]). Coincidentally, the same characteristics that make these applications prime targets for hardware specialization (no pointer chasing, low degrees of memory and control flow divergence) also makes them amenable to additional energy savings thanks to approximations.

Quality-Specialization (a.k.a. use-case specialization). Another trend in specialization is to insert accuracy-efficiency tradeoffs in the micro-architecture [92, 171] or circuits [96, 97] of accelerators to make them *quality-programmable*. By exposing quality knobs in these accelerator's ISA, the software stack can scale how much energy is spent as long as quality constraints aren't violated. Many proposals for quality-programmable accelerators were made in the context of *approximate computing research*, which advocates for systems to expose quality vs. efficiency tradeoffs across the stack to bring their operation closer to a Pareto-optimal point.

This growing trend of quality-programmable accelerators exposes a second axis in the specialization spectrum: the *quality-specialization* axis. By making accelerators less quality-specialized, and therefore more quality-programmable, a human programmer can better tune the amount of

energy spent on a given problem applied to a specific *use-case*. For instance, a Sobel filter used in a smartphone photography filter would be designed under different QoR constraints than a Sobel filter used in a vision pipeline for cancerous cell identification.

The good news is that approximate computing techniques are best applied to specialized accelerators, rather than general purpose processors. In general purpose processors, dominant control and instruction fetching/decoding overheads impose an "Amdahl limitation" to how much energy savings can be achieved in compute and memory. *Disciplined approximate computing* dictates that control flow branches, arithmetic and memory operations that impact memory references have to be executed precisely to avoid catastrophic errors [155]. Coincidentally, applications that make excellent targets for specialization are also the ones that would benefit highly from fine grained approximations as Figure 1.1 shows. Take a graph processing for instance: the high amount of pointer chasing, and low amount of arithmetic operations per memory operations means that not only would accelerators do poorly due to irregular memory accesses and low arithmetic operations, but also that approximations would have limited impact on reducing efficiency since pointer chasing has to be performed precisely to avoid catastrophic de-references.

Challenges. The proliferation of accelerator designs across the domain-specialization and quality-specialization axes shown in Figure 1.2 makes it challenging to build software support for this broad range of hardware designs. For instance, fixed-function accelerators are challenging to map applications to, and generally require ad-hoc software libraries to be used effectively. These software libraries take much engineering effort to build and optimize. As a result, many accelerator designs that are taped out never leave the ”micro-benchmark” stage of evaluation, because they were not designed with programmability in mind. In addition, spatially programmed accelerators like FPGAs offer flexibility, but programming them is notoriously difficult since they require experience in hardware design and optimization. Although High Level Synthesis (HLS) tools [194, 9] have raised the level of abstraction to describe hardware, they still require programmers to think like hardware designers. Finally, quality-programmable accelerators [96, 92] expose knobs to scale energy according to quality requirements, but is is not straightforward to derive

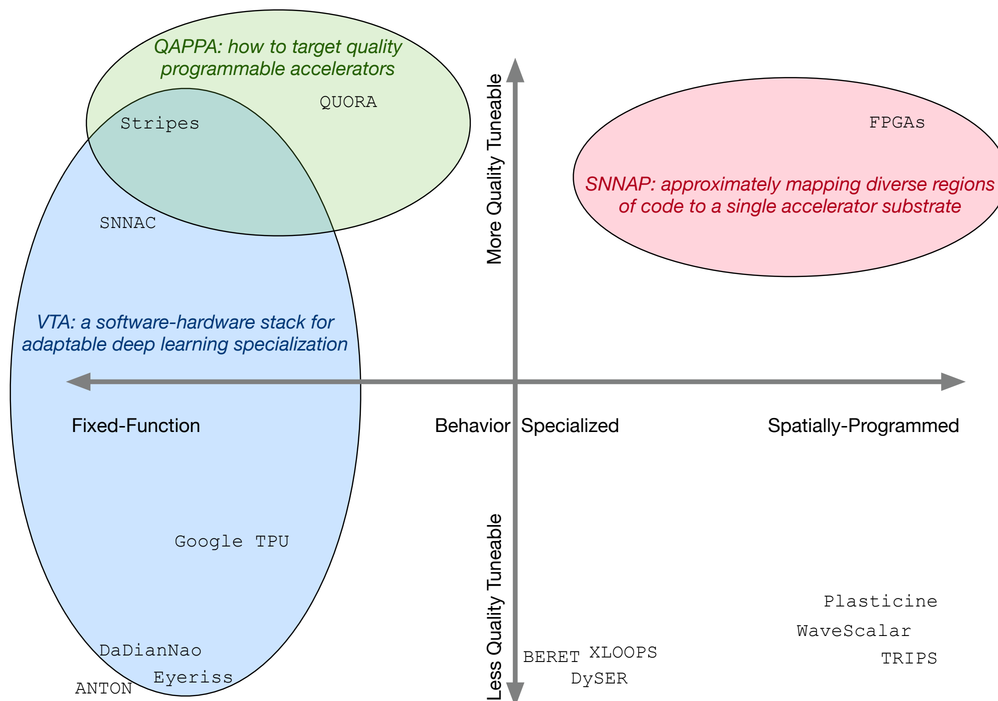


Figure 1.2: Specialization can take different forms and for that reason constitutes an adoption challenge across the system stack. This high-level overview of specialization categorizes accelerators across two axes: (1) domain-specialization, i.e. how tailored an accelerator is to an application domain, and (2) quality-specialization, i.e. how tailored an accelerator is to a given use-case. This thesis dissertation explores how to make accelerators easier to program and extend as workloads evolve. SNNAP (Chapter 2) touches on targeting spatially programmable accelerators by approximately mapping diverse regions of code to a single hardware substrate, via an algorithmic transformation that uses neural networks to memoize the code target. QAPPA (Chapter 3) aims to make it easy to target quality programmable accelerators translate programmer-defined application quality constraints, down to ISA-level quality settings. Finally, VTA (Chapter 4) presents an domain-specialized software/hardware stack that make it easy to efficiently offload diverse deep learning workloads onto a modular deep learning architecture that can be tailored to specific application scenarios. VTA exploits layered intermediate representations to make the stack extensible to evolving deep learning workloads.

these knob settings from high-level application QoR requirements. Much of the derivation for these knobs are derived from ad-hoc Matlab or Python scripts [96, 92] which require applications to be entirely re-written.

It has become evident that we need better tools and methodologies to map application code to new accelerator designs in ways that would maximize efficiency, and maintainability. This dissertation addresses three challenges that affect the adoption of accelerators in the modern compute stack:

Programmability Challenge Hardware architectures are only as good as the software stacks built to program them. For *domain-specific accelerators*, mapping regions of code to specialized hardware primitives requires hand-crafted compiler support. Specifically, building highly tuned libraries that expose a simple programming API to software programmers can take significant amounts of engineering, given that accelerators and workloads can evolve over time. An example of expertly crafted software library for accelerators is NVIDIA's CuDNN deep learning library, which provides finely tuned implementation of common GPU deep learning operators (kernels) to framework builders. Enabling fast development of these critical software libraries requires careful co-design of domain specific languages (DSLs), optimizing compilers, code-generators, and the hardware-software interface of these accelerators.

Spatially programmed accelerators on the other hand let programmers map divergent code to a single accelerator substrate. This "generality" lets applications reuse the same hardware resources and also mitigates the risks of over-specializing hardware to a given applications domain. The most common example of spatially programmed accelerators are FPGAs, which provide a balance between flexibility and efficiency [137], but rely upon tedious hardware description languages and toolchains for programming. In order to facilitate the wide-scale adoption of FPGA hardware, we want to maintain a software programming paradigm which programmers are familiar with.

Adaptability Challenge Design, verification and tape-out costs for *domain specialized accelerators* have increased exponentially at every process node shrink [94]. For rapidly evolving application domains like deep learning, this makes hardware specialization a risky undertaking. One approach to designing adaptable accelerators is to expose the right amount of

software-defined programmability in hardware to allow for some degree of post-tapeout flexibility. In addition, templatized hardware designs can define a parameterization space that can be explored to quickly respond to shifting application needs. In software, accommodating for a large space of templatized hardware designs requires flexible abstractions that can adapt to divergent hardware intrinsics and data representations. Consequently, achieving adaptability in domain specific accelerators requires careful co-design of hardware and software layers.

Pareto-Efficiency Challenge *Quality-programmable accelerators* that expose approximation knobs open up a quality vs. efficiency space that can be challenging to navigate. In particular, translating user-defined application quality requirements (e.g. SNR for signal processing applications) down to low-level accuracy knobs (e.g. number of mantissa bits for a floating point add) that minimize energy requires compiler support, well defined quality metrics, and automated tuning techniques.

Fixed-function accelerators on the other hand can take advantage of domain specificity across the compute stack to gracefully mitigate low-level errors. Deep learning systems for example are known to be very resilient to imprecisions: models are known to be forgiving to errors [176, 97], or aggressive quantization [51, 142] thanks to their ability to be retrained around imperfections. With full flexibility over the software and hardware stack, many synergistic Pareto-optimization opportunities open up. Cross stack decisions include: what model architecture, hyper-parameters, parameter compression, operator (e.g. convolution vs. depthwise convolution), scheduling knobs, and micro-architectural optimizations to use. In this deep learning, the design space of models, schedules, and hardware designs can be intractably large to navigate.

Overview of Contributions The goal of this dissertation is to propose hardware and software techniques that work in concert to facilitate the integration of diverse hardware accelerators in the system stack. Specifically, we look at three bodies of work that address the aforementioned

challenges of programmability, adatability and Pareto-efficiency.

- In **Chapter 2**, I discuss SNNAP [126, 97], a hardware design coupled with a familiar software API that approximately offloads diverse compute-intensive regions of code to a tightly coupled FPGA to deliver fixed-function accelerator levels of energy savings. SNNAP addresses the programmability challenge for spatially programmed accelerators, by making it much easier to target FPGAs for software programmers compared to HLS tools [194], as long as they can express quality bounds for their target application.

- In **Chapter 3**, I propose QAPPA [121, 122] an C/C++ compiler framework that can target quality-programmable accelerators, i.e. accelerator designs that expose quality knobs in their ISA. QAPPA addresses the Pareto-efficiency challenge for quality programmable hardware by safely translating application-level quality bounds into instruction-level quality settings via a quality auto-tuner. Program safety is ensured by ACCEPT [154]'s guarantees on disciplined approximate execution [155]. QAPPA also lets programmers derive energy savings bounds for architectures that expose the right set of hardware quality scaling mechanisms, and lets us qualitatively compare these quality scaling techniques to other fine-grained approximations, like voltage under-scaling.

- In **Chapter 4**, I describe the VTA hardware-software stack designed for extensible deep learning acceleration as data sets, models, and numerical representations evolve. VTA addresses the programmability and adaptability challenges for deep learning accelerators, by jointly designing a complete software-hardware stack. This stack is composed of hardware agnostic intermediate representations, IR transformation primitives, a low-level JIT compiler, a flexible ISA, and explicitly controlled decoupled access-execute micro-architecture. By allowing hardware, software schedule, and neural network model customization, VTA tackles the Pareto-efficiency challenge by facilitating synergistic quality-efficiency exploration across the stack.

In the following appendix subsections, we provide background information on both approximate computing and deep learning specialization research. In Section 1.A, we discuss a taxonomy of approximate computing techniques across the stack [124]. In Section 1.B, we discuss salient properties of deep learning accelerators and how they help us inform the design of a generic deep learning hardware-software stack.

CHAPTER APPENDIX

1.A Background: A Taxonomy of General Purpose Approximate Computing Techniques

Published As: Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger and Adrian Sampson, *A Taxonomy of General Purpose Approximate Computing Techniques*, IEEE Embedded Systems Letters, 10(1):2-5, 2018.

Abstract: *Approximate computing is the idea that systems can gain performance and energy efficiency if they expend less effort on producing a "perfect" answer. Approximate computing techniques propose various ways of exposing and exploiting accuracy–efficiency trade-offs. In this chapter, we present a taxonomy that classifies approximate computing techniques according to salient features: visibility, determinism, and coarseness. These axes allow us to address questions about the correctability, reproducibility, and control over accuracy–efficiency tradeoffs of different techniques. We use this taxonomy to inform research challenges in approximate architectures, compilers, and applications.*

1.A.1 Introduction

Approximate computing encompasses a broad spectrum of techniques that relax accuracy to improve efficiency. Although the term is new, the principle is not: floating-point numbers, for example, efficiently but approximately represent the real numbers in the digital domain. Efficiency–accuracy trade-offs are also commonplace in digital signal processing, where techniques such as quantization and decimation are crucial for tractable designs.

Opportunities abound for exploiting efficiency–accuracy trade-offs at every layer of the system stack, from compilers to circuit design. Cross-cutting concerns about energy efficiency and the future of CMOS scaling have created a boom in approximate computing research. While exciting, the multitude of approaches complicates discussions and obscures common patterns. A single monolithic "approximate computing" label, spanning ideas as disparate as voltage overscaling [60], tweaking floating-point precision [150], and code perforation [164], is too broad to identify the foundations of the field.

This appendix section presents a taxonomy of general-purpose approximate computing tech-

Software Technique	Visible	Deterministic	Coarse
Approximate GPU Kernels [153, 109]	Y	Y	Y
Approximate Synthesis [26, 117]	Y	Y	Y
Algorithm Selection [17, 14]	Y	Y	Y
Code Perforation [164]	Y	Y	Y
Lossy Compression / Packing [153]	Y	Y	Y
Parallel Pattern Replacement [152]	Y	Y	Y
Bit-Width Reduction [150]	Y	Y	N
Float-to-Fixed Conversion [2]	Y	Y	N
Approximate Parallelization [30]	Y	N	Y
Statistical Query [5]	Y	N	Y
Synchronization Elision [146]	Y	N	Y
Hardware Technique	**Visible**	**Deterministic**	**Coarse**
Digital Neural Acceleration [61]	Y	Y	Y
Interpolated Memoization [120]	Y	Y	Y
Approximate Warp Deduplication [191]	Y	Y	Y
Clock Overgating [99]	Y	Y	N
Load Value Approximation [116]	Y	Y	N
Approximate Cache Coherence [147]	Y	Y	N
Concise Loads and Stores [86]	Y	Y	N
Instruction Memoization [11]	Y	Y	N
Precision Scaling [186, 86, 93]	Y	Y	N
Logical Simplifications [187]	Y	Y	N
Reduced-Precision FPU [178]	Y	Y	N
Analog Neural Acceleration [171]	Y	N	Y
Approx. Processors [108, 199]	Y	N	N
Voltage Overscaling [60, 107]	Y	N	N
Stochastic Logic [66]	Y	N	N
Approx. PCM Multi-Level Cells [156]	Y	N	N
SRAM Soft Error Exposure [60]	Y	N	N
Approximate Value Dedup. [115, 158]	N	Y	Y
Low-Refresh DRAM [112]	N	N	N

Table 1.A.1: Taxonomy of approximate computing techniques.

niques. An approximate computing technique is deemed general if it is not specific to a given algorithm or application domain. We classify techniques along three axes: correctability of the approximation effects, reproducibility of the approximate results, and control over the efficiency–accuracy trade-offs.

1.A.2 Motivation

Our taxonomy characterizes approximation techniques around three practical concerns:

1. **Correctability:** How can the effects of an approximation technique be detected and corrected?

2. **Reproducibility:** How easily can the results of an approximation technique be reproduced for testing?

3. **Control:** How much confidence over the error magnitude does an approximation technique provide?

In this section, we present examples to highlight the importance of these questions and demonstrate how they distinguish techniques that may seem similar at first glance.

Correctability of the Approximation Effects Correctability reflects the cost and complexity of detecting and compensating for approximation errors. The degree of correctability varies widely between techniques. For example, consider two seemingly similar techniques: (1) low supply voltage SRAM [60], which allows for soft errors when accessing data in SRAM; and (2) low refresh DRAM [112], which allows for soft errors in DRAM data cells. For low supply voltage SRAM, errors are introduced when an instruction reads or writes the data. A precise check can thus be invoked on each approximate load and store instruction in order to recover from a faulty operation. On the other hand, for low refresh DRAM, the error can be introduced at any point in the lifetime of the data independent of any instruction's execution. This uncertainty makes error

management more costly and less prompt. Our taxonomy distinguishes these two approaches (Section 1.A.3) in terms of their *architectural visibility*.

Reproducibility of the Approximate Results Reproducibility is the degree to which error can be measured during development and generalized to production. It can be difficult to reason about the error introduced by an approximation technique. We often rely on measurements from test systems to decide whether or not the error is within an acceptable range. For example, code perforation [164] is an approximation technique that omits instructions during execution. In general, its impact on error is the same regardless of the underlying system on which it is executed, so its reproducibility is straightforward. On the other hand, synchronization elision [30] omits calls to synchronization primitives like locks. We can measure the error of synchronization elision on a test system and deem it satisfactory, but we may find that error increases on a different production system. Our taxonomy distinguishes reproducibility between *deterministic* techniques like code perforation and *nondeterministic* techniques like synchronization elision (Section 1.A.3).

Control over the Accuracy–Efficiency Tradeoffs Control reflects how easily a technique can trade accuracy for efficiency gains. All approximate computing techniques enable such a trade-off. However, they fall all along the accuracy–efficiency curve; some favor efficiency while others favor accuracy. Consider a program that performs many floating-point computations. We can approximate this program either via fuzzy function memoization [120] or via fuzzy floating-point instructions [11]. Both techniques seem similar, yet they offer very different error–efficiency trade-offs. Function memoization can elide code regions that are as small as one or two instructions or as large as entire functions, which can lead to arbitrary errors if not tested exhaustively. Fuzzy floating-point instructions, on the other hand, limit efficiency gains due to control overheads but also confine errors to the execution of individual instructions, meaning that traditional techniques such as interval analysis can be used to guarantee control over the error introduced by the technique. To characterize control over errors, our taxonomy distinguishes between techniques based on their *granularity* (Section 1.A.3).

1.A.3 Taxonomy

We guide our taxonomy with the motivation questions detailed in Section 1.A.2—*(1) correctability, (2) reproducibility, (3) control*—and list three orthogonal taxonomy axes that address them: *(1) architectural visibility vs. invisibility, (2) deterministic vs. nondeterministic, (3) coarse-grained vs. fine-grained.* For each taxonomy dimension, we provide a formal definition, examples and discuss practical implications. Table 1.A.1 lists a set of recent approximation techniques we surveyed and classified along these three dimensions. In this table, note that we classify techniques as software or hardware; we do not elaborate on this as a taxonomy axis since it does not inform any interesting new insights or properties.

Correctability: Architecturally Visible vs. Invisible

Definition 1. *Consider a program as a sequence of instructions that operate on data. An approximation technique is* ***architecturally invisible*** *if it can introduce error even when the sequence of instructions is null. Otherwise it is an* ***architecturally visible*** *technique.*

Architecturally visible techniques introduce errors during the execution of a specific instruction, and architecturally invisible techniques introduce errors silently. Naturally, visible errors are simple to detect: they can be traced to a specific moment in time. On the other hand, invisible errors are attributed to a phenomenon that occurs below the architectural stack, e.g., a micro-architectural event, or a physical event occurring at the circuit level. Consequently, architecturally invisible techniques can require expensive error detection and correction mechanisms and are harder to monitor dynamically.

Revisiting the examples in Section 1.A.2, low supply voltage SRAM [60] is architecturally visible. It approximates (via bit upsets) only upon memory operations; thus, detecting and managing error is straightforward. For write upsets, for example, adding a precise check after a write operation can immediately catch (and roll back) any erroneous approximations. On the other hand, low refresh DRAM [112] is architecturally invisible: since it yields bit flips at arbitrary times, a precise check after a write operation cannot draw any conclusions about error. Even if the precise

check passes, an erroneous bit-flip can still occur some time later.

Though errors are invisible, an advantage of architecturally invisible techniques is that they are not on the critical path; thus their latency costs can be made invisible as well. Architecturally visible techniques can introduce run time overheads, whereas invisible approximations can be performed in the background. For example, the Doppelgänger cache [115] is an architecturally invisible technique; it generates approximate values silently upon a microarchitectural event without stalling memory requests.

This taxonomy axis informs trade-offs in error correctability. Architecturally visible techniques benefit from errors which are easier to detect and correct. On the other hand, architecturally invisible techniques benefit from generating approximations off the critical path of program execution.

Reproducibility: Deterministic vs. Nondeterministic

Definition 2. *An approximation technique is* ***deterministic*** *if, given the same initial state, for every input I_j, it yields constant error E_j. An approximation technique is* ***nondeterministic*** *if, given the same initial state, there exists some input I_j for which it yields more than one error value $E_{j0}, ..., E_{jn}$.*

Nondeterministic techniques can pose a challenge for testing and debugging. When developing techniques, the conventional approach is to evaluate error and efficiency on a test system and extrapolate to production systems. This is effective for deterministic techniques since they produce the same approximations regardless of the underlying system; errors are *reproducible*. It is possible for a user to declare any error threshold ϵ and concretely evaluate whether or not it is always satisfied for a given input. However, this is not true for nondeterministic techniques. For a given input, error can only be probabilistically evaluated; ϵ must be accompanied by some probability and confidence.

Nondeterministic techniques have limited reproducibility. Such approximations are possible via exposing analog noise, asynchrony and race conditions to the program. Revisiting the examples in Section 1.A.2, synchronization elision [30] is a nondeterministic technique while code

perforation [164] is deterministic. Whereas perforating computations yields the same output on any system, eliding synchronization primitives exposes race conditions. This increases the number of possible outputs and limits reproducibility. The amount of error via synchronization elision can vary greatly across systems depending on the amount of thread-level parallelism. Nondeterministic techniques can also expose analog noise. For example, voltage-overscaled ALUs [60] generate approximations by risking exposure to the analog domain. This has low reproducibility; error cannot be concretely evaluated and must be empirically measured. In comparison, precision-scaled ALUs [186] are deterministic. Scaling precision in the digital representation of data yields the same output on any system.

As a trade-off, nondeterministic techniques can generally offer more opportunity for efficiency gains. By exposing the stochastic nature of the physical world, they avoid the expensive digital abstraction tax. For example, voltage-overscaled ALUs significantly improve efficiency by relaxing the safety margins enforced by digital circuitry.

This taxonomy axis informs trade-offs in reproducibility. Deterministic techniques benefit from high reproducibility, simplifying testing and debugging. On the other hand, nondeterministic techniques benefit from more opportunities for approximation that only exist outside the digital domain.

Error Control: Coarse-Grained vs. Fine-Grained

Definition 3. *An approximation technique is* ***coarse-grained*** *if it reduces the data footprint or the number of dynamic instructions in a program. Otherwise, it is* ***fine-grained****.*

Control over the error introduced by a technique depends on the *granularity* at which an approximation technique is employed. Fine-grained techniques lower the cost of executing an instruction or storing a word of data. Coarse-grained techniques replace a set of instructions or a block of data with a more efficient or compact representation.

Coarse-grained techniques offer more opportunity for error–efficiency trade-offs. Revisiting the examples in Section 1.A.2, fuzzy floating-point instructions [178] are fine-grained while fuzzy

function memoization [120] is coarse-grained. Whereas the former improves the efficiency of individual instructions, the latter can improve the efficiency of an entire block or function. The latter, in the most extreme case, can memoize the entire program for the highest efficiency. In terms of storage, fine-grained techniques, such as low refresh DRAM [112], generate approximations in individual bits. Coarse-grained techniques, such as approximate deduplication [115], reduce data footprint. The latter can be more aggressively tuned for efficiency gains, to the point where the entire data footprint is deduplicated into a single data block.

Naturally, the coarser the granularity of a technique, the higher the risk of error. Fine-grained techniques do not remove any data nor instructions. Conversely, coarse-grained techniques risk information loss as more data and more instructions are omitted. In the previous examples, though memoizing an entire program yields highest efficiency, it also yields highest error. Holistically approximating regions of code can disregard rarely-used control-flow paths when not exercised. Neural approximation [61] is an example of a coarse-grained technique that can subsume entire functions, including potentially complex control flow. This coarseness makes testing and analysis challenging.

This taxonomy axis informs trade-offs in error control. Coarse-grained techniques benefit from greater opportunities for aggressive efficiency gains. On the other hand, fine-grained techniques can limit error and are generally better suited for programs where quality constraints are conservative.

1.A.4 Discussion

We highlight the applicability of our proposed taxonomy by suggesting how it can inform future research in approximate computing. We formulate a three-pronged answer that address the questions across layers of the compute stack: *(1) architecture, (2) compilers and runtimes and (3) applications.*

How Can It Inform Architecture Research? Research on new approximation techniques motivates the need for approximation-aware ISAs (A-ISA). Since the days of the IBM System/360,

architects have distinguished between architecture and implementation to guarantee the *forward-compatibility* of their hardware. An A-ISA can express instruction-level error bounds that need to be respected when deployed on current or future hardware. Such an abstraction layer would allow hardware designers to modify the implementation of approximations down the road in a way that remains invisible to the software. We make a distinction between two types of A-ISAs: strict A-ISAs and statistical A-ISAs. Strict A-ISAs are applicable to deterministic fine-grained techniques and provide strict error bounds on the execution of an instruction. Examples of A-ISAs include the Quality-Programmable ISA [186], which provides strict error bounds relative to the maximum output value of the instruction. Statistical A-ISAs, on the other hand, are applicable to nondeterministic fine-grained techniques and provide statistical failure guarantees. Such an ISA would have to include probability bounds as well as confidence bounds.

How Can It Inform Compilers/Runtimes Research? Research on approximation techniques motivates the development of frameworks to make approximations *safe* to use. Such frameworks include new languages, compilers and runtimes. We discuss how each taxonomy can inform the applicability of framework proposals.

Architectural visibility is relevant to frameworks that focus on detecting and recovering from hardware faults. Relax [55], for instance, can only work on top of architecturally visible techniques because errors must be locally correctable [169]. Online monitoring proposals [148] that rely on precise replay are also only applicable to architecturally visible techniques.

Determinism and coarseness are relevant to formulating statically-derived or empirically-observed application-level error bounds. Nondeterministic techniques require statistical methods like probabilistic assertions [157], while deterministic techniques can rely on hard assertions. Fine-grained techniques can inherit from the wealth of tools developed in numerical analysis research [150]. More specifically, deterministic fine-grained techniques have the advantage of providing strict error bounds at an instruction granularity. Thus, they can provide hard worst-case error bounds for many algorithmic patterns, as opposed to empirically derived average-case error bounds. Coarse-grained techniques have seen a wealth of frameworks [14, 17, 152, 154, 114]

that generally rely on empirical error measurements to provide varying levels of error guarantees via quality autotuning.

How Can It Inform Applications Research? Research on new approximation techniques motivates better understanding on the *applicability* of such techniques. Application designers care about (1) whether a technique can be applied to their algorithms, and (2) whether a technique can meet the quality guarantees they wish to enforce.

Coarseness correlates to how general a technique is to algorithmic patterns. Fine-grained techniques are broadly generalizable: any approximate floating-point algorithm can make use of reduced-precision FPUs. Coarse-grained techniques, on the other hand, have to adhere to specific code patterns: neural acceleration only applies to precise-pure regions of code, while loop-perforation applies to loops free of early exits [154].

Determinism and coarseness will both determine the error behavior that the application will see. Nondeterministic techniques generally yield large rarely-occurring errors while deterministic techniques yield small frequently-occurring errors. Nondeterministic techniques would generally not be used in mission-critical systems. The magnitude of an error is generally better controlled on deterministic fined-grained techniques as opposed to deterministic coarse-grained techniques.

1.A.5 Conclusion

A wealth of approximate computing techniques has been proposed in architecture, circuits, languages, and compilers research. We present a taxonomy that categorizes approximate computing techniques based their most salient properties: visibility, determinism, and coarseness, to better inform cross-stack research in architecture, tools, and applications.

1.B Background: An Survey of Domain Specialized Deep Learning Accelerators

Over the last few years years, comprehensive deep learning accelerator designs have been proposed in academia [42, 110, 56, 41, 77, 90]. These domain-specific architectures present a *programming* challenge to traditional compiler frameworks since they expose unconventional hardware intrinsics. For instance, compilers built to generate scalar code are naturally ill-equipped to uncover tensor operations as they would require higher-level abstractions.

Related Work	Tile Type	Compute Tiles	MAC/ Tile	Data Types	Parameter Storage	Activation Storage	Accumulator Storage	DRAM Latency Hiding	Other Operators	Applications
DaDianNao	Vector Dot Product	16	256	int16/int32	distributed 32MB (eDRAM)	4MB (eDRAM)	distributed 512kB	n/a	activation, pooling, normalization	MLP, CNN
PuDianNao	Vector Dot Product	64	16	fp16/fp32	16kB	8kB	8kB	ping pong buffering	thresholding, general math, k-sorting	MLP, general machine learning
ShiDianNao	2D mesh, output-stationary	1	64	int16	128kB	64kB	64kB	n/a	activation, pooling, normalization	CNN, MLP
Eyeriss	2D mesh, row-stationary	1	168	int16	108kB of unified global cache + 84kB of distributed register files			global buffer prefetch	pooling	CNN, MLP
EIE	Sparse Vector Dot Product	1	64	int4/int16	distributed 10MB sparse representation	distributed 128kB	distributed 2kB	n/a	ReLU	MLP
TPU	2D Systolic Matrix-Matrix Multiplication	1	64k	int8/int32	256k FIFO	24MB	4MB	4-stage pipeline + explicit synchronization	activation, pooling, normalization	MLP, LSTM, CNN

Figure 1.B.1: Salient characteristics of recently-proposed deep learning accelerators.

We summarize prominent works in Figure 1.B.1 in terms of their most salient features: dense tensor computation fabric, data type specialization, memory subsystem specialization, latency hiding mechanisms and general purpose-ness. These salient features explain why programming domain-specialized accelerators in the context of deep learning requires re-thinking the software stack.

Dense Tensor Computation Fabric. While most accelerators reviewed under Table 1.B.1 appear to support different basic operations, ranging from an array of vector dot product tiles [42, 110], to spatially programmed 2D meshes of processing elements [56, 41, 90, 77], they all perform matrix-matrix multiplication from a programming abstraction perspective.

Most of the complex low-level processing element control and data movement orchestration can be hidden away under a CISC-like ISA abstraction to present the programmer with high-level dense linear algebra intrinsics, as done in the Google TPU [90] or in Cambricon ISA work [111].

Data Type Specialization. Deep learning is amenable to quantization, particularly for inference workloads which many of the accelerators in Table 1.B.1 are optimized for. Most accelerators use two levels of precision: low-precision for map-like operators (e.g., multiplication in a dot product), and high-precision for reduce-like operators (e.g., addition in a dot product). As a result, it is common to store neural network weights and activations at lower precision settings, while aggregated data is stored at a higher precision. Concretely, these two precision levels are `int8/int32` in the case of the TPU [90], i.e. 8-bit multiplications, and 32-bit accumulations.

On-Chip Memory Subsystem Specialization. While on-chip memory subsystems can seemingly vary drastically between proposed architectures, they generally provide separate and disjoint storage structures for activations, parameters and accumulator values (as opposed to a unified register file or data cache in the case of CPUs, and GPUs). This allows the memory subsystem to be finely tuned to the on-chip bandwidth (number of SRAM banks and read ports) and the overall capacity requirements for each data type that the dense tensor core processes. Dadiannao [42] and EIE [77] are architectures that are tuned to evaluate Multi-Layer Perceptrons (MLPs). Consequently they provide vast amounts of on-chip storage to store parameters (i.e. kernel weights) to minimize off-chip DRAM accesses. Other designs like the TPU offer practically no on-chip parameter storage due to the assumption that the parameters of large models cannot realistically fit on-chip. This design assumption is typical of accelerators optimized for 2D convolution layers, where the memory requirements for activations vastly outweighs those of kernel weights.

DRAM Latency Hiding. Several designs in Table 1.B.1 assume that all parameters or interme-

diate activations must fit on chip ([42, 56, 77]). While remaining within an on-chip storage budget can maximize efficiency, it yields constraints that are not realistic to satisfy as neural networks are getting deeper, and the number of parameters keeps growing. In most cases, on-chip storage limitations will inevitably lead to spilling of either activations, or parameters. In order to maintain high utilization of computation resources when memory spilling occurs, latency hiding mechanisms is required. Unsurprisingly, a variety of latency hiding techniques exists in the deep learning accelerator design literature. PuDianNao [110] performs streaming of activations and weights from DRAM into ping-pong buffers which are partitioned into a read bank and write bank that alternate during overlapping execution phases. Eyeriss [41] relies on a classical data prefetching approach into bring data its global buffer in a timely fashion. The Google TPU [90] relies on a high-level 4-stage pipeline of load, compute, activation, and store stages to hide all non-compute latencies. For that reason, the TPU pipeline relies on explicit dependences [168] to insert stalls if necessary.

General-Purposeness. Deep learning accelerators also incorporate support for other operators beyond dense linear algebra, including activation, pooling, and normalization. As new network models are proposed and new operators are introduced, adding ”general purposeness” with an ALU [56] or function interpolation module [110] can help future-proof deep learning accelerator designs to some extent.

In order to foster the development of software stacks adapted to deep learning domain-specialized accelerators, we propose in Chapter 4 a generic accelerator designs that can be seen by a compiler as a superset of the designs proposed in [42, 110, 56, 41, 77, 90]. The VTA design can for instance be parameterized to implement dense linear operations such as vector dot product [110], vector-matrix multiply [130], and matrix-matrix multiply[90].

In addition VTA design offers a customizable on-chip memory subsystem allowing for the implementation of the different memory organizations found in designs like the TPU [90] where activation storage dominates, or DaDianNao [42] where parameter storage dominates. Finally in terms of latency hiding, VTA exposes a software-driven dependency tracking management

similar to the one found in TPU [90]. Supporting latency hiding via low-level dependence tracking mechanisms requires all layers of the stack to work together.

Chapter 2

SNNAP: APPROXIMATELY MAPPING DIVERSE REGIONS OF CODE TO A SINGLE FPGA-BASED SUBSTRATE VIA NEURAL ACCELERATION

"All problems in computer science can be solved by another level of indirection."

– David Wheeler

Published As: Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze and Mark Oskin, *QAPPA: A Framework for Navigating Quality-Energy Tradeoffs with Arbitrary Quantization*, IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015.

Abstract: *Many applications that can take advantage of accelerators are amenable to approximate execution. Past work has shown that neural acceleration is a viable way to accelerate approximate code [61]. In light of the growing availability of on-chip field-programmable gate arrays (FPGAs), this chapter explores neural acceleration on off-the-shelf programmable SoCs. We describe the design and implementation of SNNAP, a flexible FPGA-based neural accelerator for approximate programs. SNNAP is designed to work with a compiler workflow that configures the neural network's topology and weights instead of the programmable logic of the FPGA itself. This approach enables effective use of neural acceleration in commercially available devices and accelerates different applications without costly FPGA reconfigurations. No hardware expertise is required to accelerate software with SNNAP, so the effort required can be substantially lower than custom hardware design for an FPGA fabric and possibly even lower than current "C-to-gates" high-level synthesis (HLS) tools. Our measurements on a Xilinx Zynq FPGA show that SNNAP yields a geometric mean of 3.8× speedup (as high as 38.1×) and 2.8× energy savings (as high as 28×) with less than 10% quality*

loss across all applications but one. We also compare SNNAP with designs generated by commercial HLS tools and show that SNNAP has similar performance overall, with better resource-normalized throughput on 4 out of 7 benchmarks.

2.1 Introduction

In light of diminishing returns from technology improvements on performance and energy efficiency [58, 134], researchers are exploring new avenues in computer architecture. There are at least two clear trends emerging. One is the use of *specialized logic* in the form of accelerators [188, 189, 70, 75] or programmable logic [138, 137, 44], and another is *approximate computing*, which exploits applications' tolerance to quality degradations [155, 186, 60, 153]. Specialization leads to better efficiency by trading off flexibility for leaner logic and hardware resources, while approximate computing trades accuracy to enable novel optimizations.

The confluence of these two trends leads to additional opportunities to improve efficiency. One example is *neural acceleration*, which trains neural networks to mimic regions of approximate code [61, 171]. Once the neural network is trained, the system no longer executes the original code and instead invokes the neural network model on a *neural processing unit* (NPU) accelerator. This leads to better efficiency because neural networks are amenable to efficient hardware implementations [136, 59, 89, 159]. Prior work on neural acceleration, however, has assumed that the NPU is implemented in fully custom logic tightly integrated with the host processor pipeline [61, 171]. While modifying the CPU core to integrate the NPU yields significant performance and efficiency gains, it prevents near-term adoption and increases design cost/complexity. This chapter explores the performance opportunity of NPU acceleration implemented on off-the-shelf *field-programmable gate arrays* (FPGAs) and without tight NPU–core integration, avoiding changes to the processor ISA and microarchitecture.

On-chip FPGAs have the potential to unlock order-of-magnitude energy efficiency gains while retaining some of the flexibility of general-purpose hardware [166]. Commercial parts that incorporate general purpose cores with programmable logic are beginning to appear [195, 10, 85]. In light of this trend, this chapter explores an opportunity to accelerate approximate programs via

an NPU implemented in programmable logic.

Our design, called SNNAP (**s**ystolic **n**eural **n**etwork **a**ccelerator in **p**rogrammable logic), is designed to work with a compiler workflow that automatically configures the neural network's topology and weights instead of the programmable logic itself. SNNAP's implementation on off-the-shelf programmable logic has several benefits. First, it enables effective use of neural acceleration in commercially available devices. Second, since NPUs can accelerate a wide range of computations, SNNAP can target many different applications without costly FPGA reconfigurations. Finally, the expertise required to use SNNAP can be substantially lower than designing custom FPGA configurations. In our evaluation, we find that the programmer effort can even be lower than for commercially available "C-to-gates" high-level synthesis tools [194, 9].

We implement and measure SNNAP on the Zynq [195], a state-of-the-art programmable system-on-a-chip (PSoC). We identify two core challenges: communication latency between the core and the programmable logic unit, and the difference in processing speeds between the programmable logic and the core. We address those challenges with a new throughput-oriented interface and programming model, and a parallel architecture based on scalable FPGA-optimized systolic arrays. To ground our comparison, we compare benchmarks accelerated with SNNAP to custom designs of the same accelerated code generated by a high-level synthesis tool. Our HLS study shows that current commercial tools still require significant effort and hardware design experience. Across a suite of approximate benchmarks, we observe an average speedup of 3.8×, ranging from 1.3× to 38.1×, and an average energy savings of 2.8×.

2.2 Programming

There are two basic ways to use SNNAP. The first is to use a high-level, compiler-assisted mechanism that transforms regions of approximate code to offload them to SNNAP. This automated *neural acceleration* approach requires low programmer effort and is appropriate for bringing efficiency to existing code. The second is to directly use SNNAP's low-level, explicit interface that offers fine-grained control for expert programmers while still abstracting away hardware details. We describe both interfaces below.

2.2.1 Compiler-Assisted Neural Acceleration

Approximate applications can take advantage of SNNAP automatically using the *neural algorithmic transformation* [61]. This technique uses a compiler to replace error-tolerant subcomputations in a larger application with neural network invocations.

The process begins with an approximation-aware programming language in which code or data can be marked as approximable. Language options include Relax's code regions [55], EnerJ's type qualifiers [155], Rely's variable and operator annotations [31], or simple function annotations. In any case, the programmer's job is to express where approximation is allowed. The neural-acceleration compiler trains neural networks for the indicated regions of approximate code using test inputs. The compiler then replaces the original code with an invocation of the learned neural network. Lastly, quality can be monitored at run-time using application-specific quality metrics such as Light-Weight Checks [72].

As an example, consider a program that filters each pixel in an image. The annotated code might resemble:

```
APPROXFUNC double filter(double pixel);
...
for (int x = 0; x ¡ width; ++x)
  for (int y = 0; y ¡ height; ++y)
    outimage[x][y] = filter(inimage[x][y]);
```

where the programmer uses a function attribute to mark `filter()` as approximate.

The neural-acceleration compiler replaces the `filter()` call with instructions that instead invoke SNNAP with the argument `inimage[x][y]`. The compiler also adds setup code early in the program to set up the neural network for invocation.

2.2.2 Low-Level Interface

While automatic transformation represents the highest-level interface to SNNAP, it is built on a lower-level interface that acts both as a compiler target and as an API for expert programmers. This section details the instruction-level interface to SNNAP and a low-level library layered on

top of it that makes its asynchrony explicit.

Unlike a low-latency circuit that can be tightly integrated with a processor pipeline, FPGA-based accelerators cannot afford to block program execution to compute each individual input. Instead, we architect SNNAP to operate efficiently on batches of inputs. The software groups together invocations of the neural network and ships them all simultaneously to the FPGA for pipelined processing. In this sense, SNNAP behaves as a *throughput-oriented* accelerator: it is most effective when the program keeps it busy with a large number of invocations rather than when each individual invocation must complete quickly.

Instruction-level interface. At the lowest level, the program invokes SNNAP by enqueueing batches of inputs, invoking the accelerator, and receiving a notification when the batch is complete. Specifically, the program writes all the inputs into a buffer in memory and uses the ARMv7 `SEV` (send event) instruction to notify SNNAP. The accelerator then reads the inputs from the CPU's cache via a cache-coherent interface and processes them, placing the output into another buffer. Meanwhile, the program issues an ARM `WFE` (wait for event) instruction to sleep until the neural-network processing is done and then reads the outputs.

Low-Level asynchronous API. SNNAP's accompanying software library offers a low-level API that abstracts away the details of the hardware-level interface. The library provides an ordered, asynchronous API that hides the size of SNNAP's input and output buffers. This interface is useful both as a target for neural-acceleration compilers and for expert programmers who want convenient, low-level control over SNNAP.

The SNNAP C library uses a callback function to consume each output of the accelerator when it is ready. For example, a simple callback that writes a single floating-point output to an array can be written:

```
static int index = 0;
static float output[...];
void cbk(const void *data) –
  output[index] = *(float *)data; ++index;
```

Then, to invoke the accelerator, the program configures the library, sends inputs repeatedly, and then waits until all invocations are finished with a barrier. For example:

```
snnapstreamt stream = snnapstreamnew(
  sizeof(float), sizeof(float), cbk);
for (int i = 0; i ¡ max; ++i) –
  snnapstreamput(stream, input);

snnapstreambarrier(stream);
```

The snnapstreamnew call creates a stream configuration describing the size the neural network's invocation in bytes, the size of each corresponding output, and the callback function. Then, snnapstreamput copies an input value from a **void*** pointer into SNNAP's memory-mapped input buffer. Inside the put call, the library also consumes any outputs available in SNNAP's output buffer and invokes the callback function if necessary. Finally, snnapstreambarrier waits until all invocations are finished.

This asynchronous style enables the SNNAP runtime library to coalesce batches of inputs without exposing buffer management to the programmer or the compiler. The underlying SNNAP configuration can be customized with different buffer sizes without requiring changes to the code. In more sophisticated programs, this style also allows the program to transparently overlap SNNAP invocations with CPU code between snnapstreamsend calls.

This low-level, asynchronous interface is suitable for expert programmers who want to exert fine-grained control over how the program communicates with SNNAP. It is also appropriate for situations when the program explicitly uses a neural network model for a traditional purpose, such as image classification or handwriting recognition, where the SNNAP C library acts as a replacement for a software neural network library. In most cases, however, programmers need not directly interact with the library and can instead rely on automatic neural acceleration.

2.3 Architecture Design for SNNAP

This work is built upon an emerging class of heterogeneous computing devices called Programmable System-on-Chips (PSoCs). These devices combine a set of hard processor cores with programmable logic on the same die. Compared to conventional FPGAs, this integration provides a higher-bandwidth and lower-latency interface between the main CPU and the programmable logic. However, the latency is still higher than in previous proposals for neural acceleration [61, 171]. Our objective is to take advantage of the processor–logic integration with efficient invocations, latency mitigation, and low resource utilization. We focus on these challenges:

- The NPU must use FPGA resources efficiently to minimize its energy consumption.
- The NPU must support low-latency invocations to provide benefit to code with small approximate regions.
- To mitigate communication latency, the NPU must be able to efficiently process batches of invocations.
- The NPU and the processor must operate independently to enable the processor to hibernate and conserve energy while the accelerator is active.
- Different applications require different neural network topologies. Thus, the NPU must be reconfigurable to support a wide range of applications without the need for reprogramming the entire FPGA or redesigning the accelerator.

The rest of this section provides an overview of the SNNAP NPU and its interface with the processor.

2.3.1 SNNAP Design Overview

SNNAP evaluates *multi-layer perceptron* (MLP) neural networks. MLPs are a widely-used class of neural networks that have been used in previous work on neural acceleration [61, 171]. An MLP

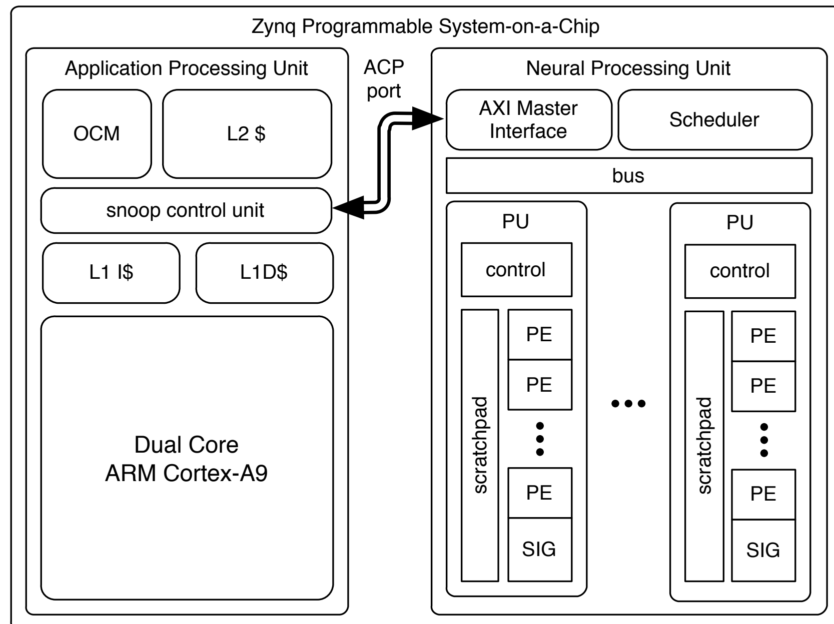


Figure 2.1: SNNAP system diagram. Each Processing Unit (PU) contains a chain of Processing Elements (PE) feeding into a sigmoid unit (SIG).

is a layered directed graph where the nodes are computational elements called *neurons*. Each neuron computes the weighted sum of its inputs and applies a nonlinear function, known as the *activation function*, to the sum—often a sigmoid function. The complexity of a neural network is reflected in its *topology:* larger topologies can fit more complex functions while smaller topologies are faster to evaluate.

The SNNAP design is based on *systolic arrays*. Systolic arrays excel at exploiting the regular data-parallelism found in neural networks [45] and are amenable to efficient implementation on modern FPGAs. Most of the systolic array's highly pipelined computational datapath can be contained within the dedicated multiply–add units found in FPGAs know as *Digital Signal Processing* (DSP) slices. We leverage these resources to realize an efficient pipelined systolic array for SNNAP in the programmable logic.

Our design, shown in Figure 2.1, consists of a cluster of *Processing Units* (PUs) connected

through a bus. Each PU is composed of a control block, a chain of *Processing Elements* (PEs), and a sigmoid unit, denoted by the SIG block. The PEs form a one-dimensional systolic array that feeds into the sigmoid unit. When evaluating a layer of a neural network, PEs read the neuron weights from a local scratchpad memory where temporary results can also be stored. The sigmoid unit implements a nonlinear neuron-activation function using a lookup table. The PU control block contains a configurable sequencer that orchestrates communication between the PEs and the sigmoid unit. The PUs operate independently, so different PUs can be individually programmed to parallelize the invocations of a single neural network or to evaluate many different neural networks. Section 2.4 details SNNAP's hardware design.

2.3.2 CPU–SNNAP Interface

We design the CPU–SNNAP interface to allow dynamic reconfiguration, minimize communication latency, and provide high-bandwidth coherent data transfers. To this end, we design a wrapper that composes three different interfaces on the target programmable SoC (PSoC).

We implement SNNAP on a commercially available PSoC: the Xilinx Zynq-7020 on the ZC702 evaluation platform [195]. The Zynq includes a Dual Core ARM Cortex-A9, an FPGA fabric, a DRAM controller, and a 256 KB scratchpad SRAM referred to as the on-chip memory (OCM). While PSoCs like the Zynq hold the promise of low-latency, high-bandwidth communication between the CPU and FPGA, the reality is more complicated. Zynq provides multiple communication mechanisms with different bandwidths and latencies that can surpass 100 CPU cycles. This latency can in some cases dominate the time it takes to evaluate a neural network. SNNAP's interface must therefore mitigate this communication cost with a modular design that permits throughput-oriented, asynchronous neural-network invocations while keeping latency as low as possible.

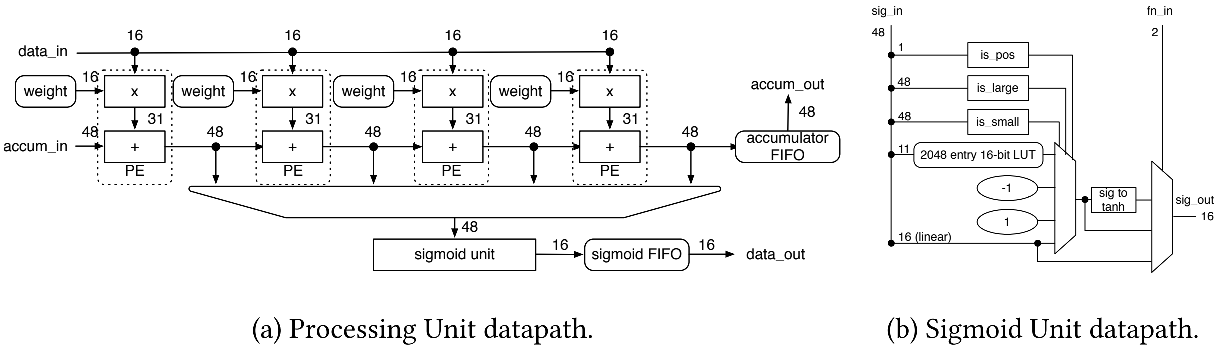
We compose a communication interface based on three available communication mechanisms on the Zynq PSoC [197]. First, when the program starts, it configures SNNAP using the medium-throughput General Purpose I/Os (GPIOs) interface. Then, to use SNNAP during execution, the program sends inputs using the high-throughput ARM Accelerator Coherency Port (ACP). The

processor then uses the ARMv7 SEV/WFE signaling instructions to invoke SNNAP and enter sleep mode. The accelerator writes outputs back to the processor's cache via the ACP interface and, when finished, signals the processor to wake up. We detail each of these components below.

Configuration via General Purpose I/Os (GPIOs). The ARM interconnect includes two 32-bit Advanced Extensible Interface (AXI) general-purpose bus interfaces to the programmable logic, which can be used to implement memory-mapped registers or support DMA transfers. These interfaces are easy to use and are relatively low-latency (114 CPU cycle roundtrip latency) but can only support moderate bandwidth. We use these GPIO interfaces to configure SNNAP after it is synthesized on the programmable logic. The program sends a configuration to SNNAP without reprogramming the FPGA. A configuration consists of a schedule derived from the neural network topology and a set of weights derived from prior neural network training. SNNAP exposes the configuration storage to the compiler as a set of memory-mapped registers. To configure SNNAP, the software checks that the accelerator is idle and writes the schedule, weights, and parameters to memory-mapped SRAM tables in the FPGA known as block RAMs.

Sending data via the Accelerator Coherency Port. The FPGA can access the ARM on-chip memory system through the 64-bit Accelerator Coherency Port (ACP) AXI-slave interface. This port allows the FPGA to send read and write requests directly to the processors' Snoop Control Unit to access the processor caches thus bypassing explicit cache flushes required by traditional DMA interfaces. The ACP interface is the best available option for transferring batches of input/output vectors to and from SNNAP. SNNAP includes a custom AXI master for the ACP interface, reducing round-trip communication latency down to 93 CPU cycles. Batching invocations help amortize this latency in practice.

Invocation via synchronization instructions. The ARM and the FPGA are connected by two unidirectional event lines `eventi` and `evento` for synchronization. The ARMv7 ISA contains two instructions to access these synchronization signals, SEV and WFE. The SEV instruction



(a) Processing Unit datapath.

(b) Sigmoid Unit datapath.

Figure 2.1: Detailed PU datapath. PEs are implemented on multiply–add logic and produce a stream of weighted sums from an input stream. The sums are sent to a sigmoid unit that approximates the activation function.

causes the `evento` signal in the FPGA fabric to toggle. The `WFE` instruction causes the processor to enter the low-power hibernation state until the FPGA toggles the `eventi` signal. These operations have significantly lower latency (5 CPU cycles) than any of the other two communication mechanisms between the processor and the programmable logic.

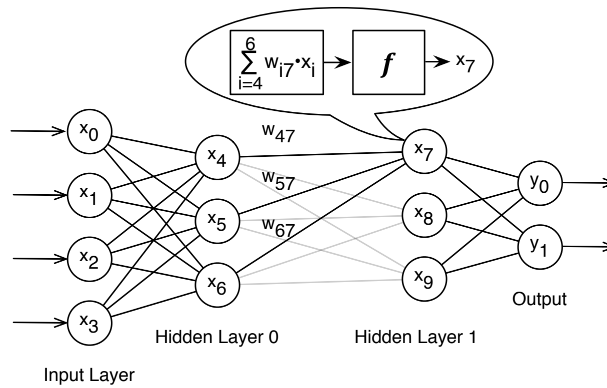
We use these instructions to invoke SNNAP and synchronize its execution with the processor. To invoke SNNAP, the CPU writes input vectors to a buffer in its cache. It signals the accelerator to start computation using `SEV` and enters hibernation with `WFE`. When SNNAP finishes writing outputs to the cache, it signals the processor to wake up and continues execution.

2.4 Hardware Design for SNNAP

This section describes SNNAP's systolic-array design and its FPGA implementation.

2.4.1 Multi-Layer Perceptrons With Systolic Arrays

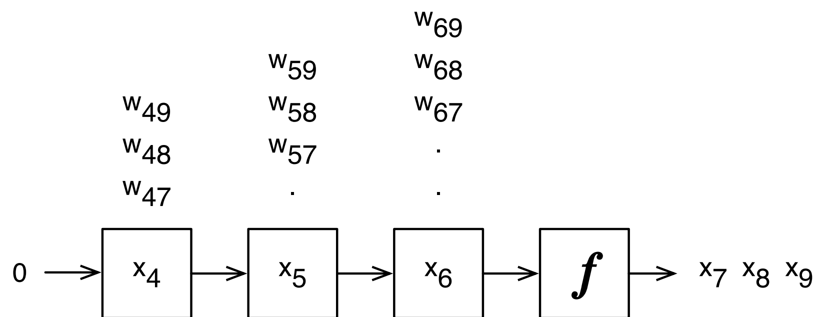
MLPs consist of a collection of neurons organized into layers. Figure 2.2a depicts an MLP with four layers: the input layer, the output layer, and two *hidden layers*. The computation of one of the neurons in the second hidden layer is highlighted: the neuron computes the weighted sum of



(a) An multilayer perceptron neural network.

$$\left(\begin{pmatrix} w_{47} & w_{57} & w_{67} \\ w_{48} & w_{58} & w_{68} \\ w_{49} & w_{59} & w_{69} \end{pmatrix} \cdot \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix}\right) = \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

(b) Matrix representation of hidden layer evaluation.



(c) Systolic algorithm on one-dimensional systolic array.

Figure 2.2: Implementing multi-layer perceptron neural networks with systolic arrays.

the values of its source neurons and applies the activation function f to the result. The resulting neuron output is then sent to the next layer.

The evaluation of an MLP neural network consists of a series of matrix–vector multiplications interleaved with non-linear activation functions. Figure 2.2b shows this approach applied to the hidden layers of Figure 2.2a. We can schedule a systolic algorithm for computing this matrix–vector multiplication onto a 1-dimensional systolic array as shown in Figure 2.2c. When computing a layer, the vector elements x_i are loaded into each cell in the array while the matrix elements elements w_{ji} trickle in. Each cell performs a multiplication $x_i \cdot w_{ji}$, adds it to the sum of products produced by the upstream cell to its left, and sends the result to the downstream cell to its right. The output vector produced by the systolic array finally goes through an activation function cell, completing the layer computation.

Systolic arrays can be efficiently implemented using the hard DSP slices that are common in modern FPGAs. Our PSoC incorporates 220 DSP slices in its programmable logic [197]. DSP slices offer pipelined fixed-point multiply-and-add functionality and a hard-wired data bus for fast aggregation of partial sums on a single column of DSP slices. As a result, a one-dimensional fixed-point systolic array can be contained entirely in a single hard logic unit to provide high performance at low power [196].

2.4.2 Processing Unit Datapath

Processing Units (PUs) are replicated processing cores in SNNAP's design. A PU comprises a chain of *Processing Elements* (PEs), a sigmoid unit, and local memories including block-RAMs (BRAMs) and FIFOs that store weights and temporary results. A sequencer orchestrates communication between the PEs, the sigmoid unit, local memories, and the bus that connects each PU to the NPU's memory interface.

The PEs that compose PUs map directly to a systolic array cell as in Figure 2.1a. A PE consists of a multiply-and-add module implemented on a DSP slice. The inputs to the neural network are loaded every cycle via the input bus into each PE following the systolic algorithm. Weights, on the other hand, are statically partitioned among the PEs in local BRAMs.

The architecture can support an arbitrary number of PEs. Our evaluation discusses the optimal number of PEs per PU by discussing throughput-resources trade-offs.

Sigmoid unit. The sigmoid unit applies the neural network's activation function to outputs from the PE chain. The design, depicted in Figure 2.1b, is a 3-stage pipeline comprising a lookup-table and some logic for special cases. We use a $y = x$ linear approximation for small input values and $y = \pm 1$ for very large inputs. Combined with a 2048-entry LUT, the design yields at most 0.01% normalized RMSE.

SNNAP supports three commonly-used activation functions: a sigmoid function $S(x) = \frac{k}{1+e^{-x}}$, a hyperbolic tangent $S(x) = k \cdot \tanh(x)$, and a linear activation function $S(x) = k \cdot x$, where k is a steepness parameter. Microcode instructions (see Section 2.4.3) dictate the activation function for each layer.

Flexible NN topology. The NPU must map an arbitrary number of neurons to a fixed number of PEs. Consider a layer with n input neurons, m output neurons and let p be the number of PEs in a PU. Without any constraints, we would schedule the layer on n PEs, each of which would perform m multiplications. However, p does not equal n in general. When $n < p$, there are excess resources and $p - n$ PEs remain idle. If $n > p$, we time-multiplex the computation onto the p PEs by storing temporary sums in an *accumulator FIFO*. Section 2.4.3 details the process of mapping layers onto PEs.

The partial sums of the first p input neurons are computed and stored in the accumulator FIFO; and later retrieved and added to the next p partial sums before being stored back into the accumulator FIFO etc. This process repeats until the last input neuron is mapped to a PE; at that point the completed sum is sent to the sigmoid unit.

A similar time-multiplexing process is performed to evaluate neural networks with many hidden layers. We buffer sigmoid unit outputs in a *sigmoid FIFO* until the evaluation of the current layer is complete; then they can be used as inputs to the next layer. When evaluating the final layer in a neural network, the outputs coming from the sigmoid unit are sent directly to the

memory interface and written to the CPU's memory.

The BRAM space allocated to the sigmoid and accumulator FIFOs limit the maximum layer width of the neural networks that SNNAP can execute.

Numeric representation. SNNAP uses a 16-bit signed fixed-point numeric representation with 7 fraction bits internally. This representation fits within the 18×25 DSP slice multiplier blocks. The DSP slices also include a 48-bit fixed-point adder that helps avoid overflows on long summation chains. We limit the dynamic range of neuron weights during training to match this representation.

The 16-bit width also makes efficient use of the ARM core's byte-oriented memory interface for applications that can provide fixed-point inputs directly. For floating-point applications, SNNAP converts the representation at its inputs and outputs.

We found that one sigmoid unit was sufficient for our NPU design. Out of the 6 application benchmarks we used in our evaluation, only the FFT benchmark schedule experienced contention for the Sigmoid Unit from two PEs, thus introducing a one-cycle bubble in the schedule and increasing the FFT neural network computation latency by 4.8%.

2.4.3 Processing Unit Control

Microcode. SNNAP executes a static schedule derived from the topology of a neural network. This inexpensive scheduling process is performed on the host machine before it configures the accelerator. The schedule is represented as microcode stored in a local BRAM.

Each microcode line describes a command to be executed by a PE. We distinguish architectural PEs from physical PEs since there are typically more inputs to each layer in a neural network than there are physical PEs in a PU (i.e., $n > p$). Decoupling the architectural PEs from physical PEs allow us to support larger neural networks and makes the same micro-code executable on PUs of different PE length.

Each instruction comprises four fields:

1. `ID`: the ID of the architectural PE executing the command.

2. MADD: the number of multiply–add operations that must execute to compute a layer.

3. SRC: input source selector; either the input FIFO or the sigmoid FIFO.

4. DST: the destination of the output data; either the next PE or the sigmoid unit. In the latter case, the field also encodes (1) the type of activation function used for that layer, and (2) whether the layer is the output layer.

Sequencer. The sequencer is a finite-state machine that processes microcoded instructions to orchestrate data movement between PEs, input and output queues, and the sigmoid unit within each PU. Each instruction is translated by the sequencer into commands that get forwarded to a physical PE along with the corresponding input data. The mapping from architectural PE (as described by the microcode instruction) to the physical PE (the actual hardware resource) is done by the sequencer dynamically based on resource availability and locality.

Algorithm 1 shows this scheduling process. The sequencer only needs to wait on the first physical PE, PE_0. For fully connected neural networks, PEs receive the same MADD count in each layer. Consequently, if a given PE is ready to do work at cycle t, the next downstream PE is guaranteed to be ready do work at cycle $t + 1$.

Scheduler optimizations. During microcode generation, we use a simple optimization that improves utilization by minimizing pipeline stalls due to data dependencies. The technique improves overall throughput for a series of invocations at the cost of increasing the latency of a single invocation.

Consider a simple PU structure with two PEs and a one-stage sigmoid unit when evaluating a 2–2–1 neural network topology. Table 2.1 presents two schedules that map this neural network topology onto the available resources in the pipeline diagram. Each schedule tells us which task each functional unit is working on at any point in time. For instance, when PE_1 is working on x_2, it is multiplying $x_1 \times w_{12}$ and adding it to the partial sum $x_0 \times w_{02}$ computed by PE_0.

Algorithm 1: Sequencer algorithm.

```
1  while inputs are ready do
2      if a new layer is being processed then
3          idx_PE ← 0
4      end
5      foreach microcode command do
6          if idx_PE == 0 then
7              while PE[idx_PE] is busy do
8                  wait
9              end
10         end
11         map architectural ID to physical PE[idx_PE]
12         dequeue input or sigmoid FIFO based on SRC
13         send command with DST and MADD to PE[idx_PE]
14         idx_PE ← (idx_PE + 1) mod len(PE)
15     end
16 end
```

Executing one neural network invocation at a time results in a inefficient schedule as illustrated by the *naive schedule* in Table 2.1. The pipeline stalls here result from (1) dependencies between neural network layers and (2) contention over the PU input bus. Data dependencies occur when a PE is ready to compute the next layer of a neural network, but has to wait for the sigmoid unit to produce the inputs to that next layer.

We eliminate these stalls by interleaving the computation of layers from multiple neural network invocations as shown in the *efficient schedule* in Table 2.1. Pipeline stalls due to data dependencies can be eliminated as long as there are enough neural network invocations waiting to be executed. SNNAP's throughput-oriented workloads tend to provide enough invocations to

Schedule	FU	0	1	2	3	4	5	6	7
Naive	PE_0	$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$		$x_2^{(1)}$	$x_3^{(1)}$
	PE_1		$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$		$x_2^{(1)}$
	SIG			$x_2^{(0)}$	$x_3^{(0)}$			$x_4^{(0)}$	
Efficient	PE_0	$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$		$x_2^{(2)}$
	PE_1		$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$	
	SIG			$x_2^{(0)}$	$x_3^{(0)}$	$x_2^{(1)}$	$x_3^{(1)}$	$x_4^{(0)}$	$x_4^{(1)}$

Table 2.1: Static PU scheduling of a 2–2–1 neural network. The naive schedule introduces pipeline stalls due to data dependencies. Evaluating two neural network invocations simultaneously by interlacing the layer evaluations can eliminate those stalls.

justify this optimization.

2.5 Evaluation

We implemented SNNAP on an off-the-shelf programmable SoC. In this section, we evaluate our implementation to assess its performance and energy benefits over software execution, to characterize the design's behavior, and to compare against a high-level synthesis (HLS) tool. The HLS comparison provides a reference point for SNNAP's performance, efficiency, and programmer effort requirements.

2.5.1 Experimental setup

Applications. Table 2.2 shows the applications measured in this evaluation, which are the benchmarks used by Esmaeilzadeh et al. [61] along with `blackscholes` from the PARSEC benchmark suite [23]. We offload one approximate region from each application to SNNAP. These regions are mapped to neural network topologies used in previous work [61, 39]. The table shows a hypothetical "Amdahl speedup limit" computed by subtracting the measured runtime of the

Application	Description	Error Metric	Topology	Config. Size	Error	Amdahl Speedup (×)
`blackscholes`	option pricing	mean error	6–20–1	6308 bits	7.83%	> 100
`fft`	radix-2 Cooley-Tukey FFT	mean error	1–4–4–2	1615b	0.1%	3.92
`inversek2j`	inverse kinematics for 2-joint arm	mean error	2–8–2	882b	1.32%	> 100
`jmeint`	triangle intersection detection	miss rate	18–32–8–2	15608b	20.47%	99.65
`jpeg`	lossy image compression	image diff	64–16–4	21264b	1.93%	2.23
`kmeans`	k-means clustering	image diff	6–8–4–1	3860b	2.55%	1.47
`sobel`	edge detection	image diff	9–8–1	3818b	8.57%	15.65

Table 2.2: Applications used in our evaluation. The "NN Topology" column shows the number of neurons in each MLP layer. The "NN Config. Size" column reflects the size of the synaptic weights and microcode in bits. "Amdahl Speedup" is the hypothetical speedup for a system where the SNNAP invocation is instantaneous.

kernel to be accelerated from the overall benchmark runtime.

Target platform. We evaluate the performance, power and energy efficiency of SNNAP running against software on the ZYNQ ZC702 evaluation platform described in Table 2.3. The ZYNQ processor integrates a mobile-grade ARM Cortex-A9 and a Xilinx FPGA fabric on a single TSMC 28nm die.

We compiled our benchmarks using GCC 4.7.2 at its `-O3` optimization level. We ran the benchmarks directly on the bare metal processor.

Monitoring performance and power. To count CPU cycles, we use the event counters in the ARM's architectural performance monitoring unit and performance counters implemented in the FPGA. The ZYNQ ZC702 platform uses Texas Instruments UCD9240 power supply controllers, which allow us to measure voltage and current on each of the board's power planes. This allows us to track power usage for the different sub-systems (e.g., CPU, FPGA, DRAM).

Zynq SoC	
Technology	28nm TSMC
Processing	2-core Cortex-A9
FPGA	Artix-7
FPGA Capacity	53KLUTs, 106K Flip-Flops
Peak Freqs	667MHz A9, 167MHz FPGA
DRAM	1GB DDR3-533MHz

Cortex-A9	
L1 Cache Size	32kB I$, 32kB D$
L2 Cache Size	512kB
Scratch-Pad	256kB SRAM
Interface Port	AXI 64-bit ACP
Interface Latency	93 cycles roundtrip

NPU	
Number of PUs	8
Number of PEs	8
Weight Memory	1024×16-bit
Sigmoid LUT	2048×16-bit
Accumulator FIFO	1024×48-bit
Sigmoid FIFO	1024×16-bit
DSP Unit	16×16-bit mul, 48-bit add

Table 2.3: Microarchitectural parameters for the Zynq platform, CPU, FPGA and NPU.

NPU configuration. Our results reflect a SNNAP configuration with 8 PUs, each comprised of 8 PEs. The design runs at 167 MHz, or 1/4 of the CPU's 666MHz frequency. For each benchmark, we configure all the PUs to execute the same neural network workload.

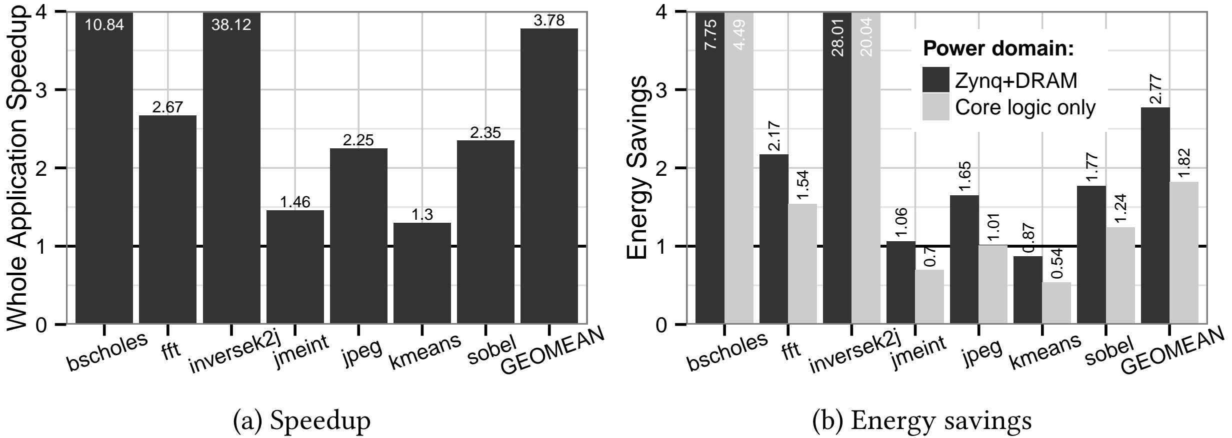
High-Level Synthesis infrastructure. We use Vivado HLS 2014.2 to generate hardware kernels for each benchmark. We then integrate the kernels into SNNAP's bus interface and program the FPGA using Vivado Design Suite 2014.2.

2.5.2 Performance and Energy

This section describes the performance and energy benefits of using SNNAP to accelerate our benchmarks.

Performance. Figure 2.1a shows the whole application speedup when SNNAP is used to execute each benchmark's target region, while the rest of the application runs on the CPU, over an all-CPU baseline.

The average speedup is 3.78×. Among the benchmarks, `inversek2j` has the highest speedup (38.12×) since the bulk of the application is offloaded to SNNAP, and the target region of code includes trigonometric function calls that take over 1000 cycles to execute on the CPU and that a small neural network can approximate. Conversely, `kmeans` sees only a 1.30× speedup,



(a) Speedup

(b) Energy savings

Figure 2.1: Performance and energy benefit of SNNAP acceleration over an all-CPU baseline execution of each benchmark.

mostly because the target region is small and runs efficiently on a CPU, while the corresponding neural network is relatively deep.

Energy. Figure 2.1b shows the energy savings for each benchmark over the same all-CPU baseline. We show the savings for two different energy measurements: (1) the SoC with its DRAM and other peripherals, and (2) the core logic of the SoC. On average, neural acceleration with SNNAP provides a 2.77× energy savings for the SoC and DRAM and a 1.82× savings for the core logic alone.

The *Zynq+DRAM* evaluation shows the power benefit from using SNNAP on a chip that already has an FPGA fabric. Both measurements include all the power supplies for the Zynq chip and its associated DRAM and peripherals, including the FPGA. The FPGA is left unconfigured for the baseline.

The *core logic* evaluation provides a conservative estimate of the potential benefit to a mobile SoC designer who is considering including an FPGA fabric in her design. We compare a baseline consisting only of the CPU with the power of the CPU and FPGA combined. No DRAM or peripherals are included.

On all power domains and for all benchmarks except `jmeint` and `kmeans`, neural acceleration on SNNAP results in energy savings. In general, the more components we include in our power measurements, the lower the relative power cost and the higher the energy savings from neural acceleration. `inversek2j`, the benchmark with the highest speedup, also has the highest energy savings. For `jmeint` and `kmeans` we observe a decrease in energy efficiency in the core logic measurement; for `kmeans`, we also see a decrease in the Zynq+DRAM measurement. While the CPU saves power by sleeping while SNNAP executes, the accelerator incurs more power than this saves, so a large speedup is necessary to yield energy savings.

2.5.3 Characterization

This section supplements our main energy and performance results with secondary measurements to the primary results in context and justify our design decisions.

Impact of parallelism. Figure 2.2 shows the performance impact of SNNAP's parallel design by varying the number of PUs. On average, increasing from 1 PU to 2 PUs, 4 PUs, and 8 PUs improves performance by 1.52×, 2.03×, and 2.40× respectively. The `sobel`, `kmeans` and `jmeint` benchmarks require at least 2, 4, and 8 PUs respectively to see any speedup.

Higher PU counts lead to higher power consumption, but the cost can be offset by the performance gain. The best energy efficiency occurs at 8 PUs for most benchmarks. The exceptions are `jpeg` and `fft`, where the best energy savings are with 4 PUs. These benchmarks have a relatively low "Amdahl speedup limit" , so they see diminishing returns from parallelism.

Impact of batching. Figure 2.3 compares the performance of batched SNNAP invocations, single invocations, and zero-latency invocations - an estimate of the speedup if there were no communication latency between the CPU and the accelerator.

With two exceptions, non-batched invocations lead to a slowdown due to communication latency. Only `inversek2j` and `jpeg` see a speedup since their target regions are large enough to outweigh the communication latency. Comparing with the zero-latency estimate, we find that

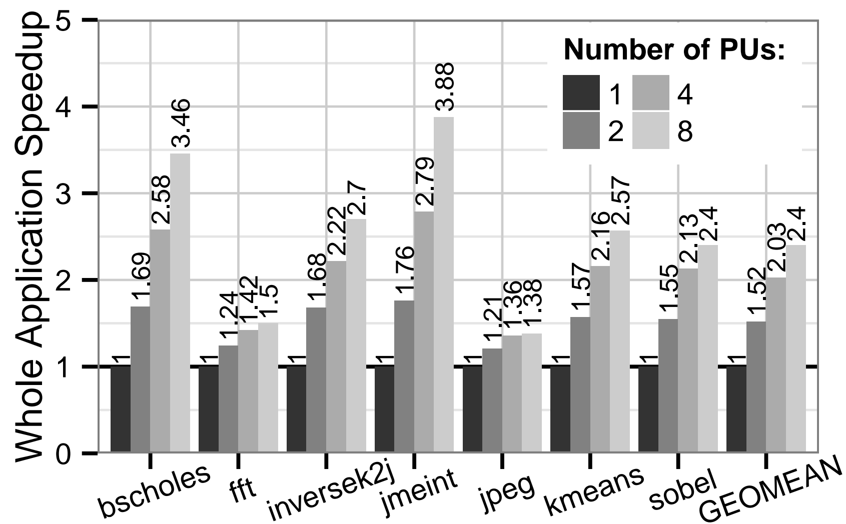


Figure 2.2: Performance of neural acceleration as the number of PUs increase.

batch invocations are effective at hiding this latency. Our 32-invocation batch size is within 11% of the zero-latency ideal.

Optimal PE count. Our primary SNNAP configuration uses 8 PEs per PU. A larger PE count can decrease invocation latency but can also have lower utilization, so there is a trade-off between fewer, larger PUs or more, smaller PUs given the same overall budget of PEs. In Figure 2.5a, we examine this trade-off space by sweeping configurations with a fixed number of PEs. The NPU configurations range from 1 PU consisting of 16 PEs (1×16) through 16 PUs each consisting of a single PE (16×1). The 16×1 arrangement offers the best throughput. However, resource utilization is not constant: each PU has control logic and memory overhead. The 16×1 NPU uses more than half of the FPGA's LUT resources, whereas the 2×8 NPU uses less than 4% of all FPGA resources. Normalizing throughput by resource usage (Figure 2.5b) indicates that the 2×8 configuration is optimal.

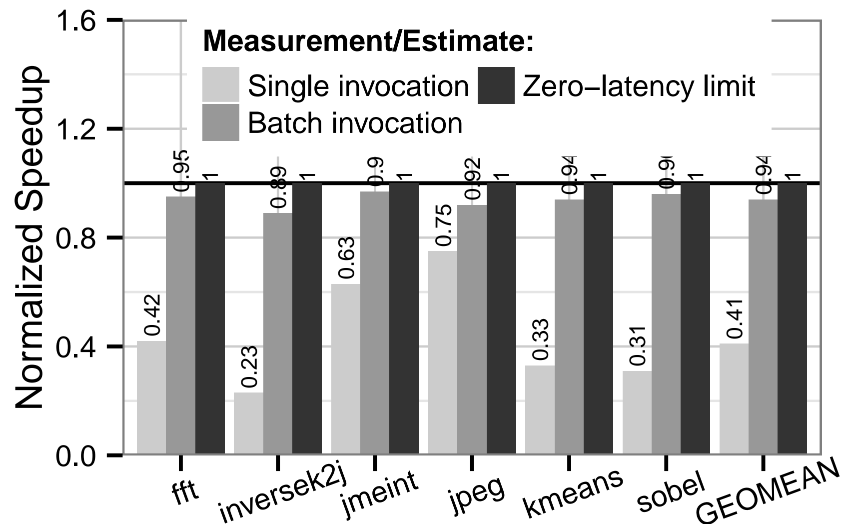


Figure 2.3: Impact of batching on speedup.

2.5.4 Design Statistics

FPGA utilization. Figure 2.5d shows the FPGA fabric's resource utilization for varying PU counts. A single PU uses less than 4% of the FPGA resources. The most utilized resources are the slice LUTs at 3.92% utilization and the DSP units at 3.64%. With 2, 4, 8, and 16 PUs, the design uses less than 8%, 15% 30% and 59% of the FPGA resources respectively and the limiting resource is the DSP slices. The approximately linear scaling reflects SNNAP's balanced design.

Memory Bandwidth. Although the Zynq FPGA can accommodate 16 PUs, the current ACP interface design does not satisfy the bandwidth requirements imposed by compute-resource scaling for benchmarks with high bandwidth requirements (e.g. `jpeg`). This limitation is imposed by the ACP port used to access the CPU's cache hierarchy. During early design exploration, we considered accessing memory via higher-throughput non-coherent memory ports, but concluded experimentally that at a fine offload granularity, the frequent cache flushes were hurting performance. As a result, we evaluate SNNAP at 8-PUs to avoid being memory bound by the

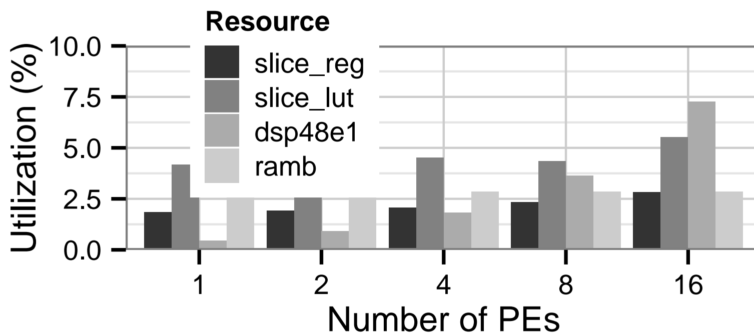


Figure 2.4: Resource Utilization for a 1-PU NPU containing 1 to 16 PEs.

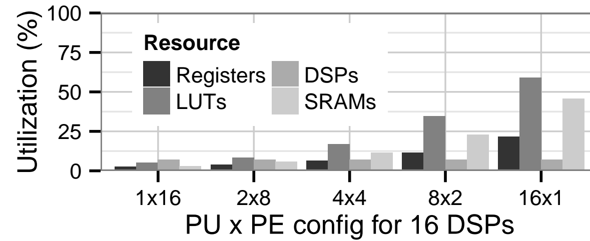
ACP port. We leave interface optimizations and data compression schemes that could increase effective memory bandwidth as future work.

Output quality. We measure SNNAP's effect on output quality using application-specific error metrics, as is standard in the approximate computing literature [155, 60, 61, 164]. Table 2.2 lists the error metrics.

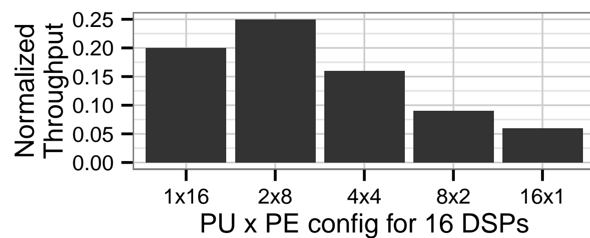
We observe less than 10% application output error for all benchmarks except `jmeint`. `jmeint` had high error due to complicated control flow within the acceleration region, but we include this benchmark to fairly demonstrate the applicability of neural acceleration. Among the remaining applications, the highest output error occurs in `sobel` with 8.57% mean absolute pixel error with respect to a precise execution. To put this error in context, Figure 2.6 shows the output from the original and SNNAP-accelerated executions of the benchmark. Qualitatively, the program still produces reasonable results.

2.5.5 HLS Comparison Study

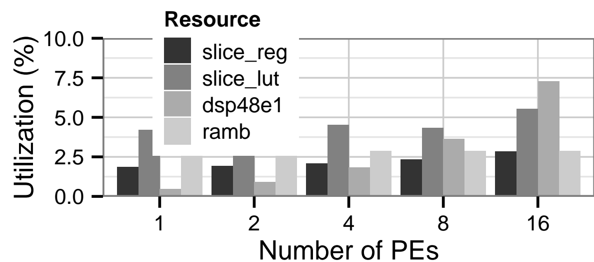
We compare neural acceleration with SNNAP against Vivado HLS [194]. For each benchmark, we attempt to compile using Vivado HLS the same target regions used inneural acceleration. We synthesize a precise specialized hardware datapath and integrate it with the same CPU–FPGA



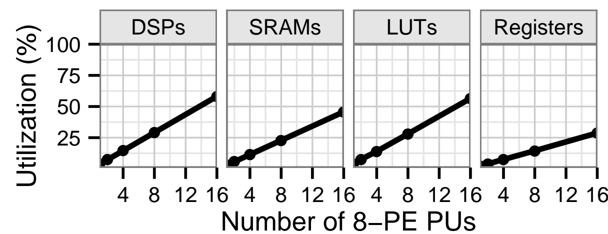
(a) Static resource utilization for multiple configurations of 16 DSP units.



(b) Peak throughput on jmeint normalized to most-limited FPGA resource for each configuration.



(c) Resource utilization as number of PEs increase in a single PU.



(d) Resource utilization as number of PUs increase, each PU consisting of 8-PEs.

Figure 2.5: Exploration of SNNAP static resource utilization.

Logic Utilization	Used	Available	Util
Occupied Slices	625	13300	4%
Slice Registers	2055	106400	2%
Slice LUTs	1650	53200	3%
RAMB18E1	13	280	4%
RAMB36E1	4	140	2%
DSP48E1	8	220	3%

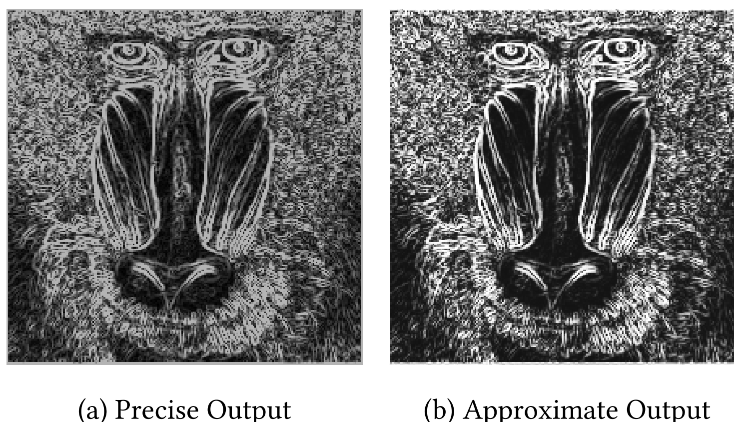
Table 2.1: Post-place-and-route FPGA utilization.

interface we developed for SNNAP and contrast whole-application speedup, resource-normalized throughput, FPGA utilization, and programmer effort.

Speedup. Table 2.2 shows statistics for each kernel we synthesized with Vivado HLS. The kernels close timing between 66 MHz and 167 MHz (SNNAP runs at 167 MHz). We compare the performance of the HLS-generated hardware kernels against SNNAP.

Figure 2.7a shows the whole-application speedup for HLS and SNNAP. The NPU outperforms HLS on all benchmarks, yielding a 3.78× average speedup compared to 2× for HLS. The `jmeint` benchmark provides an example of a kernel that is not a good candidate for HLS tools; its dense control flow leads to highly variable evaluation latency in hardware, and the HLS tool was unable to pipeline the design. Similarly, `jpeg` performs poorly using HLS due to DSP resource limitations on the FPGA. Again, the HLS tool was unable to pipeline the design, resulting in a kernel with long evaluation latency. HLS nearly matches SNNAP's speedup on `blackscholes` and `fft` as it is able to generate fully pipelined, low latency kernels for each.

Resource-normalized kernel throughput. To assess the area efficiency of SNNAP and HLS, we isolate FPGA execution from the rest of the application. We compute the theoretical through-



(a) Precise Output (b) Approximate Output

Figure 2.6: Output of `sobel` for a 220x220 pixel image.

put (evaluations per second) by combining the *pipeline initiation interval* (cycles per evaluation) from functional simulation and the $f_{\max}$ (cycles/second) from post-place-and-route timing analysis. We obtain post-place-and-route resource utilization by identifying the most-used resource in each design. The resource-normalized throughput is the ratio of these two metrics.

Figure 2.8 compares the resource-normalized throughput for SNNAP and HLS-generated hardware kernels. Neural acceleration does better than HLS for `blackscholes`, `inversek2j`, `jmeint` and `jpeg`. In particular, while HLS provides better absolute throughput for `blackscholes` and `inversek2j`, the kernels also use an order of magnitude more resources than a single SNNAP PU. `kmeans` and `sobel` have efficient HLS implementations with utilization roughly equal to one SNNAP PU, resulting in 2–5× greater throughput.

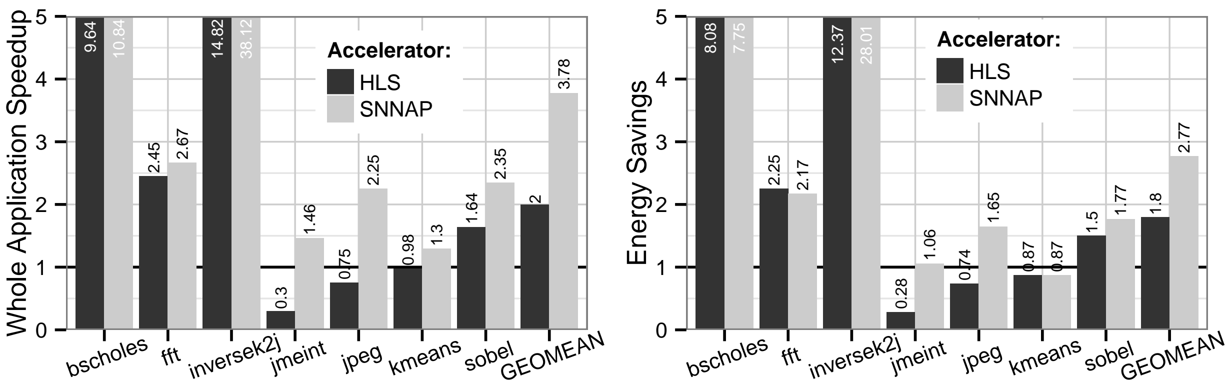
Programming experience. "C-to-gates" tools are promoted for their ability to hide the complexity of hardware design. With our benchmarks, however, we found hardware expertise to be essential for getting good results using HLS tools. Every benchmark required hardware experience to verify the correctness of the resulting design and extensive C-code tuning to meet the tool's requirements.

Table 2.2 lists the number of working days required for a student to produce running hardware

Application	Effort	Clock	Pipelined	Util.
`blackscholes`	3 days	148 MHz	yes	37%
`fft`	2 days	166 MHz	yes	10%
`inversek2j`	15 days	148 MHz	yes	32%
`jmeint`	5 days	66 MHz	no	39%
`jpeg`	5 days	133 MHz	no	21%
`kmeans`	2 days	166 MHz	yes	3%
`sobel`	3 days	148 MHz	yes	5%

Table 2.2: HLS-kernel specifics per benchmark: required engineering time (working days) to accelerate each benchmark in hardware using HLS, kernel clock, whether the design was pipelined, most-utilized FPGA resource utilization.

for each benchmark using HLS. The student is a Masters researcher with Verilog and hardware design background but not prior HLS experience. Two months of work was needed for familiarization with the HLS tool and the design of a kernel wrapper to interact with SNNAP's custom memory interface. After this initial cost, compiling each benchmark took between 2 and 15 days. `blackscholes`, `fft`, `kmeans`, and `sobel` all consist of relatively simple code, and each took only a few days to generate fast kernels running on hardware. The majority of the effort was spent tweaking HLS compiler directives to improve pipeline efficiency and resource utilization. Accelerating `jmeint` was more involved and required 5 days of effort, largely spent attempting (unsuccessfully) to pipeline the design. `jpeg` also took 5 days to compile, which was primarily spent rewriting the kernel's C code to make it amenable to HLS by eliminating globals, precomputing lookup tables, and manually unrolling some loops. Finally, `inversek2j` required 15 days of effort. The benchmark used the arc-sine and arc-cosine trigonometric functions, which are not supported by the HLS tools, and required rewriting the benchmark using mathematical identities with the supported arc-tangent function. The latter exposed a bug in the HLS workflow



(a) Single HLS kernel and 8-PU NPU whole-application speedups over CPU-only execution baseline.

(b) Energy savings of single HLS kernel and 8-PU NPU over CPU-only baseline for Zynq+DRAM power domain.

Figure 2.7: Performance and energy comparisons of HLS and SNNAP acceleration.

that was eventually resolved by upgrading to a newer version of the Vivado tools.

In an ideal world, using “C-to-gates” HLS tools would be as simple as using a traditional C compiler—no RTL programming experience required. The reality is different. For our benchmark suite, we found hardware expertise to be essential for troubleshooting the resulting design and to achieve good performance. Every benchmark required extensive tuning of C-code to meet the tool’s requirements.

Table 2.2 lists the number of working days taken by a student to get each benchmark running on hardware using HLS. The student in question is an Masters researcher with no prior experience in HLS, but some Verilog background. It took over two months to get familiarized with the tool flow and design the kernel wrapper to interact with a memory interface. After this initial cost, each benchmark took between 2 days and 3 weeks. Four benchmarks—`kmeans`, `fft`, `blackscholes`, and `sobel`—consist of relatively simple code: it took less than a week to get fast kernels from these benchmarks running on hardware. This effort was spent on tweaking compiler directives to get more efficient pipelining and resource utilization. Accelerating

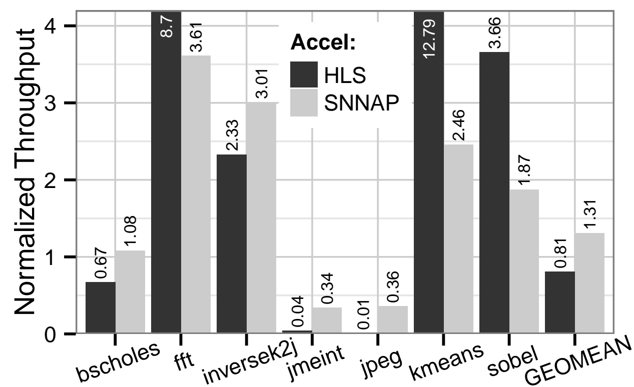


Figure 2.8: Resource-normalized throughput of the NPU and HLS accelerators.

`jmeint` was more involved and required a full week of effort, and even then the design could not be successfully pipelined. `jpeg` also took a full week, which was primarily spent rewriting the kernel's C code to make it amenable to HLS by eliminating globals, precomputing some tables, and unrolling loops. Finally, `inversek2j` required three weeks of effort. The benchmark used the arc-cosine and arc-sine trigonometric functions, which are not supported by the HLS tools, so we rewrote the benchmark using mathematical identities to simplify the code. The (supported) arc-tangent function exposed a bug in the HLS workflow that was, in the end, solved by upgrading to a newer version of the Vivado suite.

Discussion. While HLS offers a route to FPGA use without approximation, it is far from flawless: significant programmer effort and hardware-design expertise is still often required. In contrast, SNNAP acceleration uses a single FPGA configuration and requires no hardware knowledge. Unlike HLS approaches, which place restrictions on the kind of C code that can be synthesized, neural acceleration treats the code as a black box: the internal complexity of the legacy software implementation is irrelevant. SNNAP's FPGA reconfiguration-free approach also avoids the overhead of programming the underlying FPGA fabric, instead using a small amount of configuration data that can be quickly loaded in to accelerate different applications. These advantages make neural acceleration with SNNAP a viable alternative to traditional C-to-gates approaches.

2.6 Related Work

Our design builds on related work in the broad areas of approximate computing, acceleration, and neural networks.

Approximate computing. A wide variety of applications can be considered *approximate:* occasional errors during execution do not obstruct the usefulness of theprogram's output. Recent work has proposed to exploit this inherent resiliency to trade off output quality to improve performance or energy consumption using software [17, 164, 14, 118, 119, 83] or hardware [55, 112, 60, 108, 127, 33, 61, 155, 72] techniques. SNNAP represents the first work (to our knowledge) to exploit this trade-off using tightly integrated on-chip programmable logic to realize these benefits in the near term. FPGA-based acceleration using SNNAP offers efficiency benefits that complement software approximation, which is limited by the overheads of general-purpose CPU execution, and custom approximate hardware, which cannot be realized on today's chips.

Neural networks as accelerators. Previous work has recognized the potential for hardware neural networks to act as accelerators for approximate programs, either with automatic compilation [61, 171] or direct manual configuration [39, 176, 20]. This work has typically assumed special-purpose neural-network hardware; SNNAP represents an opportunity to realize these benefits on commercially available hardware. Recent work has proposed combining neural transformation with GPU acceleration to unlock order-of-magnitude speedups by elimiating control flow divergence in SIMD applications [73, 72]. This direction holds a lot of promise in applications where a large amount of parallelism is available. Until GPUs become more tightly integrated with the processor core, their applicability remains limited in applications where the invocation latency is critical (i.e. small code offload regions). Additionally the power envelope of GPUs has been traditionally high. Our work targets low power accelerators and offers higher applicability by offloading computation at a finer granularity than GPUs.

Hardware support for neural networks. There is an extensive body of work on hardware implementation of neural networks both in digital [136, 59, 202, 40, 49, 24] and analog [27, 159, 174, 89] domains. Other work has examined fault-tolerant hardware neural networks [79, 176]. There is also significant prior effort on FPGA implementations of neural networks ([202] contains a comprehensive survey). Our contribution is a design that enables automatic acceleration of approximate software without engaging programmers in hardware design.

FPGAs as accelerators. This work also relates to work on synthesizing designs for reconfigurable computing fabrics to accelerate traditional imperative code [138, 143, 46, 63]. Our work leverages FPGAs by mapping diverse code regions to neural networks via neural transformation and accelerating those code regions onto a fixed hardware design. By using neural networks as a layer of abstraction, we avoid the complexities of hardware synthesis and the overheads of FPGA compilation and reconfiguration. Existing commercial compilers provide means to accelerate general purpose programs [194, 9] with FPGAs but can require varying degrees of hardware expertise. Our work presents a programmer-friendly alternative to using traditional "C-to-gates" high-level synthesis tools by exploiting applications' tolerance to approximation.

2.7 Conclusion

SNNAP enables the use of programmable logic to accelerate approximate programs without requiring hardware design. Its high-throughput systolic neural network mimics the execution of existing imperative code. We implemented SNNAP on the Zynq system-on-chip, a commercially available part that pairs CPU cores with programmable logic and demonstrate 3.8× speedup and 2.8× energy savings on average over software execution. The design demonstrates that approximate computing techniques can enable effective use of programmable logic for general-purpose acceleration while avoiding custom logic design, complex high-level synthesis, or frequent FPGA reconfiguration.

CHAPTER APPENDIX

2.A SNNAC: An Error-Tolerant Low-Voltage SRAM Neural Network Accelerator ASIC

Results and figures borrowed from : Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze and Visvesh Sathe, *MATIC: Learning Around Errors for Efficient Low-Voltage Neural Network Accelerators*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018.

While the main application for SNNAP was to approximately offload compute intensive regions of code to a flexible FPGA acceleration fabric, the design of the SNNAP architecture can be applied to more general deep learning applications, specifically fully-connected deep neural networks (DNNs). We present one ASIC design based on SNNAP that was used to explore SRAM approximation knobs in the context of deep learning inference. SNNAC (Systolic Neural Network ASIC) is an error-tolerant low-voltage ASIC implementation of the SNNAP accelerator design coupled with an MSP-430. It is aimed at offloading fully connected neural networks inference and neurally-approximated software kernels [97].

2.A.1 SNNAC Motivation

State of the art DNNs can have millions or billions of learnable weights, implying that judiciously managing data movement and data storage is critical in minimizing energy consumption. Chen et al. [42] noted that maximizing data reuse on chip is critical to achieving efficient inference. Follow up work on convolutional neural networks [56] showed that maximizing weight reuse on SRAM made edge-inference for vision applications possible. Other ASIC proposals have explored compression-based approaches to store the weights of ever growing deep neural networks [77]. Figure 2.A.1 shows how much power is dedicated to storing weights on chip in state of the art deep learning ASIC designs [40, 42, 56, 77].

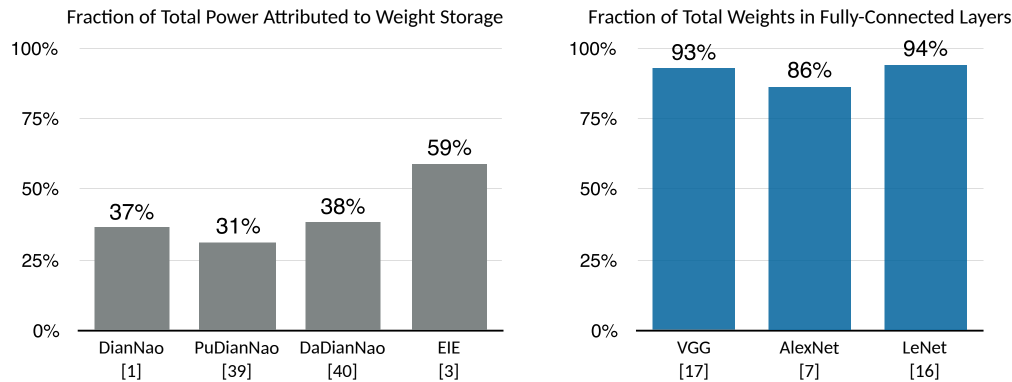


Figure 2.A.1: **(left)** The fraction of total power dissipated by weight storage SRAMs, and **(right)** the fraction of total SRAM used to store fully-connected weights. On-chip weight storage accounts for a significant fraction of the total power dissipation in state-of-the-art DNN accelerators. Even for Conv-DNNs such as AlexNet, weight storage is dominated by fully-connected layers.

2.A.2 SNNAC Overview

SNNAC was designed to overcome the challenge of the ever-growing cost of storing DNN weights in on-chip SRAMs. The key idea behind SNNAC was to expose voltage scaling knobs to minimize static and dynamic power dissipation in SRAM. However, as the voltage gets lowered, so does the chance of encountering a read upset or a write error increase. The key in being able to leverage voltage under-scaled SRAMs in SNNAC is to exploit (1) proactive error mitigation strategies that consist of learning weights around statically profiled errors, and (2) reactive error detection mechanisms that use dummy logic circuits (i.e. canaries) to identify imminent failures. The details of the error tolerant approach, referred to as MATIC by Kim et al. is described in much detail in [97]. Figure 2.A.2 summarizes MATIC's key principles: SRAM cells are operated at voltages that will lead to high rates of bit-level errors, but uses adaptive training approaches to minimizes the effect of those physical errors on the execution of the high-level algorithm.

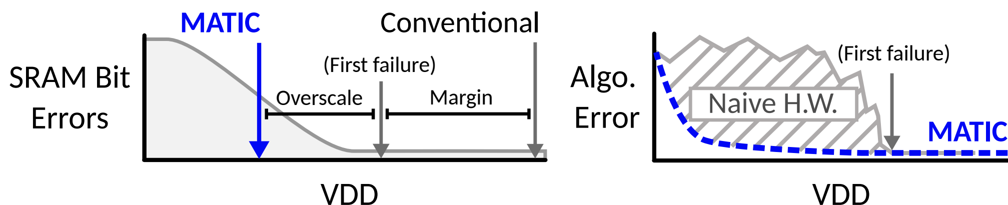


Figure 2.A.2: (left) MATIC [97] increases energy-efficiency by aggressively scaling supply voltages of on-chip weight SRAMs. (right) Compared to hardware paired with conventionally-trained neural network models, MATIC leverages an adaptive training process to recover from errors caused by voltage overscaling.

2.A.3 SNNAC ASIC Implementation

To demonstrate the effectiveness of MATIC on real hardware, SNNAC was implemented in 65 nm CMOS technology (Figure 2.A.4). The SNNAC core consists of a fully-programmable Neural Processing Unit (NPU) based on the SNNAP design [126]. The NPU contains eight multiply-accumulate (MAC)-based Processing Elements (PEs) which are arranged in a 1D systolic ring that maintains high compute utilization during inner-product operations. Energy-efficient arithmetic in the PEs is achieved with 8-22 bit fixed-point operands, and each PE includes a dedicated voltage-scalable SRAM bank to enable on-chip storage of all synaptic weights. The systolic ring is attached to an activation function unit (AFU), which minimizes energy and area footprint with piecewise-linear approximation of activation functions (e.g., sigmoid or ReLU).

The operation of the PEs is coordinated by a lightweight control core that executes statically compiled microcode. To achieve programmability and support for a wide range of layer configurations, the computation of wide DNN layers is time-multiplexed onto the PEs in a systolic ring. When the layer width exceeds the number of physical PEs, PE results are buffered to an accumulator that computes the sum of all atomic MAC operations in the layer. SNNAC also includes a sleep-enabled OpenMSP430-based microcontroller (µC) to handle runtime control, debugging functions, and off-chip communication with a UART serial interface. To minimize data movement, NPU input and output data buffers are memory-mapped directly to the µC data-memory

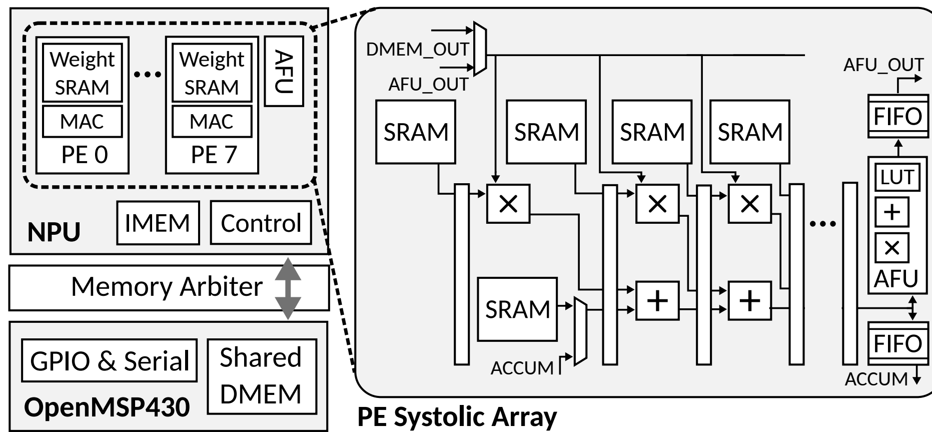
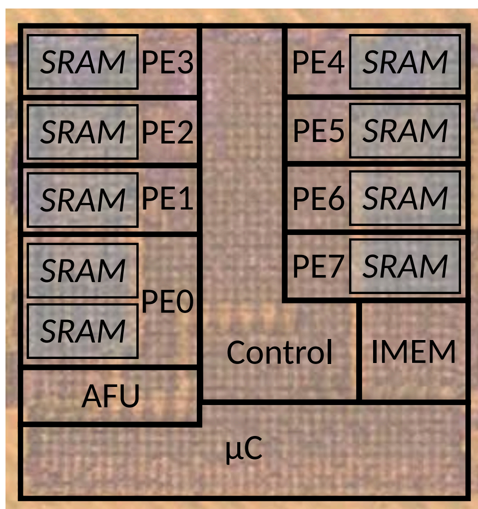


Figure 2.A.3: Architecture of the SNNAC DNN accelerator. The SNNAP design is tightly integrated with an OpenMSP430 micro-controller.

address space.

2.A.4 SNNAC Results Summary

The combination of an efficient ASIC design, coupled with memory supply-voltage voltage overscaling, and error correction mechanism to minimize the effect of bit-level errors on application error leads to highly efficient neural network inference. As demonstrated on SNNAC, MATIC[97] reports $3.3\times$ total energy reduction, and $5.1\times$ energy reduction in SRAM, or 18.6x reduction in application error. This indicates that SNNAC, when used with error compensation techniques can achieve *graceful* quality degradation as errors appear. By taking advantage of application specificity (e.g. deep learning's ability to learn around errors), a thoughtfully crafted application stack can elegantly mitigate errors as they manifest in physical layer (e.g. SRAM bit-level errors).



Technology	TSMC GP 65 nm
Core Area	1.15×1.2 mm
SRAM	9 KB
Weight Prec.	8-bit
Activation Prec.	22-bit
Voltage	0.9 V
Frequency	250 MHz
Power	16.8 mW
Energy	67.1 pJ/cycle

Figure 2.A.4: (a) Microphoto of a fabricated SNNAC test chip, and (b) summary of test chip characteristics. The baseline voltage, power, frequency, and energy efficiency are reported.

Chapter 3

QAPPA: QUALITY AUTOTUNER FOR PRECISION PROGRAMMABLE HARDWARE ACCELERATORS

"Our treatment of this science will be adequate, if it achieves the amount of precision which belongs to its subject matter."

– Aristotle, *Nicomachean Ethics, 7.*

Published As: Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi and Luis Ceze, *Exploiting Quality-Energy Tradeoffs with Arbitrary Quantization*, Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2017.

Abstract *Quantization naturally exposes knobs in hardware to trade fidelity for efficiency: the more bits that are used to represent the data, the higher the storage and computation overheads. With the emergence of approximate computing research, we set out to answer the following question: how effective is quantization in trading quality for efficiency, and how does it compare to other approximation techniques? This chapter makes the case for quantization as a general approximation technique that exposes quality vs. energy tradeoffs and provides practical error guarantees. We assume* arbitrary *quantization levels, and focus on the hardware subsystems that are affected by quantization: memory and computation. We present QAPPA (Quality Autotuner for Precision Programmable Accelerators), an autotuner for C/C++ programs that automatically tunes the precision of each arithmetic and memory operation to meet user defined application level quality guarantees. QAPPA integrates energy models of quantization scaling mechanisms to produce bandwidth and energy savings estimates for custom accelerator designs. We use the analysis produced by QAPPA to*

compare the effectiveness of arbitrary quantization against voltage overscaling and neural approximation. Our analysis shows that when using the right quantization scaling mechanisms in hardware, quantization provides significant energy efficiency benefits over voltage overscaling and comparable energy efficiency gains over neural approximation. Additionally, quantization offers more predictable error degradation and fully tunable error bounds.

3.1 Introduction

Energy efficiency is a first-class concern in data centers, embedded systems and sensory nodes. To improve energy efficiency, numerous cross-stack techniques have been proposed to bring hardware and software systems closer to their quality-energy Pareto-optimal design point. Navigating quality-energy tradeoffs is fundamental to digital systems design, and often starts with data representation, i.e. how to map a set of real values to a compact and finite digital representation. This process is called *quantization*, and is essential in keeping computation tractable in digital systems. Quantization offers a natural way to trade quality for energy efficiency by tweaking the number of bits needed to represent data. Using more bits leads to higher fidelity, but also larger compute, data movement and memory overheads.

This chapter argues towards adopting arbitrary quantization as a general approximation technique for its effectiveness in delivering smooth quality-energy tradeoffs, and practical error guarantees. Quantization is often overlooked as an effective way to improve quality-energy optimality due to the limited quantization levels available in hardware (e.g. single and double precision floating point), and the large control overheads found in general purpose processors. This chapter bypasses those limitations by assuming *arbitrary* quantization, i.e. bit-granular precision tunability, and by targeting hardware accelerators where control overheads are minimal.

We introduce QAPPA (Quality Autotuner for Precision Programmable Accelerators), a precision auto-tuner for C and C++ programs that finds bit-granular quantization requirements for each program instruction while meeting user-defined application-level quality guarantees. QAPPA leverages ACCEPT [154] in order to guarantee isolation of approximation effects based on lightweight user annotations. We survey a set of hardware precision scaling techniques and eval-

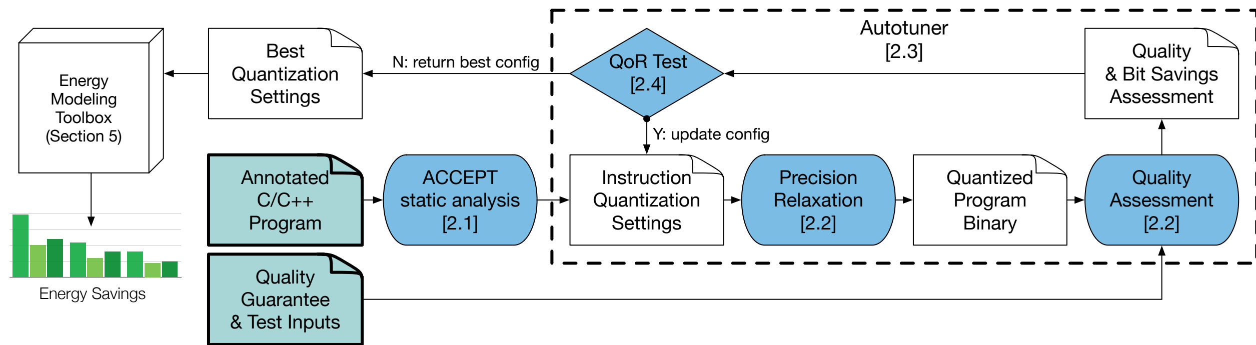


Figure 3.1: QAPPA Autotuner System Architecture.

uate their ability to improve quality-energy optimality using detailed RTL models. We feed those hardware models into QAPPA to identify energy savings opportunities that arise from adopting precision scaling techniques in hardware accelerator designs. QAPPA isolates arithmetic energy savings and memory bandwidth savings, preserving the orthogonality between savings due to specialization and savings due to approximation in hardware accelerators.

We analyze the PERFECT benchmark suite [18] with QAPPA to unveil significant precision reduction opportunities; about 74%, 57%, and 48% of total precision bits can be dropped to achieve 10%, 1%, and 0.1% average relative error. Respectively, we suggest hardware precision-scaling mechanisms for hardware accelerators that provide 7.7×, 4.8×, and 3.6× energy reduction in arithmetic units, and 4.4×, 3.3×, and 2.8× memory bandwidth reduction.

Finally, we argue that arbitrary quantization compares favorably against other approximation techniques in terms of quality-energy optimality and error guarantees. Our comparative study of approximation techniques includes a SPICE-level characterization of voltage scaling-induced faults, and an analytical evaluation of neural acceleration in terms of hardware resource utilization. Our evaluation reveals that arbitrary quantization outperforms voltage overscaling in terms of quality-energy optimality, and provides performance that is on par with neural acceleration.

3.2 QAPPA: A Quantization Autotuner

QAPPA is a precision autotuning framework built using ACCEPT [154], the LLVM-based approximate compiler for C and C++ programs. In a nutshell, QAPPA takes an annotated C/C++ program and user-specified, high-level quality guarantees to greedily derive quantization requirements for each program instruction. We discuss the design and implementation of QAPPA as illustrated in Figure 3.1. Section 3.2.1 describes the annotation model used by QAPPA to identify instructions that are safe to approximate and guarantee program safety. Section 3.2.2 describes how QAPPA instruments programs to quantify quality loss that results from arbitrary quantization. Section 3.2.3 describes the autotuner search algorithm and how it is used to find quantization requirements. Section 3.2.4 describes the quality guarantees that QAPPA provides. The energy modeling toolbox is later discussed in Section 3.4, where we evaluate different hardware techniques that enable energy scaling.

3.2.1 Annotation Model and Static Analysis

QAPPA leverages ACCEPT [154] to provide type-safety and error isolation guarantees. These isolation guarantees are essential to prevent crashes or catastrophic errors from occurring. QAPPA utilizes the APPROX type qualifiers for approximate data used by ACCEPT. By default, all program variables are assumed to be precise, so approximations have to be specified as an opt-in property. Consequently, it is the programmer's responsibility to annotate what variables hold data that is safe to approximate. The compiler then uses flow analysis to infer which instructions are *approximable* from data annotations.

Figure 3.1 shows how one would annotate a simple convolution kernel. Intuitively, data types such as pixels and filter coefficients can be relaxed, but integer variables that are used to index arrays should remain precise to avoid out-of-array writes. In the convolution example, the compiler infers that the instructions that perform convolution are safe to approximate (instructions from `l.9` and `l.10`). In addition, it identifies that the loads from the image source and the stores to the image destination are also safe to approximate. These approximable instructions will later

be used by the autotuner as *knobs* to minimize precision in the target program.

3.2.2 Assessing Quantization Effects

The QAPPA autotuner relies on a trial-and-error approach to find locally optimal quantization settings that satisfy user-defined accuracy metrics. In order to properly assess quantization effects on a given program execution, QAPPA statically instruments the target program with code that applies arbitrary quantization to individual arithmetic and memory instructions. This can be done in LLVM by replacing all uses of a given static single assignment (SSA) register with its quantized counterpart. In order to perform floating point to fixed point conversion, QAPPA performs an initial dynamic profiling step on the target program by measuring the value range of each variable.

The degree of quantization and the rounding policy (i.e. up, down, towards zero, away from zero, nearest) are defined for each static instruction in a quantization settings file. The quantization settings dictate how QAPPA applies varying levels of quantization to each instruction in the target program. The instrumented program gets compiled by QAPPA to produce an approximate binary. The approximate binary can then be executed on user-provided input datasets to produce output data on which to quantitatively assess quality degradation with user-defined quality metrics.

3.2.3 Autotuner Design

The goal of the autotuner is to maximize quantization while satisfying user-specified quality requirements.

Bit Savings We define *bit savings* as a hardware-agnostic metric that quantifies how much total precision can be trimmed-off in a program over its execution. QAPPA attempts to maximize bit savings while keeping application accuracy within user-specified margins.

Bit savings are calculated with the following formula:

$$BitSavings = \sum_{i=1}^{N} \frac{(r_i - q_i)}{r_i} \times \frac{e_i}{\sum_{j=1}^{N} e_j}$$

```
0:  void conv2d (APPROX pix *in, APPROX pix *out, APPROX flt *filter)
1:    for (row)
2:      for (col)
3:        APPROX flt sum = 0
4:        int dstPos = …
5:        for (row_offset)
6:          for (col_offset)
7:            int srcPos = …
8:            int fltPos = …
9:            sum += in[srcPos] * filter[fltPos]
10:       out[dstPos] = sum / normFactor
```

Figure 3.1: Program annotation with APPROX type qualifier. Variables that are safe to approximated are annotated by the user. The compiler then infers the program instructions that can be approximated.

where r_i and q_i denote the precision in bits of the reference, and quantized instruction i, e_i denotes the number of times instruction i executes, and N denotes the total approximable instructions in the target program. For instance, if a program executes only one single precision floating point instruction, and that QAPPA quantizes that instruction down to 6 bits, the total bit savings will be $\frac{32-6}{32} \times \frac{1}{1} = 81.25\%$.

Autotuner Search Algorithm The challenge in the design of an arbitrary quantization autotuner lies in the exponentially large problem search space. Let us consider a program containing m static instructions, where each instruction can be tuned to n different precision levels. In order to find a globally optimal configuration that maximizes bit savings, the autotuner needs to traverse an exponential search space with n^m possible quantization settings, each with different tradeoffs between quality and bit savings. Instead of resorting to a brute-force search to find the optimal configuration, we use a greedy search which finds a local optimum in $O(m^2 * n)$ worst-case time by selecting the path of least quality degradation.

The greedy iterative search algorithm is similar to the approach proposed in Precimonious [150]

which uses a trial and error tuning approach to selecting the precision of floating point data. At each step of the search, the QAPPA autotuner identifies the instruction that affects output the least, and relaxes its precision by a single bit. The autotuner repeats the process until it finally reaches a point where decreasing the precision of any instruction violates user-defined quality requirements. We discuss the different quality tests that can be used to guide this search process in Section 3.2.4. Finally, the autotuner reports locally-optimal instruction quantization settings along with bit savings estimates. Those quantization settings can then be fed into an energy modeling toolbox, which we discuss later in Section 3.4.

Autotuner Complexity The autotuner greedy-search can be improved with runtime optimizations including search parallelization, stochastic search, the *delta-debugging* algorithm [150]. The autotuner can parallelize each stage of its iterative search, across m machines where m is the number of safe to approximate instructions in a given program. We found that in kernels we analyzed, m is relatively small (¡200), which makes it possible to run the analysis on a cluster, allowing the search algorithm to run in O($m*n$) worst-case time on m nodes. If one were to run the autotuner on a single node, the autotuner could be extended to make use of the delta-debugging algorithm, which finds an optimal solution in $O(n * m * \log m)$ average times. A final trick to improve runtime of the autotuner is to use logarithmic precision increments instead of using 1-bit increments at each stage of the autotuning search. This improves average runtime of the parallel autotuner to $O(m * \log n)$ and the serial delta-debugging autotuner to $O(m * \log m * \log n)$.

3.2.4 Quality of Result (QoR) Guarantees

Approximation techniques are only practical if they provide *accuracy guarantees* to the programmer. Guarantees are used as a contract between the tools and the programmer to ensure that the relaxations applied by the tool to the target program will not violate QoR requirements. Guarantees can come under different forms: empirical, statistical and hard guarantees. Hard guarantees provide the strongest guarantees by assuming worst-case error accumulation. A method to ensure hard guarantees is interval analysis [184], which can be applied to small functions that do

not exhibit asymptotic behavior or long chains of operations that could lead to high error accumulation. While hard guarantees are the most desirable to the user, they assume worst-case error accumulation, which are often not representative of real-world inputs. For that reason, QAPPA offers empirical or statistical guarantees.

Empirical Guarantees Empirical guarantees provide guarantees that are as good as the datasets provided by the user. This puts more pressure on the programmer to provide satisfactory input coverage, akin to what test engineers do in industry to ensure that code is properly tested, or that learning models are properly trained. This class of guarantees are prevalent in approximate computing literature, due to the complexity involved in providing stricter guarantees [61, 154].

QAPPA provides empirical guarantees by default. The user has to provide a training dataset, and a validation dataset. QAPPA's autotuner traverses the search path of least quality degradation measured on the training input set, but decides when to stop its search when error thresholds are violated on the validation dataset. Having disjoint test and validation sets prevents overfitting issues. QAPPA also provides statistical guarantees, which we discuss next.

Statistical Guarantees Statistical guarantees provide a way to reason about unlikely quality violations. Some applications scenarios may tolerate rarely occurring errors if that means achieving significant energy savings. While statistical guarantees make the most sense in the context of non-deterministic approximations [157] and statistical sampling-based approximations [5], they can also be used on deterministic techniques [114]. In the latter case, the rarely occurring quality violation would be the result of a corner case input that would lead to worst case error accumulation.

We augment QAPPA to provide statistical error guarantees in the form of confidence intervals. For example, a confidence interval may imply that the output has an error of at most 10% with a confidence that is equal or greater than 95%. To derive a statistical guarantee, QAPPA measures $N_{violation}$, the number of times the error has exceeded a given error bound δ across N input samples that it has sampled from the user-provided distribution. QAPPA then provides a

statistical bound to the user by computing the Clopper-Pearson interval [48] to find an upper bound ε of the probability of getting errors that are larger than δ. We have:

$$\varepsilon = \beta(1 - \frac{\alpha}{2}; N_{violation} + 1, N - N_{violation}) \tag{3.1}$$

where β denotes the beta distribution and α is a constant that determines the confidence of the Clopper-Pearson interval. In all our experiments, we set $\alpha = 0.01$. Equation 3.1 entails

$$Pr[error < \delta] > 1 - \varepsilon$$

in which $Pr[*]$ denotes the probability of an event. Equivalently, we can say that the *error is within* δ *with* $1 - \varepsilon$ *confidence.*

3.2.5 Floating-Point to Fixed Point Conversion

We discuss a motivation use-case for QAPPA: deriving cheap fixed point specification of floating-point and math heavy kernels for hardware implementation (which will evaluated in Section 3.5.2). Fixed point computation is significantly cheaper than its floating-point counterpart and is thus preferred in custom hardware designs to maximize energy efficiency [2]. Floating-point data representations provide high-precision across a very wide dynamic range of values, which makes the implementation of certain algorithms possible without using very long integer types. In practice however, application programmers choose floating-point over fixed point computation for its practicality and ease-of-use. Deriving fixed point precision requirements from floating point programs can be tricky, and generally requires analysis from domain experts, numerical analysts, or the use of application-specific tools. We describe how we augment QAPPA to produce precision minimal fixed point specifications of floating-point program by performing (1) floating-point to fixed point conversion, (2) precision minimization and (3) piece-wise polynomial approximation of standard math functions. The end result of these transformations is a cheap fixed point specification of the input program where all non-linear operations (i.e. standard math, or division) are replaced with cheap linear operations (i.e. addition and multiplication).

Fixed Point Emulation and Precision Minimization QAPPA can convert all safe-to-approximate floating point instructions with fixed point instructions via dynamic profiling and static instrumentation. First, QAPPA analyzes the dynamic range of exponents for each floating-point variable by inserting instrumentation code that tracks the range of binary exponents for each target variable. Then, once QAPPA has recorded the largest exponent e_{max} value for each variable, it generates a fixed point version of that instruction with precision n by dynamically setting the mantissa width m of the variable at run-time to $m = n - e_{max} + e - 1$ where e denotes the current exponent value of the variable. Finally, the autotuner performs its standard search to find a precision-minimal configuration that satisfies the QoR requirements set by the user. There is one limitations to this fixed point emulation approach: fixed point precisions cannot exceed the original floating point type's mantissa width. This limitation can be alleviated with the use of double precision floating point types.

Piece-Wise Polynomial Approximation of Standard Math Functions Standard math is prevalent in many application domains, but comes at a high computational cost in its standard library implementation. Precision reduction provides an opportunity to significantly reduce the cost of math. QAPPA uses the final instruction-level precision requirements produced by the autotuner to generate a piece-wise-polynomial approximations for each math function in the target program. We implemented a custom optimization library for QAPPA that finds the cheapest piece-wise polynomial approximation for each math function while ensuring that QoR is not violated as a result of the code transformation.

3.3 PERFECT Application Study

We use QAPPA on the PERFECT benchmark [18] kernels to quantify the opportunity for quantization on compute intensive workloads. We answer the following questions:

Section 3.3.3 How long does the autotuner take to run on the target program?

Section 3.3.4 How does increasing the strength of guarantees diminish opportunities for precision re-

App.	Kernel	Use Case	User Annotations	Static Approx. Insn.	Dynamic Approx. Insn.	Approx Runtime Overhead	Autotuner Steps (40dB)	Autotuner Runtime
PA1	2D Convolution	Convolutional NNs	6	6	33%	8.9x	26	233x
	DWT	JPEG compression	10	27	44%	3.3x	94	315x
	Histogram Eq.	PDF estimation	12	13	50%	1.5x	71	109x
STAP	Outer Product	Covariance Estimation	26	142	81%	10.3x	1143	11762x
	System Solver	Weight Generation	47	77	77%	10.1x	929	9420x
	Inner Product	Adaptive Weighting	41	84	83%	10.5x	974	10256x
SAR	Interpolation 1	Radar	25	42	65%	6.4x	402	2588x
	Interpolation 2	Radar	21	41	50%	6.5x	528	3437x
	Backprojection	Radar	18	45	82%	6.2x	569	3517x
WAMI	Debayer	Photography	22	124	31%	12.3x	228	2793x
	Lucas-Kanade	Motion Tracking	34	129	51%	4.3x	772	3322x
	Gaussian MMs	Change Detection	25	134	58%	8.1x	107	870x
Required	FFT-1D	Signal Processing	18	43	49%	1.1x	578	642x
	FFT-2D	Signal Processing	18	43	49%	3.1x	1084	3357x
Average			23	68	57%	5x	338	1836x

Table 3.1: PERFECT kernels overview. "Annotations" refer to how many lines of code had to be altered with ACCEPT-style type annotations. "Static Approx. Insn." refers to the total number of instructions that were deemed safe to approximate by ACCEPT. "Dynamic Approx. Insn." refers to the percentage of overall instructions that are safe to approximate over the course of the program execution. "Approx. Runtime Overhead" refers to the slowdown experienced after approximate code injection by QAPPA over the original kernel. "Autotuner Steps" indicates the number of tuning steps taken to find a configuration that could not be approximated further without violating a 40dB quality target. "Autotuner Runtime" indicates how long it takes to tune each kernel as a multiple of its original runtime.

duction?

Section 3.3.5 What dynamic portion of those applications is safe to quantize?

Section 3.3.6 For the set of instructions that can be relaxed, how much precision can be dropped at different quality constraints?

Section 3.3.8 How does increasing the strength of guarantees diminish opportunities for precision re-

duction?

3.3.1 Benchmark Overview and Quality Metrics

PERFECT is a benchmark suite composed of compute-intensive application kernels that span image processing, signal processing, compression, and machine learning. Table 3.1 provides an overview of the PERFECT kernels. For instance, the Wide Area Motion Imagery (WAMI) application represents a typical processing pipeline performed on giga-pixel scale imagery. WAMI comprises an RGB image generation kernel based on the debayer algorithm, an image registration kernel based on the Lucas-Kanade algorithm, and a change detection algorithm based on Gaussian Mixture Models.

PERFECT Application 1 (PA1) is composed of image processing kernels often executed in standard image processing and vision pipelines. It includes a 2d-convolution kernel used in convolutional neural networks, a discrete-wavelet-transform kernel used in image compression and a histogram-equalization kernel.

Space Time Adaptive Processing (STAP) is based on the extended factored algorithm, and is used for mitigating the impact of ground clutter on signals of interest in airborne radar systems. The application includes an covariance estimation kernel, a linear system solver kernel and an adaptive weighting kernel.

Synthetic Aperture Radar (SAR) is a radar-based image formation application for achieving high-resolution imagery. The SAR application is composed of two polar format algorithm interpolation kernels (range and azimuth), and a back-projection kernel.

Wide Area Motion Imagery (WAMI) represents a typical processing pipeline performed on giga-pixel scale imagery. The WAMI application comprises an RGB image generation kernel based on the debayer algorithm, an image registration kernel based on the Lucas-Kanade algorithm and a change detection algorithm based on Gaussian Mixture Models.

3.3.2 Quality Assessment

For quality assessment, we follow the PERFECT manual guidelines for quality assessment [18], and use a uniform Signal-to-Noise Ratio (SNR) quality metric across all benchmarks to measure quality degradation.

$$SNR_{dB} = 10\log_{10}\left(\frac{\sum_{k=1}^{N}|r_k|^2}{\sum_{k=1}^{N}|r_k - q_k|^2}\right) \tag{3.2}$$

The formula used to assess SNR in our benchmarks is provided in Equation 3.2, where r_k and q_k denote the kth reference and quantized output value. SNR provides an average measure of relative error. It is also worth noting that SNR measures error in a logarithmic scale, i.e. an increase of 20dB corresponds to a $10\times$ relative error reduction. Some kernels do not use an SNR metric by default: `gmm` of the WAMI benchmark measures the number of foreground pixels that have been misclassified. For the sake of uniformity, we convert the classification metric to a logarithmic scale. In order to produce a uniform error scale, we convert classification rates to SNR by using the conversion formula in Equation 3.3.

$$SNR_{dB} = 20\log_{10}\left(\frac{pos_{false} + neg_{false}}{pos_{true}}\right) \tag{3.3}$$

3.3.3 Annotation Effort

The QAPPA framework relies on ACCEPT to apply quantization on program instructions that are deemed to be safe to approximate. The set of approximable instructions are identified via data type annotations by ACCEPT, as discussed in Section 3.2.1. ACCEPT dictates that approximations must be applied as an opt-in decision. This places the burden of expressing to the compiler what data can be affected by approximation on the user. We argue that the burden is necessary to ensure the safety of a program [155]. Thankfully, the code annotations effort is reasonable: we counted the amount of code annotations that we had to insert in each PERFECT kernel, which are enumerated in Table 3.1 under the “User Annotations” column. Overall, annotations were

minimal for each kernel. Most of the time, it came down to annotating all floating point variables and integer variables that hold data (as opposed to an address or index) as approximate.

3.3.4 Autotuner Runtime

Table 3.1 summarizes the runtime overhead of the autotuner. The autotuner runtime is dictated by how many steps the autotuner gets to run and how much slower the instrumented approximate program runs at each autotuning step. The goal of the quantization instrumentation step is to faithfully emulate the error resulting from quantization, not to improve performance of the original program. We report an at-most 12.3× slowdown from instrumentation under the "Instrumentation Overheads" column. We report the total number of search steps taken under the "Autotuner Search Step" column where we used a 40dB target.

Table 3.1 summarizes the total autotuner overhead as a multiple of the original program runtime under the "Autotuner Runtime" column. At worst, the autotuner will take 10,000× longer to perform the precision tuning compared to the original runtime, but in the common case it takes about 1000× longer. This runtime overhead isn't too bad considering that we ran the autotuner on microbenchmarks which take less than a second to run, and that this slowdown is comparable to the slowdown that many architectural simulators introduce. The QAPPA autotuner was designed to be run once on programs of interests, but we plan to improve its runtime to make it more practical across more challenging applications.

3.3.5 Approximation Opportunity

Table 3.1 summarizes application characteristics of the PERPECT kernels derived using QAPPA. The "Static Quantized Instruction Count" column lists the number of static instructions that are safe to approximate according to QAPPA. Each approximable instruction serves as a knob that the autotuner can tune to find a precision-minimal configuration that meets quality requirements. The more precision knobs, the larger the search space for the autotuner.

The "Dynamic Quantized Instruction Ratio" is the ratio of approximable instructions to total instructions, measured over the dynamic execution of the target kernel. The higher the ratio,

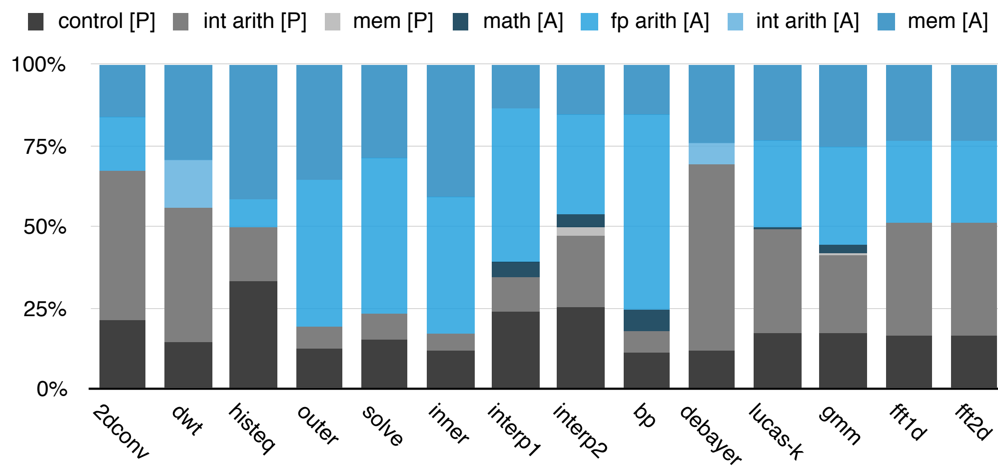


Figure 3.1: Dynamic instruction category mix of the PERFECT kernels. The approximable instructions are colored in shades of blue, and the precise instructions categories are colored in gray.

the larger the opportunity to apply quantization in a given program. Figure 3.1 shows a detailed instruction category breakdown for each PERFECT kernel. Each category is split between approximable and precise classes, which are respectively colored in blue and gray. The approximate instruction ratio is on average 64% which indicates that the PERFECT benchmark suite is a compelling target for approximate computing.

More importantly, the approximable instructions are for the most part composed of *expensive* operations, such as floating-point arithmetic, loads and stores to memory, and standard C math functions (LLVM IR treats math functions as instructions since back-end architectures may or may not have hardware support for those). Most floating-point and memory operations can be approximated. The kernels mostly access memory to store data, rather than pointers, which are more common in graph applications where pointer-chasing is necessary. The PERFECT kernels mostly perform regular bulk data processing which is not only compelling for hardware acceleration, but also for approximate computing. The bulk of the precise instructions are composed of control instructions and integer arithmetic used for address computation, neither of which can

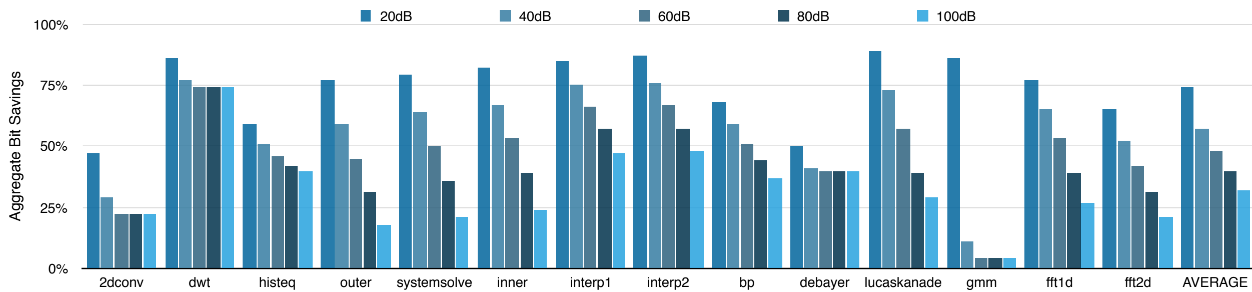


Figure 3.2: Aggregate bit-savings for 14 PERFECT kernels over a 20dB to 100dB SNR range.

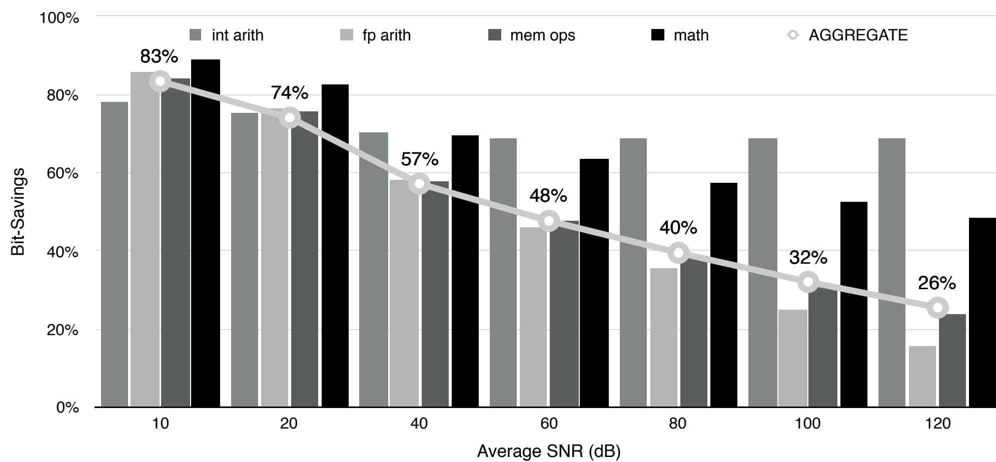


Figure 3.3: Bit-savings vs. SNR averaged over PERFECT kernels, for integer arithmetic, FP arithmetic, memory ops and math functions.

be approximated without compromising the safety of the program. That said, the PERFECT kernels are highly regular in terms of memory and control flow divergence. Consequently address computation and control can easily be handled by simple finite-state machines in hardware accelerator designs, and their energy overheads should remain small next to arithmetic, and memory operations.

3.3.6 Bit Savings

Figure 3.2 shows the aggregate bit-savings obtained on approximable instructions that QAPPA was able to obtain on each PERFECT application kernel, on SNR targets from 100dB down to 20dB (0.001% up to 10% average relative error). In general, the lower the quality target, the higher the bit-savings. On average, a 74%, 57%, and 48% average bit-savings can be obtained at 20dB, 40dB and 60dB respectively (10%, 1%, and 0.1% average relative error). We observe that integer benchmarks (`2dconv`, `dwt`, `histeq`, and `debayer`) offer relatively high bit-savings at high SNR requirements (100dB). This is indicative of the common use of wide integer types (e.g. 32-bit) to handle narrow pixel data (e.g. 8-bit) for image processing benchmarks (`PA1`). We also notice that `changedet` provides minimal bit-savings until we lower error to 40dB and 20dB error (1% and 10% misclassification rate). The remaining floating-point kernels all exhibit a smooth tradeoff relationship between bit-savings and quality. We observe that quantization can meet very stringent quality thresholds that are often not achievable with other approximation techniques. For instance, the PERFECT manual recommends 100dB (0.001% relative error) degradation as a quality target from applying compiler optimizations. We do not know of any approximation techniques that can meet such stringent accuracy guarantees.

Figure 3.3 shows the average bit-savings obtained across the PERFECT benchmark for approximable integer arithmetic, floating-point arithmetic, memory ops and standard math functions over the same range of SNR targets. Again, integer arithmetic has an overall higher bit-savings at 100dB SNR target because trimming off the MSB bits of many integer variables has no effect on output quality. Math functions also exhibit high bit-savings at 100dB but for different reasons: most standard math in the PERFECT benchmark suite uses double-precision implementations which is generally overkill over our range of quality targets. All benchmark exhibit a smooth quality vs. bit-savings trade-off curve. which offers opportunities for energy, bandwidth and storage savings in hardware designs.

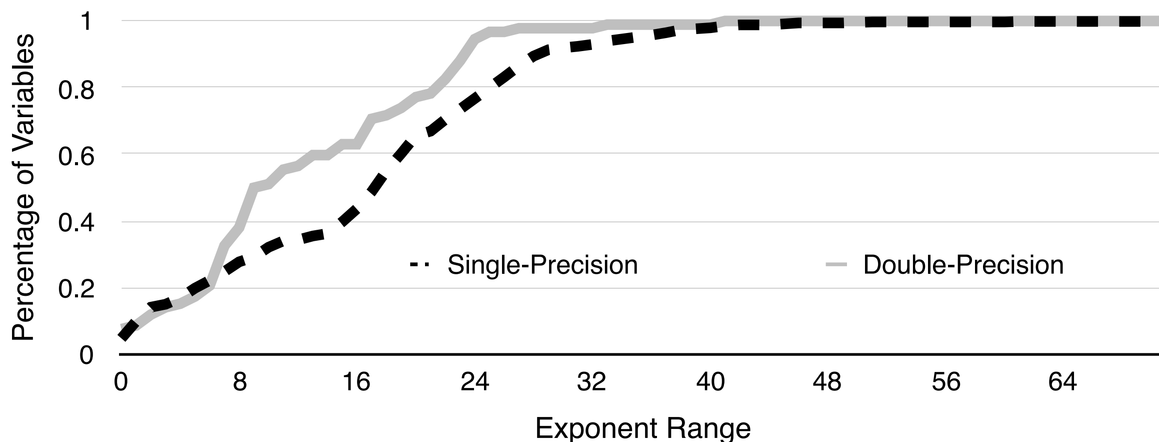


Figure 3.4: CDF of exponent value range of all floating-point variables in the PERFECT benchmark suite.

3.3.7 Fixed Point Conversion

We apply fixed-point conversion of the floating-point PERFECT kernels to evaluate the viability of using fixed-point hardware exclusively. This type of analysis would be beneficial to a hardware designer or an FPGA programmer who wants to produce a highly efficient fixed-point data-path for a given kernel.

Figure 3.4 shows a histogram of the exponent range for all of the PERFECT benchmark suite variables, when executing each program on the provided input sets. Over 92% and 97% of single-precision and double-precision floating-point variables have a binary exponent range below 32, while the IEEE754 floating-point supports exponent ranges of 255 and 2047 for single-precision and double-precision floating-point respectively (excluding sub-normals). This shows that most PERFECT kernels can be ported to fixed-point without affecting quality too drastically.

3.3.8 Guarantees

In Section 3.2.4, we discussed two ways to express QoR guarantees: empirical tests — used so far in this evaluation — and statistical tests, which we discuss in this section. Statistical error

PA1 Kernel	medium quality (20dB)		high quality (40dB)	
	conf > 90%	conf > 99%	conf > 90%	conf > 99%
2D Conv.	-4.10%	-13.25%	-4.37%	-8.73%
DWT	-12.50%	-22.47%	-2.51%	-2.73%
Hist. Eq.	-3.02%	-7.35%	-2.91%	-6.76%

Table 3.1: Bit-savings loss from using a empirical guarantee to statistical guarantee at 90% and 99% confidence. We vary the quality target at medium (20dB) a high (40dB) settings on the PA1 kernels.

guarantees capture the uncertainty that arises from measuring error in a non-exhaustive way. To express a statistical guarantee, the user needs to provide an error threshold δ, and a confidence threshold $1-\varepsilon$. QAPPA then applies the Clopper-Pearson (CP) test to ensure that both δ and $1-\varepsilon$ are satisfied.

Demanding higher confidence leads to more conservative precision relaxations and thus lower bit-savings. We conduct an experiment to quantify the loss in bit-savings when demanding a statistical guarantee at different confidence levels. The baseline bit-savings for this experiment is obtained using empirical error guarantees. We chose the PA1 kernels to conduct our experiment for two reasons: (1) it was straightforward to produce a generative model for image data, and (2) processing each image requires hundred of thousands of kernel invocations which provided enough samples for QAPPA to run the CP test on at high confidence levels.

We conduct our experiment at two quality levels: a medium quality setting at 20dB (10% error) and a high quality setting at 40dB (1% error). Table 3.1 shows the bit-saving loss at two confidence levels ($1-\varepsilon=\{90\%, 99\%\}$), relative to the bit-savings obtained with empirical guarantees. We evaluate the bit-savings loss using both quality levels, with error thresholds ($\delta=\{10\%, 1\%\}$). Overall, we notice a reduction in bit-savings going from empirical guarantees to statistical guarantees, as the confidence interval increases. These results confirm that stronger statistical guar-

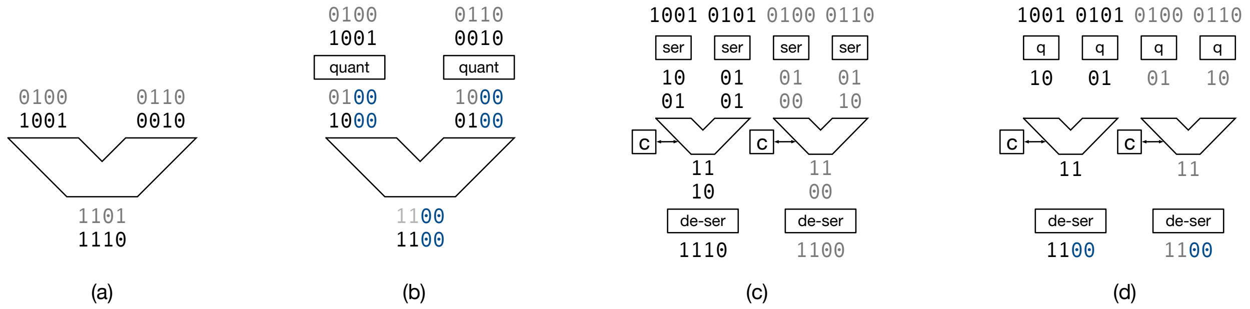


Figure 3.1: Quantization scaling mechanisms overview. (a) Default wide addition on wide adder. (b) Narrow addition on wide adder. (c) Wide addition on narrow adder (d) Narrow addition on narrow adder.

antees diminish bit-savings returns.

To complete our sensitivity analysis, we specify worst-case relative error bounds, to guide QAPPA's precision minimization search, at error thresholds of 10% and 1%. This corresponds to a confidence level of 100%. In both cases, no reduced-precision configuration meets either worst case guarantee. This indicates that using worst-case error measurement are too pessimistic to be of practical use. In general, we recommend the use of statistical guarantees, which strike a balance between strength of error guarantee, and expected bit-savings.

3.4 Dynamic Quantization Scaling

We survey *dynamic quantization* mechanisms in hardware and discuss the savings in arithmetic energy and memory bandwidth that these mechanisms achieve on hypothetical accelerator designs executing the PERFECT kernels. We isolate the subsystems that are affected by quantization, namely the arithmetic and the memory subsystems. *Arithmetic energy* denotes the fraction of energy that is consumed by arithmetic units in a given hardware design, e.g. ALUs and processing elements. What this study does not focus on are control overheads, which are specific to a given hardware implementation.

The aim of this study is to motivate the adoption of quantization scaling mechanisms in hard-

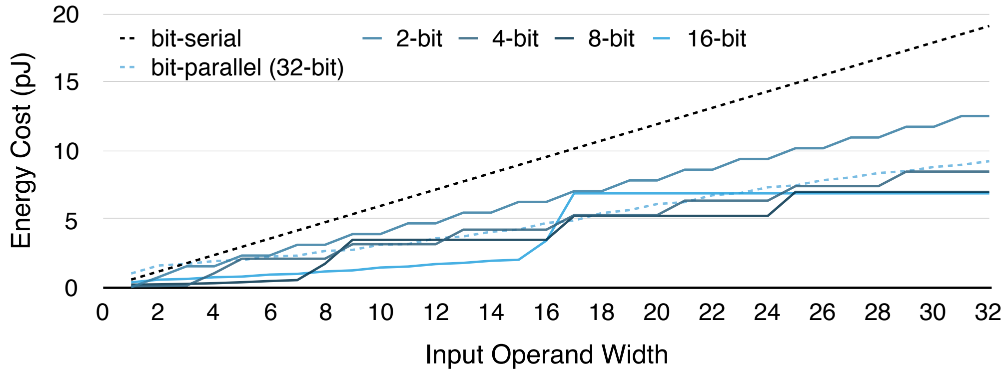


Figure 3.2: Energy vs. precision relationship for precision-scaled multiplier designs (32 bit baseline).

ware accelerators, where data bandwidth requirements far surpass the instruction bandwidth requirements. General purpose processors spend much of their energy budget in instruction fetching and decoding. Augmenting the ISA of a general processor with bit-granular quantization settings would counteract much of the energy savings that quantization would enable. Thus, this survey targets designs such as vector processors, systolic arrays, or fixed-function accelerators that could incorporate dynamic quantization scaling mechanisms in order to respond to dynamic energy or quality constraints.

3.4.1 Scaling Quantization in Compute

We evaluate two quantization scaling hardware mechanisms that provide energy reduction on quantized arithmetic operations. The first technique, operand narrowing, aims to minimize power by reducing transistor switching [186] on wide compute units. The second technique, bit slicing (or operator narrowing), utilizes narrow compute unit in parallel to time-multiplex the computation of wider operations, effectively scaling throughput with precision on data-parallel workloads [92]. We compare the energy savings obtained by each technique at different operand quantization levels, over a standard 32 bit arithmetic unit.

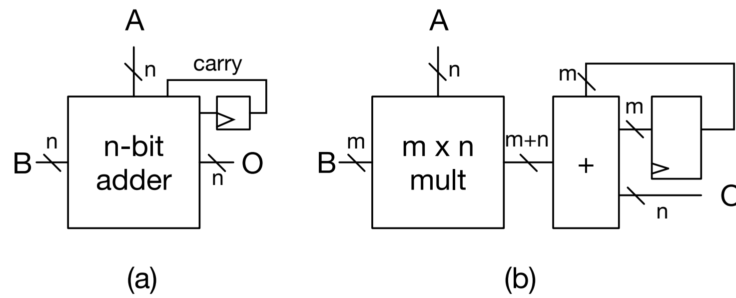


Figure 3.3: Simplified schematic of (a) bit-sliced adder and (b) bit-sliced multiplier.

Reducing Power with Operand Narrowing Operand narrowing is a precision scaling technique that can reduce dynamic switching in standard bit-parallel arithmetic units [186]. The idea is to apply quantization on the input operands of the arithmetic units by zeroing the LSBs that correspond to the desired quantization level. This in turns limits the amount of transistor switching in the arithmetic unit's logic, as the lower slices of the datapath remain inactive.

Figure 3.1.b shows how operand narrowing sets the least significant bits (LSBs) of the input operands to zero, to underutilize the arithmetic unit's lower slices. LSB-zeroing is the precision scaling mechanism proposed in the Quora vector processor [186]. While operand narrowing reduces the amount dynamic power, it does not provide throughput improvements. Next, we discuss a quantization scaling technique that achieves throughput scaling when data parallelism is available.

Increasing Throughput with Bit Slicing Bit slicing is a technique used to perform wide arithmetic operations using narrower arithmetic units. The advantage of bit slicing lies in its ability to scale throughput nearly linearly with precision requirements. Given an narrow n bit adder, a wide m bit addition can be done in $O(m/n)$ time, while an m bit multiplication can be done in $O(m/n)$ time on a an $m \times n$ multiplier. The simple design of a bit-sliced adder and bit-sliced multiplier is shown in Figure 3.3. Bit slicing reduces arithmetic unit power while increasing computational delay, thus making baseline precision computation on a wide ALU and a bit-sliced ALU

roughly equivalent in terms of energy. Bit slicing excels at reducing energy at lower-precisions settings, since lower precision lead to lower computation delays. Bit slicing comes at a cost however, which we will refer to as the *bit-serialization tax*. The bit-serialization tax is attributed to the extra registers needed to time-multiplex a narrow compute unit for wide computation. The additional hardware requirement can be seen in Figure 3.3 as a small register in the bit-sliced adder case and an m-bit register and $m + n$ bit adder for the bit-sliced multiplier. In addition, increasing the delay of a given operation has negative effects outside of the ALU itself, as the rest of the hardware needs to remain powered on. Bit slicing is best applied in applications that have SIMD parallelism, where bit parallelism can be exchanged for increased SIMD parallelism. This results in designs that have similar area footprint and the ability to dynamically increase throughput as precision requirements go down [92]. Finally bit slicing can achieve the added benefit of reducing critical path delay in some hardware designs. This in turn allows hardware designers to increase the maximum frequency of their designs if the critical path was previously in one of the arithmetic units of the design.

3.4.2 Quantization Scaling Energy Evaluation

Methodology We synthesize adder and multiplier designs of varying widths using the Synopsys Design Compiler with the TSMC-65nm library. To model power, we collect switching activities in simulation when adding/multiplying input operands streams of varying widths, from 1 bit to 32 bits. We set a target frequency of 500MHz and perform place and route on each simulated design with ICC. We use PrimeTime PX to accurately model the impact that switching activity has on power.

Multiplier Case Study We evaluate the energy cost of performing arithmetic operations on input streams with varying bit widths. The energy per operation vs. input width relationship for a 32-bit multiplier design is shown as a dotted black line in Figure 3.2. The linear increase in energy reflects an increase in switching activity when the multiplier processes wider input operands.

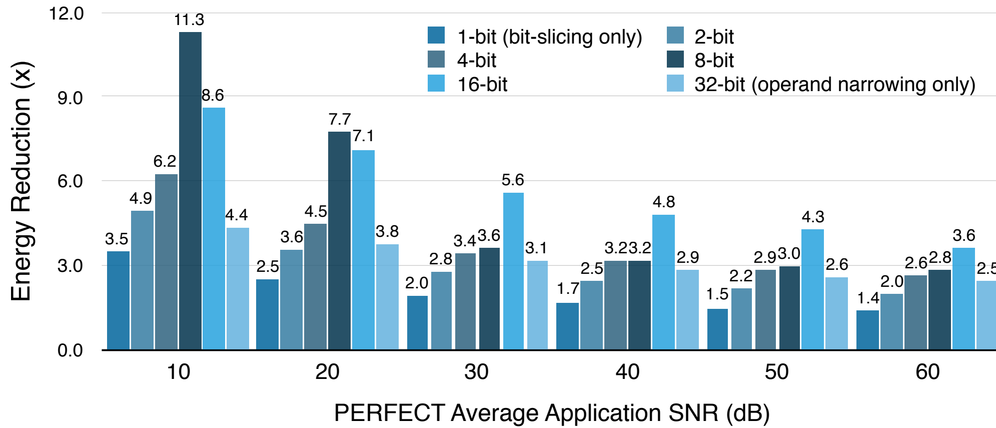


Figure 3.4: Arithmetic energy reduction on the PERFECT benchmark at different bit slicing granularities and at different SNR targets (higher is better).

Next we look at bit slicing: we vary the granularity at which computation is sliced from 1 bit (bit serial) to 32 bits (bit parallel). The relationship between the energy cost and the input width for a 32 bit multiplier is shown as colored lines in Figure 3.2 for different bit slicing granularities. When the input operand width is narrower than the arithmetic unit width, the energy scales linearly with the input width because of lower switching activity. Conversely, when the input operand width exceeds the width of the serial arithmetic unit width, the energy increases discretely at every n-bit increments, where n denotes the width of the slice. Bit-serial evaluation – i.e. arithmetic unit width of 1 – is a corner case where the relationship between energy and operand width is linear. It is worth noting that no single slice width produces better efficiency than others across the entire input widths range.

Energy Evaluation on PERFECT We use the PERFECT benchmark suite to guide our choice of an energy-optimal precision scaling mechanism at different quality targets from 60dB down to 10dB.

Figure 3.4 shows energy savings across all PERFECT benchmarks over a standard arithmetic unit executing 32 bit arithmetic operations. Performing operand narrowing exclusively as in Quora [186] on a bit-parallel arithmetic unit results in significant energy reduction over the pre-

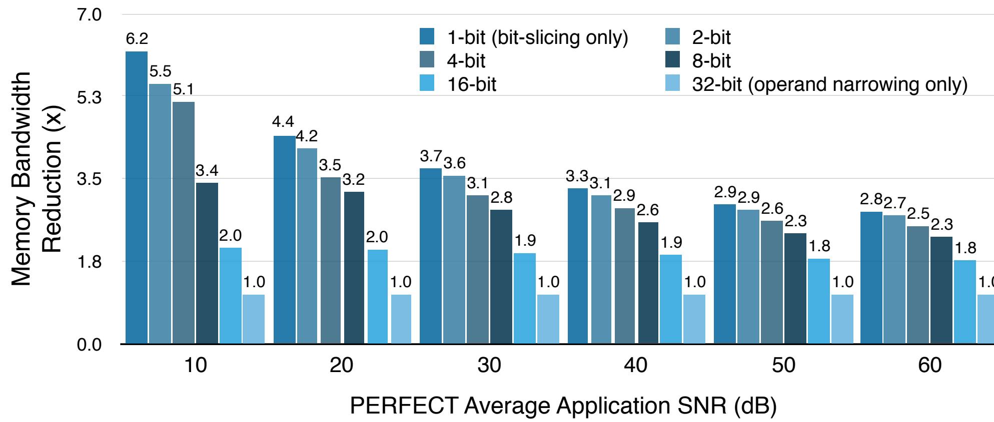


Figure 3.5: Ideal bandwidth reduction on PERFECT benchmark suite at different data packing granularities and at different SNR targets (higher is better).

cise, non quantization scalable baseline: $3.8\times$, $2.9\times$ and $2.5\times$ at 20dB, 40dB and 60dB respectively. These energy reductions are improved by combining bit slicing and operand narrowing: a slice width of 16 bits yields optimal energy reductions by $3.6\times$ and $4.8\times$ at 40dB and 60dB while a slice width of 8 bits yields $7.7\times$ energy reduction at 20dB over the baseline arithmetic unit. Finally, we make the observation that applying bit slicing at a 1 bit granularity yields suboptimal energy results at all quality targets.

3.4.3 Scaling Quantization in Memory

Much of the energy spent in processors and accelerators is associated with data movement to and from memory [42, 77]. Scaling precision in programs can help mitigate memory bandwidth requirements.

Data packing can maximize bandwidth efficiency at arbitrary precision settings. Recent work has proposed hardware packing and unpacking mechanisms to store variable precision weights in neural network accelerators [91]. The idea is to store variable precision data into fixed-width memory, by packing data at a coarse granularity (e.g. an array of coefficients) to mitigate overheads. Figure 3.6 shows how reduced precision data can be efficiently padded in fixed-width

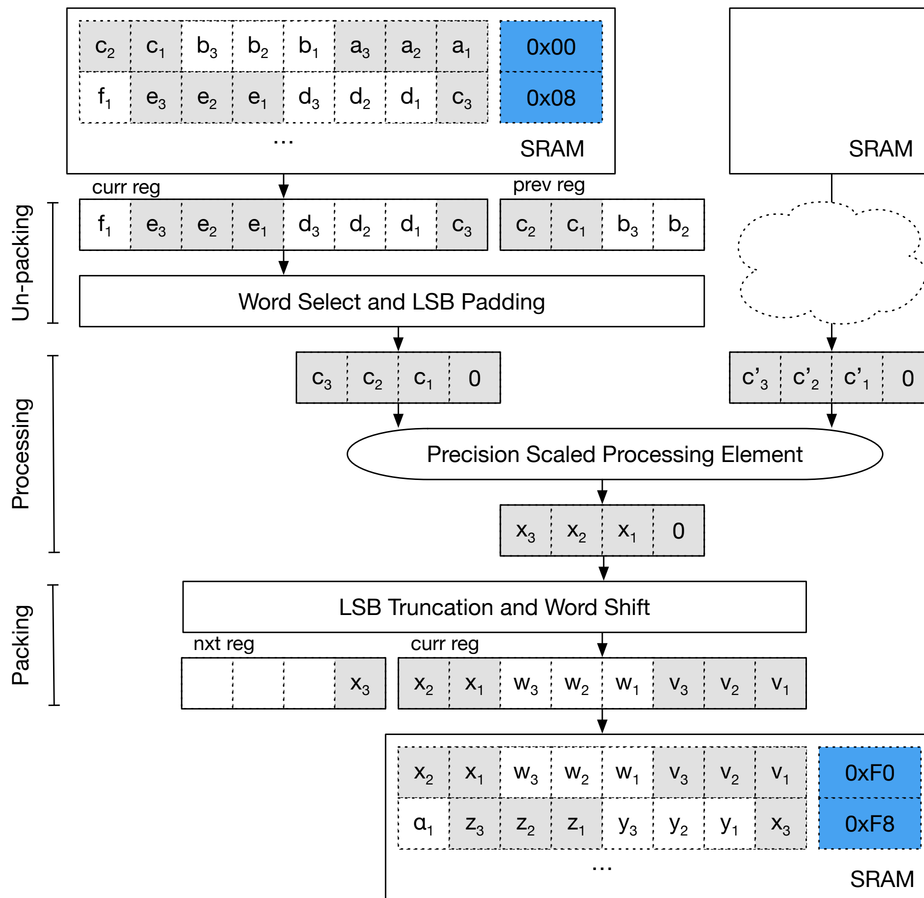


Figure 3.6: Example of quantization-scalable pipeline: memory packing and unpacking mechanism used in Proteus [91] combined with operand narrowing used in Quora [186]. The input and output data can be loaded in its packed format to save memory bandwidth.

SRAM modules, unpacked for processing, and re-packed before being stored to SRAM again. This results in more effective use of bandwidth and storage, but adds complexity when accessing data. This complexity can be mitigated in hardware accelerators that perform regular data access on large portions of memory, where precision settings can be set on coarse structures. We assume that the data is read and written to DRAM in a dense format, simplifying the on-chip to off-chip storage communication pipeline.

Applying quantization to data can significantly reduce memory bandwidth. Figure 3.5 shows

bandwidth savings on a cache-less accelerator. We vary the data packing granularity from 1 to 32 bits and derive the resulting bandwidth reduction. A data packing granularity of 1 bit can achieve $4.4\times$, $3.3\times$, and $2.8\times$ average memory bandwidth reduction on the PERFECT kernels at 20dB, 40dB and 60dB. Data packing at fine granularities can increase both software and hardware overheads for packing and unpacking. A hardware designer might therefore want to align the data packing granularity with the bit slicing width of the precision scalable compute units to minimize control overheads. The optimal data granularity can be determined by the target system energy breakdown between memory, computation, and control which differs for different classes of accelerators and workloads.

3.5 Approximation Study

Approximate Computing encompasses a wide variety of software and hardware techniques that expose quality-efficiency trade-offs in compute-intensive applications. It seems fitting given the emergence in recent approximate computing trends to compare how the classical approach of fine-grained precision minimization compares with more recent proposals of approximate computing optimizations.

In this section, we conduct a comparative evaluation of approximation techniques. We evaluate precision reduction against nondeterministic voltage overscaling [127, 60] and coarse grained neural approximation [171], and compare the quality vs. energy tradeoffs achieved with each technique.

3.5.1 Voltage Overscaling

We compare the energy savings obtained by quantization against voltage overscaling and contrast the energy savings obtained at different quality targets on the PERFECT benchmark kernels.

Motivation: Determinism vs. Nondeterminism Nondeterministic approximations can introduce errors in a random or pseudo-random fashion [127, 60, 34, 108, 199]. While nondeterministic approximations pose a testing and debugging challenge, they can be modeled using proba-

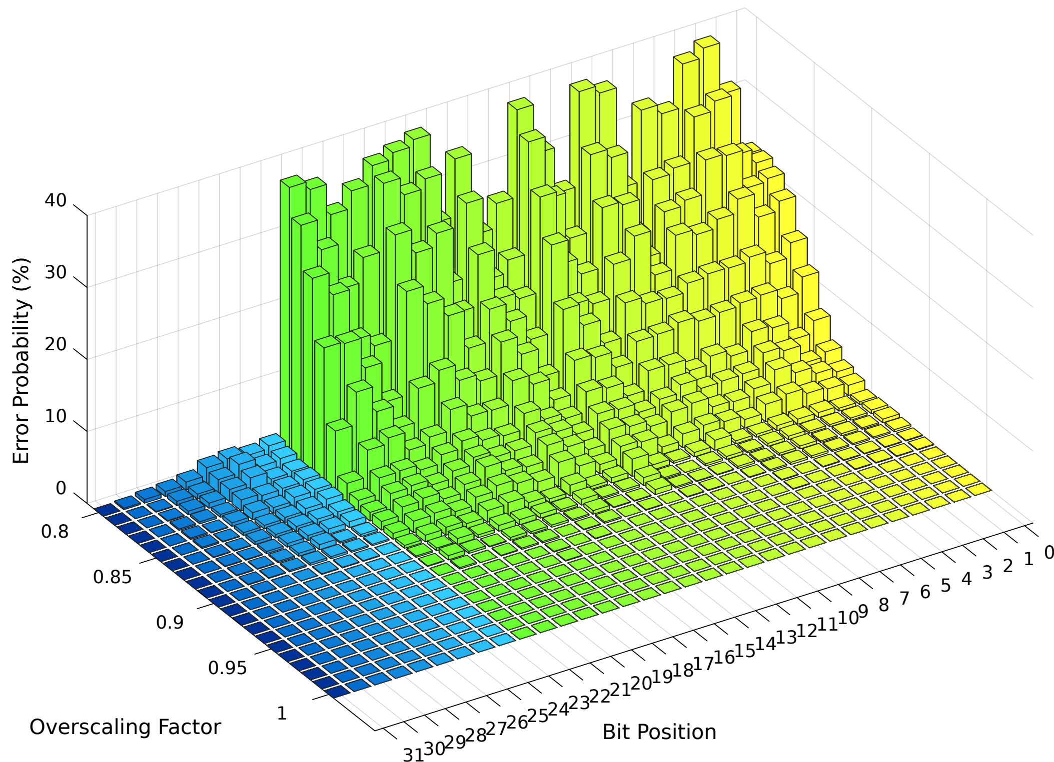


Figure 3.1: Bit-flip probabilities of each output bit for a single-precision floating point adder at voltage overscaling factors [0.8-1.0]. Sign and exponent bits are in blue, mantissa bits are in green/yellow.

bilistic distributions [157]. We investigate nondeterministic *voltage overscaling*, a popular approximation technique that reduces compute power at the risk of increasing timing violations. Our evaluation of voltage overscaling relies on (1) characterizing the energy vs. error relationship of voltage overscaling and (2) analyzing how low level timing violations affects application quality.

Characterizing Overscaling Error We quantify the effects of voltage overscaling on fixed point and floating point arithmetic designs taken from the Synopsis DesignWare IP library. We simulate those circuits in CustomSim-XA, built on top of FastSpice to perform transistor level power and fault characterization. The circuits are built in Synopsys Design Compiler with a 65nm

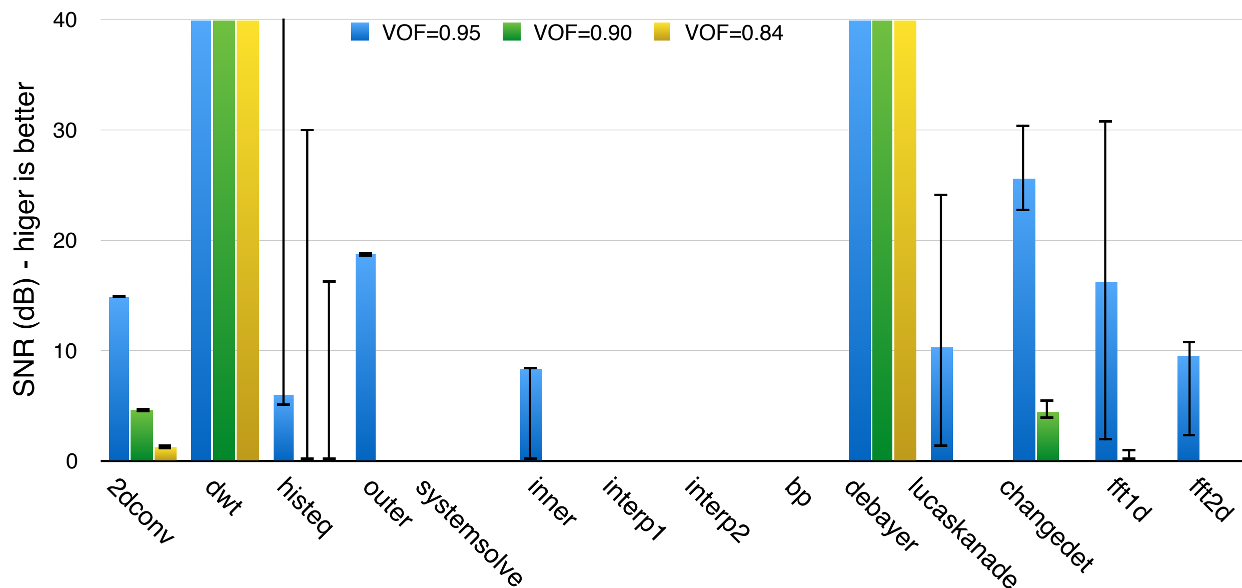


Figure 3.2: PERFECT kernel SNR at voltage overscaling factors of 0.95, 0.90 and 0.84 corresponding to 10%, 20% and 30% energy savings. SNR is measured collected over 100 runs, values represent median SNR, and error bars represent min and max error.

process and synthesized using a timing constraint of 2GHz. Registers latch the inputs and outputs of the arithmetic units and a synchronizer is used to settle errors caused by metastability. We synthesize a parallel prefix architecture for the fixed point adder and a Booth-encoded Wallace-tree architecture for the fixed point multiplier. We generate 10^5 random input pairs as stimuli to the circuits and profile timing violation errors at three representative voltage overscaling factors ($0.95\times$, $0.90\times$, and $0.84\times$), corresponding to 10%, 20%, and 30% power savings respectively. We measure the probability of a timing violation induced bit-flip for each output bits to produce a statistical error model of the voltage overscaled circuit. Figure 3.1 shows the bit-flip probability distribution for a floating point adder, measured at different voltage overscaling factors, with different color coding to highlight the sign, exponent and mantissa bits.

Comparative Evaluation on PERFECT We feed the error models derived above into QAPPA's error injection framework to quantify the effect of voltage overscaling on the application output.

We execute each benchmark 100 times on the same input data to obtain an error distribution.

The results of the experimental runs are displayed in Figure 3.2 and show the effects of voltage overscaling on application quality at 10%, 20% and 30% energy savings. Applying the same voltage overscaling factor to each PERFECT kernel can lead to vastly different errors because of nondeterminism. Integer benchmarks such as `dwt` and `debayer` are mostly unaffected by overscaling. The integer circuits have shorter critical paths than their floating point counterparts, and therefore are less affected by voltage overscaling. Other benchmarks including the SAR kernels and `systemsolve` produce data that contain erroneous output values (`inf` and `NaN`) which lead to a 0dB SNR. Voltage overscaling does well on simple single-stage functions (`2dconv`), in which errors have localized effects. Multi-stage kernels (`lucaskanade`) on the other hand pose a challenge since errors can propagate and snowball into large output errors.

Discussion Quantization provides better energy efficiency at preferable SNR levels for all PERFECT kernels. In addition, the deterministic nature of quantization allows for sounder guarantees and more predictable behavior. We conclude that *is it difficult to justify incorporating voltage overscaling in hardware designs without some form of error correction.* The unbounded errors simply don't justify the energy savings. A hybrid approach of combining fine grained precision requirements with error correction mechanisms proposed in [57] could selectively correct a timing violation error based on what bits are affected, thereby reducing the amount of hardware rollbacks. We reserve the evaluation of such error correction mechanisms for future work.

3.5.2 Neural Approximation

We discuss how quantization scaling could improve the efficiency and programmability of programmable accelerators and compare the energy benefits of quantization against neural approximation. Neural approximation has limited applicability when it comes to approximating arbitrary functions at arbitrarily low error levels. We evaluate the AxBench [198] benchmark suite at suggested error levels (10% relative) to ground the comparison between quantized acceleration and neural acceleration.

Figure 3.3: Approximating the inverse kinematics kernel: (a) default DFG, (b) optimized fixed point DFG with PWP, (c) neural approximation DFG. Operations that read data from local SRAM are colored in gray.

Motivation: Fine vs. Coarse Approximation Coarse grained approximation attempts to approximate an entire code region using a regression model (e.g. polynomials, neural networks). Neural acceleration [61, 171] uses neural networks to approximate functions via learning, and utilizes hardware accelerators for efficient execution. Much of the previous studies on neural acceleration have not isolated the efficiency gains attributed to specialization from approximation.

We compare two approximation approaches: (1) fine grained approximation with piecewise polynomial (PWP) approximation of math functions, and (2) coarse grained approximation with neural approximation. In both cases, we assume a hardware accelerator composed of fixed point adders, multipliers and local SRAM storage. We quantify arithmetic energy, and SRAM requirements to draw a cost comparison between the two techniques. We motivate our study with the inverse kinematics (`inversek2j`) function example, which dataflow graph (DFG) is shown in

Figure 3.3.a.

Comparative Evaluation on AxBench We use QAPPA to derive the quantization requirements in each target application at the error rate recommended by AxBench. Quantization provides an opportunity to significantly reduce the cost of standard math function invocations. We leverage QAPPA to derive the accuracy requirements and the input range of standard math functions (e.g. cos, $sqrt$, reciprocal etc.) in each target program. We use those requirements to produce piecewise polynomial approximations with a custom math approximation toolbox that we built in Python. The degree of the polynomial dictates computational requirements, while the number of pieces dictates the memory requirements for storing the polynomial coefficients. The DFG of an example quantized program is shown in Figure 3.3.b. In this example, all nonlinear operators (represented as circles) have been replaced replaced with a piecewise degree-one polynomial approximation.

We run our study on a set of AxBench [198] benchmarks due to the limited applicability of neural approximation on the PERFECT benchmark kernels. We exclude `jmeint` and `jpeg` from our study since the input dimensionality of these kernels is too high to be approximated with coarse PWP approximation. We assume a spatially laid-out accelerator design (i.e. each static instruction is mapped to a single processing element, or load/store unit) for each approximation technique and measure hardware efficiency in two key metrics: (1) compute energy and (2) SRAM storage requirements. We use the RTL computation cost models obtained in Section 3.4 to analytically evaluate energy costs associated with each approximate acceleration technique. We quantitatively measure on-chip SRAM requirements for storing the neural network weights, and piecewise polynomial approximation coefficient tables. Finally we use the neural approximation errors reported in previous literature [171] as quality targets for quantization.

We use two modeling assumptions to estimate the computation and storage costs of neural acceleration. The realistic model based on digital implementations of NPUs [61, 126] assumes 16 bit weights, and a 16-piece linear approximation of the activation function. The optimistic model assumes 8 bit weights, a linear activation function and no quality loss with respect to the realistic

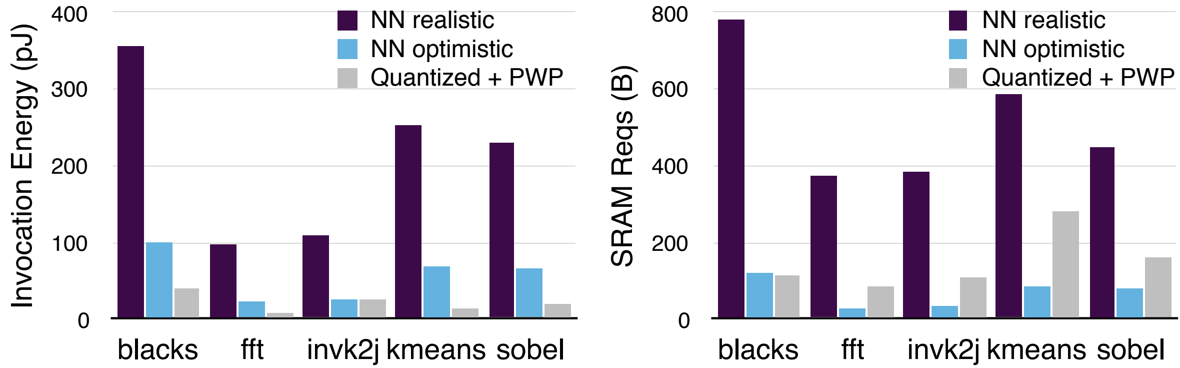


Figure 3.4: Energy and storage comparison of quantized acceleration vs. neural acceleration on AxBench kernels (lower is better).

model. We leverage QAPPA to produce a reduced precision quantized program specification for each AxBench kernel. Our compute cost model assumes a quantization scalable 8 bit ALU, that applies either operand narrowing or bit-serial computation depending on the quantization requirements.

We summarize our evaluation of neural approximation vs. reduced precision acceleration in Figure 3.4. Reduced precision acceleration is more energy-efficient than neurally approximated acceleration for all of the reviewed AxBench kernels. The storage requirements of the quantized kernels lie between the realistic and optimistic neural network accelerator cost models, except for `blackscholes` where quantized acceleration beats neural acceleration in both cost modeling scenarios.

Discussion While there is not a clear answer as to which technique is more efficient in terms of both energy and storage, we can claim that quantized acceleration offers comparable efficiency benefits to neural acceleration. Neural acceleration provides the benefit of programmability as it requires one hardware accelerator to evaluate any neurally approximated piece of code [126]. Conversely, neural networks have limited success at approximating code at arbitrarily low error levels, as there are no examples in literature that show successful approximations with neural

networks below 1% relative error [61, 171, 126, 71]. Quantization and PWP approximation could improve programmability in spatial accelerators by simplifying complex operators such as math functions, down to simple linear operators. The simplified kernel can then be more easily mapped onto a programmable acceleration substrate composed of simple arithmetic functions [21, 129]. Finally, improving the quality guarantees of neurally-approximated programs is the object of much on-going research and remains a challenge for high-dimensional functions [95, 71, 114, 96]. Quantization on the other hand benefits from mature numerical analysis frameworks that provide error analysis and guarantees which programmers are familiar with [54, 53, 184].

3.6 Related Works

Tools and Frameworks Precimonious [150] is a dynamic program analysis tool that suggests cheaper floating point operations to improve the performance of floating point-heavy functions. QAPPA differs from Precimonious in that it supports approximate type qualifiers to ensure program safety, and that it applies arbitrary quantization to either floating point or integer types. AHLS [107] is a high-level synthesis framework for synthesizing RTL implementations of energy efficient circuits given a high-level C specification. Similarly to QAPPA, AHLS considers narrow fixed-point operations to minimize the overall energy of a given circuit. While AHLS focuses on synthesizing fixed-function and fixed-precision accelerators, QAPPA serves as a framework to navigate quality-efficiency tradeoffs for precision-scalable accelerators. Approxilyzer [185] helps improve hardware resiliency to approximation errors by quantifying the impact of single bit errors on output quality. QAPPA assumes deterministic value truncation or rounding as opposed to random bit-errors. QAPPA is not focused on improving resiliency, but rather aims to expose opportunities to reduce energy and bandwidth in hardware accelerators.

Error Guarantees Approximate computing has embraced statistical guarantees [114, 157] to provide common-case error bounds. Our work inspires itself from past work to provide statistical error bounds. Numerical analysis exploits interval analysis [54, 53] to reason about quantization and rounding errors in floating point programs. dco/scorpio [184] is a framework that automates

significance analysis to identify computation tasks that have high contribution to output quality. QAPPA could be augmented with such frameworks to provide stricter error bounds.

Precision-Scaling Hardware Techniques Quora [186] is a precision scalable SIMD architecture that delivers energy precision trade-offs in parallel applications. Stripes and Proteus [92, 91] propose precision scalable compute and storage mechanisms that can improve the energy efficiency of DNN accelerators. QAPPA can be used as a software compiler for such precision scalable architectures, by automatically deriving precision requirements and providing statistical guarantees. Our comparative evaluation of precision-scaling mechanisms aims to motivate the inclusion of precision scalable architectures like Quora and Stripes.

3.7 Conclusion.

We present QAPPA, a framework that fine-tunes quantization requirements of C/C++ programs, while meeting user defined, application level quality guarantees. We analyze the PERFECT benchmark suite with QAPPA and find that much precision can be discarded while meeting reasonable quality targets. We evaluate hardware mechanisms that can reduce compute energy and memory bandwidth in hardware accelerator designs. We then perform a comparative study of quantization as a viable alternative to voltage overscaling and neural approximation. We show that precision reduction rivals these techniques in terms of energy savings, while exhibiting predictable error and providing practical quality guarantees.

Chapter 4

VTA: FLEXIBLE ARCHITECTURE & RUNTIME CO-DESIGN FOR DEEP LEARNING SPECIALIZATION

"Anyone can build a fast CPU. The trick is to build a fast system."

– Seymour Cray

Work under submission to ISCA 2019 with the collaboration of Thierry Moreau, Tianqi Chen, Lianmin Zheng, Eddie Yan, Josh Fromm, Luis Vega, Jared Roesch, Ziheng Jiang, Luis Ceze, Carlos Guestrin and Arvind Krishnamurthy.

Abstract *Specialized Deep Learning acceleration stacks, designed for a specific set of frameworks, model architectures, operators, and data types, offer the allure of high performance while sacrificing flexibility. Changes in algorithms, models, operators, or numerical representations pose the risk of making custom hardware quickly obsolete. We propose VTA (Versatile Tensor Accelerator), a customizable deep learning architecture designed to be flexible in the face of evolving workloads. VTA achieves this flexibility via a two-level ISA: (1) a task-ISA to explicitly orchestrate concurrent compute and memory tasks and (2) a microcode-ISA which lets us describe a wide variety of operators with single-cycle tensor-tensor operations. Effective use of VTA's two-level ISA is enabled by a complete runtime system equipped with a JIT compiler for flexible code-generation. Our evaluation demonstrates that VTA can be integrated into a complete state-of-the-art deep learning stack, providing flexibility for diverse models and divergent hardware backends. We deploy optimized deep learning models used for object classification and style transfer, on edge-class FPGAs. We propose a flow that performs hardware and schedule design space exploration to generate a customized hardware architecture and software library that can be leveraged by learning frameworks. We show that VTA*

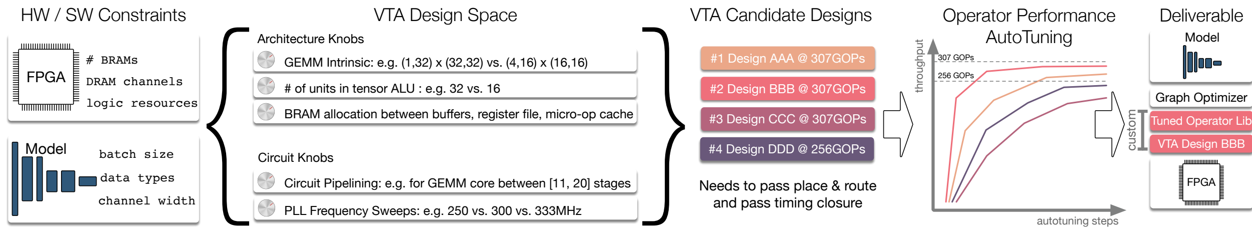


Figure 4.1: VTA provides flexibility with respect to hardware targets and deep learning models. This flow diagram shows the steps in adapting a given model to a hardware backend by exploring VTA hardware configurations, and performing operator autotuning on the top hardware candidates (section 4.5). This process generates the pieces necessary to deploy VTA in any deep learning framework (section 4.2).

deployed on an Xilinx Ultra96 FPGA outperforms highly optimized edge deep learning libraries on Mali-T860 GPUs.

4.1 Introduction

Deep learning has triggered a revolution in computer vision, natural language processing, speech processing, and artificial intelligence [105]. These advancements have been achieved by steady progress in collecting better data, training better models, and designing better systems. In particular, hardware specialization [42, 77, 90, 41] has been instrumental for improving the capabilities of systems in the datacenter and on the edge. Unfortunately, the gains hardware specialization provides in terms of efficiency come at the cost of flexibility. The evolution of deep learning workloads, via new network architectures [101, 151, 145, 165, 139], new operators [80, 47], new numerical representations [100, 103, 142], and new algorithmic optimizations [82, 78, 104] can rapidly push deep learning accelerators into obsolescence. In order to reconcile fast progress in deep learning workloads with the explosion of specialized architectures, we need a software-hardware co-design approach which strikes a balance between flexibility and specialization.

Despite the multitude of hardware/software systems in industry [90, 3], open-source deep

learning accelerator designs [180, 131, 181], and model-to-ASIC/FPGA compilers [161], we argue that supporting *flexibility* is a challenge that takes several forms and remains to be solved. Supporting novel deep learning models and operators requires cross-stack modifications to the compiler, ISA, and as far down as the micro-architecture. For example, supporting custom data types requires proper handling of data layout requirements across the software stack and operator support in hardware. Changing either a hardware target, FPGA device, or ASIC technology node creates another set of challenges in terms of logic, memory resources, and bandwidth allocation and management.

In turn, hardware changes affect how software libraries are written, scheduled, and optimized. As a result, taking off-the-shelf deep learning hardware designs not engineered for customization typically requires tedious modifications across the hardware and software stack to adapt to new workloads. Performing modifications for each new workload is time consuming and unsustainable.

This chapter advocates for a sustainable approach to building and updating a specialized deep learning systems stack, as workloads evolve over time. We propose VTA (Versatile Tensor Accelerator), a flexible deep learning architecture co-designed with a runtime that delivers the benefits of hardware specialization while exposing a programming interface that accommodates changing workloads. The key underlying design philosophy of VTA is to keep the hardware as simple as possible, offloading control and scheduling to the software runtime through its two-level-ISA: task-ISA and microcode-ISA.

The task-ISA provides software-defined control over task scheduling and encodes multi-cycle memory and compute operations similar to the ISAs presented in the Google TPU and Cambricon [90, 111] work. Additionally, the task-ISA covers operation dependencies, enabling the runtime and compiler to exploit *task-level pipeline parallelism*. The microcode-ISA provides control over the provisioning of various hardware resources for tasks. The two-level ISA is instrumental in supporting the multitude of operators, data layouts, and input shapes found in deep learning workloads.

The VTA runtime bridges the gap between the explicitly programmed hardware accelerator

and state-of-the-art deep learning compilers [37, 190, 183]. These compilers can be used to explore the space of program scheduling strategies. In order to support exploration at the program level, the runtime's JIT compiler exposes a code-generation API that lets compilers explicitly define concurrent task contexts, and generate micro-kernels from high-level schedules. The stratified design enables support for widely adopted deep learning frameworks [36, 3, 133].

This chapter presents the following contributions:

- We present VTA, a novel architecture and runtime co-designed around a two-level ISA that provides software-defined task scheduling and operator implementation. We show that VTA can be easily integrated with a deep learning framework to support a diverse set of inference workloads.[1] These workloads cover object classification, natural language processing, style transfer, deep reinforcement learning, and object detection.

- We show VTA's flexibility to adapt to different workloads on two edge FPGAs. Figure 4.1 demonstrates the process used to map workloads to FPGAs with the VTA flexible architecture and runtime. This process explores VTA design variants, and autotunes software workloads onto each candidate to generate a well optimized hardware design and software library combo that can be deployed in a deep learning framework. Finally, we show VTA's ability to outperform edge GPUs with edge FPGA that have similar costs.

In section 4.2, we provide background on state-of-the-art deep learning specialization stacks, and how VTA fits into this picture. In section 4.3, we give an overview of the VTA architecture and two-level ISA. In section 4.4, we describe how the VTA runtime, based on a JIT compiler, schedules instructions down to the hardware. In section 4.5, we perform an end-to-end evaluation of our system and demonstrate its flexibility in light of evolving workloads, hardware backends, and accuracy constraints.

Definitions In order to better guide the reader moving forward, we provide a list of useful definitions:

[1] Training while possible within our framework is left to future work.

- **Workload** A given model architecture: e.g. ResNet-18, or DCGAN.

- **Operator** A layer of a neural network: e.g. `conv2d`, or `batchnorm`.

- **Task** A hardware multi-cycle CISC operation that describe a given operator: e.g. a `conv2d` operator can be broken down into a sequence of tile load, tile `conv2d`, and tile store tasks.

- **Micro-Kernel** A small micro-coded program composed of micro-ops that defines a compute task.

- **Micro-Op** A hardware tensor-tensor RISC instruction: e.g. general matrix-multiply (GEMM) hardware instruction, or broadcast add.

- **Schedule** An operator-level plan that describes how to execute tasks in hardware. A schedule defines how loops that describe computation are split, re-ordered, and unrolled to better execute on the target hardware resources.

4.2 Background: Anatomy of A Deep Learning System Stack

In order to understand how VTA's architecture and runtime fit within a deep learning system stack, we present a top-down view of a reference stack implementation and explain the role of each layer in light of emerging deep learning workloads and hardware accelerator designs. Figure 4.1 provides an overview of a generic deep learning system stack, how it compares to Google's TensorFlow-XLA-TPU highly integrated stack, what each layer of the stack achieves, and where VTA fits.

Deep Learning Frameworks Popular deep learning frameworks like TensorFlow [3], MxNet [36], PyTorch [133], CNTK [200] and Theano [7] provide a high-level programming interface for programmers to express deep learning models in a declarative fashion. These frameworks have been critical for increasing programmer productivity over the recent years: they let programmers focus

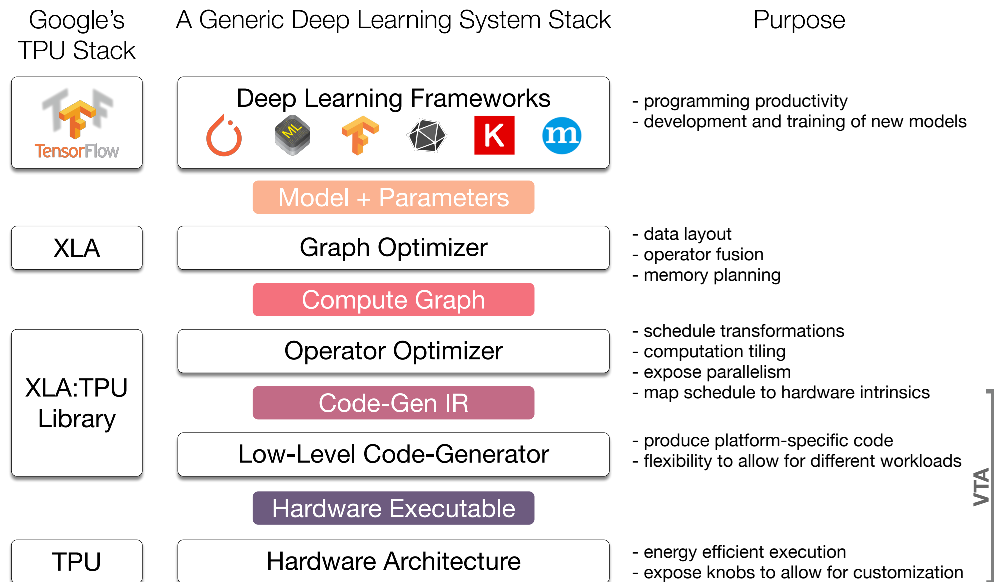


Figure 4.1: Anatomy of a Modern Deep Learning System Stack. We provide Google's TensorFlow-XLA-TPU industrial stack as a reference point. VTA's architecture and runtime provide a reference implementation of a hardware design, and low-level code generator respectively. Operator-level schedule optimizations, and graph-level optimizations are handled by the upper layers of the software stack which VTA is integrated with.

on exploring better model architectures and tweaking training hyper-parameters to produce improved training results. Features like *auto-differentiation* [19] have eliminated the need to specify a separate program to train a given model. In addition, much of the hardware-specific performance optimization details are hidden away from the programmer underneath a graph optimizer and off-the-shelf operator libraries that provide highly tuned performance for different hardware back-ends.

Graph Optimizers Workloads are represented internally as computational graphs, which provide a global view of *operators* (e.g. 2D convolution, batch-normalization, etc.). Recently, graph optimizers including Google XLA [68], Intel nGraph [52], Relay [149], and NNVM [37] have

emerged to provide optimizations on computational graphs that minimize overall memory footprint and execution latency via *memory planning* and *operator fusion* respectively. Graph optimizers can also apply *data layout transformations* to massage data into forms that are friendly to different *tensorized* hardware architectures. When performing inference on highly quantized data types, *data packing transformations* are necessary to support bit-serial inference [179]. Most importantly, graph optimizers provide flexibility over how each operator is implemented on any given hardware back-end by invoking different operator libraries.

Operator Libraries and Optimizing Compilers Traditionally, in order for computational graphs to efficiently utilize hardware resources like GP-GPUs, device manufacturers provide highly tuned deep learning operator libraries that enable out-of-the-box optimized performance. NVIDIA's cuDNN [43] deep learning library provides highly GPU-optimized operator primitives for computational graphs to call into. ARM's ComputeLibrary [15] exposes a similar API for ARM-based CPU and GPUs. These ad-hoc operator libraries present two limitations. First, they are mostly black-box, making it difficult for graph optimizers to apply operator fusion. Second, they cannot be easily extended to support new workloads and new hardware architectures by developers or researchers.

A sustainable alternative is to *generate* custom deep learning operator libraries with the use of domain-specific optimizing compilers. Recent domain specific compilers like TVM [37], DLVM [190], TensorComprehensions [183], and Glow [62] provide the tooling and optimization knobs to easily generate highly custom operator libraries and support new models and hardware architectures. Automated learning techniques [141, 38] exploit genetic algorithms or boosting [35] to automatically optimize operator libraries for evolving workloads, operators, and hardware backends (GPUs, CPUs, FPGAs, and ASICs).

Low-Level Code Generators Deep learning optimizing compilers will generate code targeting a platform-specific IR, generating code for CPUs, GPUs, and specialized accelerators. In the case of TVM [37], x86 and ARM CPU binaries are generated using LLVM, while GPU binaries are

obtained with NVIDIA's CUDA compiler and AMD's ROCm compiler.

Specialized Hardware Backends While CPUs and GPUs have been targets of choice in deep learning applications, we are seeing an emergence of specialized ASIC [42, 90, 41, 77] designs and model-to-FPGA compilers [64, 25, 4, 161]. In spite of the recent strides in open-sourcing deep learning hardware [180, 181, 131, 193, 161], hardware designs either (1) lack a reference software stack, or (2) implement a complete ad-hoc compiler that is too rigid to modify and extend.

Where VTA Fits in the Stack In this chapter, we advocate for leaving the graph and operator optimizations to a competent, off-the-shelf, and cross-platform optimization stack like TVM or Glow. We suggest coupling hardware architectures with a multi-purpose runtime API that can provide flexible code-generation to adapt new workloads to a fluid hardware design. We highlight below how layers of the mainstream industrial deep learning system stack accommodate VTA:

Frameworks Hardware designs are mostly hidden from DL frameworks. When targeting inference tasks on ultra-low-precision VTA variants (now common in hardware accelerators [181, 182, 162]), the programmer may need to tweak training recipes and insert quantization operators to produce a model that maintains acceptable accuracy.

Graph Optimizers. Graph optimizers fuse operators to minimize data movement off of the VTA design, as well as managing the data layout requirements dictated by VTA's choice of tensor hardware intrinsic and data type width.

Operator Optimizers. This step automates the tedious process of scheduling workloads onto VTA. Scheduling is important for three reasons. First, it tiles the computation in a way that satisfies on-chip memory constraints. Second, it enables thread parallelism that can be translated by VTA's runtime into task-level pipeline parallelism. Finally, it massages a computation into sub-computations which can be mapped to hardware accelerator intrinsics like bulk DMA loads or tensor operations in order to directly invoke the underlying VTA runtime API.

Low Level Code-Generators. VTA's runtime performs JIT code generation. It explicitly manages task scheduling to hide memory latency, and assembles and manages compact micro-kernels for every fused operator that needs to run on VTA. The runtime is described in much more depth in section 4.4.

Hardware Backends. VTA provides a flexible and customizable deep learning acceleration design with a two-level ISA. It can expose various memory subsystem organizations, tensor computation hardware intrinsics, and sets of arithmetic operations to support new deep learning operators. The hardware and its ISA is described in much more depth in section 4.3.

We have open-sourced the VTA hardware architecture and runtime and hope that VTA can serve as a blueprint for integrating open-source deep learning accelerators into open industrial-strength deep learning stacks.

4.3 VTA Hardware Architecture

VTA is a simple and customizable deep learning accelerator architecture. Its main highlights are:

- A modular, decoupled access-execute organization [168] that allows for concurrent memory and compute tasks (subsection 4.3.2).
- A compute module that is programmed like a simple RISC processor, but performs operations on special tensor registers, rather than scalar registers (subsection 4.3.3).
- A novel two-level ISA that provides control to the software over explicit compute and memory task scheduling (subsection 4.3.2), and how compute tasks are specified via micro-coding (subsection 4.3.3).

VTA was built to be maintainable by software and hardware developers alike. Its hardware modules are entirely specified in HLS-C and take up less than 1k LoC, providing design transparency and readability.

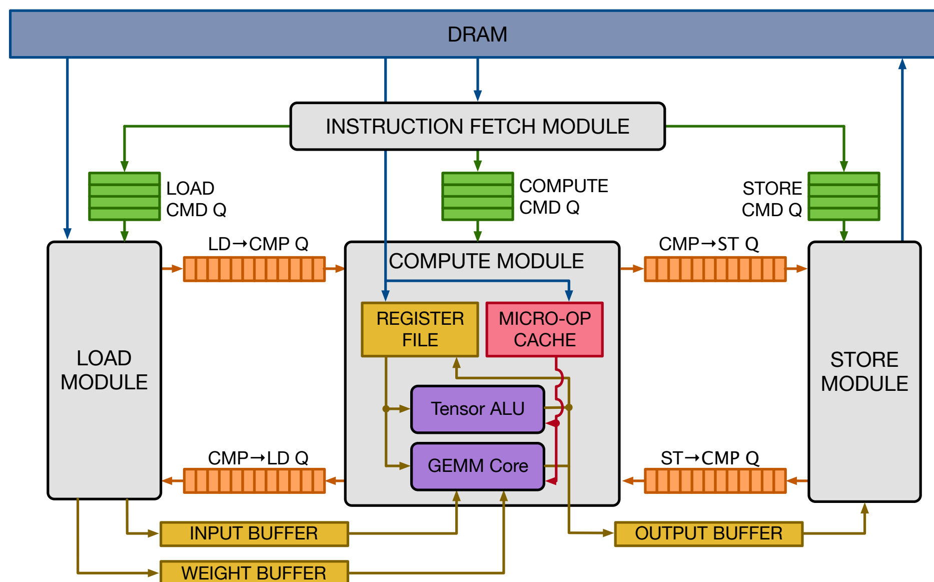


Figure 4.1: The VTA hardware organization. VTA is composed of modules that communicate via queues and SRAMs. This enables task-level pipeline parallelism, which helps maximize compute resource utilization.

4.3.1 Architecture Overview

Figure 4.1 gives a high-level overview of the VTA hardware organization. VTA is composed of four modules that can be synchronized between each other via dependence FIFO queues that act as semaphores. SRAM buffers serve as unidirectional data channels between concurrently executing modules. The `fetch` module loads instruction streams from DRAM and dispatches those instructions into one of the three command queues. The `load` module loads input and weight tensors from DRAM into data-specialized on-chip memories. The `compute` module performs both dense linear algebra computation via a GEMM core and tensor ALU operations. This module also loads data from DRAM into the register file and loads micro-op kernels into the micro-op cache. Lastly, the `store` module writes back results to DRAM.

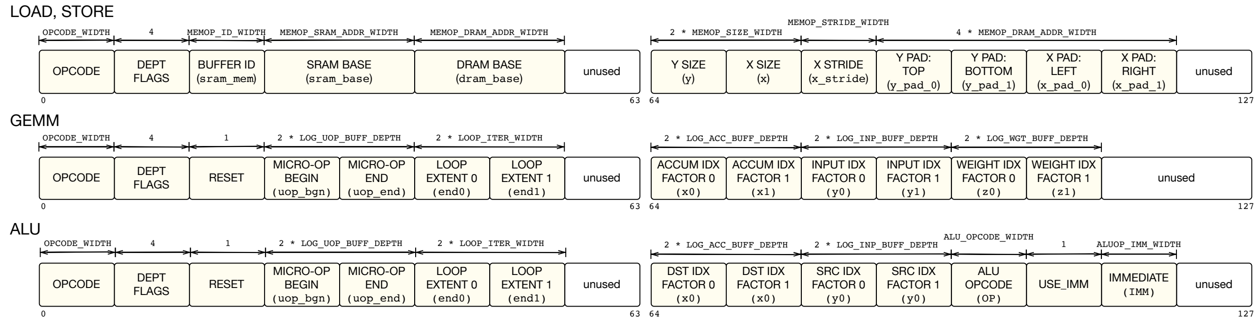


Figure 4.2: The VTA high-level instruction fields. LOAD and STORE instructions perform 2D strided DMA reads/writes between DRAM and SRAM. GEMM instructions are used to perform matrix multiplication and 2D convolutions while ALU instructions can perform a wide range of activation, normalization, and pooling tasks.

Task-ISA VTA's Task-ISA encodes multi-cycle compute and memory tasks, similar to other CISC deep learning ISAs [111, 90], including LOAD, GEMM, ALU, and STORE instructions. Figure 4.2 describes the fields of each instruction. LOAD and STORE instructions describe how data from DRAM is loaded and stored into the specific on-chip SRAMs. Strided memory access is supported to load tensor tiles without having to change memory layout. GEMM and ALU instructions call into micro-coded kernels, providing software-defined flexibility to implement various types of compute tasks. These tasks span operators, input shapes, and data layout. The micro-coded kernel are discussed in more detail in subsection 4.3.3. Finally, the dependence fields found in every task instruction express read-after-write and write-after-read dependences between memory and compute tasks. subsection 4.3.2 walks us through how these dependences are enforced to hide memory access latency.

4.3.2 Decoupled Access-Execute Organization for Task-Level Pipeline Parallelism

Managing memory movement to keep compute resources busy is the key to efficient hardware acceleration. For this reason, Task-Level Pipeline Parallelism (TLPP) is an important pattern in VTA architecture, allowing simultaneous use of compute and memory resources in order to maximize

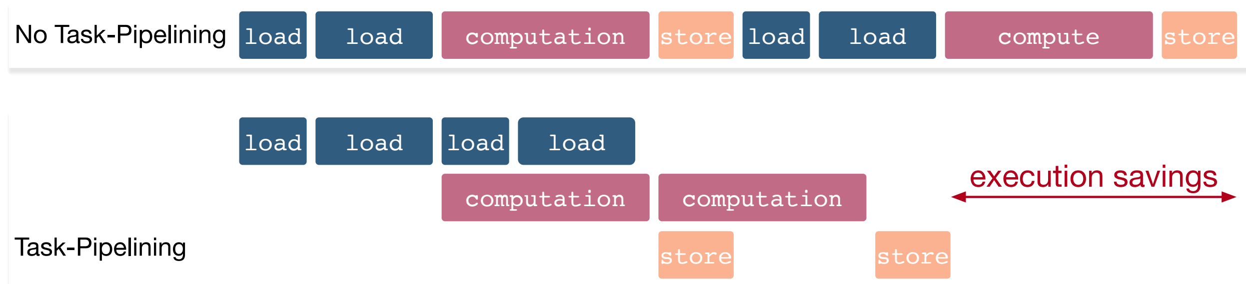


Figure 4.3: Decoupled access-execute allows concurrent utilization of compute and memory resources in hardware: this uncovers task-level pipeline parallelism.

their utilization. TLPP is achieved via access-execute decoupling [168], which is the mechanism employed by Google's TPU [90] to maximize compute resource utilization.

Latency Hiding Figure 4.3 demonstrates the benefits of performing access-execute decoupling in order to unlock task-level pipeline parallelism. By allowing instructions to execute concurrently in separate hardware modules rather than within a monolithic module, the memory operations can be executed concurrently with non-interfering compute operations. This is particularly useful in deep learning workloads with high-arithmetic intensity, because it hides memory access latency.

Data Dependences Implementing a decoupled access-execute hardware pipeline requires explicit data dependences between instructions. In order 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. The execution context partitioning is described in section 4.4. Figure 4.4 shows the benefit of inserting *read-after-write* (RAW) and *write-after-read* (WAR) dependences between tasks from the same execution context. The insertion of these dependencies guarantees timely and correct execution for decoupled access-execute instruction streams.

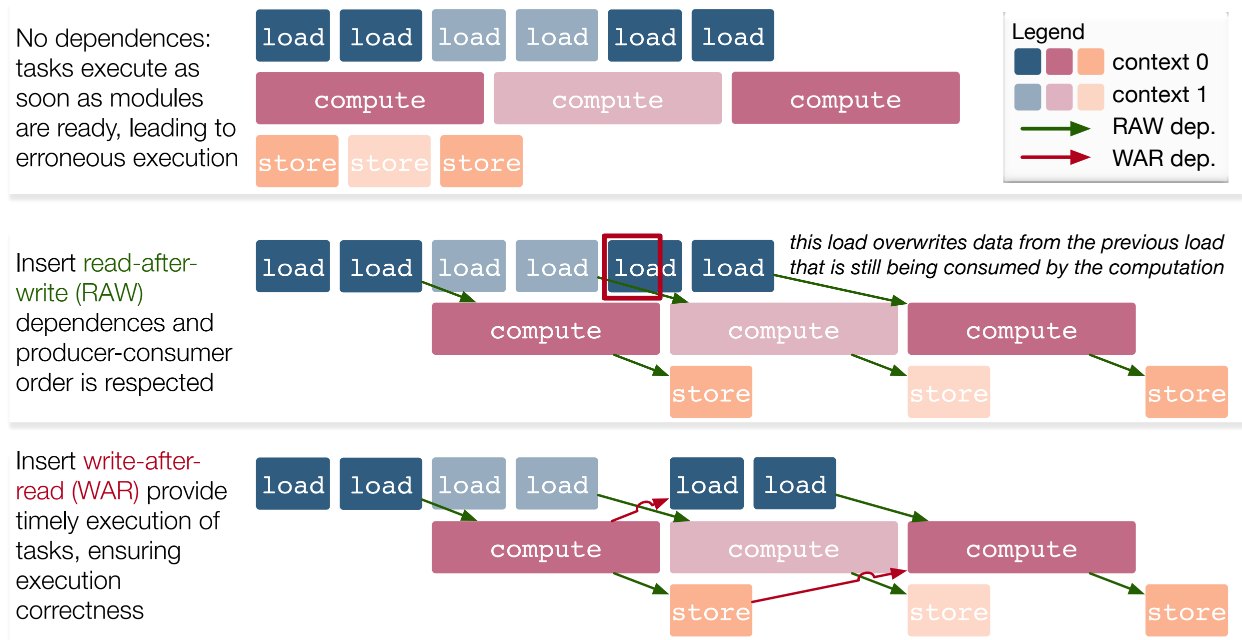


Figure 4.4: Inserting data dependences between instructions is essential to ensure execution correctness for a decoupled access-execute instruction stream.

Concurrent Execution VTA uses dependence queues between hardware modules to synchronize the execution of concurrent tasks. Figure 4.5 shows how hardware modules can execute concurrently in a dataflow fashion through the use of dependence queues, and single-reader/single-writer SRAM buffers. Figure 4.1 describes VTA architectural organization via a 3-stage task pipeline (`load-compute-store`). Each module can execute concurrently in a dataflow fashion through the use of dependence queues, and single-reader/single-writer SRAM buffers. This method of connecting hardware modules can build task pipelines of arbitrary depth as long as the single producer-single consumer principle is upheld.

4.3.3 Compute Module

Two functional units perform operations on the register file: the tensor ALU and the GEMM core. The tensor ALU performs tensor operations such as element-wise addition, activation, normaliza-

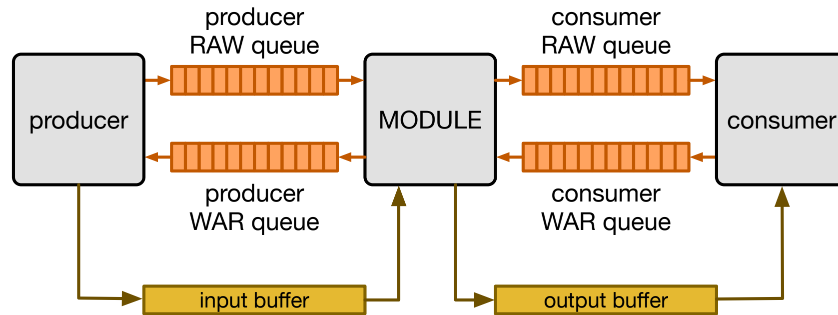


Figure 4.5: Each module is connected to its consumer and producer via RAW and WAR dependence queues and unidirectional SRAM data channel.

tion, and pooling tasks. The GEMM core performs high-arithmetic intensity matrix multiplication over data from the input and weight buffers. As outputs are written to the register file, they are concurrently flushed to the output buffer and eventually written to DRAM via the store module.

Microcode-ISA The `compute` core executes RISC micro-ops from the micro-op cache, which describes how computation is performed over data. There are two types of compute micro-ops: ALU and GEMM operations. To minimize the footprint of micro-op kernels while avoiding 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. This compression approach helps reduce the micro-kernel instruction footprint, and it is applied to both matrix multiplication and 2D convolution, which are common primitives in neural network operators.

GEMM Core The GEMM core evaluates GEMM instructions, by executing a micro-code sequence in a 2-level nested loop as described in Figure 4.6's pseudo-code block, at a rate of one input-weight matrix multiplication per cycle. The input-weight matrix multiplication logic is implemented as parallel vector multiplications that are reduced in a pipelined reduction tree. The instruction fields for the GEMM instruction are described in Figure 4.2. The compiler lowers a

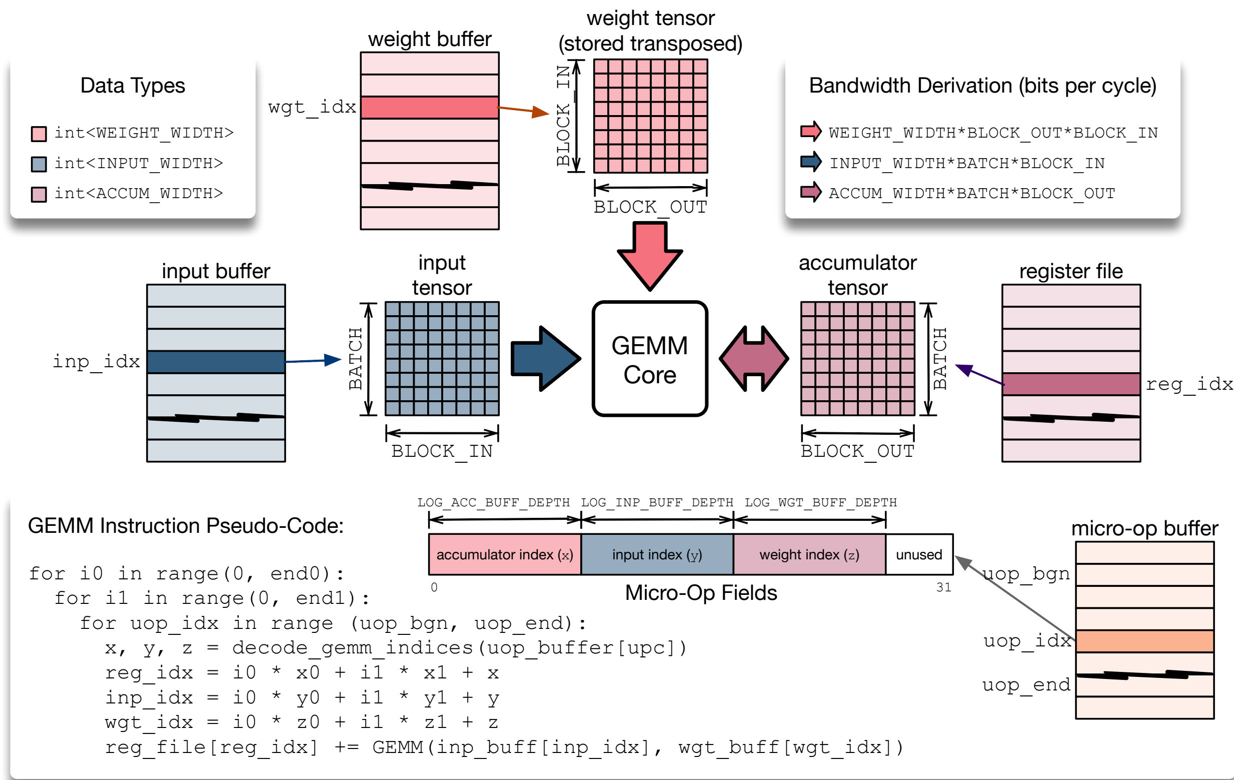


Figure 4.6: The VTA GEMM core can perform one dense matrix multiplication over an input tensor and a weight tensor, and add the result into a register file tensor. The data addressing pattern is specified by a micro-coded sequence: this allows us to map very different deep learning operators onto a single hardware tensor intrinsic.

computation schedule onto a hardware *tensorization intrinsic* defined by the dimensions of the single-cycle matrix multiplication. Each data type can have a different integer precision: weight and input types are low-precision (8-bits or fewer), while the accumulator tensor has a wider type (32-bit or fewer) to prevent overflow.

Tensor ALU. Figure 4.7 details the range of operators that the Tensor ALU supports to implement common activation, normalization, and pooling operations. VTA being a modular design, the range of operators that the Tensor ALU supports can be extended for higher operator cov-

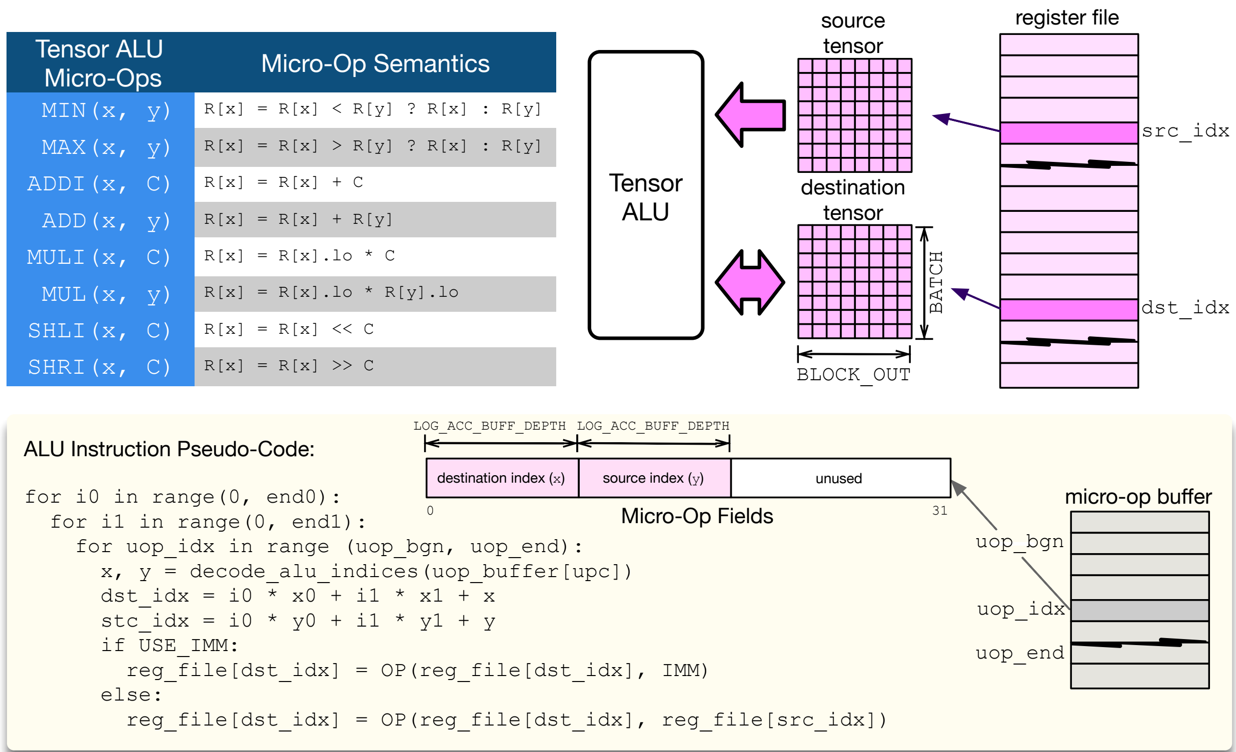


Figure 4.7: The VTA tensor ALU can implement tensor-tensor element wise operations, or tensor-scalar operations.

erage, at the expense of higher resource utilization. The Tensor ALU can perform tensor-tensor operations, as well as tensor-scalar operations on an immediate value. The opcode of the tensor ALU, and the immediate value are specified by the high-level CISC instruction which fields are listed in Figure 4.2. The micro-code in the context of tensor ALU computation only takes care of specifying data access patterns, as shown in the ALU instruction pseudo-code block in Figure 4.7.

In terms of computational throughput, the Tensor ALU does not execute at a rate of one operation per cycle. The limitation comes from the lack of read-ports: since one register file tensor can be read per cycle, the tensor ALU has an initiation interval of at least 2 (i.e. performs at most 1 operation every 2 cycles). In addition, performing a single tensor-tensor operation at once can be expensive especially given that register file types are wide, typically 32-bit integers.

As a result, in order to balance the resource utilization footprint of the Tensor ALU with the GEMM core, a tensor-tensor operation is by default performed via vector-vector operations over multiple cycles.

4.3.4 VTA Memory Subsystem

VTA has a single-level on-chip memory hierarchy composed of SRAM memories that are data-specialized, e.g. weight and input activations are stored in different physical SRAM modules. Figure 4.1 shows that SRAM buffers serve as unidirectional data channels between hardware modules. Each buffer has a single reader, single writer to allow for concurrent execution of both modules.

Bandwidth Considerations. In order to keep the GEMM core busy, the input buffer, weight buffer, and register file have to expose sufficient read and write bandwidth, as derived in Figure 4.6. VTA has a single-level memory hierarchy composed of multiple SRAMs that load and store weights and input activations separately, providing the right amount of memory bandwidth to GEMM.

For instance, with 8-bit inputs and weights, 32-bit accumulators, BATCH=`2`, BLOCK_`IN`=`16`, and BLOCK_OUT=`16`, the bandwidth required to keep a GEMM core clocked at 200MHz busy is: 51.2 Gb/s, 409.6Gb/s, and 204.8Gb/s for each of the input buffer, weight buffer and register file SRAM memories. This divergence in bandwidth requirements is the reason why VTA relies on data-specialized SRAM memories, rather than reading and writing data from a single memory structure.

Memory Access Latency Hiding. VTA's `load` and `store` modules perform DMA transfers from DRAM to the input and weight SRAM buffers, and from the SRAM output buffer to the DRAM respectively. These operations can be performed while computation is taking place in the `compute` core using the latency-hiding mechanisms described in Section 4.3.2.

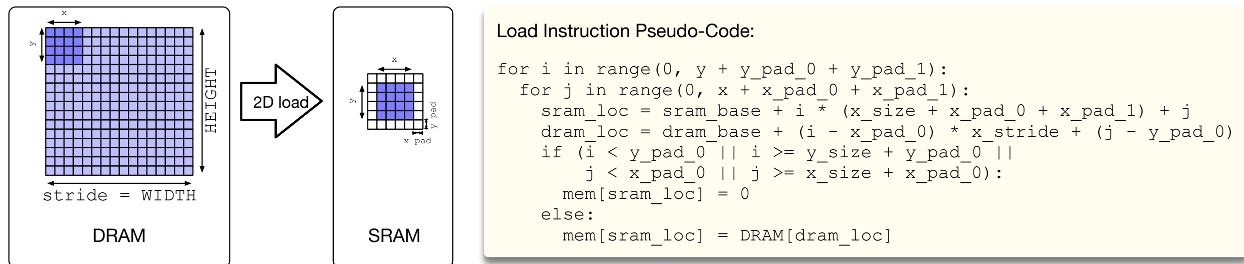


Figure 4.8: The `load` module can perform 2D DMA loads with a strided access pattern from DRAM to SRAM. In addition, it can insert 2D padding on the fly, which is useful when blocking 2D convolution. This means that VTA can tile 2D convolution inputs without paying the overhead of re-laying data out in DRAM to insert spatial padding around input and weight tiles.

Tiled Access Patterns. The `load` and `store` modules can perform strided 2D accesses from and to DRAM as Figure 4.8 shows. This feature is useful for describing cached reads and writes over tiled tensor data with a single instruction. The `load` module can also dynamically insert padding as Figure 4.8 suggests, in order to tile input and weight tensors in the context of 2D convolution without paying the overhead of spatial packing.

4.4 VTA JIT Runtime System

The guiding design principle of VTA's hardware and runtime co-design is to offload control complexity to the software stack. This complexity includes dynamic dependence tracking, ensuring timeliness of instruction cache reads, and explicitly describing compute tasks via micro-coding. The runtime gives us the flexibility to offload a multitude of deep learning workloads, just by modifying the software stack rather than changing hard-coded hardware features. VTA can customize operation coverage via micro-kernel JIT-ing, workload size via instruction cache management, and memory latency hiding via dependence insertion from virtual thread contexts.

The VTA runtime has allowed us, for instance, to extend the functionality of VTA's original vision-centric design to supporting operators found in Generative Adversarial Networks (GANs) [140] without requiring modifications to the hardware.

Function	Usage
`BufferLoad2D()`	prepare a load from DRAM instruction
`BufferStore2D()`	prepare a store to DRAM instruction
`PushUop()`	push micro-kernel op
`DepPush()`	mark instruction dependence source
`DepPop()`	mark instruction dependence destination
`Synchronize()`	send the instructions/micro-kernels

Table 4.1: Instruction stream management runtime functions

Function	Usage
`UopInst()`	prepare a compute instruction
`UopLoopBegin()`	mark uop loop begin
`UopLoopEnd()`	mark uop loop end

Table 4.2: Compute micro-kernel generation functions

Runtime API The VTA runtime exposes an API that the host program can use to generate task instruction streams and micro-kernels. The runtime acts as the glue between the VTA low-level hardware programming interface and the deep learning system stack. In a typical configuration, the runtime is used as a dynamic code generator by a high level optimizing compiler [37, 190, 62, 183] to produce high-performance CuDNN-like deep learning libraries for the VTA hardware. The APIs (shown in Table 4.1 and Table 4.2) can be used to perform the following tasks:

- Direct Memory Access (DMA) transfers between main memory (DRAM) and accelerator memory (SRAM). It supports shared memory systems both with coherent and non-coherent accesses between the CPU and VTA's accelerator.

- Explicit dependence management in the instruction stream.

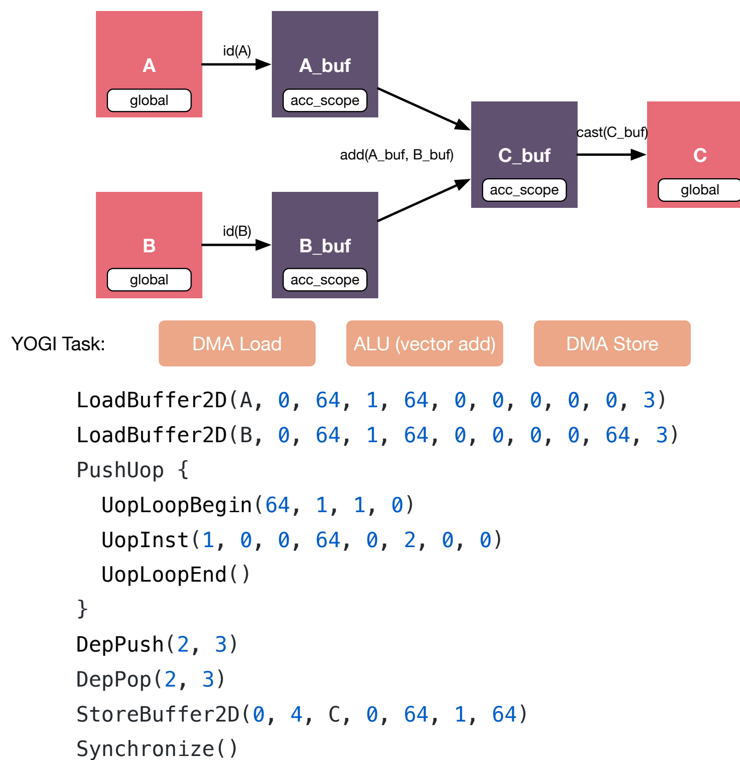


Figure 4.1: Simple vector addition dataflow graph and corresponding low level calls into JIT runtime. A and B are stored in global memory (DRAM) and are copied via DMA into the register file (accumulator memory scope, a.k.a. register file). The vector add computes the results in the local register file, which are written back to DRAM via a DMA copy.

- Synchronization between the target CPU and VTA.
- Micro-op kernel generation and caching.

We discuss VTA's runtime functionality in light of a lowered code example that performs vector addition, dictated by the graph in Figure 4.1. The code is the result of schedule optimizations and lowering. The resulting code calls into to the VTA JIT runtime.

The VTA runtime exposes the `LoadBuffer2D()` and `StoreBuffer2D()` APIs to generate LOAD and STORE VTA instructions. In the vector add code example in Figure 4.1, these

functions are inserted when preparing the SRAM buffers, and when sending the results back to DRAM.

On-the-fly micro-kernel generation is handled using the `UopLoopBegin()`, `UopLoopEnd()`, and `UopInst()` APIs, listed in the vector addition example. The `Begin` and `End` functions prepare and package the micro-op kernel to produce a CISC compute task that will call into the micro-kernel. The `DepPop()` and `DepPush()` calls set dependence flags of the in-flight tasks inserting a dependence edge between two tasks. Finally, the `Synchronize()` runtime call finishes preparing the instruction stream and micro-kernels, and hands-off control to the accelerator.

Micro-Kernel Compression VTA hardware provides a nested loop construct to reduce the microcode footprint for those which consist of multiple levels of nested loops. We enhanced the TVM compiler to detect these loop patterns and fold them into the hardware-imposed loop construct. After folding the loop nest outer loops, the internal code is unrolled during runtime. This provides a key advantage—we can unroll the inner loop dependent predicates and simplify away control flow.

Explicit Micro-Kernel Management The runtime generates each micro-kernel once and caches them in on-chip DRAM throughout execution enabling reuse of micro-kernels across multiple kernel launches. While most modern CPU architectures manage instruction caches automatically, we implement a minimalist design which trims down this control unit. We use the VTA JIT runtime to manage the micro-op cache instead, swapping the cache on each invocation and using a simple LRU cache replacement policy.

Reconfigurable Runtime and Automated Optimization For a given VTA hardware design and deep learning operator specification, there are multiple ways to implement an efficient schedule. In order to automate this process we need a mechanism to explore the scheduling space to find the best implementation of a given operator on VTA. Most existing automated schedule explorers require reliable performance measurement on hardware. This requirement can be problematic

Board	Process	Cost	FPGA Device	ARM CPU	Logic Cells	SRAM	DSPs
Pynq-Z1	28nm	$65	Zynq-7000 SoC Z-7020	Dual-core Cortex-A9	85K	4.9Mb	200
Ultra-96	20nm	$249	Zynq UltraScale+ MPSoC ZU3EG	Quad-core Cortex-A53	154K	7.6Mb	360

Table 4.3: We target three classes of edge FPGAs, ranging different cost tiers, and that provide different amounts of resources.

for accelerators since the cost of generating a hardware design can be expensive (a few hours of FPGA place and route time).

We built a VTA behavioral simulator that lets us to run any given workload program. The simulator supports a quick profiling mode that skips the computation but returns the performance counter metrics for the program. The performance counter can be used to predict the actual hardware runtime quite accurately, which we show in the evaluation below. The simulator has proven to be crucial in pruning bad schedules while performing schedule space exploration. The simulator's development benefits have helped decrease the time to design by reliably catching bad programs (ones that can cause runtime errors), and providing quick estimations of runtime costs.

Importantly, we make both the runtime and simulator dynamically reconfigurable. This property enables our automated optimization system to directly run different variants of VTA hardware and programs without human intervention.

4.5 Evaluation

We integrated the VTA runtime into a state-of-the art deep learning stack and evaluated a variety of deep learning models on a set of edge FPGA devices. Without loss of generality, we chose MxNet [36] as our front-end deep learning framework, Relay [149] as the graph-level optimizer, and TVM [37] as the low level operator optimizer. It is worth noting Relay's model importers provide access to a wide variety of other front-ends. Our detailed evaluation showcases the following benefits of using VTA to build a deep learning specialization stack:

HW Constraint	SW Constraints		Results								
Device	Type	Batching	GEMM Shape	Pipeline Stages	ALU Size	Logic Util	BRAM Util	DSP Util	Frequency	Peak Throughput
Pynq	W8A8	N	(1,16)x(16,16)	11	8	74.95	93.93	100	100MHz	51.2 GOps
Pynq	W8A8	Y	(4,8)x(8,8)	10	16	82.65	93.93	100	100MHz	51.2 GOps
Ultra-96	W8A8	N	(1,16) x (16,16)	16	8	54.65	90.97	78.33	333MHz	170.5 GOps
Ultra-96	W8A8	Y	(2,16) x (16,16)	16	4	95.25	85.42	100	300MHz	307.2 GOps
Ultra-96	W4A8	N	(1,32) x (32,32)	12	8	99.34	96.53	100	250MHZ	512.0 GOps

Table 4.4: We provide a sample of VTA configurations considered favorable that we compiled for Ultra-96/Pynq-Z1 and software constraints defined by data type, or batch size. The data shows promise in terms of adapting the design to (1) occupy available hardware resources, (2) successfully closing timing at high clock targets, (3) exposing different GEMM tensor intrinsics, and (4) scaling peak throughput when reducing precision.

- VTA parameterized hardware design can target different generations of edge FPGA devices, identifying and balancing resources in the target FPGAs (subsection 4.5.1).
- VTA's runtime is used to generate the optimized operator library (subsection 4.5.2).
- VTA supports multiple data types, scaling end-to-end performance on networks that were trained to take advantage of different weight precisions (subsection 4.5.3).
- VTA runs various model architectures that out-compete highly optimized edge CPU and GPU systems (subsection 4.5.4).

4.5.1 Hardware Exploration

One way to showcase VTA's flexibility is to target different FPGA platforms. FPGAs are becoming more accessible than ever, with sub-$100 development boards, and FPGA cloud computing instances becoming ubiquitous [13]. Table 4.3 provides an overview of the FPGAs VTA has been implemented for. Currently, VTA's runtime is designed to take advantage of a tight CPU-FPGA integration. Therefore we reserve support for discrete or PCIe-based FPGAs for follow-up work,

as they require extending VTA's runtime to manage memory allocation and PCI-E data transfers efficiently.

Figure 4.1 shows the high level exploration process that we employ to generate a VTA design given hardware and workload constraints. The VTA design offers multiple architectural customization parameters that are listed in Figure 4.1. These customization knobs define a hardware design space in the 100s to 1000s of design points that are exhaustively explored on a compute cluster. We perform this exploration in a sequence of stratified steps. First use a simple FPGA resource model to prune infeasible configurations. After pruning, each candidate hardware design is compiled, placed and routed. Table 4.4 shows a set of VTA hardware configurations that our design space exploration selects by ranking raw peak throughput estimates. We picked one design for each $\{fpga \times dtype \times batch\}$ combination, but typically our exploration returns a handful of promising candidates — the rest of the designs either failed placement, routing or timing closure, or resulted in low peak performance. On this final set we perform operator autotuning to obtain the workload's performance profile, which we discuss next.

The FPGA utilization numbers show that the VTA hardware template can occupy hardware resources given different hardware or software constraints. In addition, our design template can close timing at high FPGA clock speeds. Naturally, high device contention will result into lower clock speeds due to more effort being required to place and route the design. It's worth noting as well that exploring different software constraints leads to our search producing hardware designs with different GEMM intrinsic shapes. A key to exploiting the changing intrinsic shapes is our flexible runtime coupled with a flexible operator scheduling template. Finally the data shows that when data type precision is reduced, (W4A8 stands for 4bit weights, 8bit activations) we were able to scale resources to take advantage of the smaller multipliers that were being generated.

4.5.2 Schedule Exploration for Operator Autotuning

The analytical model of *peak performance* used to initially filter hardware design points is based on theoretical throughput and frequency assuming compute resources are 100% utilized. This modeling does not capture the complexity of obtaining performance for real-world deep learn-

ing accelerators: in order to saturate compute resources, data must be loaded from DRAM, often resulting in compute resources being stalled on memory. Depending on the workload mix, operators like `conv2d` with large window sizes may exhibit high arithmetic intensity (measured in Op/Byte). Operations with high arithmetic intensity translate to high utilization, and therefore are close to peak performance. Operators which exhibit low arithmetic intensity, like `conv2d`, with a window size of 1, are memory bandwidth constrained. In these situations we are able to use latency hiding to mitigate performance loss (c.f. subsection 4.3.2).

Schedule Autotuning Schedule autotuning is the process by which an automated search algorithm attempts to optimize a given workload towards peak hardware performance. We perform autotuning by applying different memory tiling, loop transformations (e.g. splitting, reordering, unrolling), vectorization/tensorization, and parallelization strategies [38]. We use the TVM compiler to express schedule templates for each operator (e.g. `conv2d`, `conv2d_transpose`, `group_conv2d`, `fc`) we support in hardware. We then leverage TVM's automated scheduling library to obtain schedules that maximize performance for a given configuration of operator, tensor shapes, and hardware variant.

Figure 4.1 shows the autotuning search process when optimizing different VTA hardware candidates for single ResNet layer. We used the XGBoost [35] search algorithm to find the best schedules for each hardware variant in a limited number of trials. Each workload's layers are then tuned on each hardware candidate, we use aggregate inference time to guide which VTA hardware variant is best for a given network model.

Full Network Optimization It takes several hours to exhaustively tune a network on a single hardware variant. Given the large numbers of VTA hardware designs to test, and large numbers of 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, based on `SuccessiveHalving` [87]. Instead of choosing among hy-

perparameters that define a network architecture, we apply this technique to choose among 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.

We evaluate the hierarchical optimization scenario on the ResNet-18 workload, over a set VTA candidates generated given an W8A8 (8-bit weights, 8-bit activations) data representation. 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 is a difficult task, as the relative performance differences between hardware designs may be small early on. After a moderate number of iterations, `SuccessiveHalving` is able to converge to the best candidate hardware design, as shown in Figure 4.2.

4.5.3 Kernel Weight Quantization Analysis

We assess the workload-level effects of modifying kernel weight precision from a baseline 8-bit which is now standard in accelerators, down to 4 and 2 bits. By carefully co-optimizing the the hardware in tandem with the program schedules we can achieve noticeable speedups when reducing weight precision. We denote a design that uses 4-bit weights and 8-bit activation with the $W4A8$ notation. We selected the best combination of VTA hardware designs and schedule for each data point in Figure 4.3, for ResNet-18 running on the Pynq-Z1.

Overall performance is a function of how quantization affects compute throughput, effective memory bandwidth, and effective on-chip storage. For example, changing the weight precision leads to linear compute scaling because of narrower fused multiply-add (FMA) units. Moreover, reducing the weight precision increases effective weight storage linearly, which improve tile reuse when optimizing schedules. This reuse can be very effective in layers of ResNet with higher weight-to-activation ratios.

To better understand the effect of quantization, we complement our analysis with a roofline model of the hardware design at each precision setting (Figure 4.4). A roofline analysis visualizes

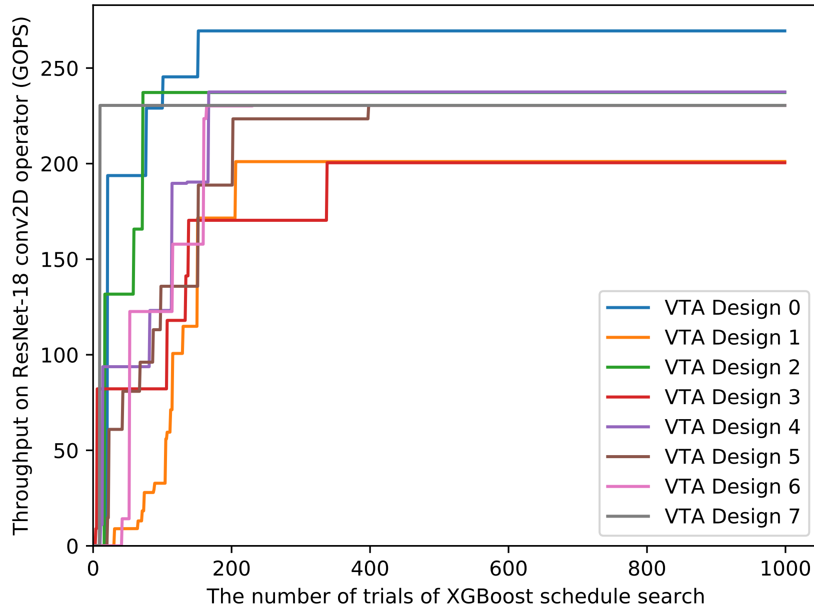


Figure 4.1: Schedule exploration with XGBoost for a single ResNet-18 layer on Ultra-96. Eight VTA design candidates with (2,16)x(16,16) and (8,8)x(8,8) GEMM intrinsic at W8A8 are considered. Layer is conv2d: IC=256, OC=256, H=W=14, KW=KH=3, stride=(1,1), padding=(0,0).

how efficiently a given workload is running on hardware (i.e. how close it it to the theoretical peak performance), and whether the bottleneck is memory or compute bound.

Figure 4.4 highlights how the rooflines of the VTA variants change as we reduce the bitwidth of kernel weights: as expected, the compute roofline's peak shifts up to higher throughput values, and so do the data points under the roofline (each represents a workload performance number). Peak throughput scaling is linear at a clock rate of 100MHz; we achieve 200GOps at $W2A8$, 100GOps at $W4A8$, and 50GOps at $W8A8$.

The knee of the roofline diagram marks the point from which a memory-constrained workload shifts to being compute-constrained. We can see that the knee of the diagram is moving towards the right, a sign that the memory bandwidth cannot keep up with compute scaling. This shift

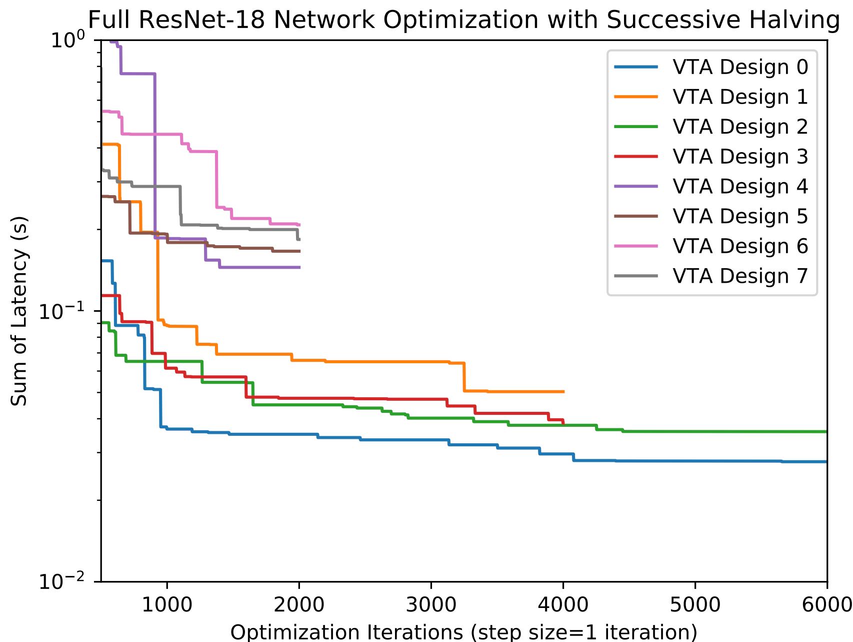


Figure 4.2: We use `SuccessiveHalving` for choosing among potential hardware designs on a complete ResNet-18 inference workload. Here, we begin with promising VTA hardware variants. `SuccessiveHalving` converges to the optimal hardware design while using a fraction of the optimization time required to exhaustively evaluate each design.

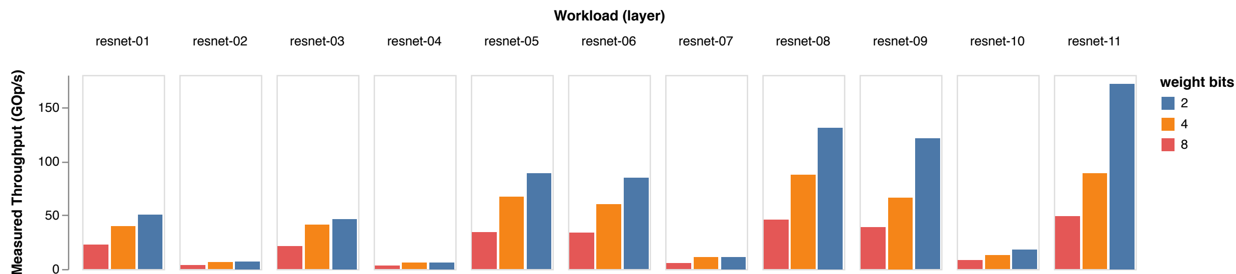


Figure 4.3: Throughput improvement on each ResNet-18 convolution layer versus integer precision of kernel weights (8-bit down to 2-bits) running on Pynq-Z1.

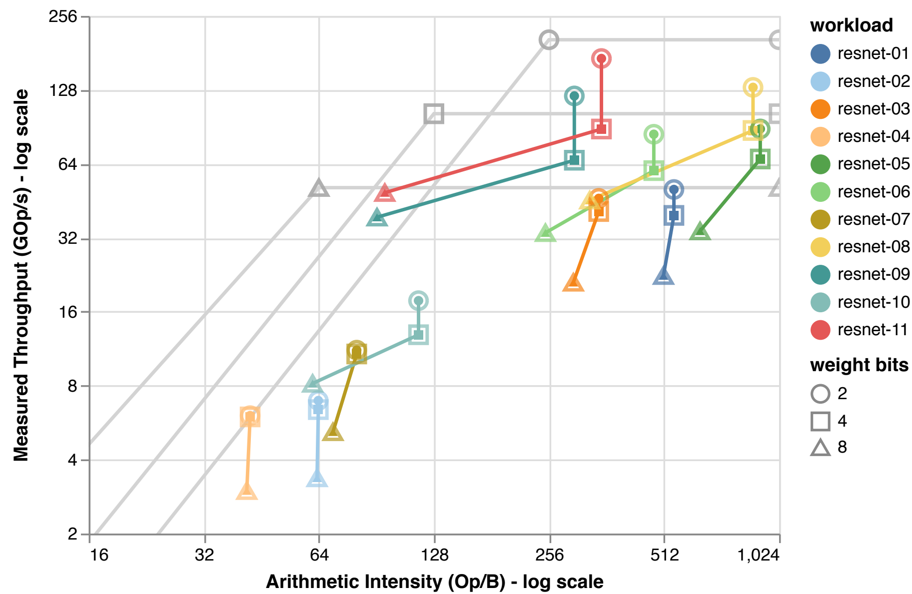


Figure 4.4: Improvement in compute throughput of ResNet-18 layers as we decrease the kernel weight accuracy of VTA designs variants, seen under their respective rooflines, on Pynz-Z1.

implies workloads need a high arithmetic intensity to take advantage of the reduced precision.

Quantizing the weights shifts the arithmetic intensity of each workload to the right because we perform the same amount of computation with less bytes moved in memory. For layers with high weight to activation ratio, like `resnet-11`, the bar chart in Figure 4.3 shows linear scaling. Decreasing weight bitwidth effectively increases weight storage, and allows for more effective data reuse. In Figure 4.4, moving from $W8A8$ to $W4A8$, the red `resnet-11` shifts far to the right, validating that we have an increase in arithmetic intensity. For layers where activations dominate data movement, like `resnet-1`, performance scaling is sub-linear. The roofline shows that `resnet-1` barely shifts on the x axis, demonstrating little benefit to quantization on overall data movement requirements, while compute throughput itself is improving.

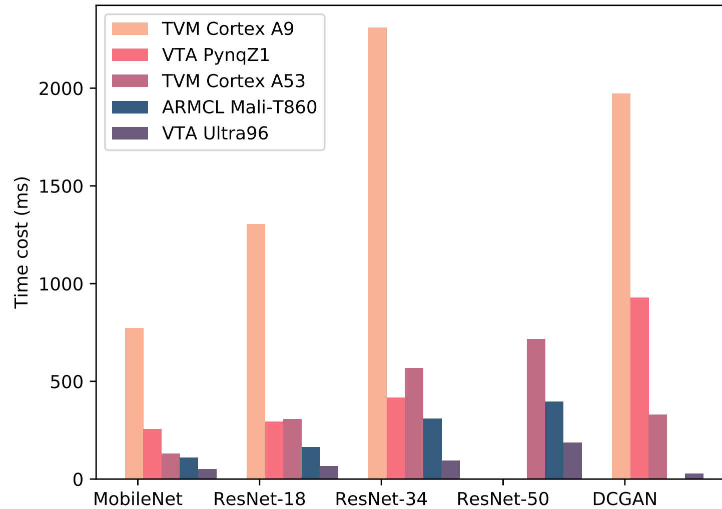


Figure 4.5: End to end performance evaluation over multiple CPU, GPU and FPGA-equipped edge systems. For comparable systems, VTA provides a significant performance edge over conventional CPU and GPU-based inference.

4.5.4 End-to-End Evaluation

As the landscape of deep learning continues to evolve it is important to support emerging models. We evaluate VTA's ability to support two novel model architectures beyond classical deep convolution nets. First, we evaluate MobileNet [84] a recent model architecture that uses grouped convolution to reduce the total computation overhead of the network. We evaluate a variant of MobileNet we call MobileNetG that groups channels by the vector factor of the VTA's GEMM core. Second, we implement a Convolutional Generative Adversarial Network [140] (DCGAN) model that is used for image-to-image translation and generation.

Both models require non-trivial extensions to support new operators. MobileNetG requires support for grouped convolution that exhibits block sparse patterns on channel groups. DCGAN requires support for 2D convolution transpose which has a sparsity pattern on the spatial locations. Specialized accelerators need to specifically support these novel access patterns to avoid unnecessary computations and achieve maximum performance. The runtime can readily

make use of schedules to generate micro-kernels that support these novel access patterns without changing the hardware.

Figure 4.5 shows a performance comparison across these models, comparing VTA-accelerated execution against a highly optimized ARM CPU and GPU platforms that rely on industry-strength deep learning libraries [15, 37]. 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) boards. For the VTA hardware designs, 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, multi-threading and state of the art scheduling tricks (spatial tiling, Winograd transform etc.). For our GPU baseline, we use the ARM ComputeLib v18.03 and exploit 16-bit floating-point (FP16) library support. 8-bit support is unfortunately missing in both RMCL and TVM. ARM ComputeLib is missing support `conv2d` transpose for DCGANs, demonstrating VTA's ability to stay ahead of the curve for unconventional workloads.

Figure 4.5 shows end-to-end results that can be discussed in two groups of comparable devices in terms of cost: (1) VTA on the Pynq vs. Cortex-A9 (sub-$100), and (2) VTA on Ultra96 vs. Cortex-A53 and Mali-T860 GPU ($100). First off, VTA on the Pynq-Z1 outperforms the Cortex-A9 CPU by 3.0x, 4.4x, 5.3x and 2.1x on MobileNet, ResNet-18, ResNet-34 and DCGAN. Second, VTA on the Ultra-96 outperforms the Cortex-A53 by 2.5x, 4.7x, 6.0x, 3.8x and 11.5x on MobileNet, ResNet-18, ResNet-34, ResNet-50 and DCGAN. In addition, VTA on the Ultra-96 outperforms the mobile-class Mali-T860 GPU by 2.1x, 2.5x, 3.2x and 2.1x on MobileNet, ResNet-18, ResNet-34 and ResNet-50.

Overall, VTA demonstrates that the flexibility of a two-level ISA and low hardware complexity can offer high performance and form a promising path forward for accelerating diverse workloads on edge devices.

4.6 Related Work

Deep Learning Accelerators There has been an explosion of deep learning accelerator design proposals from academia [42, 41, 77, 110, 56] and industry [90, 64, 74, 131, 111]. Rather than

focusing solely on hardware design, VTA proposes a new two-level ISA with a simple hardware architecture and provides a complete blueprint for building a custom deep learning accelerator stack. Our work provides a co-designed runtime and hardware design which leverages off-the-shelf graph and operator compilers to facilitate the generation of optimized deep learning libraries for diverse workloads.

Model-to-FPGA Compilers Model to FPGA translation is an active area of research [161, 181, 64, 25, 4]. DNNWeaver [161] implements a compiler from Caffe to static templatized FPGA accelerator, guided by a heuristic algorithm that aims to minimize off-chip memory accesses. Our approach differs from DNNWeaver optimization framework, because we offload graph/schedule optimization to existing optimization toolchains (Relay [149], TVM [37]), and design a runtime that exposes necessary knobs to offload graph and schedule optimizations to the upper layers of the stack, allowing for end-to-end tuning. To the best of our knowledge, model-to-FPGA compilers do not cover graph and operator optimization tools [37, 190, 183, 62]. Most importantly, though, our work on VTA is not about spatially porting models to FPGAs but about a principled two-level ISA design and low-complexity micro-architecture, which is applicable to both FGPA-based deployment as well as custom silicon.

BrainWave [64] is a cloud-scale DNN acceleration framework that relies on a spatially distributed microarchitecture that can scale across many FPGAs while keeping neural network parameters on chip, reducing data movement from DRAM. VTA, on the other hand, targets edge class FPGAs that have limited storage ($<$ 1MB). For that reason, we are reliant upon tiling, and latency hiding strategies to maximize compute resource utilization.

Narrow Precision and Model Compression There has been extensive work on maximizing hardware inference efficiency using binarization [181], and weight and activation quantization [92, 144]. Much of the complexity involved in supporting hardware inference lies in improving training techniques [142, 51, 201, 29], and adapting a deep learning framework to support new data layouts imposed by quantized data types. In VTA's evaluation, we tackle the challenges of

training quantized networks and modifying a complete deep learning stack to support end-to-end inference.

Model compression is a popular technique that prunes model parameters [78, 77, 8, 144] with minimal accuracy loss to reduce data movement and inference energy. VTA only supports dense operators, but could be extended to support popular sparse matrix representations to take advantage of these compression techniques.

4.7 Conclusion

In this chapter, we introduced VTA, a customizable deep learning architecture designed for flexibility in the presence of evolving workloads. We adopted a design principle of minimal hardware that shifts explicit control to a co-designed runtime via a two-level ISA interface. We show that VTA can effectively target different FPGAs, multiple workloads, and leverage deep learning compilers to integrate optimized software with specialized hardware.

Chapter 5
CONCLUSION

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."

– John Von Neumann, 1949

5.1 Accomplished Research Summary

The development of big data, machine learning and ubiquitous computing is driving the demand for efficient computer systems that can process massive amounts of data under stringent energy budgets. Hardware accelerators offer the promise of quantum leaps in efficiency gains for stable applications and algorithms. But hardware accelerators pose programmability, adaptability and Pareto-efficiency challenges to system designers, to *durably* incorporate acceleration into the system stack.

In Chapter 2 we propose SNNAP, an FPGA-based accelerator design that can map diverse application targets approximately. The idea is to train a neural network to approximate the execution of the target region of code, and run it efficiently on a single neural network accelerator substrate [61]. Having a single accelerator substrate for diverse regions of code essentially emulates the flexibility of behavior-specialized accelerators [129]. We showed through a programmability study that taking advantage of neural acceleration is much easier than programming FPGAs from scratch, even when it involves using friendlier HLS tools. By implementing approximate region detection with approximate compilers like ACCEPT [154], we can automate the task of identifying candidate regions of code from simple EnerJ-style type annotations. SNNAP alleviates the burden of programming FPGAs as those become ubiquitous in the datacenter [137], and in mobile

SoCs [44].

In Chapter 3 we propose QAPPA, a compilation framework that helps programmers navigate quality-efficiency tradeoffs when targeting quality programmable accelerators [92, 186]. QAPPA builds on top of ACCEPT's ability to guarantee safe execution, augments it with the ability to emulate errors from hardware quality-scaling mechanisms, and applies autotuning techniques [150] to navigate design tradeoffs for general purpose C/C++ kernels. We performed a qualitative study of different quality scaling mechanisms backed by Spice circuit simulations and post-place and route detailed power estimates. Notably we compare arbitrary quantization [92, 96, 2] against voltage overscaling [60, 155] and demonstrate that with the right compilation framework, quality-scalable architectures can provide much superior quality-efficiency Pareto-optimality over voltage scaled designs.

In Chapter 4 we propose the VTA stack, a hardware-software stack built around a versatile deep learning accelerator that captures the salient features of deep learning domain-specialized accelerator designs [42, 110, 56, 41, 77, 90]. VTA helps inform the design of domain-specific compilers like the TVM compiler [37] for which we implemented specific scheduling passes to target deep learning architectures, and tackle challenges of mapping computation down to hardware intrinsics (like tensor operations), explicitly managing memories with automated scheduling [38], and hiding memory latency via high-level threading abstractions. The VTA stack forms a blueprint for how to integrate accelerators into deep learning frameworks, and has been open sourced [1] for the community to use. Our evaluation of VTA showed that it enables significant speedups over highly tuned CPU implementations, and that it could enable accuracy-efficiency tradeoffs by scaling throughput as the bit-width used to store weights gets minimized. Additionally, we showed that by supporting a wide variety of models, VTA offers adaptability to evolving workloads: although it was designed to mostly accelerate traditional dense 2D convolution operations which are common in vision neural network, we demonstrated that we could use the highly flexible software stack to adapt VTA to execute other operators such as conv2d-transpose

[1] VTA is available at: `http://tvm.ai/vta`

and grouped-conv2d found in Mobilenets [84] and DCGANs [139] respectively. This flexibility is achieved thanks to VTA's philosophy of "keep the hardware simple, and offload the complexity of the software stack". Following this philosophy, VTA's JIT runtime helps offload tasks like low-level dependence synchronization to increase task parallelism, micro-kernel generation to achieve flexible operator coverage, and explicit instruction cache management to minimize accesses to fetch instruction data.

5.2 Beyond Academic Research

One objective of this dissertation is to facilitate the integration of customized accelerators and foster the exploration energy-efficiency tradeoffs in systems design and implementation. I discuss two ways in which my work as a Ph.D. candidate has aimed to make accelerator design and optimization more approachable. The first approach puts together class material and assignments aimed at introducing graduate level students to accelerator implementation and pareto-efficient design in the context of machine learning. The second approach, aims to popularize the reproducibility and artifact evaluation of deep learning systems research, and facilitate multi-objective comparisons of research artifacts.

5.2.1 Teaching Pareto-Efficient Deep Learning Optimization

As a graduate student I had the chance to TA the Computer Architecture class for graduate students under Luis Ceze [2] and design my own class assignment focusing on machine learning systems optimization. The materials for the assignment are freely available on GitHub [3], and have been reused by other academics teaching architecture including Trevor Carlson at NUS.

Assignment Objectives The problem statement was to implement an FPGA based inference accelerator for MNIST hand-written digit recognition [106] that is both fast and accurate. This

[2] `http://courses.cs.washington.edu/courses/cse548/17sp/`

[3] (`http://github.com/uwsampa/cse548-labs`)

simple problem makes it easy to quickly explore a wide space of classifiers and designs for students. The learning objectives of the assignment were three-fold:

- To help students explore hardware/software co-design methodologies on FPGA-equipped systems.
- To provide intuition necessary to uncover and exploit accuracy/performance tradeoffs.
- To learn how to identify system performance bottlenecks and tackle them accordingly.

In terms of **constraints**, we had to assume that the assignment had to be completed in under two weeks, and that no prior FPGA experience or machine learning background experience was required to complete the assignment in time.

Assignment Overview The assignment provided some scaffolding to help streamline the process of programming FPGAs. Students had flexibility over the design of the accelerator thanks to the productivity of HLS, but the integration of the accelerator within the FPGA shell was provided via custom TCL scripts, and predefined bus interfaces. We provide each student with an FPGA kit that includes the Pynq FPGA board, which was sold to academics for 65 US dollars. The Pynq board houses a dual core ARM Cortex A-9 SoC that runs Linux and consumes less than 2W of power. The ARM SoC houses an FPGA fabric with 630kB of on-chip SRAM storage, 220 digital signal processing (DSP) units, and 53k Look-Up Tables (LUTs).

The assignment was composed of three parts:

- **Part 1: Pipeline Optimization** Students are asked to implement a linear classifier that performs floating point computation. Students are familiarized with the process of optimizing hardware pipelines, increasing pipeline and SIMD parallelism, and optimizing for memory throughput via smart banking and buffering.
- **Part 2: Fixed Point Optimization** Students are introduced to the notion of fixed-point optimization. Specifically, they are asked to change the base design implemented in 32bit

floating point to use 8bit integers. This has implications on the way data is packed in memory, but also requires changes to the algorithm to make sure that it still works under the new data representation.

- **Part 3: Open Ended Design Optimization** Students are given free range to explore software optimizations (e.g. improved training), hardware optimizations (narrower integer types, input feature compression), or co-design techniques (new classifiers that require new hardware).

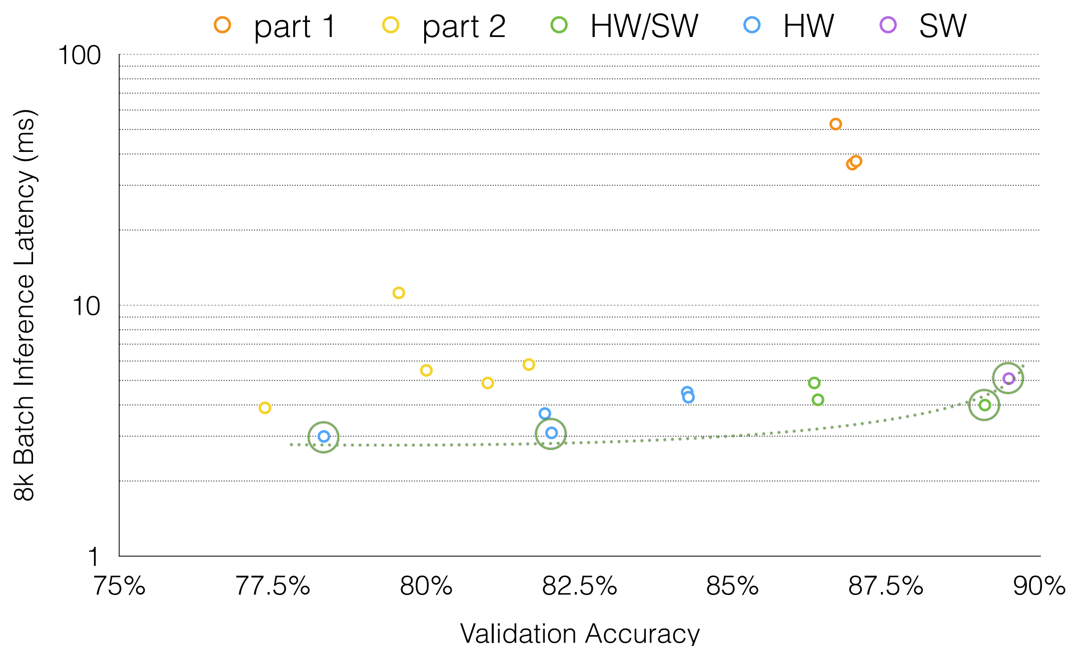


Figure 5.1: Each of the students assignment submissions according to their efficiency (8k batch inference latency) and validation accuracy. The Pareto frontier is represented as a green dotted line.

Submission Overview Upon receiving the student submissions, we analyzed each system based on their accuracy and performance characteristics. Each solution is plotted on Figure 5.1.

Students that only got to finishing Part 1 or Part 2 due to lack of time define two clusters of solutions, that present different accuracy-performance tradeoffs. Part 1 solutions yields relatively accurate but slow classifiers. Part 2 solutions yield much faster classifier but the naive post-training quantization affects classification accuracy.

The students that attempted the open ended optimization challenge are categorized according to the optimization approach they took: hardware only (HW), software only (SW), or hardware/-software co-optimization (SW+HW).

- For the HW submissions, students implemented more aggressive quantization using int4 data types, or by further compressing the input from 256 down to only 144 features. These optimizations don't change the algorithm and therefore result in very fast, but less accurate solutions.

- For the SW submission (only one student attempted this approach), the student implemented a more powerful classifier on top of the existing hardware. They replaced the linear classifier with an SVM which is better adapted to the problem.

- Finally for the SW+HW submissions, students implemented more ambitious classifiers such as multi-layer perceptrons (MLPs), which are a class of neural networks. One person even implemented a fully binarized implementation of an XNOR net.

Overall 4 submissions were deemed Pareto optimal, and interestingly covered all 3 approaches to systems optimization, namely: software only, hardware only, and hardware-software. One could argue that the design that lives on the "knee" of the curve is the one that employs both software and hardware optimizations.

The success of this assignment led to a follow-up class dedicated to building and optimizing specialized deep learning systems [4] that I helped teach with Luis Ceze, where many of the concepts from this assignment were carried over.

---

[4] `https://courses.cs.washington.edu/courses/cse599s/18sp/`

5.2.2 ReQuEST: A Workshop for Reproducible Deep Learning Systems Artifacts and Multi-Objective Comparisons

Published As: Thierry Moreau, Anton Lokhmotov and Grigori Fursin, *Towards Reproducible and Reusable Deep Learning Systems Research Artifacts*, Machine Learning Open Source Software 2018: Sustainable communities (co-located with NIPS), 2018.

I had the chance to co-organize the ReQuEST workshop at ASPLOS 2018 with Grigori Fursin, Anton Lokhmotov, and the help of Luis Ceze, Natalie Enright Jerger, Babak Falsafi, Adrian Sampson and Phillip Stanley Marbell. I discuss the results and insights from the 1st ReQuEST workshop, a collective effort to promote reusability, portability and reproducibility of deep learning research artifacts within the Architecture/PL/Systems communities. ReQuEST (Reproducible Quality-Efficient Systems Tournament) exploits the open-source Collective Knowledge framework (CK) to unify benchmarking, optimization, and co-design of deep learning systems implementations and exchange results via a live multi-objective scoreboard. Systems evaluated under ReQuEST are diverse and include an FPGA-based accelerator, optimized deep learning libraries for x86 and ARM systems, and distributed inference in Amazon Cloud and over a cluster of Raspberry Pis. We finally discuss limitations to our approach, and how we plan improve upon those limitations for the upcoming SysML artifact evaluation effort.

5.2.3 ReQuEST Overview

The quest to continually optimize deep learning systems has introduced new deep learning models, frameworks, DSLs, libraries, compilers and hardware architectures. In this frantically changing environment, is has become critical to quickly reproduce, deploy, and build on top of existing research. While open-sourcing research artifacts is one step in the right direction, it is not sufficient to guarantee ease of *reproducibility* and *reusability.* To enable reproducible and reusable research, we need to provide complete, customizable, and portable *workflows* that combine off-the-shelf and custom layers of the system stack and deploys them in a push-button fashion to generate end-to-end metrics of importance.

In an effort to promote reproducible, reusable, and portable workflows in deep learning systems research, we introduced the ReQuEST workshop at the ACM ASPLOS 2018 (for multidisciplinary systems research spanning computer architecture and hardware, programming languages and compilers, operating systems and networking). The goal was to have computer architects, compilers, and systems researchers submit deep learning research artifacts (code, data, and experiments) using a unified Collective Knowledge (CK) workflow framework [65] to produce a *multi-objective scoreboard* that would rank submissions under varied cost metrics that include: ImageNet validation (50,000 images), latency (seconds per image), throughput (images per second), platform price (dollars), and peak power consumption (Watts). To keep the task of collecting artifacts tractable, we focused on a single problem: ImageNet classification, but gave complete freedom over what models, frameworks, libraries, compilers and hardware platforms were being used to solve the classification problem.

The most important difference of ReQuEST from other related workshops and tournaments such as DawnBench [50] and LPIRC [67] is that we not only publish final results but also share portable and customizable workflows (i.e. not just Docker images) with all related research components (models, data sets, libraries) to let the community immediately reuse, improve, and build upon them.

The first iteration of the ReQuEST workshop led to five artifact submissions that were unified under the CK framework and evaluated (reproduced) by the organizers. What the submissions lacked in quantity, they made up for in terms of diversity: (1) submissions spanned architecture, compilers, and systems research, (2) utilized x86, ARM, and FPGA-based platforms; and (3) were deployed on single-node systems as well as distributed nodes.

5.2.4 Unifying Artifacts and Workflows with CK

ReQuEST aims to promote reproducibility of experimental results and reusability/customization of systems research artifacts by standardizing evaluation methodologies and facilitating the deployment of efficient solutions on heterogeneous platforms. For that reason, packaging artifacts (scripts, libraries, frameworks, data sets, models) and experimental results requires a bit more

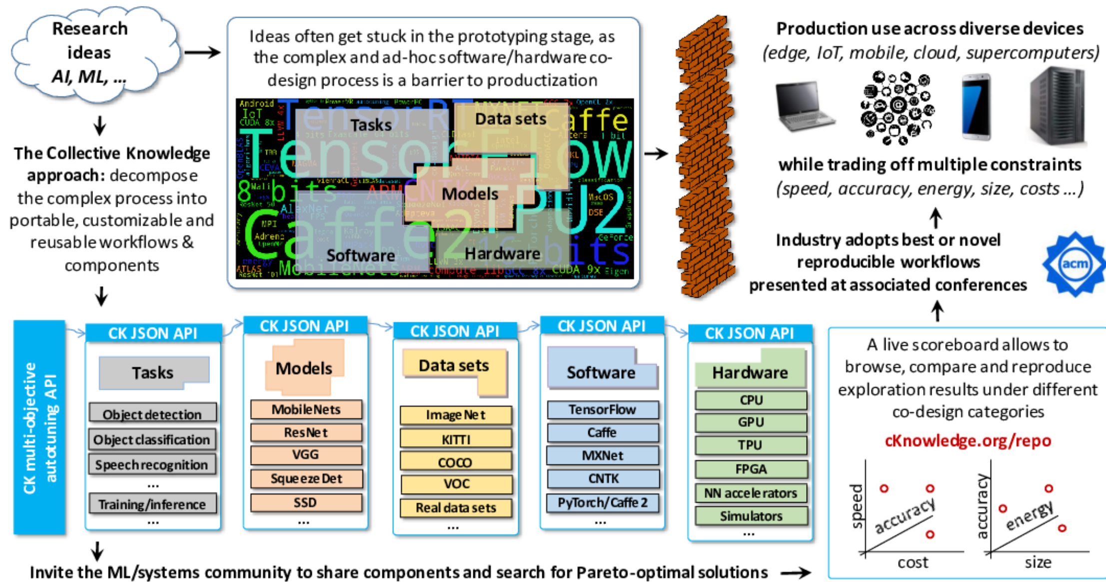


Figure 5.2: We leverage the open Collective Knowledge workflow framework (CK) and the rigorous ACM artifact evaluation methodology (AE) to allow the community collaboratively explore quality vs. efficiency trade-offs for rapidly evolving workloads across diverse systems.

involvement than sharing some CSV/JSON files or checking out a given GitHub repository. That is why we build our competition on top of CK [65] to provide unified evaluation and a real-time leader-board of submissions. CK is an open-source portable workflow framework, used as standard ACM artifact evaluation methodology from ACM and IEEE systems conferences (CGO, PPoPP, PACT, SuperComputing).

CK works a Python wrapper framework to help users share their code and data as customizable and reusable plugins with a common JSON API, meta description and an integrated package manager, adaptable to a user platform with Linux, Windows, MacOS and Android. Researchers can then quickly prototype experimental workflows from shared components, crowd-source benchmarking and autotuning across diverse models, data sets and platforms, exchange results via public scoreboards, and generate interactive reports [1].

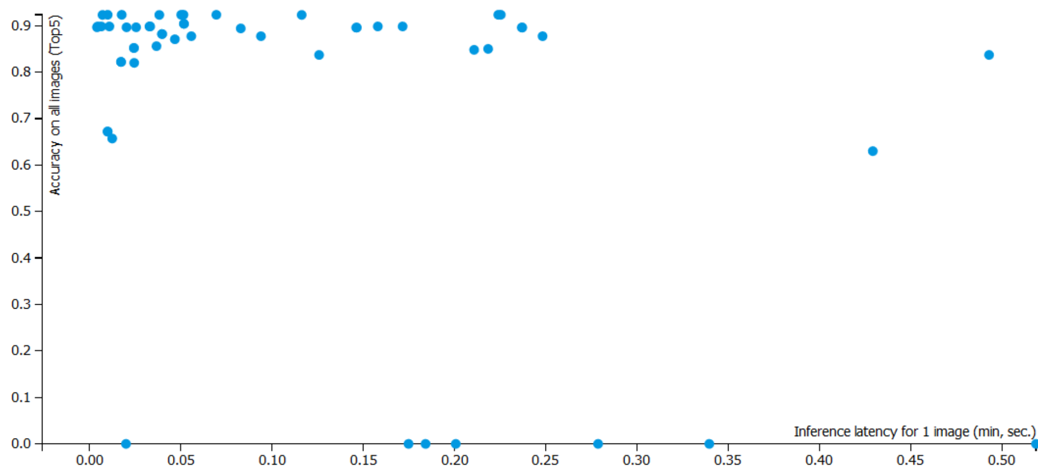


Figure 5.3: A live scoreboard can produce a scatterplot of system implementations across any two dimensions among accuracy, latency, throughput, batch size, price, model size, peak power, clock frequency.

5.3 Artifact Submissions Overview

The ReQuEST-ASPLOS'18 proceedings, available in the ACM Digital Library, include five papers with Artifact Appendices and a set of ACM reproducibility badges.

The CK repository for all ReQuEST-ASPLOS'18 artifacts are documented and available at the following link: https://github.com/ctuning/ck-request-asplos18-results. The interactive live scoreboard can be accessed under the followig URL: http://cKnowledge.org/request-results. The proceedings are accompanied by snapshots of Collective Knowledge workflows covering a very diverse model/software/hardware stack:

- **Models:** MobileNets, ResNet-18, ResNet-50, Inception-v3, VGG16, AlexNet, SSD.
- **Data types:** 8-bit integer, 16-bit floating-point (half), 32-bit floating-point (float).
- **AI frameworks and libraries:** MXNet, TensorFlow, Caffe, Keras, Arm Compute Library, cuDNN, TVM, NNVM.

- **Platforms:** Xilinx Pynq-Z1 FPGA, Arm Cortex CPUs and Arm Mali GPGPUs (Linaro HiKey960 and T-Firefly RK3399), a farm of Raspberry Pi devices, NVIDIA Jetson TX1 and TX2, and Intel Xeon servers in Amazon Web Services, Google Cloud and Microsoft Azure.

In addition all of the submission account for a wide spectrum across the following objective functions:

- **Latency:** 4 .. 500 milliseconds per image
- **Throughput:** 2 .. 465 images per second
- **Top 1 accuracy:** 41 .. 75 percent
- **Top 5 accuracy:** 65 .. 93 percent
- **Model size (pre-trained weights):** 2 .. 130 megabytes
- **Peak power consumption:** 2.5 .. 180 Watts
- **Device frequency:** 100 .. 2600 megahertz
- **Device cost:** 40 .. 1200 dollars
- **Cloud usage cost:** 2.6E-6 .. 9.5E-6 dollars per inference

The community can now access all the above CK workflows under permissive licenses and continue collaborating on them via dedicated ReQuEST'18 GitHub projects. First, the workflows can be automatically adapted to new platforms and environments by either detecting already installed dependencies (e.g. libraries) or rebuilding dependencies via an integrated package manager supporting Linux, Windows, MacOS and Android. Second, the workflows can be customized by swapping in new models, data sets, frameworks, libraries, and so on. Third, the workflows can be extended to expose new design and optimization choices (e.g. quantization), as well as evaluation metrics (e.g. power or memory consumption).

Finally, the workflows can be used for collaborative autotuning ("crowd-tuning") to explore huge optimization spaces using devices such as Android phones and tablets, with best solutions being made available to the community on the online CK scoreboard.

5.4 Lessons Learned and Future Work

Our overwhelmingly positive experience has also allowed us to critically assess limitations to the scalability to our approach. Fair competitive benchmarking between different platforms, frameworks, and models is hard work. It requires carefully considering model equivalence (e.g. performing the same mix of operations), input equivalence (e.g. preprocessing the inputs in the same way), output equivalence (e.g. validating the outputs for each input, not just calculating the usual aggregate accuracy score), etc. Formalizing the benchmarking requirements and encapsulating them in shared CK components (e.g. using a framework-independent model representation such as ONNX) and workflows (e.g. for input conversion and output validation), should help standardize and automate the benchmarking process.

Thorough artifact evaluation can take several person-weeks. Each submitted workflow needs to be studied in detail in its original form and then converted into a common format. However, the more reusable CK components (such as workflows, modules/plugins, packages) are shared by the community, the easier the conversion becomes. For example, we have successfully reused several previously shared components for models, frameworks and libraries, as well as the universal CK workflow for program benchmarking and autotuning. We propose to introduce a new ACM reproducibility badge for such unified ”plug&play” components. This could eventually lead to creating a ”marketplace” for Pareto-efficient implementations (code and data) shared as portable, customizable and reusable CK components.

Finally, full experimental evaluation can take many days/weeks. The AE committee can collaborate with the authors to determine a *minimally useful scope* for evaluation whichwould still provide insights to the community. The community can eventually crowdsource full evaluation. In other words, AE can be ”staged” with a quick check that the artifacts are ”functional” before the camera-ready deadline followed by full evaluation using the ReQuEST methodology. In fact, ReQuEST can grow into a non-profit service to conferences and journals. Sponsorship should help attract experienced full-time evaluators, as well as part-time volunteers, to work on unifying and evaluating artifacts and workflows.

Our experience at ReQuEST-ASPLOS'18 will be repurposed to organize SysML's AE, but at a larger scale. Our long-term vision is to dramatically reduce the complexity and costs of the development and deployment of AI, ML, and other emerging workloads. We believe that having an open repository (marketplace) of customizable workflows with reusable components helps to bring together the multidisciplinary community to collaboratively co-design, optimize, and autotune computer systems across the full model/software/hardware stack. Systems integrators will also benefit from being able to assemble complete solutions by adapting such reusable components to their specific usage scenarios, requirements, and constraints. We envision that our community-driven approach and decentralized marketplace will help accelerate adoption and technology transfer of novel AI/ML techniques similar to the open-source movement.

5.5 Concluding Remarks: An Outlook to the Future

Hardware acceleration has become a critical component of modern computer systems, particularly for scaling their capabilities as Moore's law is running out of steam. As a result, we are living through a *renaissance era* for domain-specialized hardware designs with countless new accelerators being proposed and implemented in varied domains like deep learning [56, 41, 90, 77], data bases [44, 192], graph processing [76, 6, 88] etc. In addition, we are seeing novel spatially programmable hardware accelerators that aim to make hardware acceleration flexible across application domains [137, 135, 173, 132, 28].

This Cambrian explosion of hardware designs pushes our research community to rethink how the software stack is built. To support this proliferation of domain-specific accelerators, we will need faster software integration via high-performance libraries, which will be mostly automated thanks to better DSLs [37], design space exploration tools [38], and modular software-managed accelerator designs [123]. In addition, the push to eliminate more inefficiencies across the stack will favor the design of accelerators that are quality-programmable [97, 60, 92, 186] and therefore can respond to dynamic changes like fluctuating electricity costs in a datacenter, or low battery levels in a smartphone. In order to derive low-level hardware approximation settings from user-defined quality of results constraints, we will need new programming models [155], com-

pilers [154], and quality auto-tuning frameworks [122, 121] that can navigate energy-efficiency tradeoffs and provide viable error guarantees. Finally for designs that are highly customized to specific use-case scenarios in which energy constraints are stable, it makes sense to tailor the entire system stack to quality bounds dictated by its use-case. In domains like deep learning, we will see more uses of vertical optimization across the stack: highly tailored quantized models [142, 51] running on specialized low-power hardware architectures [181]. Eventually, the generation of this domain and use-case specialized stack will be automated with improved search and modeling techniques.

These strategies for eliminating waste across the stack via specialization and Pareto-optimization will help scale the performance and capabilities of systems until a viable replacement for CMOS technology will revitalize the computing landscape. And the lessons we have learned from designing a more efficient CMOS stack will carry across these future technologies.

BIBLIOGRAPHY

[1] Industrial and academic use-cases of Collective Knowledge. `http://cKnowledge.org/partners.html`, 2018.

[2] Tor M Aamodt and Paul Chow. Compile-time and instruction-set methods for improving floating-to fixed-point conversion accuracy. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):26, 2008.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[4] Mohamed S Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane OConnell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C Ling, et al. Dla: Compiler and fpga overlay for neural network inference acceleration. *arXiv preprint arXiv:1807.06434*, 2018.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[6] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News*, 43(3):105–117, 2016.

[7] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016.

[8] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 1–13. IEEE Press, 2016.

[9] Altera Corporation. Altera OpenCL Compiler.

[10] Altera Corporation. Altera SoCs.

[11] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[12] Mariano Alvira and Ryan Rifkin. An empirical comparison of snow and svms for face detection. 2001.

[13] Amazon. Amazon EC2 F1 FPGA Cloud Computing Platform, 2018.

[14] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[15] ARM. ARM Compute Library For Computer Vision and Machine Learning, 2018.

[16] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[17] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[18] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, et al. Perfect (power efficiency revolution for embedded computing technologies) benchmark suite manual. *Pacific Northwest National Laboratory and Georgia Tech Research Institute*, 2013.

[19] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[20] Bilel Belhadj, Antoine Joubert, Zheng Li, Rodolphe Héliot, and Olivier Temam. Continuous real-world inputs can open up alternative accelerator designs. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 1–12. ACM, 2013.

[21] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[22] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, 2010.

[23] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[24] Hugh T Blair, Jason Cong, and Di Wu. Fpga simulation engine for customized construction of neural microcircuits. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 607–614. IEEE, 2013.

[25] Michaela Blott, Thomas Preusser, Nicholas Fraser, Giulio Gambardella, Kenneth O'Brien, and Yaman Umuroglu. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *arXiv preprint arXiv:1809.04570*, 2018.

[26] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, volume 51, pages 775–788. ACM, 2016.

[27] Bernhard E Boser, Eduard Sackinger, Jane Bromley, Yann Le Cun, and Lawrence D Jackel. An analog neural network processor with programmable topology. *IEEE Journal of Solid-State Circuits*, 26(12):2017–2025, 1991.

[28] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[29] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. *arXiv preprint arXiv:1702.00953*, 2017.

[30] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 235–245. IEEE Computer Society, 2015.

[31] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, volume 48, pages 33–52. ACM, 2013.

[32] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Designing a coarse-grained reconfigurable architecture for

power efficiency. In *Department of Energy NA-22 University Information Technical Interchange Review Meeting*, 2007.

[33] Lakshmi N Chakrapani, Bilge ES Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) soc architectures based on probabilistic cmos (pcmos) technology. In *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, volume 1, pages 1–6. IEEE, 2006.

[34] Lakshmi N Chakrapani, Pinar Korkmaz, Bilge ES Akgul, and Krishna V Palem. Probabilistic system-on-a-chip architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):29, 2007.

[35] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[36] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[37] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end compilation stack for deep learning. In *SysML Conference*, 2018.

[38] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018.

[39] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. Benchnn: On the broad potential application scope of hardware neural network accelerators. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 36–45. IEEE, 2012.

[40] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014.

[41] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[42] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[43] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[44] Eric S Chung, John D Davis, and Jaewon Lee. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 261–272. ACM, 2013.

[45] Jai-Hoon Chung, Hyunsoo Yoon, and Seung Ryoul Maeng. A systolic array exploiting the inherent parallelisms of artificial neural networks. *Microprocessing and Microprogramming*, 33(3):145–159, 1992.

[46] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 30–40. IEEE Computer Society, 2004.

[47] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[48] Charles J Clopper and Egon S Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[49] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[50] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[51] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[52] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[53] Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in scala. In *Acm Sigplan Notices*, volume 46, pages 325–344. ACM, 2011.

[54] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Transactions on Computers*, 60(2):242–253, 2011.

[55] Marc De Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. *ACM SIGARCH Computer Architecture News*, 38(3):497–508, 2010.

[56] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[57] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Computer Society, 2003.

[58] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[59] Hadi Esmaeilzadeh, Pooya Saeedi, Babak Nadjar Araabi, Caro Lucas, and Seid Mehdi Fakhraie. Neural network stream processing core (nnsp) for embedded systems. In *ISCAS*, 2006.

[60] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, volume 47, pages 301–312. ACM, 2012.

[61] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[62] Facebook. Glow: A Community-Driven Approach to AI Infrastructure, 2018.

[63] Kevin Fan, Manjunath Kudlur, Ganesh Dasika, and Scott Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 313–322. IEEE, 2009.

[64] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[65] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective knowledge: towards r&d sustainability. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 864–869. EDA Consortium, 2016.

[66] Brian R Gaines. Stochastic computing systems. In *Advances in information systems science*, pages 37–172. Springer, 1969.

[67] Kent Gauen, Rohit Rangan, Anup Mohan, Yung-Hsiang Lu, Wei Liu, and Alexander C Berg. Low-power image recognition challenge. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 99–104. IEEE, 2017.

[68] Google. Google TensorFlow XLA, 2018.

[69] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, et al. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31(2):86–95, 2011.

[70] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 503–514. IEEE, 2011.

[71] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 615–626. IEEE, 2015.

[72] Beayna Grigorian and Glenn Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pages 248–255. IEEE, 2014.

[73] Beayna Grigorian and Glenn Reinman. Accelerating divergent applications on simd architectures using neural networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):2, 2015.

[74] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to fpga: Software-hardware co-design for efficient neural network acceleration. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–27. IEEE, 2016.

[75] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 12–23. ACM, 2011.

[76] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[77] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.

[78] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[79] Atif Hashmi, Hugues Berry, Olivier Temam, and Mikko Lipasti. Automatic abstraction and fault tolerance in cortical microachitectures. In *ACM SIGARCH computer architecture news*, volume 39, pages 1–10. ACM, 2011.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[81] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[82] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[83] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

[84] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[85] Intel Corporation. Disrupting the data center to create the digital services economy.

[86] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md E Haque, Michael A Laurenzano, Scott Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 41. IEEE Press, 2016.

[87] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[88] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 228–241. IEEE, 2015.

[89] Antoine Joubert, Bilel Belhadj, Olivier Temam, and Rodolphe Héliot. Hardware spiking neurons design: Analog or digital? In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–5. IEEE, 2012.

[90] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[91] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, page 23. ACM, 2016.

[92] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[93] Andrew B Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Proceedings of the 49th Annual Design Automation Conference*, pages 820–825. ACM, 2012.

[94] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: Nre optimization in asic clouds. *ACM SIGOPS Operating Systems Review*, 51(2):511–526, 2017.

[95] Daya S Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: An online quality management system for approximate computing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 554–566. IEEE, 2015.

[96] Daya Shanker Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Quality control for approximate accelerators by error prediction. *IEEE Design & Test*, 33(1):43–50, 2016.

[97] Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze, and Visvesh Sathe. Matic: Learning around errors for efficient low-voltage neural network accelerators. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 1–6. IEEE, 2018.

[98] Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze, and Visvesh S Sathe. Energy-efficient neural network acceleration in the presence of bit-level memory errors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, (99):1–14, 2018.

[99] Younghoon Kim, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Designing approximate circuits using clock overgating. In *Proceedings of the 53rd Annual Design Automation Conference*, page 15. ACM, 2016.

[100] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 1742–1752, 2017.

[101] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[102] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[103] Hamed F Langroudi, Zachariah Carmichael, John L Gustafson, and Dhireesha Kudithipudi. Positnn: Tapered precision deep learning inference for the edge. 2018.

[104] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[105] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[106] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

[107] Seogoo Lee, Lizy K John, and Andreas Gerstlauer. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 187–192. European Design and Automation Association, 2017.

[108] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A Jacobson, and Subhasish Mitra. Ersa: Error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1560–1565. European Design and Automation Association, 2010.

[109] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. Sfu-driven transparent approximation acceleration on gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, page 15. ACM, 2016.

[110] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[111] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 393–405. IEEE Press, 2016.

[112] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. Flicker: Saving refresh-power in mobile devices through critical data partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Citeseer, 2009.

[113] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 178–190. IEEE, 2016.

[114] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. *ACM SIGARCH Computer Architecture News*, 44(3):66–77, 2016.

[115] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 50–61. ACM, 2015.

[116] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 127–139. IEEE Computer Society, 2014.

[117] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, volume 49, pages 309–328. ACM, 2014.

[118] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):88, 2013.

[119] Sasa Misailovic, Daniel M Roy, and Martin C Rinard. Probabilistically accurate program transformations. In *International Static Analysis Symposium*, pages 316–333. Springer, 2011.

[120] Asit K Mishra, Rajkishore Barik, and Somnath Paul. iact: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[121] Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. Exploiting quality-energy tradeoffs with arbitrary quantization: special session paper. In *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*, page 30. ACM, 2017.

[122] Thierry Moreau, Felipe Augusto, Patrick Howe, Armin Alaghi, and Luis Ceze. Qappa: A framework for navigating quality-energy tradeoffs with arbitrary quantization. Technical report, Technical Report CMU/CSE-17-03-02, 2017.

[123] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Vta: An open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188*, 2018.

[124] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. A taxonomy of general purpose approximate computing techniques. *IEEE Embedded Systems Letters*, 10(1):2–5, 2018.

[125] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Luis Ceze, Natalie Enright Jerger, and Adrian Sampson. A taxonomy of approximate computing techniques. *UW CSE Technical Report*, pages 1–5, 2016.

[126] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. Snnap: Approximate computing on programmable socs via neural

acceleration. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 603–614. IEEE, 2015.

[127] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L Jones. Scalable stochastic processors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 335–338. IEEE, 2010.

[128] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*, pages 96–96. IEEE, 2004.

[129] Tony Nowatzki and Karthikeyan Sankaralingam. Analyzing behavior specialized acceleration. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 697–711. ACM, 2016.

[130] NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture: The Worldś Most Advanced Data Center GPU, 2017.

[131] NVIDIA Corporation. NVDLA Open Source Project, 2018.

[132] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 142–153. ACM, 2013.

[133] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[134] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. Gated-v dd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 90–95. ACM, 2000.

[135] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 389–402. IEEE, 2017.

[136] K Wojtek Przytula and Viktor K Prasanna. *Parallel digital implementations of neural networks*. Prentice-Hall, Inc., 1993.

[137] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[138] Andrew R Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. Chimps: A high-level compilation flow for hybrid cpu-fpga architectures. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 261–261. ACM, 2008.

[139] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[140] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[141] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[142] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[143] Rahul Razdan and Michael D Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 172–180. ACM, 1994.

[144] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 267–278. IEEE Press, 2016.

[145] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[146] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, pages 41–50. ACM, 2012.

[147] Prasanna Venkatesh Rengasamy, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. Exploiting staleness for approximating loads on cmps. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 343–354. IEEE, 2015.

[148] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *ACM SIGPLAN Notices*, volume 50, pages 399–411. ACM, 2015.

[149] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68. ACM, 2018.

[150] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12. IEEE, 2013.

[151] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*, 2014.

[152] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. *ACM SIGPLAN Notices*, 49(4):35–50, 2014.

[153] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. ACM, 2013.

[154] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. *University of Washington Technical Report UW-CSE-15-01*, 1, 2015.

[155] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[156] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[157] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. *ACM SIGPLAN Notices*, 49(6):112–122, 2014.

[158] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The bunker cache for spatio-value approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[159] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. Wafer-scale integration of analog neural networks. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pages 431–438. IEEE, 2008.

[160] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[161] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *the Workshop on Cognitive Architectures*, 2016.

[162] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[163] David E Shaw, Martin M Deneroff, Ron O Dror, Jeffrey S Kuskin, Richard H Larson, John K Salmon, Cliff Young, Brannon Batson, Kevin J Bowers, Jack C Chao, et al. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, 2008.

[164] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[165] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[166] Scott Sirowy and Alessandro Forin. Where's the beef? why fpgas are so fast. *Microsoft Research, Microsoft Corp., Redmond, WA*, 98052, 2008.

[167] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[168] James E Smith. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, volume 10, pages 112–119. IEEE Computer Society Press, 1982.

[169] Vilas Sridharan, Dean A Liberty, and David R Kaeli. A taxonomy to enable error recovery and correction in software. In *Workshop on Quality-Aware Design*. Citeseer, 2008.

[170] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural specialization for inter-iteration loop dependence patterns. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 583–595. IEEE Computer Society, 2014.

[171] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. *ACM SIGARCH Computer Architecture News*, 42(3):505–516, 2014.

[172] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[173] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[174] Simon M Tam, Bhusan Gupta, Hernan A Castro, and Mark Holler. Learning on an analog vlsi neural network chip. In *Systems, Man and Cybernetics, 1990. Conference Proceedings., IEEE International Conference on*, pages 701–703. IEEE, 1990.

[175] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35, 2002.

[176] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. *ACM SIGARCH Computer Architecture News*, 40(3):356–367, 2012.

[177] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[178] Jonathan Ying Fai Tong, David Nagle, and Rob A Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):273–286, 2000.

[179] Andrew Tulloch and Yangqing Jia. High performance ultra-low-precision convolutions on mobile devices. *arXiv preprint arXiv:1712.02427*, 2017.

[180] UCBarchlab. OpenTPU Project, 2018.

[181] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[182] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Sjalander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. *arXiv preprint arXiv:1806.08862*, 2018.

[183] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[184] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. Towards automatic significance analysis for approximate computing. In *Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on*, pages 182–193. IEEE, 2016.

[185] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 42. IEEE Press, 2016.

[186] Swagath Venkataramani, Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2013.

[187] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1367–1372. IEEE, 2013.

[188] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[189] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*, pages 163–174. IEEE, 2011.

[190] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[191] Daniel Wong, Nam Sung Kim, and Murali Annavaram. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 176–187. IEEE, 2016.

[192] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. Q100: The architecture and design of a database processing unit. In *Acm Sigplan Notices*, volume 49, pages 255–268. ACM, 2014.

[193] Xilinx. CHaiDNN: HLS based Deep Neural Network Accelerator Library for Xilinx Ultrascale+ MPSoCs, 2018.

[194] Xilinx, Inc. Vivado high-level synthesis.

[195] Xilinx, Inc. Xilinx all programmable SoC.

[196] Xilinx, Inc. Zynq UG479 7 series DSP user guide.

[197] Xilinx, Inc. Zynq UG585 technical reference manual.

[198] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, 34(2):60–68, 2017.

[199] Yavuz Yetim, Margaret Martonosi, and Sharad Malik. Extracting useful computation from error-prone processors for streaming applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 202–207. EDA Consortium, 2013.

[200] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

[201] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[202] Jihan Zhu and Peter Sutton. Fpga implementations of neural networks–a survey of a decade of progress. In *International Conference on Field Programmable Logic and Applications*, pages 1062–1066. Springer, 2003.