BREAKING THE ABSTRACTIONS FOR PRODUCTIVITY AND PERFORMANCE IN THE ERA OF SPECIALIZATION

A thesis
Presented to
The Academic Faculty

By

Jongse Park

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology
August 2018

BREAKING THE ABSTRACTIONS FOR PRODUCTIVITY AND PERFORMANCE IN THE ERA OF SPECIALIZATION

Approved by:

Dr. Hadi Esmaeilzadeh
School of Computer Science
College of Computing
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
College of Computing
Georgia Institute of Technology

Dr. Milos Prvulovic
School of Computer Science
College of Computing
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
College of Engineering
Georgia Institute of Technology

Dr. Nam Sung Kim
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign

Date Approved:
July 16, 2018

All things are difficult before they are easy

Thomas Fuller

Dedicated to my wife, Tsuping.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Hadi Esmaeilzadeh for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee, Prof. Hyesoon Kim, Prof. Tushar Krishna, Prof. Milos Prvulovic, and Prof. Nam Sung Kim, for their encouragement, insightful comments, and hard questions.

I am grateful to my internship mentors for offering me amazing opportunities in their groups and leading me working on diverse exciting projects. At Microsoft Research, I had the privilege to closely work with Dr. Doug Burger and Dr. Eric Chung. At NVIDIA Research, I had the privilege to closely work with Steve Keckler, Arslan Zulfiqar, and Eiman Ebrahimi.

I thank my fellow students in ACT Lab, Amir Yazdanbaksh, Bradley Thwaites, Divya Mahajan, Hardik Sharma, Joon Kyung Kim, Jacob Sacks, Emmanuel Amaro, Anandhavel Nagendrakumar, Soroush Ghodrati, and Ahmed Elthakeb, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years. I should also thank Prof. Mayur Naik and Dr. Xin Zhang for giving me tremendous help in growing as a researcher in the early phase of my Ph.D. study.

I am also deeply grateful to Dr. Seungryoul Maeng and Dr. Jaehyuk Huh for enlightening me the first glance of research.

I thank Kwanjeong Educational Foundation Scholarship for generous financial support for last five years and giving me opportunities to meet fellow researchers from various fields.

I would like to thank my family. My father, Sangkyu Park, my mother, Jungsuk Kong, my sister, Yoonse Park, my in-laws, Chungfu Wang, Hungnan Li, Tsufu Wang, Tsuming Wang, Tsuyu Wang, Tsui Wang, Sam Jon, my nephew, Logan Jon, and my unborn niece, Kidong, have given me unconditional love, trust, and support.

Last but not the least, I would like to express my deepest love and gratitude to my wife, Tsuping. I cannot imagine how my life as a Ph.D. student would have been without her. Her dedication, encouragement, and endurance helped me complete everything that I have achieved in recent years. Moreover, I thank my wife for sharing with me every moment of the last 10 years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Over the last decades, general-purpose computing stack and its abstractions have provided both performance and productivity, which have been the main drivers for the revolutionary advances in IT industry. However, the computational demand of emerging applications grows rapidly and the rate of data generation exceeds the level where the capabilities of current computing systems can match. The challenges have coincided with the Dark Silicon era in which conventional technologies offer insufficient performance and energy efficiency. Thus, it is timely to move beyond conventional techniques and explore radical approaches that can overcome the limitations of general-purpose systems and deliver large gains in performance and efficiency. One such approach is *specialization*, where the hardware and systems are developed for a domain of applications. However, the specialization creates a tension between the performance and productivity, since (1) programmers need to delve into the details of specialized hardware, and (2) perform low-level programming. Hence, the objectives are (1) delivering large gains in performance and efficiency (2) while retaining automation and productivity through high-level abstractions. Achieving both of these conflicting objectives is a crucial challenge to place the specialization techniques in a position of practical utility, which is the main focus of this dissertation research. My works offer algorithm-driven computing stacks, which span from algorithms and languages to micro-architectural designs. I have primarily focused on two paradigms of specialization: *acceleration* and *approximation*.

Chapter 1

INTRODUCTION

Conventionally, general-purpose computing has offered both *performance* and *productivity*, which delivered numerous capabilities for our lives. However, the computational demand of emerging applications grows rapidly and the rate of data generation exceeds the level where the capabilities of current computing systems can match. The challenges have coincided with the Dark Silicon era in which conventional technologies offer insufficient performance and energy efficiency. To tackle these challenges, community has started exploring radical approaches that can overcome the limitations of general-purpose systems while providing large performance and efficiency benefits. In this dissertation, we focus on one such solution, hardware specialization. While the specialization techniques offer significant gains in performance and efficiency, they create a tension between the performance and productivity, since (1) programmers need to delve into the details of specialized hardware, and (2) perform low-level programming. Hence, the objectives are (1) delivering large gains in performance and efficiency (2) while retaining automation and productivity through high-level abstractions. Achieving both of these conflicting objectives is a crucial challenge to place the specialization techniques in a position of practical utility, which is the main focus of this dissertation research. My works offer algorithm-driven computing stacks, which span from algorithms and languages to micro-architectural designs. I have primarily focused on two paradigms of

specialization: *acceleration* and *approximation*. My efforts in acceleration leverage algorithmic insights to redefine the hardware-software abstractions and enable programmers to automatically utilize hardware accelerators (e.g., FPGAs) in scale-out setting for emerging workloads, such as data analytics and machine learning. For approximation, I have devised programming language and crowdsourcing software engineering solutions that improve the productivity and utility of approximation technologies, and bridges the gap between unconventional research innovations and practical real-world applications.

1.1 Resolving Performance and Productivity for Acceleration of Machine Learning

A growing number of commercial and enterprise systems increasingly rely on compute-intensive Machine Learning (ML) algorithms. Hardware accelerators offer several orders of magnitude higher performance than general-purpose processors and provide a promising path forward to accommodate the needs of ML algorithms. Even software companies have begun to incorporate various forms of accelerators in their data centers. Microsoft's Project Brainwave integrated FPGAs in datacenter scale for real-time AI calculations and Google developed the TPU as a specialized matrix multiplication engine for machine learning. However, not only do the benefits come with the cost of lower programmability, but also the acceleration requires long development cycles and extensive expertise in hardware design. Moreover, conventionally, accelerators are integrated with the existing computing stack by profiling hot regions of code and offloading the computation to the accelerators. This approach is suboptimal since the stack is designed and optimized merely for CPUs, the sole processing platform up until very recently. To tackle these challenges, we

developed cross-stack and algorithm-hardware co-designed solutions that rebuild the computing stack for acceleration of machine learning. These solutions break the conventional abstractions of computing stack by reworking the entire layers of computing stack, which include programming language, compiler, system software, accelerator architecture, and circuit generator.

1.1.1 Full Stack Solution for Scale-Out Acceleration of Learning

Recently, community has started exploring mostly single-node acceleration techniques to meet the massive compute demand of ML. In a concurrent yet disjoint effort, others have also explored the use of distributed general-purpose systems (e.g., Spark and Hadoop) as a mean to scale the learning frameworks. However, there is a gap between these accelerators and scale-out systems due to the lack of solutions that enable distributed acceleration of learning at scale. To bridge these two paradigms, we developed CoSMIC [1], a full computing stack that constitutes language, compiler, system software, template architecture, and circuit generators, which enable programmable acceleration of learning at scale. CoSMIC enables programmers to exploit scale-out acceleration using FPGAs and Programmable ASICs (P-ASICs) from a high-level and mathematical Domain-Specific Language (DSL). Nonetheless, CoSMIC does not require programmers to delve into the onerous task of system software development or hardware design. CoSMIC achieves three conflicting objectives of efficiency, automation, and programmability, by integrating a novel multi-threaded template accelerator architecture and a cohesive stack that generates the hardware and software code from its high-level DSL. CoSMIC can accelerate a wide range of learning algorithms that are most commonly trained using parallel variants of gradient descent. The key is to distribute partial gradient calculations of the learning algorithms across the accelerator-augmented

nodes of the scale-out system. Additionally, CoSMIC leverages the parallelizability of the algorithms to offer multi-threaded acceleration within each node. Multi-threading allows CoSMIC to efficiently exploit the numerous resources that are becoming available on modern FPGAs/P-ASICs by striking a balance between multi-threaded parallelism and single-threaded performance. CoSMIC takes advantage of algorithmic properties of machine learning to offer a specialized system software that optimizes task allocation, role-assignment, thread management, and internode communication. While accelerators gain traction, their integration in the system stack is not well understood. CoSMIC takes an initial step toward such an integration for an important class of applications, while providing generality and a high-level programming interface.

1.1.2 Algorithmic Approaches to Accelerate Machine Learning.

As a preliminary effort for the CoSMIC project, in collaboration with my fellow graduate students, we developed a single-node FPGA accelerator generation framework for data analytics, dubbed TABLA [2], which enables FPGA acceleration from high-level specifications of algorithms. We open-sourced the code and it is available at http://act-lab.org/artifacts/tabla. TABLA leverages the insight that many learning algorithms can be solved using stochastic gradient descent that minimizes an objective function. The solver is fixed while the objective function changes with the learning algorithm. Therefore, TABLA uses stochastic optimization as the abstraction between hardware and software. Consequently, programmers specify the learning algorithm by merely expressing the gradient of the objective function in our domain specific language. TABLA then automatically generates the synthesizable implementation of the accelerator for FPGA realization using a set of template designs. Real hardware measurements show orders of magnitude higher performance and power efficiency

while the programmer only writes 60 lines of code.

As a follow-on work, we developed DNNWEAVER [3], a framework that automatically generates a synthesizable accelerator for a given (DNN, FPGA) pair from a high-level specification in Caffe. To achieve large benefits while preserving automation, we devised hand-optimized design templates that the DNNWEAVER framework uses to generate the accelerators. First, DNNWEAVER translates a given high-level DNN specification to its novel ISA that represents a macro dataflow graph of the DNN. The DNNWEAVER compiler is equipped with our optimization algorithm that tiles, schedules, and batches DNN operations to maximize data reuse and best utilize target FPGA's memory and other resources. The final result is a custom synthesizable accelerator that best matches the needs of the DNN while providing high performance and efficiency gains for the target FPGA.

1.2 Improving Productivity in Approximate Computing

Approximate computing is another form of specialization, which brings forth an unconventional yet innovative computing paradigm that trades accuracy of computation for otherwise hard-to-achieve performance and efficiency. This new computing paradigm is built upon the property that emerging applications (e.g., sensor processing, translation, vision, and data analytics) are increasingly tolerant to imprecision. Leveraging this property, approximation techniques are able to provide orders of magnitude higher performance and efficiency gains, while maintaining the acceptable level of functionalities. However, these techniques are only pragmatic when (1) they are easy to use for the programmers, and (2) they produce acceptable output quality from the perspective of application users. To this end, my research efforts for approximation focus on improving

productivity and utility of approximation technologies by developing programming language and crowdsourcing-based software engineering solutions.

1.2.1 Practical Approximate Programming

While approximate computing is currently a hot area, the programmability of approximation techniques is still one of the pivotal challenges to enable their practical and prevalent use. State-of-the-art approximate programming models require extensive manual annotations on program data and operations to guarantee safe execution of approximate programs. The need for extensive manual annotations hinders the pragmatic use of approximation techniques. We developed a small set of language extensions, dubbed FlexJava, that significantly reduces the annotation effort, paving the way for practical approximate programming [4]. These extensions enable programmers to annotate approximation-tolerant method outputs. The FlexJava compiler, which is equipped with an approximation safety analysis, automatically infers the operations and data that affect these outputs and selectively marks them as approximable while providing safety guarantees. The automation and the language-compiler co-design relieve programmers from manually and explicitly annotating data declarations or operations as safe to approximate. FlexJava is designed to support safety, modularity, generality, and scalability in software development. Compared to other models, FlexJava largely reduces the number of annotations and programmers spend significantly less time annotating programs based on our user study.

1.2.2 Crowdsourcing Quality Target Determination in Approximate Computing

Approximation is useful only if its impact on application output quality is acceptable to the users. However, there is a lack of systematic solutions and studies that explore users' perspective on the effects of approximation. We sought to provide one such solution for the developers to probe and discover the boundary of quality loss that most users will deem acceptable. We proposed AxGames, a crowdsourced solution that enables developers to readily infer a statistical common ground from the general public through three entertaining games [5]. The users engage in these games by betting on their opinion about the quality loss of the final output while the AxGames framework collects statistics about their perceptions. The framework then statistically analyzes the results to determine the acceptable levels of quality for a pair of (application, approximation technique). The three games are designed such that they effectively capture quality requirements with various tradeoffs and contexts. We recruited 700 participants/users through Amazon's Mechanical Turk to play the games that collect statistics about their perception on different levels of quality. Subsequently, the AxGames framework uses the Clopper-Pearson exact method statistically project the quality level that satisfies a given percentage of users. The developers can use these statistical projections to tune the level of approximation based on the user experience.

In addition to the aforementioned works, I have worked on many collaborative projects that aim to develop hardware-software co-designed approximation techniques. These techniques seek opportunities for approximation at various components of computing stack, which span from neural accelerators [6, 7] and memory subsystems [8] to GPUs [9] and hardware description languages [10].

1.3 Thesis Contributions

This dissertation makes the following contributions.

1. We propose a series of full-stack solutions that bridge the semantic gap between the programmers and specialized accelerators by leveraging the algorithmic insights of Machine Learning algorithms. We provide CoSMIC–Computing Stack for ML acceleration In the Cloud–that offers a complete computing stack for scale-out acceleration of machine learning, which comprises a domain-specific language, a compiler, a specialized run-time system, and a multi-threaded template architecture for the accelerator. As its prior effort, we propose TABLA–a single node predecessor of CoSMIC–that enables the automated acceleration of Machine Learning on FPGAs. DnnWeaver is another work in the same line of research, which enables the acceleration of DNN inference on FPGAs from high-level specification of DNN models. **Chapter 2** describes these solutions in more detail.

2. We provide a practical and automated programming model, FlexJava, for approximation techniques, which leverages automated program analysis techniques for more effective approximate programming. Such effort is imperative to enabling the widespread applicability of approximation techniques. The FlexJava language is designed to be intuitive and support essential aspects of modern software development: safety, modularity, generality, and scalability. We believe that FlexJava takes an effective and necessary step toward leveraging approximation in modern software development. **Chapter 3** elaborates the language constructs, analysis, and evaluation in detail.

3. We provide an automated programming tool, AxGames, that methodically utilize crowdsourcing in identifying the desirable application output quality from the final users. This readily available tool provides a path for the research community to better assess their innovative approximation techniques. The framework enables developers to conveniently study user responses at scale and gain statistical confidence when deploying approximated applications. Our results from examining a variety of applications show the necessity of solutions such as AxGames since the crowd's response to approximation varies drastically across different applications, tradeoff, and context. These results suggests that AxGames can add an unexplored, yet important, dimension to the research and development in approximate computing. **Chapter 4** illustrates the three games, statistical analysis, and experiments including user studies, in more detail.

Chapter 2

SCALE-OUT ACCELERATION OF MACHINE LEARNING

2.1 Introduction

Prevalence of interconnected compute platforms has transformed the IT industry, which is now rapidly moving towards scale-out solutions that extract insights from data. Following this trend, systems that enable distributed computing on general-purpose platforms are gaining eminence (e.g., Spark [16] and Hadoop [17]). In a concurrent yet disjoint effort, due to the diminishing benefits from general-purpose processing, the community is developing mostly single-node accelerators for a variety of applications, including machine learning [18, 19, 20, 21, 22, 23, 24, 25, 3, 2]. However, there is a gap between scale-out systems and accelerators due to the lack of solutions that enable distributed acceleration at scale. Moreover, it is not enough to just design and integrate accelerators independent from algorithms and programming interfaces. We need a holistic approach that reworks the fundamental hardware-software abstractions and enables a broad community of programmers to seamlessly utilize accelerators at scale for a specific domain of applications. Reusing the traditional stack for scale-out acceleration is inadequate as the entire computing stack is designed and optimized merely for CPUs, which were the sole processing platform up until recently. To that end, this work sets out to design a full

and specialized computing stack, dubbed CoSMIC[1], for scale-out acceleration of learning.

CoSMIC offers the entire stack of layers to execute a wide range of learning algorithms on accelerator-augmented scale-out systems. These layers comprise a domain-specific language, a compiler, a specialized runtime system, and a multi-threaded template architecture for the accelerator. The template architecture can be automatically tailored for deployment on FPGAs or realization as custom Programmable ASICs (P-ASICs). FPGAs offer flexibility as well as efficiency and are becoming readily available in different markets [26, 27, 28, 29], now even in Amazon Elastic Compute Cloud (EC2) [29]. Not only have FPGAs become a lower-cost alternative to ASICs, but also serve as prototypes for custom chip design. However, designing efficient accelerators is onerous even when targeting a single-node FPGA and requires extensive expertise in both hardware design and application domain. This challenge is exacerbated in the scale-out setting due to the added complexity of task distribution and communication. Additionally, P-ASICs impose high non-recurring engineering costs over long design periods and usually need unintuitive or narrow programming interfaces. Furthermore, as technology is scaled, modern FPGAs and ASICs can harbor an ample amount of resources, whose effective utilization necessitates rethinking accelerator design paradigms. Therefore, to realize scale-out acceleration, we address the following triad of challenges when devising the CoSMIC full stack: (1) efficiently exploiting large number of on-chip resources, (2) enabling distributed acceleration using accelerator-augmented nodes, and (3) relieving programmers of distributed system coordination and the onus of hardware design. Furthermore, CoSMIC targets a wide class of learning algorithms and provides support for new learning models and algorithmic changes

[1] **CoSMIC**: **Co**mputing **S**tack for **M**L acceleration **I**n the **C**loud

to the existing ones. To realize CoSMIC we were required to address the following research challenges.

(1) How to enable scale-out acceleration of many ML algorithms, yet disengage programmers from hardware design.

To tackle this challenge, CoSMIC leverages a combination of two theoretical insights: (1) a wide range of learning algorithms are stochastic optimization problems, solved using a variant of gradient descent [30, 31, 2]; (2) differentiation is a linear mathematical operator, and thus the gradient over a set of data points can be calculated as an aggregated value over the partial gradients computed in parallel for each point [32, 33, 34, 35, 36, 37, 38]. A variety of learning algorithms can be parallelized using these two insights. Examples include, but are not limited to, recommender systems, Kalman filters, linear and nonlinear regression models, support vector machines, least square models, logistic regression, backpropagation, softmax functions, and conditional random fields. To implement these algorithms, one needs to have (1) the partial gradient calculation function, (2) the aggregation operator, and (3) the number of data points that are processed before each aggregation. The first layer of the CoSMIC stack exposes a high-level mathematical language to programmers to specify these three constructs, which capture the entirety of the learning algorithm. The next layer of the CoSMIC stack fully automates the scale-out acceleration. The CoSMIC compiler maps and schedules the operations on the distributed accelerators. The next layer, a specialized runtime system, assigns roles and tasks for the scale-out system components and orchestrates the distributed calculation of the partial gradients and their iterative aggregation. The final layer of the CoSMIC stack provides a novel multi-threaded template architecture for the accelerators. This layer can be automatically customized and tailored according to the high-level specification of the learning algorithm and

the constraints of the system.

(2) How to design customizable accelerators that efficiently exploit the large capacity of advanced process technologies.

Advanced manufacturing processes have made integration of compute and storage resources on the chip. As a result, even modern FPGAs offer large capacities—e.g. Intel Arria 10 [39] instances comprise 1,518 DSP slices with 6.6 MBytes of storage and Xilinx UltraScale+ in Amazon EC2 [29] includes 6,840 DSP slices and 43 MBytes of storage. A single instance of learning algorithm may not effectively exploit resources since it is limited by the fine-grained parallelism in its Dataflow Graph (DFG). Therefore, CoSMIC offers a novel Multiple-Instruction Multiple-Data (MIMD) multi-threaded template architecture that divides the resources across multiple instances of the learning algorithm as independent threads. The last layer of CoSMIC customizes this template and generates the final accelerator by striking a balance between the number of threads running on the chip and the resources assigned to each thread. The code generation differs for FPGAs and P-ASICs. For FPGAs, the generated core is tailored to one specific learning algorithm as the chip can be erased and reprogrammed for different applications. For P-ASICs, the generated accelerator is a programmable superset of the design that fits in the area and power budget of the chip. Any algorithm that can be expressed using the DSL can be compiled and accelerated on the generated P-ASIC. The generated code and template are in the form of Register-Transfer Level (RTL) Verilog code. The template architecture is designed, optimized, and implemented by experts once in Verilog, which ensures efficiency although CoSMIC generates the accelerators automatically. More specifically, the template is designed as a two-dimensional matrix of compute units to ensure data dependencies and within-thread communications do not curtail its scalability to rather large num-

ber of processing elements. We also designed a tree-like bus to connect the rows and allocated bidirectional communication across columns. Hence, the communication latency only grows by a logarithmic order with an increase in the number of compute units, improving on-chip scalability. Furthermore, CoSMIC's backend compiler minimizes data movement by mapping operations to where their operands are located. This hardware-software co-design that aims to maximize effective resource utilization ensures effective utilization of on-chip resources, especially when they are plentiful.

(3) How to devise the system software that is specialized for distributed multi-threaded acceleration of learning.

To be inline with the recent industry trends in integrating accelerators in datacenters [27, 28, 29], CoSMIC targets commodity distributed systems in which accelerators sit on the high-speed expansion slots (e.g., PCIe). For generality, we assume no special connectivity between the accelerators although such connectivity will most likely improve the benefits of CoSMIC. CoSMIC aims to best utilize the system-wide resource on both CPUs and accelerators. CoSMIC achieves this objective by offering a lean and specialized system software layer that exclusively supports learning algorithms that can be trained using parallel variants of stochastic gradient descent. This specialized layer allows the CoSMIC stack to assign the partial gradient calculation onto the accelerators while the CPUs perform aggregation and networking. This task assignment alleviates the use of accelerator resources for TCP/IP communication, avoids data copies to accelerator boards for aggregation, and enables using commodity distributed systems with CoSMIC. Moreover, it maximizes system-wide resource utilization as well as portability to different accelerator boards. Within each node, the system software maintains an internal thread pool. These threads handle the communication with the remote peer nodes. Internally managing this thread

pool avoids costly OS-level context switches. The system software layer also maintains another internal thread pool that asynchronously aggregates the partial gradients. In addition, this layer assigns roles to the nodes and orchestrates the exchange of partial gradients and their aggregation.

We evaluate the benefits of the CoSMIC stack using 10 different learning applications from various domains including medical diagnosis, computer vision, finance, audio processing, and recommender systems. We compare CoSMIC against Spark, a popular framework for scale-out computing using the optimized MLlib machine learning library [40]. On average, a 16-node CoSMIC with UltraScale+ VU9P FPGAs offers 18.8× speedup over a 16-node Spark system with Xeon E3 Skylake CPUs while the programmer only writes 22–55 lines of code. When scaling the nodes from 4 to 16, CoSMIC's performance improves by 2.7×, while Spark's performance scales only by 1.8×. We also compare the CoSMIC system with the distributed GPU (NVIDIA Tesla K40c) implementation. We report the benefits of CoSMIC for two P-ASIC implementations that match the compute resources and off-chip bandwidth of the FPGA and the GPU. On average, these P-ASICs offer 1.2× and 2.3× higher system-wide performance, while the GPU delivers 1.5× speedup over FPGA system. While using custom chips can improve computation time by 11.4×, the system-wide performance benefits are limited to 2.3×. Finally, with CoSMIC's novel multi-threaded accelerator architecture, the FPGA and the two P-ASIC systems respectively achieve 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than the GPU system. These results confirm that CoSMIC is an effective and vital initial step to enable acceleration of learning at scale. To this end, this work not only contributes the full stack of CoSMIC, but also defines a new multithreaded accelerator architecture, a novel communication-aware scheduling and mapping algorithm, and a lean and specialized system software for thread

management and system orchestration.

2.2 Distributed Learning

The CoSMIC stack empowers programmers to exploit accelerator-augmented distributed systems for a wide range of learning algorithms without requiring them to deal with the laborious task of hardware design and system software programming. Although providing higher performance drives this work, programmability and generality are its other two pillars. CoSMIC facilitates programming by exposing a math-oriented DSL to programmers to express various learning algorithms as stochastic optimization problems. The layers of the CoSMIC stack compile this high-level specification to generate the accelerator architecture, and offer the system software that orchestrates them for scale-out execution. This stack is not designed for a specific ML algorithm. Instead, it is adept at accelerating learning algorithms that can be trained using variants of gradient descent optimizer. This section provides the theoretical foundation of these type of algorithms.

2.2.1 Learning as Stochastic Optimization

CoSMIC targets a wide range of supervised machine learning algorithms. These algorithms have two phases: training and prediction (inference). We focus on training, as it is more complex and involves several passes of prediction-tuning over the training data. Since training involves prediction, CoSMIC can accelerate prediction as well.

Each machine learning algorithm is identified by a set of parameters (θ) and a transfer function (g), that maps an input vector (X_i) to a predicted output vector (Y_i). As Equation 2.1 illustrates, training is the process of finding θ such that

the predicted output $Y_i = g(\theta, X_i)$ has a minimum difference from the expected output Y_i^* for all input-output pairs (X_i, Y_i^*) in the training dataset.

$$\text{Find } \theta \ni \{Loss = \sum_i f(\theta, X_i, Y_i^*) = \sum_i \langle g(\theta, X_i) - Y_i^* \rangle\} \text{ is Minimized} \tag{2.1}$$

This unique loss function ($\sum_i f(\theta, X, Y^*)$) defines each of the learning algorithms in our target class. A machine learning algorithm learns the model (θ) by solving an optimization problem that minimizes this loss function ($\sum_i \langle g(\theta, X_i) - Y^* \rangle$). To learn a model ($\theta$), optimization algorithms iterate over the training data and gradually reduce the loss by adjusting the model parameters. One of the most common [31, 30, 63] optimization algorithm is Stochastic Gradient Descent (SGD). SGD is based on the observation that a function decreases fastest in the negative direction of its gradient.

$$\theta^{(t+1)} = \theta^{(t)} - \mu \times \frac{\partial (f(\theta^{(t)}, X_i, Y_i))}{\partial \theta^{(t)}} \tag{2.2}$$

As Equation 2.2 shows, each iteration t of SGD calculates $\theta^{(t+1)}$ by updating $\theta^{(t)}$ in the negative direction of the gradient (∂f) with a learning rate (μ). The process is repeated until the loss is minimized. The gradient function varies with the learning algorithm, while the rest of the process is fixed. Hence, our stack requires programmers to specify the algorithm by expressing the gradient of its loss function ($\frac{\partial f}{\partial \theta}$).

2.2.2 Parallelizing Stochastic Optimization

SGD only consumes one input-output vector (X_i, Y_i) per iteration, traversing the entire data sequentially. Thus, basic SGD is impractical for scale-out acceleration, where the training data is large and dispersed across multiple nodes.

To enable scale-out acceleration, we exploit the insight that gradient is a linear operator. Therefore, the gradient over a set of data points can be computed by aggregating partial gradients calculated over partitions of this set. Different parallel variants of SGD [32, 33, 34, 35, 36, 37, 38] have been developed, which differ in how they iterate over the partitions and aggregate the partial gradients. For instance, the batched gradient descent algorithm [34] uses summation for aggregation, whereas the parallelized SGD [33] uses averaging. Equation 2.3 shows the use of parallelized stochastic gradient descent algorithm [33], for distributed learning.

$$\underset{j:1\rightarrow n}{Parallel}\langle \theta_j^{(t+1)} = SGD\big(\{XY_1,...,XY_b\},\theta^{(t)},f\big)\rangle \tag{2.3a}$$

$$\theta^{(t+1)} = \frac{\sum_j \theta_j^{(t+1)}}{n} \tag{2.3b}$$

As shown, each node independently performs the traditional stochastic gradient descent for b input-output pairs ($\{XY_1,...,XY_b\}$) and calculates a set of partial updates, $\theta_j^{(t+1)}$. These partial updates are aggregated with averaging, which yields the overall update ($\theta^{(t+1)}$). Equation 2.3a and 2.3b are repeated until the loss function f is minimized and the model is trained. The meta parameter b, called the mini-batch size, is the amount of local data that is processed before each aggregation step. CoSMIC expects the programmer to provide the gradient ($\frac{\partial f}{\partial \theta}$), aggregation operator (σ), and mini-batch size (b). Using only this information, CoSMIC orchestrates the scale-out acceleration of the learning algorithm. The next section discusses the accelerated execution flow and the system software layer.

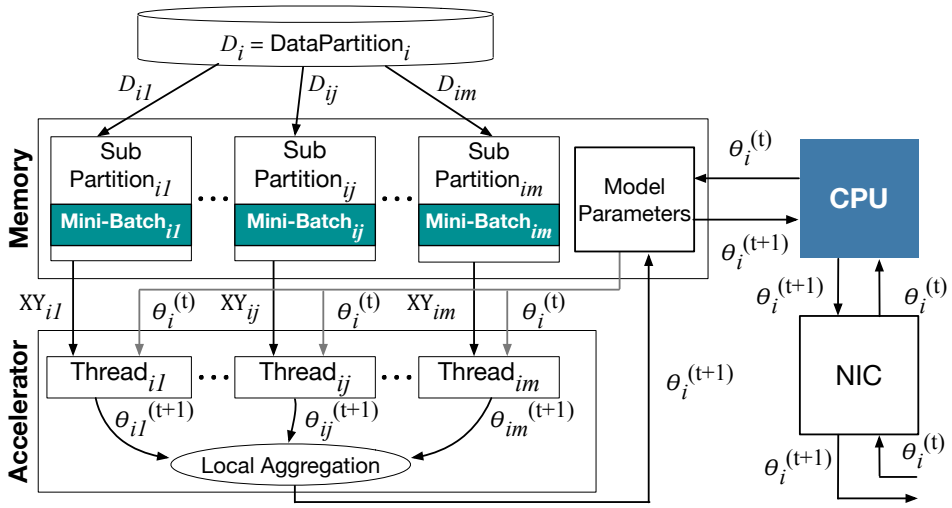


Figure 2.1: Execution and acceleration flow within each node.

2.3 CoSMIC System Software

CoSMIC targets scale-out systems with commodity nodes that use off-the-shelf CPUs. Each node hosts an accelerator board, identical across all the nodes and installed on a high-speed expansion slot such as PCIe. The nodes communicate through conventional TCP/IP stack via a Network Interface Card (NIC). We choose to use commodity host systems, networking hardware-software to alleviate dependency on a particular part. To understand the specialized system software layer of CoSMIC, we first need to delve into the overall execution flow across the nodes of the scale-out system.

2.3.1 Execution and Acceleration Flow

Figure 2.1 illustrates a single node of the system. Each node stores a partition (D_i) of the training dataset. We have devised a multi-threaded ML accelerator for the nodes, which will be discussed in Section 2.5. To utilize multi-threading in the accelerator, the node further divides its data into equally sized sub-partitions (D_{i1} , ..., D_{ij}, ... D_{im}). These data sub-partitions are simultaneously

processed by the accelerator. In Figure 2.1, each accelerator Thread$_{ij}$ calculates its own private partial gradient ($\theta_{ij}^{(t+1)}$) by consuming a sub-partition of the training data. After the partial gradient updates are calculated, the multi-threaded accelerator aggregates them locally and produces the node's partial gradient update ($\theta_i^{(t+1)}$). The host CPU sends this locally-aggregated partial gradient update ($\theta_i^{(t+1)}$) to a special node that maintains the trained model parameters for a group of nodes. We refer to this special node as a Sigma node, while other nodes are called Delta nodes. The system software layer of CoSMIC performs the aggregation in a hierarchical manner to avoid overwhelming a single Sigma node. In the first level of the hierarchy, the group Sigma node calculates the group aggregate. In the next level of the hierarchy, a master Sigma node combines the aggregates. Besides aggregation, the Sigma nodes compute their own partial gradient updates, as they are also equipped with accelerators. After the aggregation, the Sigma nodes distribute the updated model parameters back to all the nodes and threads and invoke training for the next mini-batch.

2.3.2 Task Assignment in the System Software

CoSMIC offers a lean and scalable system software layer that amortizes the cost of OS-level context switches, networking, and general thread scheduling; avoids unnecessary data copies; and matches tasks to the system resources. To devise this layer, we leverage the observation that aggregation is significantly less compute intensive than partial gradient calculations. Hence, the system software layer assigns the partial gradient calculation to the accelerators, while the CPUs perform aggregation and networking. This task assignment alleviates the use of accelerator resources for TCP/IP communication, avoids data copies to accelerator boards for aggregation, and enables using commodity distributed

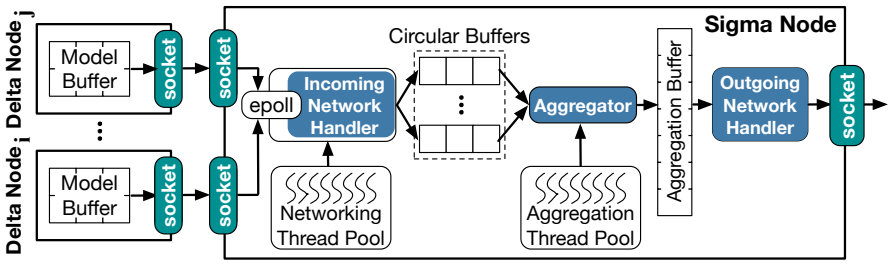


Figure 2.2: System software in a Sigma node.

systems. Moreover, it maximizes system-wide resource utilization and portability to different accelerator boards. To avoid extra data transfer with the memory and the host CPU, each accelerator internally aggregates the partial gradients for all its worker threads. Delta nodes send these partially aggregated gradients to their corresponding Sigma node. The system software workflow in the Sigma nodes is as follows.

Internal thread pools for networking and aggregation. Figure 2.2 illustrates the system software and its subroutines in the Sigma nodes. The main objective in devising these subroutines is to avoid the cost of generic thread management (creation, scheduling, and context switches) and networking by exploiting the specific execution flow of our class of learning algorithms. These subroutines need to open a socket for each communicating node. A naive approach would assign an active thread to handle each socket and spawn a thread to aggregate the received partial gradients. In contrast, the CoSMIC system software *internally* manages two thread pools, Networking Pool and Aggregation Pool as shown in Figure 2.2, limiting the number of active threads and reusing them as described below. When a Sigma node receives a partial update, our Incoming Network Handler catches the recv event using the Linux epoll system call. The epoll system call is effective since it does not require a linear scan on the list of monitored sockets. The Incoming Network Handler assigns a thread from the Networking Pool to copy the received data from the socket buffer in the kernel

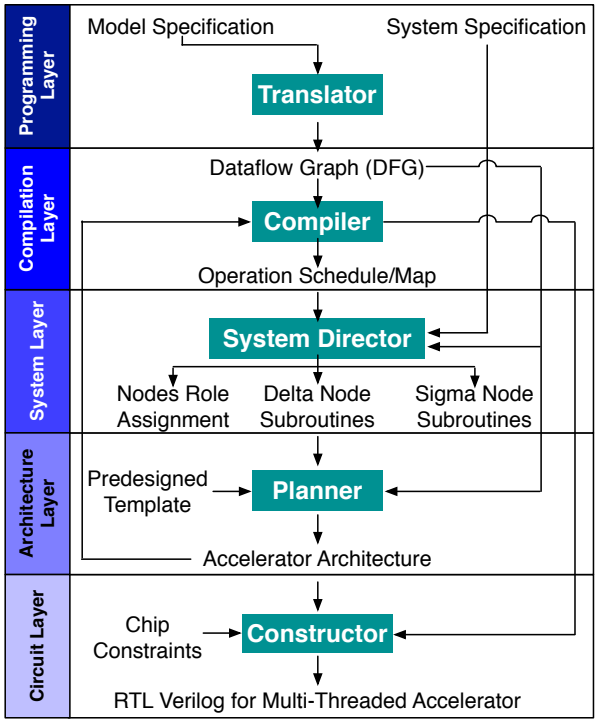
space to a Circular Buffer for aggregation (Figure 2.2). We use Circular Buffers for concurrent networking and aggregation while each corresponding thread deals with smaller portions of data. As soon as the first chunk of data is copied, a thread from the Aggregation Pool starts processing the data and updates the Aggregation Buffer. This buffer holds the results of overall aggregation. The networking threads are data producers, while the aggregation threads are the consumers. Since Sigma nodes communicate with multiple other nodes, this approach uses the multi-threading capabilities of the CPUs to improve concurrency. The Circular Buffer reduces the memory required for aggregating partial results from multiple sources while enabling overlap between communication and computation. Our internally managed thread pools (1) alleviate the need to create an active thread for each connection, limiting the number of active threads; (2) reuse threads for different connections, mitigating the cost of context switching; and (3) use a producer-consumer semantics between the two thread pools, specializing their scheduling. These techniques avert the cost of generic thread management (creation, scheduling, and context switches), which is oblivious to the execution flow of machine learning.

2.4 The CoSMIC Stack

Figure 2.3 illustrates the layers of the CoSMIC stack and their interworking that orchestrates Sigma and Delta nodes and enable scale-out acceleration. This section discusses each layer briefly.

2.4.1 Programming Layer

Our stack makes the accelerator-augmented scale-out systems programmable from a high-level DSL. With CoSMIC, programmers use our extension of the

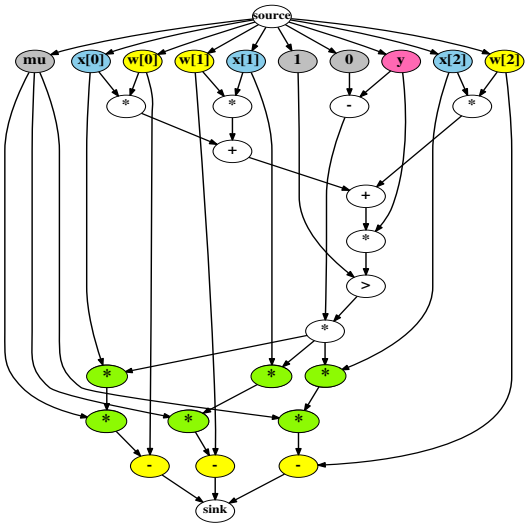


Programming Layer	Algorithmic Specification	Partial Gradient
		Aggregation Operator
		Mini-Batch Size
	System Specification	Number of Nodes
		Number of Groups
		Accelerator Type
Compilation Layer	**Translator**	Dataflow Graph (DFG)
	Compiler	Operation Schedule/Map
System Layer	**System Director**	Node Roles
	System Subroutines: Delta Nodes	Accelerator Invocation Module
		Module for Communication with Sigma Node
	System Subroutines: Sigma Nodes	Accelerator Invocation Module
		Networking Thread Pool for Communication with Delta Nodes
		Circular Buffer for Consumer-Producer Networking & Aggregation
		Aggregation Thread Pool
		Module for Communication with Next Level of Hierarchy Node
Architecture Layer	Hand-Optimized Template Design	RTL Verilog
	Performance Estimation Tool	Design Space of Possible Architectures
	Planner	Number of Threads
		Resources per Thread
		Accelerator Datapath
Circuit Layer	**Constructor**	RTL Verilog of the Multi-Threaded Accelerator

Figure 2.3: The full CoSMIC stack.

```
1  mu = 0.01; // learning rate
2  m = 3; // num of features
3  minibatch_size = 10000;
4
5  model_input x[m];
6  model_output y;
7  model w[m];
8  gradient g[m];
9  iterator i[0:m];
10
11 h = sum[i](w[i] * x[i]);
12 c = y * h;
13 g[i] = ((c > 1) * (0 - y)) * x[i];
14
15 n = 10; // number of nodes
16 aggregator(n) {
17       iterator j[0:n];
18       w[i] = (sum[j](w[i])) / n; }
```

(a)



(b)

Figure 2.4: (a) Programmer specifies the classification algorithm as its gradient and aggregation functions. (b) Translator outputs the DFG.

high-level language, developed in the prior work [2] that focuses on *single-FPGA* acceleration of learning. We chose to extend this DSL since it has a one-to-one mapping to mathematical formulations instead of providing linear algebra primitives as proposed in the past [64]. Moreover, it is open source and publicly available (http://act-lab.org/artifacts/tabla). Using the extended language, programmers express the mathematical formula of the partial gradient and the aggregation operator in a textual format. Additionally, the programmer declares the mini-batch size. Figure 2.4a illustrates how a programmer uses our stack to accelerate the training of a binary classifier based on support vector machines. The first part of the code is the textual representation of Equation 2.4.

$$Gradient_i = \begin{cases} -y \times X_i, & \left(\left(\sum_i X_i \times W_i\right) \times y\right) > 1 \\ 0, & \left(\left(\sum_i X_i \times W_i\right) \times y\right) \leq 1 \end{cases} \tag{2.4}$$

The code has three segments: data declarations, gradient formulation, and

aggregator specification. The DSL provides five data types: model_input, model_output, model, gradient, and iterator. These types denote the semantics of the variables in learning algorithms, and the statements represent the mathematical operations. For instance, the $\sum_i X_i \times W_i$ term in Equation 2.4 is implemented as sum[i](w[i] * x[i]), where x and w are declared as model_input and model, respectively. The iterator i represents the subscript in $\sum_i$. The aggregation function of the parallelized SGD, which averages the partial gradients, is specified by w[i] = sum[j](w[i]) / n. This high-level expression is then converted to a Dataflow Graph (DFG) by the Translator (Figure 2.4b).

2.4.2 Compilation Layer

In a conventional computing stack, the next natural step after translation would be compilation. However, in our specialized stack, the order of the steps is different since the architecture of the accelerator has not yet been solidified. First, the Planner (from the architecture layer) needs to produce the architectural plan of the accelerator. In the FPGA case, this plan even depends on the DFG of the learning algorithm. In the P-ASIC case, although this plan is not dependent on the DFG, it still changes according to the chip constraints. The back-edge from the architecture layer to the compilation layer in the left diagram of Figure 2.3 illustrates the dependence of Compiler to the Planner. Once the architecture is planned, the Compiler leverages our novel mapping/scheduling algorithm to statically map operations to the accelerator Processing Engines (PEs). This static mapping is converted to state machines and control units that are embedded in the accelerator code for FPGA realization. For P-ASIC, the mapping is converted to microcodes. This static scheduling strategy avoids the von Neumann overheads and significantly simplifies the hardware which is necessary for the efficiency of the accelerator. As detailed in Section 2.6, our

mapping/scheduling algorithm also minimizes on-chip communication and alleviates the need for data preprocessing or marshaling. Compiler also generates the schedule for the template architecture's programmable memory interface that feeds a large number of PEs and streams data in without the need for PEs to request the data.

2.4.3 System Layer

Section 2.3 already detailed the system layer. The topmost component of this layer is the System Director that assigns roles (Sigma or Delta) to the nodes and then configures and initiates the corresponding system subroutines. This role assignment is based on the system specification, which includes the total number of nodes, the number of groups, and the accelerator type (Figure 2.3, right).

2.4.4 Architecture Layer

In the conventional stack, this layer defines the Instruction Set Architecture (ISA) of a microprocessor. In CoSMIC, this layer is responsible for planning the architecture of the accelerator in accordance with the constraints of the target platform. The plan is generated with respect to our novel multi-threaded template architecture, which is a parametric RTL Verilog of customizable design. This template architecture can accelerate multiple instances of the partial gradients simultaneously. However, it is not specific to a learning algorithm and can be shaped according to the constraints of the acceleration platform (e.g., area) and the DFG of the algorithm in the case of FPGA acceleration. Instead, it is a two-dimensional matrix of customizable PEs that this layer needs stretches or squeezes in either dimension to match the chip specifications. The main challenge is allocating the chip resources in such a way that strikes a balance

between the single-threaded performance and multi-threaded parallelism. The Planner is responsible for this balanced plan by determining how many threads will be accelerated simultaneously; how many PEs will be allocated to each thread; and how the PE will be arranged in the 2D matrix of the accelerator. For P-ASICs, the Planner determines the largest number of PEs that fits in the area and power budget of the target chip. However, this metric depends on the PE buffer capacity that is decided according to a set of benchmarks. After determining the total number of PEs, the Planner steps are similar for P-ASICs and FPGAs. Thus, we only discuss the Planner in the context of FPGAs for brevity.

To determine these factors, the Planner takes in a high-level specification of the FPGAs, which includes the number of DSP units, the off-chip memory bandwidth, the number of on-chip Block RAMs (BRAMs), and the size of each BRAM (Figure 2.3). The first step is determining the number of columns (=# PEs in a row) and rows. The Planner uses the off-chip memory bandwidth to first set the number of columns equal to the number of words that can be fetched in parallel from memory (=off-chip bandwidth). Having fewer columns would waste bandwidth, while more would increase pressure on the internal interconnection between the PEs. The Planner will then determine the maximum row count as $row_{max} = \frac{\#\ DSPs}{\#\ of\ Columns}$.

Next, the Planner determines the number of threads and their PE allocation through design space exploration. However, this design space is prohibitively large, due to the copious amount of resources in the modern FPGAs. We prune this design space through the following intuitive design decisions. The Planner first calculates the amount of required storage and area for accelerating one worker thread based on its DFG. The ratio of total on-chip storage and area to this thread's footprint will be the upper bound on the number of simultaneous

threads. Then, we restrict the PE allocation to the row granularity, meaning each thread will have at least a row of PEs. Another parameter that affects the maximum number of threads is the programmer-provided mini-batch size, as it determines how many parallel threads can potentially be launched. The minimum of these parameters is the maximum number of possible threads:

$$t_{max} = min\Big(\frac{\#\ BRAMs \times BRAM\ Size}{DFG.storage()},\ row_{max},\ Mini-Batch\ Size \Big)$$

These design choices and the column/row arrangement restrict the design space from which the Planner needs to determine the optimal allocation of PEs to the threads. For instance, in UltraScale+, the design space is limited to 27 design points. However, the Planner still needs to explore this reduced design space. Instead of simulation, which will be intractable, we propose to equip the Planner with a performance estimation tool. The tool will use the static schedule of the operations for each design point to estimate its relative performance. This enables the Planner to choose the smallest, best-performing design point which strikes a balance between the number of cycles of data processing and off-chip data transfer. Performance estimation is viable, as the DFG does not change, there is no hardware managed cache, and the accelerator architecture is fixed during execution. Thus, there are no irregularities that can hinder estimation. As such, it takes less than five minutes to explore all the possible design points for UltraScale+. The result of this design space exploration is presented in Section 2.8. After this analysis, the Planner generates the Verilog code of the accelerator datapath from the template.

2.4.5 Circuit Layer

As Figure 2.3 depicts, the Constructor is the main module of the Circuit layer and generates the final Verilog code by adding the control logic. In the case of FPGAs, to generate the state machines and control units, the Constructor needs the Compiler to first statically map and schedule all the operations. In this case, the accelerator avoids the von Neumann overhead by bypassing instruction fetch and decode stages. Instead, the Constructor statically converts the execution schedule to state machines and control logic. In the case of P-ASICs, the Constructor adds a control logic that enables microcode execution on the PEs. Then, it inserts these control units within the datapath Verilog code generated by the Planner and produces the final synthesizable Verilog code of the accelerators. The Planner, the Constructor, and the Compiler work in tandem to make CoSMIC a cohesively co-designed stack that delivers high gains.

2.5 Template Architecture

A major challenge in acceleration is the generality across a wide range of algorithms and applications while supporting a variety of platforms (e.g., various FPGA chips). It is also crucial to offer a solution that can adapt to new algorithms and algorithmic changes. A fixed architecture cannot offer enough flexibility and is not deployable on different chips. Therefore, CoSMIC offers a template architecture to accelerate learning at scale. This template is pre-designed, yet re-organizable, providing the capability to implement different gradient calculations and parallel variants of gradient descent aggregations and updates. The template offers reusability while delivering high performance, as it is hand-crafted by experts (e.g., our team). Our stack stretches and squeezes the template to best match the DFGs and the target chip. Hence, it is modular

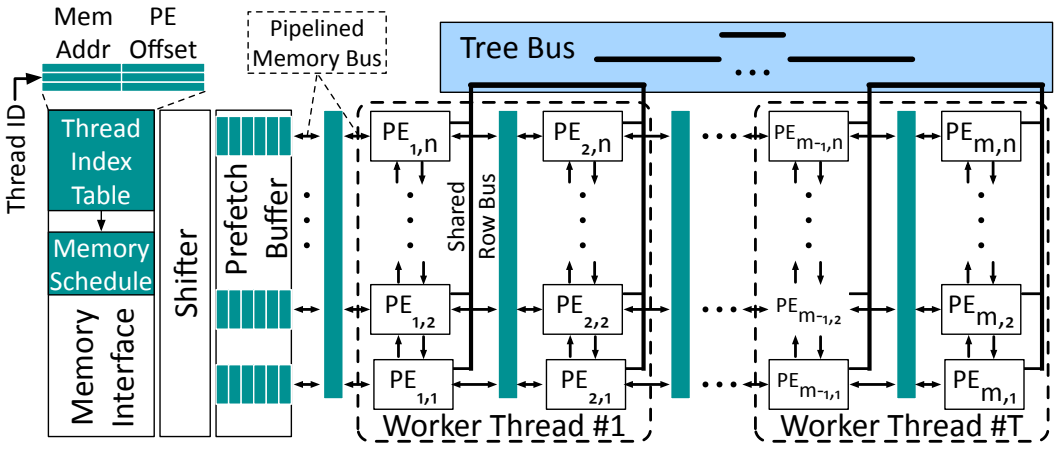


Figure 2.5: CoSMIC Multi-Threaded Template Architecture.

and scalable to maximally utilize the ample amount of resources in the server-grade FPGAs and P-ASICs.

The need for multi-threading. A single instance of a learning algorithm cannot effectively exploit as much resources, since it is limited by the level of parallelism in its DFG. The DFG of the partial gradient update dictates the number and type of operations, along with data-dependencies. However, data-dependencies in the DFG limit the number of operations that the accelerator can execute in parallel. To increase the parallelism available to the accelerator, we use the insight that partial gradient updates generated by worker threads in parallel gradient descent algorithms are independent. As such, the CoSMIC template architecture executes multiple worker threads in the FPGA accelerator; each thread, using a subset of the accelerator resources, executes the entire DFG over the thread's data sub-partition to generate an independent partial gradient update. This multi-threading limits the data-communication within a worker thread to a subset of the accelerator's DSP slices, reducing communication overhead.

2.5.1 Accelerator Organization

As depicted in Figure 2.5, the template architecture constitutes: (1) the memory interface—to transfer data to and from external memory; (2) the shifter—to align the data coming from memory; (3) the prefetch buffer—to store the aligned data; and (4) the two-dimensional array of PEs—to compute partial gradient updates and locally aggregate them. We choose this 2D topology, because it enables the Planner to modularly add or remove PEs as columns or rows. As discussed, this organization also enables an efficient design space exploration by assigning PEs to the worker threads in the rows granularity.

Connectivity and bussing. As Figure 2.5 shows, the number of PEs in each row of the template matches the off-chip bandwidth so that the memory interface can feed all the PEs in a row every cycle, maximizing parallelism. Each row of PEs connects to the memory interface using a pipelined bus, as shown in Figure 2.5. Pipelining the bus is necessary for scalability since the bus is shared by all the rows in the accelerator. In addition to data transfer between external memory and the PEs, connectivity between PEs is required to transfer intermediate results due to data-dependencies in the DFG. To facilitate the communication, PEs in a single row are connected to their adjacent PEs using bi-directional links and are also connected to the other PEs in the row via a shared bus. A hierarchical tree bus connects the shared bus for different rows. We specialize the interconnect between PEs in the template architecture for communication patterns typical for operations in stochastic gradient descent based learning algorithms. One such example of a common operation is a vector dot product, which involves element-wise multiplication followed by reduction ($\sum$). The result is then typically communicated to all PEs. While the PEs can execute the element-wise multiplication in parallel, the reduction and broadcast operations require significant communication between PEs, which can be

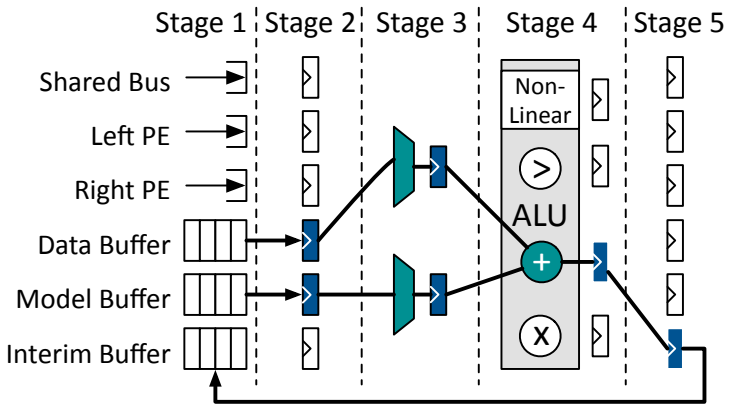


Figure 2.6: Pipelined PE. Black highlights an Add operation (InterimBuffer[i] = DataBuffer[j] + ModelBuffer[k]).

a performance bottleneck. In order to alleviate the communication overhead and ensure high utilization of the accelerator's resources, PEs possess three distinct levels of connectivity. Figure 2.5 shows these three levels of connectivity for the template architecture with (n) PEs per row and (m) rows. At the first level, the n adjacent PEs within each row can communicate using bi-directional links. Next, a shared bus connects all of the n PEs within each row. Finally, we use a tree bus to connect the shared bus of m rows of the accelerator. To further aid the reduction operation, each node in the tree bus contains an ALU to perform $\sum$ and $\prod$ operations.

PE design. Figure 2.6 details a PE, the basic unit of the template architecture responsible for executing the operations of the DFG. The rows of PEs within a worker thread exploit fine-grained parallelism in the DFG, enabling the execution of multiple independent operations in parallel. A PE consists of separate buffers for storing training data, model parameters, and intermediate results. This partitioning of buffers is necessary to enable parallel accesses required for DFG operations. The buffers are composed of on-chip SRAMs and the size of each buffer can be configured by the Planner for a given DFG. CoSMIC's Compiler statically generates the schedule of operations for each PE. The PEs

execute the scheduled operations using a five stage pipeline, orchestrated by a PE scheduler. The first pipeline stage reads the required data from PE's buffers, adjacent PE links, and shared bus links. This data is registered in the second stage. The third stage selects the input operands required by the scheduled operation. The fourth stage executes the scheduled operation using the PE's ALU. For FPGA implementation, the ALU uses DSPs blocks—the hardened on-chip arithmetic unit on the FPGA. The non-linear unit is a look-up table that implements expensive operations like sigmoid, gaussian, divide, and logarithm and is only instantiated in a PE if the Compiler schedules a non-linear operation for that PE. The output of the ALU unit is written back in the fifth and final stage of the PE pipeline. The PEs have a bypass path between the final stage and the ALU stage to forward the result of the previous operation. Figure 2.6 highlights the path taken by an add operation which reads from data and model buffers and writes back to the interim buffer.

Memory interface. Simplicity of the PEs and their highly pipelined design is vital for the efficiency of the accelerator. To further simplify the design, the template architecture prevents the PEs from initiating data requests to the memory. Instead, as illustrated in Figure 2.5, the design harbors a smart memory interface which feeds the PEs according to the schedule generated by the Compiler. This memory interface design is intended to alleviate the overhead of data marshaling, which would have been prohibitive since CoSMIC targets distributed learning with copious amounts of data. However, one issue that arises is that the vectors of data in the off-chip memory do not necessarily align with the rows of the PEs. This can lead to under-utilization of off-chip bandwidth, which is often a performance bottleneck. To avoid the overhead of padding the data to align with the PEs, we propose to use an on-chip Shifter that aligns input data after fetching it, according to the data map generated by the Compiler. In

addition to the Shifter, the memory interface will have a Prefetch Buffer. The size of the training data for each DFG is often large. Hence the time required for external memory access is significant. The Prefetch Buffer enables the accelerator to store the subsequent set of training data for the worker threads, thereby hiding the latency of memory accesses and enabling efficient MIMD execution. The memory interface can also perform broadcast writes to the PEs, as the same model needs to be sent to all the worker threads before they start calculating the new gradient updates.

2.5.2 Multi-Threaded Acceleration

The programmable memory interface plays a significant role in enabling multithreading in the accelerator without imposing significant hardware overhead. It harbors a Memory Schedule queue along with a Thread Index Table that stores thread-specific information as depicted in Figure 2.5. This information includes the memory address of each thread's data sub-partition and the base index of the first allocated PE row to the thread. In addition, each thread has its own dedicated pointer to the Memory Schedule queue. The data transfer schedule is the same for all the threads but it needs to start from different addresses and write to different PEs. The Thread Index Table enables correct and efficient data transfer from memory to all the threads while the schedule is shared. Each row of the table corresponds to one thread. The first field in each row is Mem Addr, which specifies the starting address of each thread's data sub-partition in the off-chip memory. The second field, PE Offset, specifies the index of the first PE of the thread. By walking through these rows, the memory interface controller uses the entries of the Memory Schedule and the Thread Index Table to generate memory accesses for each thread in a round-robin fashion. Each entry of the schedule stores a Base PE Index, RD/WR bit,

Broadcast bit, and Size. The index of the target physical PE is (Base PE Index + PE Offset). The latter term in the addition comes from the Thread Index Table. The memory address is also obtained from the Thread Index Table, which is updated by the size of the transferred data after it finishes. Using this table, the memory interface has the necessary information to transfer each thread's data to its allocated PEs without the need for storing multiple copies of the memory schedule. The RD/WR bit of the memory schedule entry specifies whether the memory access is a read or a write. The Broadcast bit allows a memory read to be sent to all the worker threads via the memory interface bus. This bit is particularly useful when sending model parameters from memory to all worker threads. The Size specifies the size of the data transfer. The Compiler generates the memory schedule according to the Planner-provided architecture and the DFG. The following section discusses the Compiler in detail.

2.6 CoSMIC Compilation

The Compiler is a critical layer of CoSMIC, since it statically determines a fine-grained map and schedule of all the data and operations, which significantly simplifies the hardware. This simplification is necessary for acceleration, particularly for FPGAs that incur lower frequency when design complexity grows. Furthermore, the Compiler minimizes on-chip and off-chip communication and avoids data preprocessing or marshaling. Avoiding data marshaling is crucial, since the accelerators process large amounts of data and any data transfer is costly. To this end, we propose an algorithm that minimizes data movements by statically mapping data elements to PEs before mapping the operations. Conventional mapping algorithms [65, 2] map operations before the data to find the lowest-latency schedule which adheres to the on-chip resource constraints. In

contrast, we reverse the order of mapping, thereby minimizing data movement atop latency.

The Compiler takes as input the DFG of the gradient update, the architectural plan of the multi-threaded accelerator, and the data layout of the training dataset and model parameters in the memory. Using these inputs, the Compiler generates the following for each thread:

1. Data map: assignment of inputs, outputs, model parameters, and intermediate values to the PEs.
2. Operation map: assignment of all the DFG operations to PEs.
3. Data transfer schedule: detailed schedule for memory interface and interconnection buses to send data to the appropriate PEs.

To generate the data map, the Compiler first segregates the DFG operands (graph edges) into DATA, MODEL, and INTERIM categories. These categories represent training data, model parameters, and intermediate operands, respectively. This semantic segregation enables the Compiler to provide an optimal data map without marshaling the data as follows. It starts by mapping each training data element (type DATA) to the PE that is connected to the memory interface column which brings in that element. The Compiler uses this data map to generate the schedule of data transfer from off-chip memory and embeds it into the memory interface. This map and schedule avoids marshaling by adhering to the layout of training data in the memory. Next, the Compiler generates the operation map and data map for the model parameters while minimizing the communication between PEs. We have designed Algorithm 1 for the Compiler to map the operations to the same PEs that hold their operands; hence minimizing inter-PE communication. This algorithm also maps the model parameters to the PEs that hold their corresponding operation. The intuition is to map the MODEL and INTERIM edges on to the same PE if a node operates on both

```
Input  : G: Dataflow graph (V,E)
         n_PE: Number of PEs per worker thread
Output : O: Operation map
         D: Data map
Initialize O[n_PE] ← ∅
Initialize D[n_PE] ← ∅
Initialize Graph ← G
PE_i = 0
while (graph ≠ ∅) do
    for (v ∈ Graph) do
        if (∀ p_i in v.parents = MAPPED) then
            if (∃ op_i in v.ops & op_i.type = DATA) then
                v.pe = op_i.pe
                if (∃ op_j in v.ops & op_j.type = MODEL) then
                    D[v.pe].append(v.op_j)
                Break
            else if (∃ op_i in v.ops & op_i.type = MODEL) then
                if (op_i.pe != NULL) then
                    v.pe = op_i.pe
                else
                    v.pe = w_i
                    D[v.pe].append(v.op_i)
                    PE_i = (PE_i + 1) % n_PE
                Break
            else if (∃ op_i in v.ops & op_i.type = INTERIM) then
                v.pe = op_i.pe
                Break
            O[v.pe].append(v)
            graph.remove(v)
    end
end
```

Algorithm 1: Minimum-Communication Data/Operation Mapping.

of them. After determining the data map on the PEs, the algorithm traverses the DFG and map operations according to the location of their operands, minimizing data movement. During this pass, to reduce latency, the Compiler also prioritizes scheduling operations that have the longest dependence chain. The algorithm takes in the DFG (G) and the number of PEs per thread (n_{PE}) and goes through the following steps:

1. **Initialize** the operation map ($O[n_{PE}]$) and the data map ($D[n_{PE}]$) to null and the $Graph$ variable to the DFG (G). O and D are arrays of lists that hold the maps for each PE.
2. **Select a vertex** (v) that is ready i.e. all its predecessors are mapped.
3. **Check the operand type** for this vertex (v). If any of its operands (op_i) is of type DATA, then map v to the PE containing this data, else go to step (4). Check the type of the other operand (op_j). If the other operand (op_j) is of type MODEL, then map this model parameter to v's PE and go to step (5).
4. **If operand type of the vertex (v) is MODEL**, then map v to the PE where the model parameter resides, otherwise go to step (5). If the operand is not mapped, then map this vertex and the operand op_i to a new PE (PE_i). The PE_i variable is a counter, incremented after each round of successful mapping. Incremental assignment enables parallel execution of the operations in neighboring PEs.
5. **If operand type of the vertex (v) is INTERIM**, map the vertex(v) to the PE in which the operand resides.
6. **Reiterate steps 2 through 5** until all the vertices are mapped.

Given the data and operation map, the Compiler generates the execution schedule for all the components of the accelerator, including its programmable memory interface and PE interconnects. Recall that each thread performs the same gradient update rule but uses different training data. Therefore, the Com-

piler generates the map and schedule for one thread and use it for all of them. However, to overlap off-chip data transfer with computation, the accelerator is MIMD, not SIMD. Thus, threads can be at different computation stages since they start execution as soon as they receive an operand. To enable the MIMD execution, the Planner produces a PE Offset for each thread, which is the index of the first PE that is assigned to the thread. The PE Offset and the starting address of its training data is loaded into the Thread Index Table as discussed before (see Figure 2.5). The Compiler generates only one schedule for the memory interface since the destination PE can be calculated at runtime by adding each thread's PE Offset to the PE index that is in the schedule. Finally, the Compiler uses the map of the model parameters to generate the schedule for the aggregation stage that follows partial gradient calculations.

2.7 In-Network Acceleration of Gradient Compression

While the CoSMIC's lean and specialized runtime software reduces the communication overhead of the system coordination between the accelerator nodes, the communication time still takes a significant fraction of the entire training runtime. To tackle this challenge, in collaboration with Prof. Nam Sung Kim's research group at the University of Illinois at Urbana–Champaign, we devise a in-network acceleration technique that enables compression of single-precision floating-point gradient values on FPGA-enabled NICs. In this work, we focus on the compression of gradient values since gradients are significantly more tolerant to precision loss than weights and as such lend themselves better to aggressive compression without the need for the complex mechanisms to avert any loss. Leveraging this unique property of gradient values, we propose a lossy compression algorithm, which offers high compression ratio as well as

Input : f: 32-bit single-precision FP value
Output : v: Compressed bit vector (32, 16, 8, or 0 bits)
t: 2-bit tag indicating the compression mechanism

$s \leftarrow f[31]$ // sign
$e \leftarrow f[30:23]$ // exponent
$m \leftarrow f[22:0]$ // mantissa
if *($e \geq 127$)* **then**
  $v \leftarrow f[31:0]$
  $t \leftarrow NO\_COMPRESS$ // 2'b11
else if *($e < error\_bound$)* **then**
  $v \leftarrow \{\}$
  $t \leftarrow 0BIT\_COMPRESS$ // 2'b00
else if *($error\_bound \leq e < 127$)* **then**
  $n\_shift \leftarrow 127 - e$
  $shifted\_m \leftarrow concat(1'b1,\ m) >> n\_shift$
  if *($e \geq error\_bound + \lceil (127 - error\_bound)/2 \rceil$)* **then**
    $v \leftarrow concat(s,\ shifted\_m[22:16])$
    $t \leftarrow 8BIT\_COMPRESS$ // 2'b01
  else
    $v \leftarrow concat(s,\ shifted\_m[22:8])$
    $t \leftarrow 16BIT\_COMPRESS$ // 2'b10
  end
end

Algorithm 2: Lossy compression algorithm for single-precision floating-point gradients.

hardware-friendliness due to its low complexity. Note that this algorithm is specifically designed for the hardware implementation in that there are one-to-one mappings from the data/operations in the algorithm to hardware logics. Therefore, we only discuss the algorithms and omit the hardware implementation details. Then, we discuss the integration of compression/decompression modules with the FPGA-enabled NICs.

2.7.1 Compression Algorithm

Algorithm 2 elaborates the procedure of compressing a 32-bit floating-point gradient value (f) into a compressed bit vector (v) and a 2-bit tag indicating the used compression mechanism (t). Note that this algorithm is described based on the standard IEEE 754 floating-point representation which splits a 32-bit value into 1 sign bit (s), 8 exponent bits (e), and 23 mantissa bits (m). Depending on the range where f falls in, the algorithm chooses one of the four different

compression mechanisms. If f is larger than 1.0 (i.e., $e \geq 127$), we do not compress it and keep the original 32 bits (NO_COMPRESS). If f is smaller than an error bound, we do not keep any bits from f (0BIT_COMPRESS). When the gradient values are in the range (error_bound $< f <$ 1.0), we should take a less aggressive approach since we need to preserve the precision. The simplest approach would be to truncate some LSB bits from the mantissa. However, this approach not only limits the maximum obtainable compression ratio since we need to keep at least 9 MSB bits for sign and exponent bits, but also affects the precision significantly as the number of truncated mantissa bits increases. Instead, our approach is to always set e to 127 and to not include the exponent bits in the compressed bit vector. Normalizing e to 127 is essentially multiplying $2^{(127-e)}$ to the input value; therefore, we need to remember the multiplicand so that it can be decompressed. To encode this information, we concatenate a 1-bit '1' at the MSB of m and shift it to the right by $127-e$ bits. Then we truncate some LSB bits from the shifted bit vector and keep either 8 or 16 MSB bits depending on the range of value. Consequently, the compression algorithm produces a compressed bit vector with the size of either 32, 16, 8, or 0 and 2-bit tag indicating the used compression mechanism.

2.7.2 Decompression Algorithm

Algorithm 3 describes the decompression algorithm that takes a compressed bit vector v and a 2-bit tag t. When t is NO_COMPRESS or 0BIT_COMPRESS, the decompressed output is simply 32-bit v or zero, respectively. If t is 8BIT_COMPRESS or 16BIT_COMPRESS, we should reconstruct the 32-bit IEEE 754 floating-point value from v. First, we obtain the sign bit s by taking the first bit of v. Then we find the distance from MSB to the first '1' in v, which is the multiplicand used for setting the exponent to 127 during compression. Once we get the distance,

Input : v: Compressed bit vector (32, 16, 8, or 0 bits)
t: 2-bit tag indicating the compression mechanism
Output : f: 32-bit single-precision FP value

if *($t = NO\_COMPRESS$)* **then**
  $f \leftarrow v[31:0]$
else if *($t = 0BIT\_COMPRESS$)* **then**
  $f \leftarrow 32'b0$
else
  if *($t = 8BIT\_COMPRESS$)* **then**
    $s \leftarrow v[7]$
    $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[6:0])$ $m \leftarrow concat(v[6:0] << n\_shift,\ 16'b0)$
  else if *($t = 16BIT\_COMPRESS$)* **then**
    $s \leftarrow v[15]$
    $n\_shift \leftarrow first1\_loc\_from\_MSB\ (v[14:0])$ $m \leftarrow concat(v[14:0] << n\_shift,\ 8'b0)$
  end
  $e \leftarrow 127 - n\_shift$
  $f \leftarrow concat(s,\ e,\ m)$
end

Algorithm 3: Decompression algorithm.

e can be calculated by subtracting the distance from 127. The next step is to obtain m by shifting v to left by the distance and padding LSBs with zeros to fill the truncated bits during compression. Since we now have s, e, and m, we can concatenate them together as a 32-bit IEEE 754 floating-point value and return it as the decompression output.

2.7.3 Accelerator Architecture and Integration with NIC

Our compression/decompression algorithms can be instantiated as hardware modules in accelerators that compress/decompress gradient values. Using these modules as building blocks, we develop compression/decompression engines that can process 256-bit network bursts, and integrate these engines with the existing FPGA-enabled NICs. In this section, we first introduce our NIC architecture and discuss the details of compression/decompression engines.

NIC architecture. To evaluate our system in a real world setting, we implement our accelerators on the Xilinx VC709 evaluation board [66] that offers 10Gbps network connectivity along with programmable logic. We insert our accelerator within the NIC reference design [67] that comes with the board. Figure 2.7 illus-

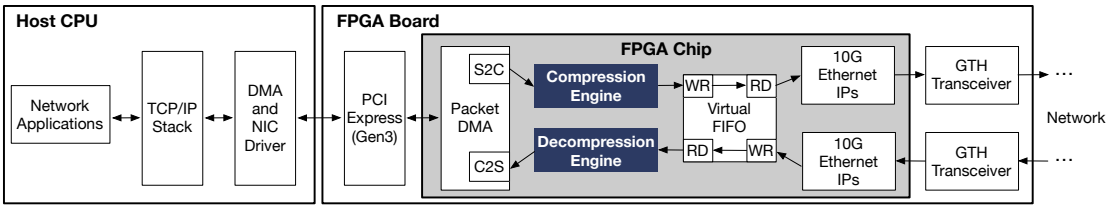


Figure 2.7: Overview of NIC architecture integrated with compressor and decompressor.

trates this integration of compression and decompression engines. For output traffic, as in the reference design, the packet DMA collects the networks data from the host system through the PCIe link. These packets then go through the Compression Engine that stores the resulting compressed data in the virtual FIFOs that are used by the 10G Ethernet MACs. These MACs drive the Ethernet PHYs on the board and send or receive the data over the network. For input traffic, the Ethernet MACs store the received data from the PHYs in the virtual FIFOs. Once a complete packet is stored in the FIFOs, the Decompression Engine starts processing and passing it to the packet DMA for transfer to the CPU. Both engines use the standard 256-bit AXI-stream bus to interact with other modules.

Although hardware acceleration of the compression and decompression algorithms is straightforward, their integration within the NIC poses several challenges. These algorithms are devised to process streams of floating-point numbers, while the NIC deals with TCP/IP packets. Hence, the accelerators need to be customized to transparently process TCP/IP packets. Furthermore, the compression is lossy, the NIC needs to provide the abstraction that enables the software to activate/deactivate the lossy compression per packet basis. The following discusses the hardware integration. The software abstractions are outside of the scope of this dissertation and therefore not discussed.

Compression Engine. Not to interfere with the regular packets that should not

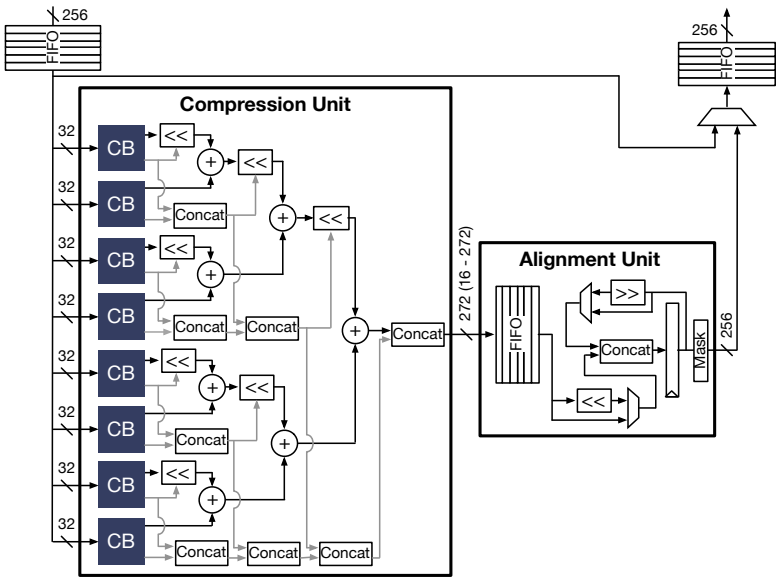


Figure 2.8: 256-bit burst compressor architecture.

be compressed, the Compression Engine first needs to identify which packets are intended for lossy compression. Then, it needs to extract their payload, compress it, and then reattach it to the packet. The Compression Engine processes packets in bursts with size of 256 bit, which is the number of bits which the AXI interface can deliver in one cycle. Our engines process the packet in this burst granularity to avoid curtailing the processing bandwidth of the NIC. The compression starts as soon as the first burst of payload arrives.

Figure 2.8 depicts the architecture of the compression hardware. The payload burst feeds into the Compression Unit equipped with eight Compression Blocks (CBs), each of which performs the compression described in Algorithm 2. Each CB produces a variable-size output in the size of either 32, 16, 8, or 0 bits, which need to be aligned as a single bit vector. We use a simple binary shifter tree that produces the aligned bit vector of which possible size is from 0 to 256. The 2-bit tags of the eight CBs are simply concatenated as a 16-bit vector. Finally, the aligned bit vector and tag bit vector are concatenated as the final output of Compression Unit, of which size is at least 16 bits and can go up to 272

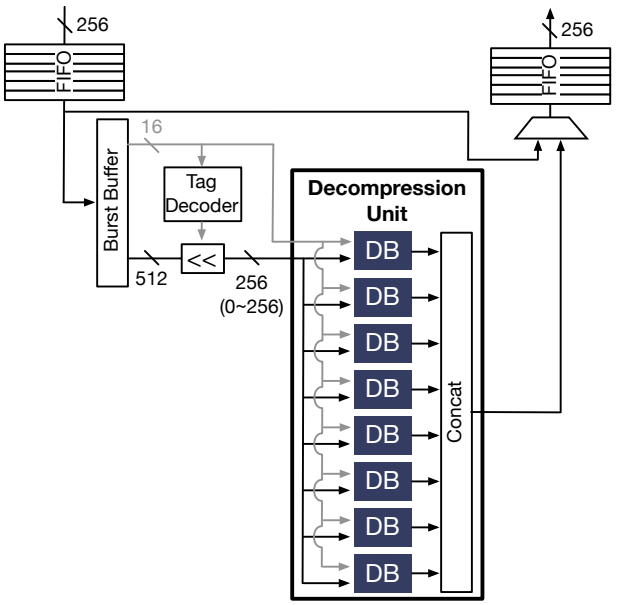


Figure 2.9: 256-bit burst decompressor architecture.

bits. For each burst, Compression Unit produces a variable-size (16 – 272) bit vector; therefore, we need to align these bit vectors so that we can transfer the 256-bit burst via the AXI interface. The Alignment Unit accumulates a series of compressed bit vectors and outputs a burst when 256 bits are collected.

Decompression Engine. Similar to the Compression Engine, Decompression Engine processes packets in the burst granularity. The payload bursts of compressible packets feed into the decompression hardware, of which architecture is delineated in Figure 2.9. Since the compressed burst that contains 8 FP numbers can overlap two consecutive bursts at the Decompression Engine, reading only a single burst could be insufficient to proceed to the decompression. Therefore, the Decompression Engine has Burst Buffer that maintains up to two bursts (i.e., 512 bits). When Burst Buffer obtains two bursts, it feeds the 16-bit tag to Tag Decoder to calculate the size of the eight compressed bit vectors. Given the sizes, the eight compressed bit vectors are obtained from the buffered 512 bits. Since each compressed bit vector has a variable size of either 32, 16, 8 or 0 bits, the possible size of the eight compressed bit vectors is

from 0 and 256. These eight compressed bit vectors (0 – 256) and the tag bit vector (16) are fed into the eight Decompression Blocks (DBs) in Decompression Unit, which executes the decompression algorithm described in Algorithm 3. Then, Decompression Unit simply concatenates the outputs from the eight DBs and transfers it via the AXI interface. For the next cycle, Burst Buffer shifts away the consumed bits and reads the next burst if a burst (i.e., 256 bits) has been consumed and the left bits are fewer than a burst.

2.8 Evaluation

We evaluate CoSMIC with 10 different machine learning benchmarks using various acceleration platforms, which consist of one FPGA (Xilinx UltraScale+ VU9P) and two P-ASICs. These accelerators are hosted in machines equipped with Intel Xeon E3 v5 processors. We first compare the scalability of the FPGA-accelerated CoSMIC systems to a popular distributed computing platform, Spark [16], while increasing the number of nodes from 4 to 8 to 16. For the scale-out experiments, we used Amazon EC2. We built a local three node system for the in-depth sensitivity studies. We also perform comparison with the distributed GPU (Nvidia K40c) implementation of the benchmarks. Table 2.2 details the specification of these platforms. Lastly, we compare the CoSMIC template architecture with TABLA [2], a single-node FPGA acceleration framework for ML.

2.8.1 Methodology

Benchmarks and training input datasets. Table 2.1 shows the list of 10 benchmarks—obtained from machine learning literature—that train two different models with each of the following five different algorithms: backpropagation,

Table 2.1: Benchmarks, algorithms, application domains, and datasets.

Name	Algorithm	Domain	Description	# Features	Model Topology	Model Size (KB)	Lines of Code	# Input Vectors	Input Data Size (GB)
mnist	Backpropagation	Image Processing	Handwritten digit pattern recognition	784	784×784×10	2,432	55	60,000	0.4
acoustic		Audio processing	Hierarchical acoustic modeling for speech recognition	351	351×1,000×40	1,527	55	942,626	5.6
stock	Linear Regression	Finance	Stock price prediction	8,000	8,000	31	23	130,503	14.7
texture		Image Processing	Image texture recognition	16,384	16,384	64	23	77,461	17.9
tumor	Logistic Regression	Medical Diagnosis	Tumor classification using gene expression microarray	2,000	2,000	8	22	387,944	10.4
cancer1		Medical Diagnosis	Prostate cancer diagnosis based on the gene expressions	6,033	6,033	24	22	167,219	13.5
movielens	Collaborative Filtering	Recommender System	Movielens recommender system	30,101	301,010	1,176	42	24,404,096	0.6
netflix		Recommender System	Netflix recommender system	73,066	730,660	2,854	42	100,498,287	2.0
face	Support Vector Machine	Computer Vision	Human face detection	1,740	1,740	7	27	678,392	15.9
cancer2		Medical Diagnosis	Cancer diagnosis based on the gene expressions	7,129	7,129	28	27	208,444	20.0

linear regression, logistic regression, collaborative filtering, and support vector machines. The benchmarks represent various application domains including image processing, audio processing, finance, medical diagnosis, recommendation systems, and computer vision. The mnist and acoustic benchmarks train Multi-Layer Perceptrons (MLPs) for handwritten digit [68, 69] and automatic speech recognition [70], respectively. The stock benchmark trains a linear regression model to predict stock prices using the tick-level data points [71]. The texture benchmark trains another linear regression model for texture recognition [72]. The tumor and cancer1 benchmarks train two different logistic regression models to detect tumors [73] and cancer [74] using the microarray gene expression data. The movielens and netflix benchmarks train recommender systems that employ the collaborative filtering algorithm on Movielens datasets [75, 76] and Netflix Prize Dataset [77]. The face benchmark trains a support vector machine for face recognition [78]. The cancer2 benchmark trains another support vector machine to detect cancer [78]. We train each benchmark for 100 epochs over its dataset. We repeat the experiments 10 times and use the average runtime. In Table 2.1, the "# of Features" column shows the number of elements in each training data vector and the "Model Topology" column denotes the model topology of each benchmark. The "Model Size" column shows the size of the model parameters. The "Lines of Code" column lists the number of lines of code that the programmer writes, which ranges from 22 to 55. Finally, the "# of Input Vectors" and "Input Data Size" columns show the number of the training vectors and the size of the training data. The model parameters for all the benchmarks fit in on-chip memory of the FPGA and the P-ASICs.

Scale-out system specification. Both CoSMIC and Spark systems are deployed on a cluster of machines, which are equipped with the high-performance quad-core Intel Xeon E3 Skylake processors with hyper-threading support that

Table 2.2: CPU, GPU, FPGA, and P-ASICs.

	CPU	GPU
Chip	Xeon E3-1275 v5	Tesla K40c
Cores	4	2,880
Memory	32 GB	12 GB
TDP	80 W	235 W
Frequency	3.6 GHz	875 MHz
Technology	14 nm	28 nm

	FPGA
Chip	UltraScale+ VU9P
DSP Slices	6,840
BRAM	44,280 KB
TDP	42 W
LUTs	1,182 K
Flip Flops	2,364 K

	P-ASIC	P-ASIC
Chip	F	G
PEs	768	2,880
Area (mm^2)	29	105
Power	11 W	37 W
Frequency	1 GHz	1 GHz
Technology	45 nm	45 nm

operates at 3.6 GHz. The detailed CPU specification is provided in Table 2.2. The machines run Ubuntu 16.04.1 LTS with the kernel version 4.4.0-47. The machines are connected through a TP-LINK 24-Port gigabit Ethernet switch (TL-SG1024) via TP-Link gigabit Ethernet network interface card (TG‘-3468). The switch supports full duplex operation on all ports (2 Gbps per port) and a combined switching capacity of up to 48 Gbps.

Spark. We compare CoSMIC with Spark version 2.1.0. Spark is selected as the point of comparison since it supports efficient in-memory processing for iterative applications. Moreover, Spark provides the MLlib [40] machine learning library. The Spark MLlib library provides the baseline implementation for back-propagation, linear regression, logistic regression, collaborative filtering, and support vector machines [40]. To optimize the performance of MLlib, we build Spark with vectorized OpenBLAS library. For all the Spark results, we use the best-performing combination of machines and threads. The best number of threads is selected for each benchmark individually.

FPGA. As Table 2.2 shows, we use Xilinx Virtex UltraScale+ VU9P for the FPGA experiments. We use Xilinx Vivado 2017.2 to synthesize the generated accelerators at 150MHz. The synthesized accelerators are connected to the external DRAM using the AXI-4 IP.

GPU. For comparison with GPUs, we extend CoSMIC’s runtime system to support GPUs since Spark does not. The alternative would have been integrating

GPUs with Spark, which is on its own a line of ongoing research [79, 80, 81, 82]. As such, we build a GPU-accelerated CoSMIC system. We had three Nvidia Tesla K40 GPUs at our disposal, which are used for this comparison (see Table 2.2 for hardware specification). For the GPU experiments, we developed highly optimized CUDA implementations using well-known libraries, including LibSVM-GPU [83] and Caffe2+cuDNN [84], as well as source code from related works [21, 2]. In all cases, we used the latest versions of each library (e.g., cuBLAS v8.0 [85] and cuDNN v7.0 [86]). We use WattsUp [87] to measure the system power following the same methodology in the prior work [88].

P-ASICs. We use Synopsys Design Compiler (L-2016.03-SP5) and TSMC 45-nm high-Vt standard cell libraries to synthesize the CoSMIC-generated architectures and obtain the area, frequency, and power results. We used CoSMIC to generate two P-ASIC designs: one with the PE count and off-chip bandwidth that match those of the FPGAs (P-ASIC-F), the other that match those of the GPUs (P-ASIC-G). Table 2.2 provides the details of these P-ASICs. We combine the system-level measurements with the synthesis and simulation/estimation results to evaluate these P-ASICs.

2.8.2 Experimental Results

2.8.2.1 Performance Comparison

Figure 2.10 shows the result of performance comparison between CoSMIC and Spark using three system configurations: 4-Node, 8-Node, and 16-Node. The baseline is a 4-Node Spark system, referred to as 4-CPU-Spark. On average, the 4-FPGA-, 8-FPGA-, 16-FPGA-CoSMIC configurations deliver 12.6×, 23.1×, and 33.8× higher performance, respectively. Whereas, increasing the number of nodes with Spark from 4 to 16 only yields 1.8× performance improvement. The performance does not scale linearly as the number of nodes

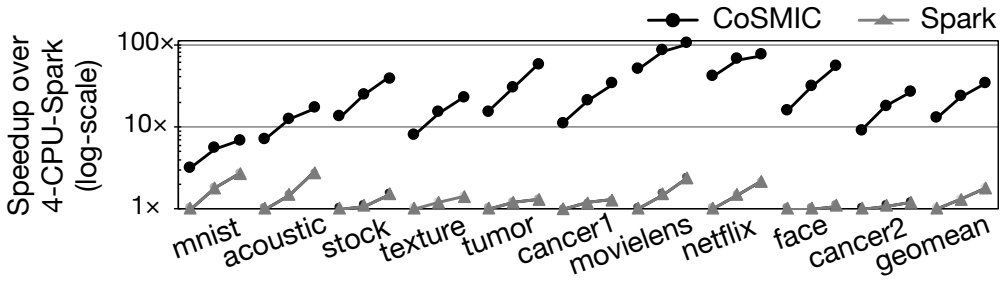
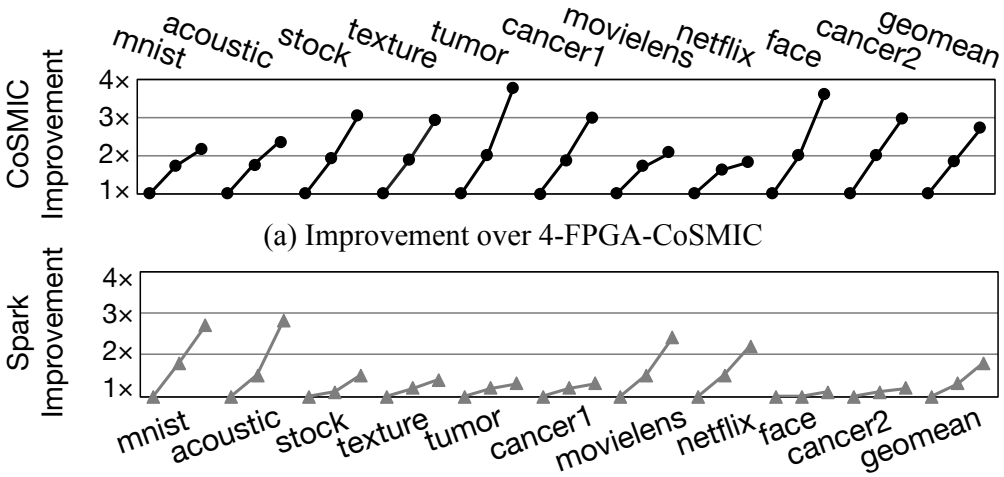


Figure 2.10: Speedup over Spark as the number of nodes increases from 4 to 8 to 16. Baseline: Spark system with 4 nodes (4-CPU-Spark).

increases due to system management overhead in networking and aggregation. The performance gains for different benchmarks depend on their model topology, parallelism, and memory footprint. For example, movielens (collaborative filtering) sees the highest speedup (100.7×) since its DFG is significantly parallel that allows CoSMIC to utilize the FPGAs resources for higher performance. On the contrary, mnist and acoustic (backpropagation) achieve relatively smaller speedup (6.8× and 16.5×) since these benchmarks require significant on-chip communication, which bottlenecks performance. These results show that CoSMIC's full-stack approach, which comes with our multithreaded accelerators, is highly effective for the scale-out acceleration of these ML applications. Furthermore, these results show that CoSMIC better utilizes the added resources and is more scalable as the number of nodes increases.

2.8.2.2 Scalability

To better compare the scalability of the two systems, Figure 2.11 shows the performance improvement over each system's own 4-Node configuration. Figure 2.11(a) shows the improvement with CoSMIC when the 4-FPGA-CoSMIC is the baseline and Figure 2.11(b) shows the improvement with Spark when 4-CPU-Spark is the baseline. On average, CoSMIC performs 1.8× and 2.7× faster when the system is scaled up to 8 and 16 nodes, respectively. As a point



(a) Improvement over 4-FPGA-CoSMIC

(b) Improvement over 4-CPU-Spark

Figure 2.11: Scalability comparison of CoSMIC and Spark as the number of nodes increases from 4 to 8 to 16.

of reference and comparison, Spark shows 1.3× and 1.8× speedup for the same increase in the number of nodes. The results from Figure 2.10 and Figure 2.11 show that CoSMIC scales well and better than Spark as the number of nodes increases. The improvement gap between Spark and CoSMIC is larger for the benchmarks that have higher ratio of communication to computation in the runtime (stock, texture, tumor, cancer1, face, and cancer2). For the other benchmarks, CoSMIC scales less steeply in comparison to Spark. These benchmarks are compute-bound and therefore acceleration is effective and adding accelerators reduces the computation time in the baseline 4-Node configuration. Since Spark does not utilize the accelerators, it benefits more from the added nodes as they bring in the necessary compute power that was missing in the 4-Node configuration. Therefore, adding more nodes helps but it is more effective for Spark. Nonetheless, as Figure 2.10 illustrates, CoSMIC significantly outperforms Spark across all the benchmarks. These results confirm that the specialization of the system software has been effective in enabling acceleration at scale.

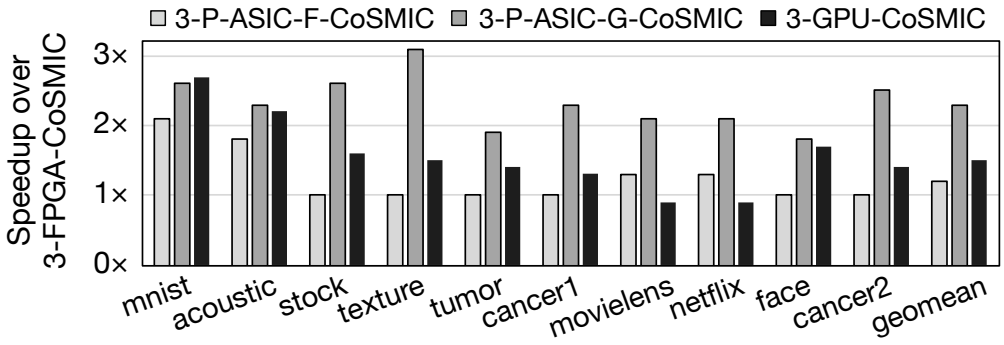


Figure 2.12: System-wide speedup over 3-FPGA-CoSMIC.

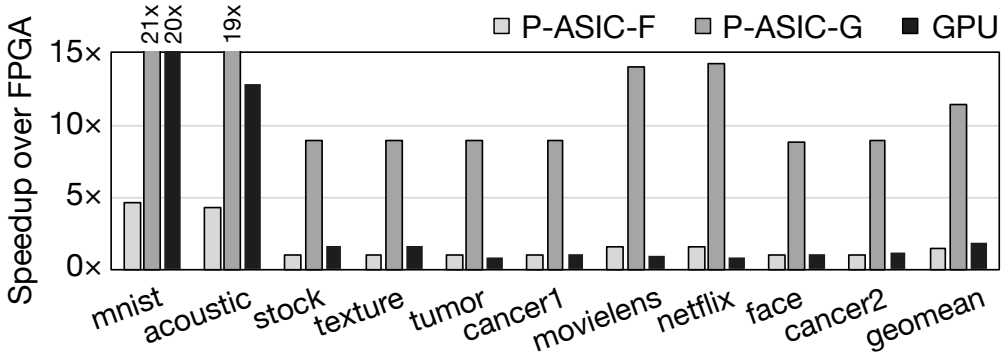


Figure 2.13: Computation speedup over FPGA.

2.8.2.3 Comparison of Different Acceleration Platforms

Figure 2.12 compares the benefits of CoSMIC with FPGAs and P-ASICs to GPUs. The results are obtained from our three-node system configuration and the baseline is the 3-FPGA-CoSMIC. On average, the 3-P-ASIC-F-CoSMIC, 3-P-ASIC-G-CoSMIC, and 3-GPU-CoSMIC systems provide average 1.2×, 2.3×, and 1.5× higher performance than the 3-FPGA-CoSMIC system, respectively. Although as expected P-ASICs and the GPU outperform the FPGA, the benefits are relatively modest. To understand this trend, Figure 2.13 shows the improvement in compute time without considering the system software. On average, P-ASIC-F, P-ASIC-G, and GPU perform 1.5×, 11.4×, and 1.9× faster than FPGA, respectively. Except for mnist and acoustic benchmarks, which use the backpropagation algorithm, the benefits from P-ASIC-F and GPU are not overwhelming. GPU provides higher speedup on two specific benchmarks (20.3×

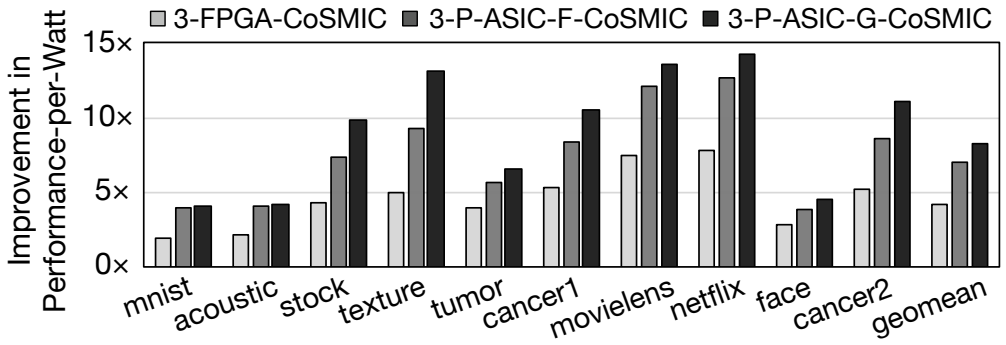


Figure 2.14: Performance-per-Watt, baseline: 3-GPU system.

for mnist and 12.8× for acoustic) as the dominant part of their computation is relatively large matrix-matrix multiplication that GPUs can compute very efficiently. P-ASIC-F offers the same number of PEs and bandwidth compared to the FPGA but at higher frequency. These results show that just improvement in frequency does not translate to proportional speedup as long as the bandwidth remains unchanged. These results also show that the coalescence of CoSMIC's Planner, Compiler, and multi-threaded accelerator design has been effective in exploiting the FPGA resources. Across all benchmarks, P-ASIC-G shows significantly higher improvement as this design point combines more PEs, higher frequency, and higher bandwidth. The PE count and bandwidth of P-ASIC-G matches the GPU and its frequency is higher than the FPGA. However, as Figure 2.12 illustrates, even in the case of P-ASIC-G, the computation speedup does not translate to proportional system-wide improvement. These results confirm the importance of system software and CoSMIC-like full-stack approaches, as accelerators gain popularity.

The speedup of 3-GPU-CoSMIC comes from the GPU's higher frequency as well as massive parallelism; however, it also comes at an expense of higher power dissipation.

Figure 2.14 highlights this power-performance tradeoff by depicting the improvement in Performance-per-Watt when comparing the FPGA- and P-ASIC-

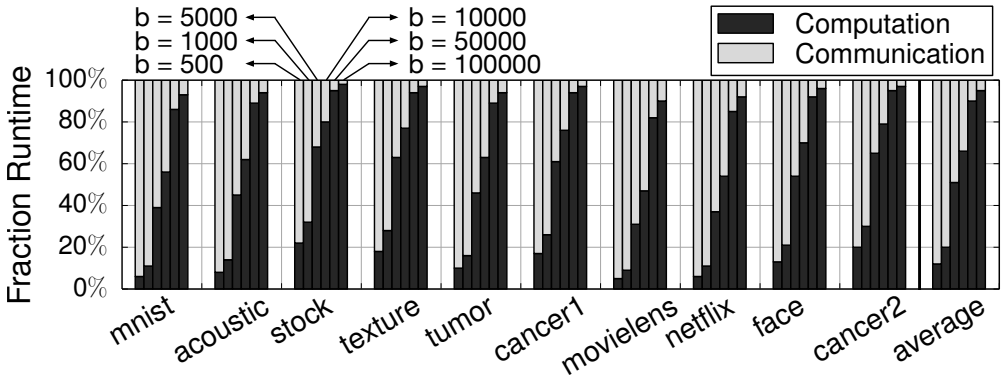
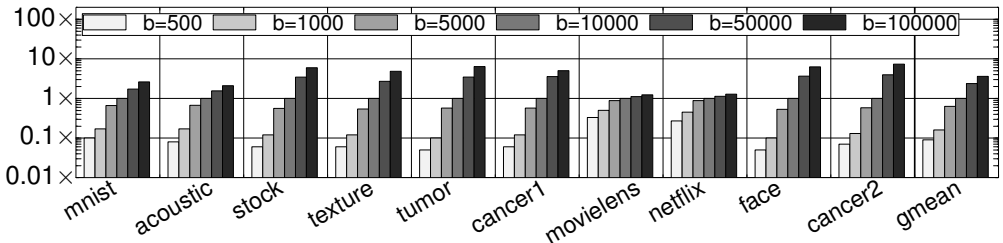


Figure 2.15: Fraction of 3-FPGA-CoSMIC runtime.

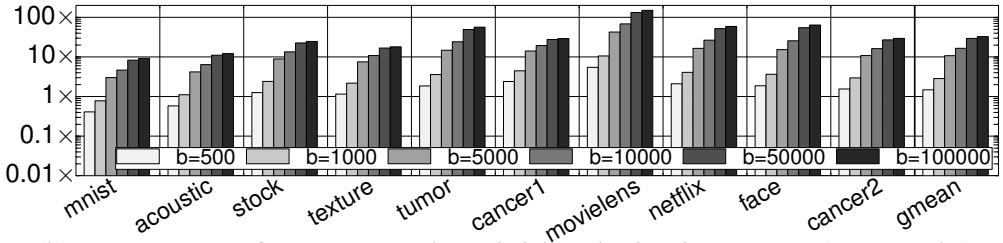
accelerated systems to the GPU-based system. The 3-FPGA-CoSMIC, 3-PASIC-F-CoSMIC, and 3-PASIC-G-CoSMIC systems achieve on average 4.2×, 6.9×, and 8.2× higher Performance-per-Watt than 3-GPU-CoSMIC, respectively. These results show that when the power-efficiency is the main concern, FPGAs or P-ASICs will be more desirable acceleration platforms than GPUs although GPUs provide higher performance than FPGAs and one of the P-ASICs, namely P-ASIC-F. Moreover, although P-ASICs provide both higher performance and power-efficiency, they impose a significant design and manufacturing cost. CoSMIC's template approach reduces the design time and cost as it offers a way to generate accelerator code. However, the cost of manufacturing may tip the scale towards FPGAs as they also offer significant benefits in both performance and power efficiency.

2.8.2.4 Sensitivity to Mini-Batch Size

We use 10,000 as the default mini-batch size as used in the machine learning literature [89, 90, 91]. However, the optimal mini-batch size depends on several variables such as model, datasets, and training iterations. Larger mini-batch size reduces the rate of aggregation, which reduces the inter-node communication, leading to higher performance. Figure 2.15 illustrates this effect by



(a) Spark performance as the mini-batch size increases. (Log-scale)



(b) CoSMIC performance as the mini-batch size increases. (Log-scale)

Figure 2.16: Performance vs. mini-batch size as it is swept from 500 to 100,000; baseline: 3-node Spark when the mini-batch size is 10,000.

segregating the fraction of runtime spent in computation and communication as the number of mini-batch size increases from b=500 to b=100,000 in the three-node runtime. On average, the computation with the mini-batch size 500 takes 12% of runtime but this increases to 95% when the mini-batch size is 100,000. However, reducing the aggregation rate can adversely affect training convergence [89, 90, 92, 91, 93]. To study the effect of mini-batch size on Spark and CoSMIC, we sweep the mini-batch size from 500 to 100,000 for three-node system configuration. Figure 2.16(a) and Figure 2.16(b) present the result of this sweep. For both figures, the baseline is the three-node Spark when mini-batch size is 10,000, our default setting. Comparing Figure 2.16(a) and Figure 2.16(b) shows that 3-FPGA-CoSMIC is faster across all combinations of benchmarks and mini-batch sizes. On average, with the same mini-batch size of b=500, CoSMIC is 16.8× faster. When the mini-batch size increases to b=100,000, CoSMIC is 9.1× faster. As the mini-batch size increases, Spark's overheads diminish. Nevertheless, CoSMIC outperforms Spark.

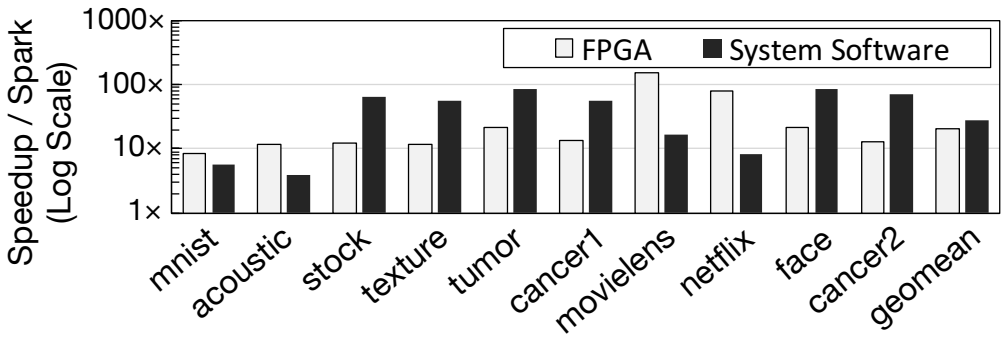


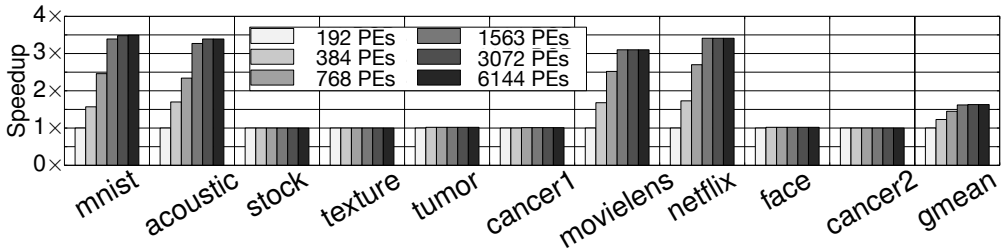
Figure 2.17: Speedup breakdown between FPGAs and system software (aggregation, networking, and management) for 3-FPGA-CoSMIC.

2.8.2.5 Sources of Speedup

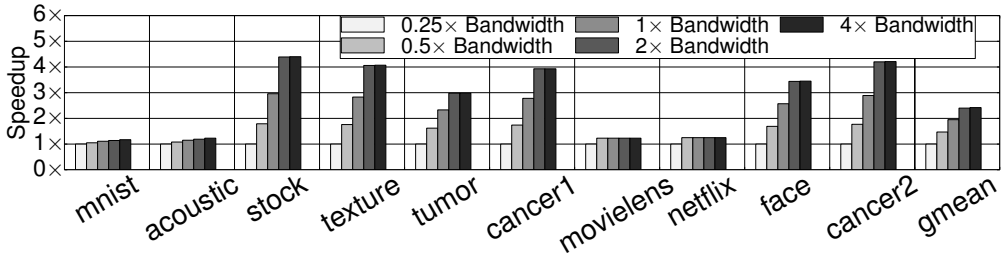
Figure 2.17 teases apart the benefits of FPGA acceleration from the benefits of the specialized system software over the three-node Spark. On average, the three FPGAs provide 20.7× speedup and the specialized system software–which also includes the aggregation part of the computation–is 28.4× faster than Spark's system software. As we discuss below, six of the benchmarks are more sensitive to data transfer and thus gain more benefits from the specialized system software compared to the benefits from FPGA. These benchmarks specifically benefit from the system software's task assignment that utilizes CPUs for both networking and aggregation of partial results from other nodes, thereby avoiding extra data transfer to the FPGAs. Nonetheless, all benchmarks gain from both FPGAs acceleration and specializing the system software.

2.8.2.6 Sensitivity to FPGA Resources and Bandwidth

CoSMIC can reshape and customize the template to match the resources of the target FPGAs or P-ASICs. The two main resources that affect performance are the number of PEs and the off-chip memory bandwidth. However, the DFG of the learning algorithm determines which resource is dominant. To study the in-



(a) Speedup of CoSMIC accelerator with increasing number of PEs
(Baseline: CoSMIC accelerator with 192 PEs)



(b) Speedup of CoSMIC accelerator as off-chip bandwidth changes
(Baseline: CoSMIC accelerator using 25% of UltraScale+ bandwidth)

Figure 2.18: Speedup comparison with varying number of PEs and memory bandwidth for CoSMIC accelerators.

terplay of algorithms and resources, we use a performance estimation tool that is validated against the hardware. Figure 2.18(a) illustrates the performance changes when the number of PEs varies from 192 to 6144 for a CoSMIC accelerator. The benchmarks that use the backpropagation (mnist and acoustic) and collaborative filtering algorithms (movielens and netflix) algorithms show performance benefits as the number of PEs increases, since they are compute-bound. The rest of the benchmarks–linear regression, logistic regression, and support vector machines do not see any performance gains when the number of DSPs increases. Although these benchmarks are offered more PEs, the limited bandwidth curtails their performance. Figure 2.18(b), which sweeps bandwidth, suggests the same categorization (bandwidth-bound vs. compute-bound) for our algorithms. These results show that a single fixed design is not the most optimal for all the algorithms. Therefore, there is a need for template architectures and solutions, such as CoSMIC, that customize the accelerator design

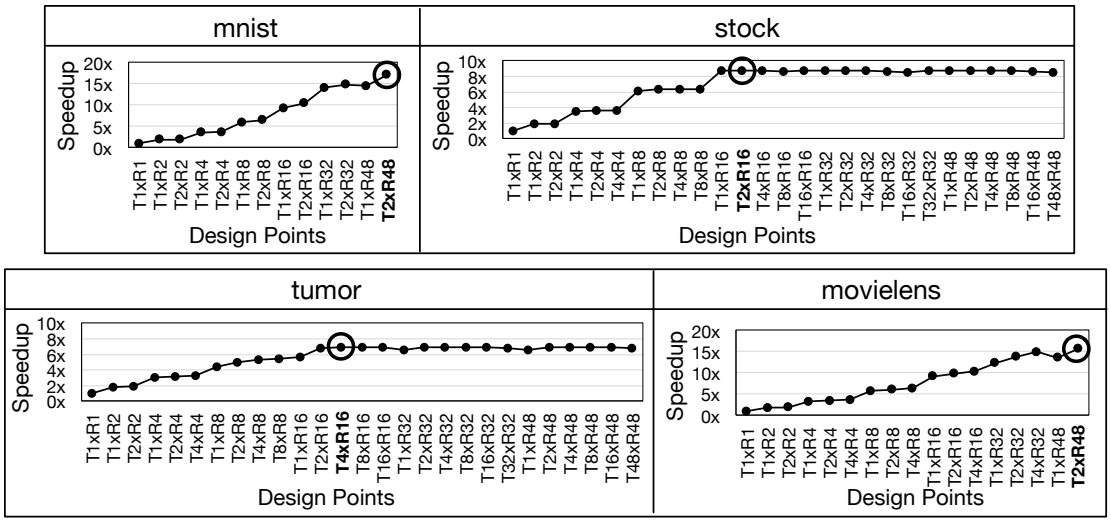


Figure 2.19: Design space exploration; Tx×Ry, x represents the number of threads and y represents the number of rows; baseline: T1×R1.

according to the algorithm. These results also suggest that modern accelerators need to strike a balance on allocating resource to off-chip communication and on-chip computation to maximize benefits for all benchmarks. Nonetheless, CoSMIC finds an optimal accelerator design considering both compute and bandwidth resources available on the FPGA.

2.8.2.7 Design Space Exploration

The Planner determines the number of PEs per thread and the number of threads in the accelerator. The Planner allocates PEs to each thread at the granularity of one row. This allocation strategy limits the design space that the Planner explores to find the optimal number of threads and rows-per-thread. In the case of UltraScale+ VU9P FPGA, the maximum number of possible design points is 27. Also, recall that the number of threads is also limited by the size of the model and not all the design points are possible. Figure 2.19 illustrates the result of this design space exploration for four different benchmarks. The performance of each design point is normalized to the design point which runs 1

Table 2.3: Number of threads and FPGA resource utilization.

Name	# Threads per FPGA	LUTs (Total: 1,182,240)		Flip Flops (Total: 2,364,480)		BRAM (Bytes) (Total: 9720 KB)		DSP Slices (Total: 6840)	
		Used	Util	Used	Util	Used	Util	Used	Util
mnist	2	851,276	72.0%	772,029	32.7%	8,640	88.9%	4,070	59.5%
acoustic	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
stock	8	278,838	23.6%	249,907	10.6%	8,640	88.9%	1,320	19.3%
texture	1	283,535	24.0%	257,005	10.9%	8,640	88.9%	1,355	19.8%
tumor	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer1	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%
movielens	2	851,276	72.0%	772,029	32.7%	8,128	83.6%	4,070	59.5%
netflix	1	851,947	72.1%	773,043	32.7%	8,128	83.6%	4,075	59.6%
face	4	281,522	23.8%	253,963	10.7%	8,640	88.9%	1,340	19.6%
cancer2	2	282,864	23.9%	255,991	10.8%	8,640	88.9%	1,350	19.7%

thread using 1 row (T1xR1) of PEs. We sweep the number of rows from 1 to 48, which is the maximum number of rows in UltraScale+ while the maximum number of threads varies for every benchmark. The optimal design points are highlighted with a concentric circle in the graphs. Benchmarks mnist and movielens see the highest speedup when they use all the 48 rows since they are compute-bound. In contrast, the performance for stock and tumor saturates beyond 16 rows. This result is commensurate with Figure 2.18(a), which shows that mnist and movielens benefit significantly with an increase in the FPGA's computational resources (PEs), while stock and tumor do not. The rest of the benchmarks show trends similar to the ones in Figure 2.19. Further, the figure shows that for a fixed number of PE rows, increasing the number of threads improves performance, which confirms the importance of multi-threading. Table 2.3 shows the resource utilization and the optimal number of threads-per-FPGA for all the benchmarks corresponding to the optimal design point chosen by the Planner. The resource utilization is highest for benchmarks that are compute-bound and lowest for the benchmarks that are bandwidth-bound. Moreover, the results show the benefits of our template-based approach that enables optimal utilization of the limited resources in the FPGA's reconfigurable fabric.

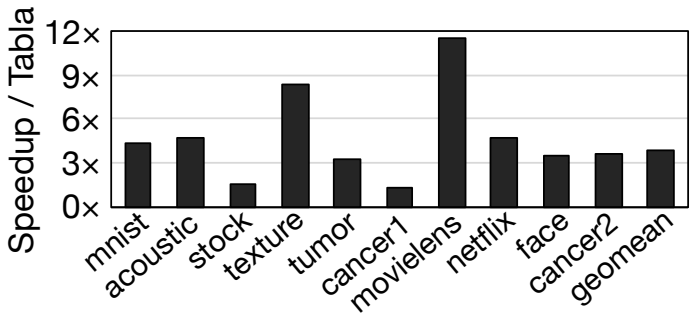


Figure 2.20: Speedup of CoSMIC's template architecture over TABLA's.

2.8.2.8 Comparison with TABLA

Prior work in TABLA [2] has explored single-node acceleration using a low-power FPGA (Zynq ZC702 with 220 DSPs). Our work, on the other hand, explores scale-out acceleration using modern high-power FPGAs (UltraScale+ with 6,840 DSPs). To provide a head-to-head comparison, we use the open-source TABLA framework [94] to generate accelerators for UltraScale+. We modify the templates for UltraScale+ and perform design space exploration to present the best results with TABLA. Figure 2.20 shows the speedup of CoSMIC compared to TABLA on UltraScale+ when using the same number of PEs. On average, CoSMIC performs 3.9× faster than TABLA. While both CoSMIC and TABLA use the same number of FPGA compute resources, the gap in performance shows that CoSMIC uses the compute resources more efficiently. The bottleneck for performance in TABLA is the communication of intermediate results due to data dependencies. As the number of DSPs in the TABLA architecture grows, the communication overhead grows significantly. To reduce the communication overhead, CoSMIC architecture uses a scalable tree-bus across rows of our 2-D PE architecture, and a bidirectional link between columns of PEs. Moreover, TABLA's compiler does not consider the overhead of data communication, which is particularly important when the number of PEs is large. CoSMIC compiler (Section 2.6) maps the operations of the learning

algorithm according to the location of data in order to reduce communication overhead. The combination of CoSMIC's scalable architecture, along with compiler optimization ensures that the FPGA's computational resources are used effectively.

2.9 Related Work

Multi-node accelerators for Machine Learning. DaDianNao [19] provides a multi-chip ASIC accelerator for DNNs. Other works use multiple FPGAs for accelerating one specific task [95, 96, 97]. Farabet et al. [95] and Donninger et al. [96] use multiple FPGAs to accelerate DNNs [95] and a chess game [96], respectively. Walters et al. [97] propose a multi-FPGA accelerator for the Hidden Markov Models [97]. Putnam et al. [27] provide an FPGA fabric for accelerating Bing's ranking algorithms [27]. Microsoft [28] also provides an infrastructure for deploying FPGAs in datacenters, which is also used for the inference phase of DNNs. This release does not deal with training nor does it offer a framework for programming. CoSMIC provides the necessary framework to utilize and program such an infrastructure [28] for machine learning algorithms without involving programmers in hardware design. Recently, Microsoft also unveiled Brainwave [98] that uses multiple FPGAs for DNN inference. In contrast, CoSMIC is a full stack to accelerate training at scale. Google's TPU [99] is a systolic array for acceleration of matrix multiplication, which is prevalent operation in ML. TPU is also programmable from Tensorflow [100] that recently supports distributed execution. In contrast, CoSMIC enables the use of FPGAs for scale-out acceleration and comes with its own template architecture.

Template-based acceleration. TABLA [2] is a single-node accelerator generator for machine learning, which also uses a template-based architecture.

As discussed in Section 2.8, TABLA, developed for a low-power FPGA (Zynq), does not effectively utilize the resources of a modern server-scale FPGA (UltraScale+). Furthermore, TABLA generates single-node FPGA accelerators which are inherently limited by the fine-grained parallelism available in the single-thread of stochastic gradient descent. In contrast, CoSMIC not only generates *scalable* accelerators for distributed systems using a novel *multi-threaded* template architecture, but also provides the necessary system software stack for scale-out acceleration. Moreover, the compilation algorithm of this work differs from TABLA. Our algorithm reduces the data communication by mapping data first. In contrast, TABLA's algorithm maps operations first to reduce the single-threaded latency. Additionally, our algorithm optimizes the mapping of operation to the FPGA's resources according to the location of data to avoid data marshaling. DNNWEAVER [3] is another template-based accelerator generator that only generates accelerators for prediction with Deep Neural Networks (DNNs). DNNWEAVER does not deal with training, multiple FPGAs, or algorithms besides DNNs. Cheng, et al. [101] propose predesigned data flow templates as the intermediate point for HLS from general C/C++ workloads. LINQits [102] provides a template architecture for accelerating database queries. The last two works [101, 102] do not focus on learning algorithms nor do they deal with scale-out systems.

Single-node accelerators for Machine Learning. There is a large body of work on single-node accelerator design for ML [41, 42, 43, 44, 45, 18, 21, 22, 23, 24, 25, 20, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 53, 54, 58, 59, 60, 61, 62]. These works mostly focus on accelerating one or a fixed number of learning algorithms. CoSMIC, on the other hand, is a full stack that targets scale-out acceleration of learning.

HLS for FPGAs. Many related works (e.g., [101, 108, 109, 62]) explore HLS

for FPGAs. HLS targets general applications while CoSMIC focuses specifically on machine learning. Therefore, HLS does not leverage any domain-specific knowledge or algorithmic insights. Using algorithmic commonalities for a range of machine learning algorithms is fundamental to our work and enables further benefits from hardware acceleration. Acceleration with HLS still requires hardware expertise. For instance, DNNWEAVER [3] reports that hardware design to optimize a Vivado HLS implementation of a deep neural network for FPGA took one month. The resulting implementation was an order of magnitude slower than a template-based accelerator for the same FPGA. A more recent work [101] uses dataflow templates as intermediate compilation target for C/C++ programs and delivers 9× higher performance than state-of-the-art HLS tools. CoSMIC takes a template-based approach that is driven by the theory of machine learning and targets distributed FPGA acceleration of training from a high-level domain-specific language.

System software for distributed FPGA acceleration. Another inspiring work [110] provides the mechanisms to integrate predesigned FPGA accelerator with Spark [16]. Melia [111] uses Altera's OpenCL-based HLS to offer a MapReduce-based framework for utilizing FPGAs in distributed systems. Another work [112] provides the framework for using Xilinx Vivado HLS tool for MapReduce [113] applications. CoSMIC does not rely on pre-developed FPGA accelerators or HLS for distributed FPGA acceleration, or generic system software.

2.10 Conclusion

While accelerators gain traction, their integration in the system stack is not well understood. CoSMIC takes an initial step toward such an integration for an important class of applications while providing generality and a high-level

programming interface. The evaluations confirm that a full-stack approach is necessary and just designing efficient accelerators does not yield proportional benefits without a co-design of the entire system stack. The traditional approaches of profiling and offloading hot-regions of code lack the flexibility to support ever-changing algorithms and the emerging scale and heterogeneity in the systems. It is clear that a full-stack design is non-trivial but deeply understanding algorithmic properties of the application domain can significantly facilitate such approaches. CoSMIC takes advantage of the algorithmic understanding to simplify the layers of its stack by specializing them and offers a cohesive hardware-software solution. The encouraging results show that this paradigm is effective but the multifaceted nature of the cross-stack approach promises an exciting yet challenging road ahead.

2.11 Algorithmic Approaches for ML Acceleration

As a preliminary effort for the CoSMIC project, we developed a single-node FPGA accelerator generation framework for data analytics, dubbed TABLA [2], which enables FPGA acceleration from high-level specifications of algorithms. We open-sourced the code and it is available at http://act-lab.org/artifacts/tabla. TABLA leverages the insight that many learning algorithms can be solved using stochastic gradient descent that minimizes an objective function. The solver is fixed while the objective function changes with the learning algorithm. Therefore, TABLA uses stochastic optimization as the abstraction between hardware and software. Consequently, programmers specify the learning algorithm by merely expressing the gradient of the objective function in our domain specific language. TABLA then automatically generates the synthesizable implementation of the accelerator for FPGA realization using a set of template designs.

Real hardware measurements show orders of magnitude higher performance and power efficiency while the programmer only writes 60 lines of code.

As a follow-on work, we developed DNNWEAVER [3], a framework that automatically generates a synthesizable accelerator for a given (DNN, FPGA) pair from a high-level specification in Caffe. To achieve large benefits while preserving automation, we devised hand-optimized design templates that the DNNWEAVER framework uses to generate the accelerators. First, DNNWEAVER translates a given high-level DNN specification to its novel ISA that represents a macro dataflow graph of the DNN. The DNNWEAVER compiler is equipped with our optimization algorithm that tiles, schedules, and batches DNN operations to maximize data reuse and best utilize target FPGA's memory and other resources. The final result is a custom synthesizable accelerator that best matches the needs of the DNN while providing high performance and efficiency gains for the target FPGA.

As the most recent effort in this line of research, we explored to leverage another algorithmic property of DNNs to open a new dimension in the design of DNN accelerators. We leverage the property that bitwidth of operations in DNNs can be reduced without compromising their classification accuracy. However, to prevent loss of accuracy, the bitwidth varies significantly across DNNs and it may even be adjusted for each layer individually. Thus, a fixed-bitwidth accelerator would either offer limited benefits to accommodate the worst-case bitwidth requirements, or inevitably lead to a degradation in final accuracy. To alleviate these deficiencies, we introduce the dynamic bit-level fusion/decomposition as the new dimension and develop Bit Fusion [114], a bit-flexible accelerator, that constitutes an array of bit-level processing elements that dynamically fuse to match the bitwidth of individual DNN layers. This flexibility in the architecture enables minimizing the computation and the communication at the

finest granularity possible with no loss in accuracy.

Chapter 3

LANGUAGE SUPPORT FOR APPROXIMATE PROGRAMMING

3.1 Introduction

Energy efficiency is a primary concern in modern systems. Mobile devices are limited by battery life and a significant fraction of the data center cost emanates from energy consumption. Furthermore, the dark silicon phenomenon limits the historical improvements in energy efficiency and performance [115]. Approximate computing is a promising approach that trades small and acceptable loss of output quality for energy efficiency and performance gains [116, 117, 118, 119, 120, 121, 122, 123]. This approach exploits the inherent tolerance of applications to occasional error to execute faster or use less energy. These applications span a wide range of domains including web search, big-data analytics, machine learning, multimedia, cyber-physical systems, speech and pattern recognition, and many more. For instance, a lossy video codec can tolerate imprecision and occasional errors when processing pixels of a frame. Practical programming models for approximation are vital to fully exploit this opportunity. Such models can provide significant improvements in performance and energy efficiency in the hardware by relaxing the abstraction of full accuracy [124, 125, 6, 126].

Safe execution of programs is crucial to the applicability of such techniques. That is, the programming model needs to guarantee that approximation will

never lead to catastrophic failures such as array out-of-bound exceptions. Recent works on approximate programming languages [119, 117] enable these techniques to provide such safety guarantees. These guarantees, however, come at the expense of extensive programmer annotations: programmers need to manually annotate all approximate variable declarations [119] or even annotate the safe-to-approximate operations [117]. This need for extensive annotations hinders the practical use of approximation techniques.

In this work, we propose a small set of language extensions that significantly lowers the annotation effort and paves the way for practical approximate programming. To achieve this goal, we identified the following challenges that need to be addressed. The extensions should enable programmers to annotate approximation-tolerant method outputs. The compiler then should automatically infer the operations and data that affect these outputs and selectively mark them approximable while providing safety guarantees. This process should be automatic and the language–compiler should be codesigned in order to relieve programmers from manually and explicitly annotating data declarations or operations. We address these challenges through the following contributions:

1. We introduce FlexJava, a small set of extensions that enables safe, modular, general, and scalable object-oriented approximate programming. It provides these features by introducing only four intuitive annotations. FlexJava supports modularity by defining a scope for the annotations based on the syntactic structure of the program. Scoping and adherence to program structure makes annotation a natural part of the software development process (Section 3.3).
2. The FlexJava annotations are designed to support both coarse-grained and fine-grained approximation, and enable programmers to specify a wide range of quality requirements, quality metrics, and recovery strategies (Section 3.3).

3. The language is codesigned with a compiler that *automatically* infers the safe-to-approximate data and operations from limited annotations on program or function outputs. The compiler statically enforces safety using a scalable dataflow analysis that conservatively infers the maximal set of safe-to-approximate data and operations. This automated analysis significantly reduces the number of annotations and avoids the need for safety checks at runtime (Section 3.5).
4. We implemented FlexJava annotations as a library to make it compatible with Java programs and tools. We extensively evaluate FlexJava using a diverse set of programs and by conducting a user study (Section 3.4 and Section 3.6).

The results of our evaluation show that FlexJava reduces the number of annotations (from 2$\times$ to 17$\times$) compared to EnerJ, a recent approximate programming language. We also conduct a user study that shows from 6$\times$ to 12$\times$ reduction in annotation time compared to EnerJ. With fine-grained approximation and small losses in quality, FlexJava provides the same level of energy savings (from 7% to 38%) compared to EnerJ. With coarse-grained approximation, FlexJava achieves even higher benefits—2.2$\times$ average energy reduction and 1.8$\times$ average speedup—for less than 10% quality loss.

A growing body of work is proposing new approximation techniques that stand to deliver an order of magnitude benefits in both energy and performance [127, 6, 128, 118, 122]. Our results suggest that practical programming solutions, such as FlexJava, are imperative for making these techniques widely applicable.

3.2 Background

Approximation techniques are broadly divided into two types: (1) fine-grained techniques that apply approximation at the granularity of individual instructions and data elements, and (2) coarse-grained techniques that apply approximation at the granularity of entire code blocks. FlexJava supports both types of techniques. We review the literature on these techniques before presenting the design of FlexJava.

3.2.1 Fine-Grained Approximation

Architectures support fine-grained approximation by allowing to execute interleaved approximate and precise instructions [119, 120, 8, 129, 117]. As Figure 3.1 shows, such architectures support both approximate operations and approximate storage. A bit in the instruction opcode identifies whether the instruction is the approximate or the precise version. Current proposals for approximate instructions lack room for enough bits to encode multiple approximation levels. As a result, we assume the prevalent binary-level approximation [119, 120, 8, 129, 117], although our approach can take advantage of multi-level approximation.

In this model, an approximate instruction has probabilistic semantics: it returns an approximate value with probability p and the precise value with prob-

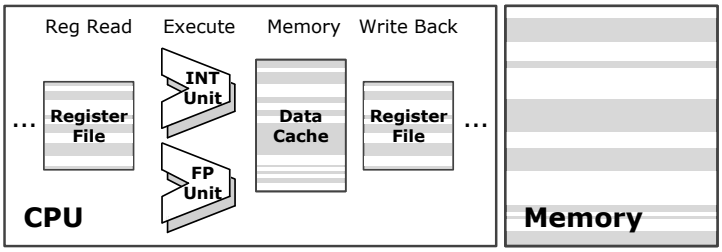


Figure 3.1: A processor that supports fine-grained approximation. The shaded units perform approximate operations or store data in approximate storage.

Table 3.1: Error probabilities and energy savings for different operations in fine-grained approximation. We consider the three hardware settings of Mild, Medium, and Aggressive from [119].

Operation	Technique		Mild	Medium	Aggressive
Integer Arithmetic/Logic	Voltage Overscaling	Timing Error Probability	10^{-6}	10^{-4}	10^{-2}
		Energy Reduction	12%	22%	30%
Floating Point Arithmetic	Bit-width Reduction	Mantissa Bits (`float`)	16 bits	8 bits	4 bits
		Mantissa Bits (`double`)	32 bits	16 bits	8 bits
		Energy Reduction	32%	78%	85%
SRAM Read/Write (Reg File/Data Cache)	Voltage Overscaling	Read Upset Probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
		Write Failure Probability	$10^{-5.6}$	$10^{-4.9}$	10^{-3}
		Energy Reduction	70%	80%	90%
DRAM (Memory)	Reduced Refresh Rate	Per-Second Bit Flip Probability	10^{-9}	10^{-5}	10^{-3}
		Memory Power Reduction	17%	22%	24%

ability $1 - p$. The approximate value may be arbitrary. The architecture also allows approximate storage, i.e., program data can be stored in approximate sections of the memory, cache, or registers. We use three such probabilistic architecture settings, shown in Table 3.1, that offer increasing energy savings with higher error probabilities. These models are similar to the ones that are used in recent works on approximate programming [119, 117].

3.2.2 Coarse-Grained Approximation

Coarse-grained approximation techniques concern approximating entire loop bodies or functions [116, 130, 128]. Loop perforation [116] is one such technique that transforms loops to skip a subset of their iterations. Green [130] substitutes functions with simpler approximate implementations or terminates loops early. NPUs [128] are a new class of accelerators that replace functions with hardware neural networks to approximately mimic the functions behavior. More generally, as the focus of the semiconductor industry shifts to programmable accelerators [131, 132, 133, 27], coarse-grained approximation can pave the way for new classes of approximate accelerators that can deliver significantly better performance and energy savings.

3.3 FlexJava Language Design

We have designed a set of language extensions for approximate programming that satisfy four key criteria:

1. **Safety.** The extensions guarantee ***safe*** execution. In other words, approximation can never lead to catastrophic failures, such as array out-of-bound exceptions.
2. **Modularity.** The extensions are ***modular*** and do not hinder modular programming and reuse.
3. **Generality.** The extensions are ***general*** and enable utilizing a wide range of approximation techniques without exposing their implementation details.
4. **Scalability.** The extensions are ***scalable*** and let programmers annotate large programs with minimal effort.

We have incorporated these extensions in the Java language. This section describes approximate programming in the resulting language FlexJava using a series of examples. In the examples, `bold-underline` highlight the safe-to-approximate data and operations that the FlexJava compiler infers automatically from the programmer annotations. Section 3.5 presents the formal semantics of the annotations and the static analysis performed by the compiler.

3.3.1 Safe Programming in FlexJava

Providing safety guarantees is the first requirement for practical approximate programming. That is, the approximation should never affect critical data and operations. The criticality of data and operations is a semantic property of the application that can only be identified by the programmer. The language must therefore provide a mechanism for programmers to specify where approximation is safe. This poses a language-compiler co-design challenge in order to

alleviate the need for manually annotating all the approximate data and operations. To address this challenge, we provide two language annotations, called **loosen** and **tighten**. These annotations provide programmers with full control over approximation without requiring them to manually and explicitly mark all the safe-to-approximate data and operations.

Selectively relaxing accuracy requirements. As discussed above, not all program data and operations are safe to approximate. Therefore, FlexJava allows each data and operation in the program to be either precise or approximate. Approximate data can be allocated in the approximate sections of memory, and an approximate operation is a variant that may generate inexact results. All data and operations are precise by default. The **loosen** annotation allows to relax the accuracy requirement on a specified variable at a specified program point. That is, any computation and data that *exclusively* affects the annotated variable is safe to approximate. For example, in the following snippet, the programmer uses **loosen**`(luminance)` to specify that the computation of `luminance` can be safely approximated.

```
float computeLuminance (float r, float g, float b) {
    float luminance = r * 0.3f + g * 0.6f + b * 0.1f;
    loosen(luminance);
    return luminance;
}
```

From this single annotation, the FlexJava compiler automatically infers that data `r`, `g`, `b`, and `luminance` can be safely allocated in the approximate memory. It also infers that all arithmetic operations, loads, and stores that contribute to calculating `luminance` are approximable. To provide memory safety and avoid null pointer exceptions, operations that calculate addresses to access `r`, `g`, and `b` are not approximable. A single annotation thus suffices to relax the accuracy of four variables and nine operations. Our language-compiler codesign

alleviates the need to manually annotate all these variables and operations.

Control flow safety.

To avoid unexpected control flow, FlexJava keeps all the computation and data that affects control flow precise by default. Consider the following example:

```
int fibonacci(int n) {
    int r;
    if (n <= 1)
        r = n;
    else
        r = fibonacci(n - 1) + fibonacci(n - 2);
    loosen(r);
    return r;
}
```

Variable `r` is annotated as an approximate output and `n` affects `r`. But since `n` also affects control flow in the conditional, it is not safe to approximate.

In many cases, conditionals represent simple control flow that can be converted to data dependence. Programmers can add explicit **loosen** annotations to mark such conditionals approximate. However, to reduce programmer effort, the FlexJava compiler automatically achieves this effect by conservatively converting control dependencies into data dependencies using a standard algorithm [134]. The following example illustrates this optimization:

```
double sobel(double[][] p){
    double x, y, g, r;
    x = p[0][0] + ...;
    y = p[0][2] + ...;
    g = sqrt(x * x + y * y);
    if (g > 0.7) r = 0.7;
    else r = g;
    loosen(r);
    return r;
}
```

```
double sobel(double[][] p){
    double x, y, g, r;
    x = p[0][0] + ...;
    y = p[0][2] + ...;
    g = sqrt(x * x + y * y);
    r = (g > 0.7) ? 0.7 : g;
    loosen(r);
    return r;
}
```

In the code snippet on the left, by annotating `r`, there are only a few opportunities for approximation since `r` depends on `g` which is used in the conditional. However, the FlexJava compiler can convert this control dependence to data dependence. This conversion is illustrated in the snippet on the right using the ternary `?:` operator. After conversion, `r` is only data dependent on `g`, which in turn makes `g` safe to approximate. Consequently, as the snippet on the right shows, all data and operations that affect `g` are also safe to approximate. As this example shows, this automation significantly increases approximation opportunities without the need for extra manual annotations.

Memory safety. Approximating address calculations may lead to memory access violations or contamination of critical data. To avoid such catastrophic failures and provide memory safety, any computation or data that affects address calculations is precise in FlexJava. Similarly, any computation or data that affects object allocation size is also precise. However, objects that do not contribute to address calculations, allocation sizes, or control flow may be allocated in approximate memory in accordance with the programmer annotations. Consider the following example:

```
int computeAvgRed (Pixel[] pixelArray) {
    int sumRed = 0;
    for(int i = 0; i < pixelArray.length; i++)
        sumRed = sumRed + (int) pixelArray[i].r;
    int avgRed = sumRed / pixelArray.length;
    loosen(avgRed);
    return avgRed;
}
```

Variables `i` and `pixelArray` are not approximable since they are used for address calculations. But the contents of the `Pixel` objects pointed to by the `pixelArray` elements, e.g., `pixelArray[i]`**.r**, are approximable due to **loosen**`(avgRed)`. As discussed before, programmers can always override the default semantics and relax these strict safety guarantees.

Restricting approximation.

FlexJava provides the **tighten** annotation which is dual to **loosen**. Annotating a variable with **tighten** makes any data or operation that affects the variable precise, *unless* a preceding **loosen** makes a subset of those data and operations approximable. The following examples illustrate the interplay between **loosen** and **tighten**:

```
float computeAvg (Pixel p) {
    float sum= p.r + p.g + p.b;
    tighten(sum);
    float avg = sum / 2.0f;
    loosen(avg);
    return avg;
}
```

```
float computeAvg (Pixel p) {
    float sum= p.r + p.g + p.b;
    loosen(sum);
    float avg = sum / 2.0f;
    tighten(avg);
    return avg;
}
```

In the left example, we relax the accuracy of data and operations that affect `avg` except those that affect `sum`. Conversely, in the right example, we relax the accuracy of data and operations that affect `sum` while keeping the last step of com-

puting `avg` precise. The FlexJava compiler automatically introduces **tighten** annotations to prevent approximating control flow and address calculations. The **tighten** annotation could also be used by programmers when critical data and operations are intertwined with their approximate counterparts. No such cases appeared when annotating the evaluated benchmarks (Section 3.6.1).

3.3.2 Modular Approximate Programming

Scoped approximation. Modularity is essential when designing a language since it enables reusability. To make approximate programming with FlexJava modular, we define a scope for the **loosen** annotation. The default scope is the code block that contains the annotation; e.g., the function or the loop body within which the **loosen** annotation is declared. As the following example illustrates, data and operations that are outside of the scope of the **loosen** annotation are not affected.

```
int p = 1;
for (int i = 0; i < a.length; i++)
    p *= a[i];
for (int i = 0; i < b.length; i++) {
    p += b[i];
    loosen(p);
}
```

Since **loosen** `(p)` is declared in the second loop that process the `b` array, the operations outside of this loop (e.g., `p *= a[i]`) are not affected and cannot be approximated. Assigning scope to the **loosen** annotation provides separation of concerns. That is, the **loosen** annotation only influences a limited region of code that makes it easier for programmers to reason about the effects of the annotation. Furthermore, the scope of approximation adheres to the syntactic structure of the program that makes annotating the code a natural part of the

program development.

To ensure safety, the scope for the **tighten** annotation is the entire program. All data and operations in the program that affect the annotated variable in **tighten** will be precise. The same principle applies to the conditionals and pointers. The FlexJava compiler automatically applies these global semantics and relieves programmers from safety concerns.

Reuse and library support in FlexJava. Composing independently developed codes to build a software system is a vital part of development. Composability must be supported for the annotations. To this end, we define two variants for the **loosen**; the default case and the invasive case (**loosen_invasive**). These variants have different semantics when it comes to function calls. If a function call is in the scope of a **loosen** annotation and its results affects the annotated variable, it may be approximated only if there are **loosen** annotations within the function. In other words, the caller's annotations will not interfere with the annotations within the callee and may only enable them. If the callee does not affect caller's annotated variable, its internal **loosen** annotations will not be enabled. With this approach, the library developers can develop general approximate libraries independently regardless of the future specific use cases. The users can use these general libraries without concerning themselves with the internal annotations of the libraries. The following examples demonstrate the effects of **loosen** for function calls.

```
static int square(int a){
  int s = a * a;
  loosen(s);
  return s;
}
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen(x);
  System.out.println(x);
}
```

```
static int square(int a){
  int s = a * a;
  loosen(s);
  return s;
}
public static void main
        (String[] args){
  int x = 2 + square(3);


  System.out.println(x);
}
```

In the left example, as highlighted, **loosen(x)** declares the local operations with the `main` function as safe-to-approximate. The annotation also enables approximation in the `square` function that was called in the scope of the **loosen(x)** annotation. Within the `square` function, the approximation will be based on the annotations that are declared in the scope of `square`. As the right example illustrates, if there are no **loosen** annotations in the caller function, `main`, nothing will be approximated in the callee function, `square`.

An expert user may want to apply approximation to the callee functions even if they do not contain any internal annotations. FlexJava provides the **loosen_invasive** for such cases. The **loosen_invasive** enables applying approximation to the conventional libraries that are not annotated for approximation. Note that **loosen_invasive** does not cause control flow or memory address calculations to be approximated as we discussed for **loosen**. The only difference is how approximation is enforced in the callee function as illustrated below.

```
static int square(int a){
  int s = a * a;
  return s;
}
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen(x);
  System.out.println(x);
}
```

```
static int square(int a){
  int s = a * a;
  return s;
}
public static void main
        (String[] args){
  int x = 2 + square(3);
  loosen_invasive(x);
  System.out.println(x);
}
```

In the left example, the **loosen(x)** annotation approximates the local operations in `main` function but will not lead to any approximation in the `square` function since it does not contain any **loosen** annotations. In contrast, in the right example, **loosen_invasive(x)** enforces safe approximation in `square` since its return value affects `x`.

Supporting separate compilation. FlexJava supports separate compilation [135]. That is, a FlexJava program can link with both annotated and unannotated precompiled code without having to re-compile it. If the precompiled code is not annotated, it executes precisely. If the precompiled code is annotated, its annotations are respected and its data and operations are approximated accordingly. Moreover, the annotations in the new program will not approximate any additional operations and data in the precompiled code other than the ones already approximated by annotations in them.

3.3.3 OO Programming in FlexJava

To this point, we have described how to use FlexJava annotations to identify approximate data and operations within methods of a class. This section describes how to declare class fields as approximate and how inheritance and

polymorphism interplay with the annotations.

Approximating class fields. Since class fields are not declared in the scope of any of the methods, we allow the programmers to *selectively* relax their semantics in the constructor of the class. The fields will be allocated in the approximate section of the memory if an *outer-level* **loosen** enables approximation in the constructor. In principle, instantiation of an object involves a function call to the constructor. The outer-level **loosen** annotations have the same effect on constructors as they have on other function calls.

```
class A {
  float x, y;
  A (float x, float y) {
    this.x = x;
    this.y = y;
    loosen(x); }
  public static void main() {
    A a = new A(1.5f, 2.0f);
    float p = 3.5f + a.x;
    loosen(p);
  }
}
```

The annotated `p` is affected by the instance of `A`. Therefore, **loosen(p)** enables approximation in the constructor. Consequently, the `x` field will be allocated in the approximation section of the memory because of the **loosen(x)** in the constructor. The `y` field will not be allocated in the approximation section since it is not annotated in the constructor.

Inheritance. When inheriting an annotated class, annotations are preserved in methods that are not overridden. Naturally, if the child class overrides a method, the overriding method must be re-annotated if approximation is desired.

Polymorphism due to approximation. Depending on the annotations, differ-

ent instances of a class and different calls to a method may carry approximate or precise semantics. The FlexJava compiler generates different versions of such classes and methods using code specialization [136].

3.3.4 Generality in FlexJava: Support for Coarse-Grained Approximation

The annotations discussed so far enable fine-grained approximation at the level of single operations and data. This section describes another form of annotations, the **begin_loose**–**end_loose** pair, that enables coarse-grained approximation in FlexJava. Any arbitrary code block that is enclosed between this pair of annotations can be approximated as a whole. Both annotations have a variable argument list. The first argument of **begin_loose**, which is a string, identifies the type of approximation that can be applied to the code block. The compiler or the runtime system then can automatically apply the corresponding approximation technique. Some approximation techniques may require programmers to provide more information. For example, function substitution [130] requires the programmer to provide an approximate version of the function. This extra information can be passed to the compiler or runtime system through the arguments of **begin_loose** or **end_loose**. This approach is flexible enough to enable a variety of coarse-grained approximation techniques. We describe how to use the approach with two such techniques: loop perforation [116] and NPUs [128, 48, 137].

Loop perforation. Loop perforation [116] allows the runtime to periodically skip iterations of loops. The programmer can set the initial rate of perforation (skipping the iterations). FlexJava annotations can be used for loop perforation as the following example shows.

```
begin_loose("PERFORATION", 0.10);
    for (int i = 0; i < n; i++)  { ... }
end_loose();
```

The **begin_loose**("PERFORATION", 0.10) and **end_loose()** annotations identify the loop that can be approximated. The first argument of **begin_loose**, "PERFORATION", declares that the desired approximation technique is loop perforation. The second argument, 0.10, identifies the rate of perforation.

Neural acceleration. Neural Processing Units (NPU) [128, 123, 6, 48, 137] are a new class of accelerators that replace compute-intensive functions with hardware neural networks. We give an overview of the NPU compilation workflow since we use them to evaluate FlexJava's coarse-grained annotations. The compiler first automatically trains a neural network on how an approximable code block behaves. Then, it replaces the original block with an efficient hardware implementation of the trained neural network or the NPU. This automatic code transformation also identifies the inputs and outputs of the region. The compiler performs the transformation in four steps:

1. **Input/output identification.** To train a neural network to mimic a code block, the compiler needs to collect the input-output pairs that represent the functionality of the block. Therefore, the first step is identifying the inputs and outputs of the delineated block. The compiler uses a combination of live variable analysis and Mod/Ref analysis [138] to automatically identify the inputs and outputs of the annotated block. The inputs are the intersection of live variables at the location of **begin_loose**("NPU") with the set of variables that are referenced within the segment. The outputs are the intersection of live variables at the location of **end_loose()** with the set of variables that are modified within the segment. In the example that follows, this analysis identifies x and y as the inputs to the block and p and q as the outputs.

2. **Code observation.** The compiler instruments the program by putting probes on the inputs and outputs of the block. Then, it profiles the instrumented program using representative input datasets such as those from a test suite. The probes log the block inputs and outputs. The logged input–output pairs form the training dataset.
3. **Training.** The compiler uses the collected input–output dataset to configure and train a multilayer perceptron neural network that mimics the approximable block.
4. **Code generation.** Finally, the compiler replaces the original block with a series of special instructions that invoke the NPU hardware, sending the inputs and receiving the computed approximate outputs.

The following example illustrates the use of FlexJava annotations for NPU acceleration.

```
Double foo(Double x, Double y) {
  begin_loose("NPU");
    p = Math.sin(x) + Math.cos(y);
    q = 2 * Math.sin(x + y);
  end_loose();
  return p + q;
}
```

The programmer uses **begin_loose**–**end_loose** to indicate that the body of function `foo` is a candidate for NPU acceleration. The first argument of **begin_loose** `("NPU")` indicates that the approximation technique is NPU acceleration.

3.3.5 Support for Expressing Quality Metrics, Quality Requirements, and Recovery

Practical and complete approximate programming languages need to provide a mechanism to specify and express quality metrics, quality requirements, and

```
package edu.flexjava;
abstract class QualityMetric {
  double acceptableQualityLoss = 0.0;
  QualityMetric(double q) { acceptableQualityLoss = q; }
  abstract void checkQuality(Object... o);
  abstract void recover(Object... o);
}
```

Figure 3.2: An abstract class for defining the quality metric.

```
package edu.flexjava;
class FlexJava {
  static void loosen(Object... o) {}
  static void loosen_invasive(Object... o) {}
  static void tighten(Object... o) {}
  static void begin_loose(String type, Object... o) {}
  static void end_loose(Object... o) {}
}
```

Figure 3.3: FlexJava annotations are implemented as a library.

recovery mechanisms. As shown in prior works on approximation, quality metrics are application dependent [119, 116, 120, 128, 130]. For example, an image processing application may use signal-to-noise ratio as the quality metric, while the quality metric for web search is relevance of the results to the search query. The quality metric for machine learning algorithms that perform classification is the misclassification rate. Consequently, the common practice in approximate computing is for programmers to specify the application quality metric and the acceptable level of quality loss. The FlexJava annotations can be naturally extended to express quality metrics and requirements.

As Figure 3.2 shows, we first provide an abstract class as a template for implementing the quality metric function. The programmer can implement this abstract class and override the `checkQuality` function to implement the quality metric. The constructor of this class can be used to set the acceptable level of quality loss, `acceptableQualityLoss`. The programmer can also override the `recover` to implement a recovery procedure for the occasions that the quality loss is greater than the requirements. Note that the quality re-

quirement can be expressed as a probability if desired. After implementing the `QualityMetric` class, the programmer can pass its instance via the last argument of **loosen**, **loosen_invasive**, or **end_loose** to the compiler or the runtime system. Clearly, the programmer need not specify the quality metric in each such annotation; it is usually specified only when annotating the final output or important functions of the application, as illustrated in the following example.

```
static int cube (int x) {
    int y = x * x * x;
    loosen(y);
    return y;
}
public static void main (String[] args) {
    int z = cube(7);
    loosen(z, new ApplicationQualityMetric(0.10));
    System.out.println(z);
}
```

Notice that the quality requirement is not specified in the function or library annotations (**loosen** `(y)`). It is specified only in the last annotation on the final output `z` of the program. In this example, the acceptable quality loss is 10%, which is passed to the constructor as `0.10`.

3.4 FlexJava Implementation

FlexJava is a small set of extensions to Java that enables safe, modular, general, and scalable object-oriented approximate programming. It achieves these goals by introducing only four intuitive annotations: **loosen**, **tighten**, **loosen_invasive**, and the **begin_loose**–**end_loose** pair. In this section, we describe our implementation of these annotations and the development environment of FlexJava.

3.4.1 Implementation of Annotations

We implemented FlexJava annotations as a library to make it compatible with Java programs and tools. Figure 3.3 illustrates this library-based implementation that provides the interface between the FlexJava language and compiler. The `FlexJava` class implements the annotations as empty variable-length argument functions. Consequently, compiling a FlexJava program with a traditional compiler yields a fully precise executable. The approximation-aware compiler, however, can intercept calls to these functions and invoke the necessary analyses and approximate transformations.

3.4.2 Integrated Highlighting Tool

FlexJava is coupled with a static approximation safety analysis that automatically infers the safe-to-approximate operations and data from the programmer annotations. We have developed an integrated tool that highlights the source code with the result of this analysis. By visualizing the result, this tool further facilitates FlexJava programming and can help programmers to refine their annotations. In its current form, the integrated tool adds comments at the end of each line showing which of the line's operations are safe to approximate. It is straightforward to convert this visual feedback to syntax highlighting. In fact, we used the result of this tool to highlight the examples in Section 3.3.

3.5 Approximation Safety Analysis

In this section, we define the formal semantics of approximation safety for annotated programs in FlexJava. We define a core language with **loosen** and **tighten** annotations. We give a concrete semantics parameterized by the set of operations to be approximated in an annotated program in the language. The

$$
\begin{array}{rr}
(\textit{real constant})\ r \in \mathbb{R} & (\textit{variable})\ v \in \mathbb{V} \\
(\textit{real expression})\ e \in \mathbb{R} \cup \mathbb{V} & (\textit{operation label})\ l \in \mathbb{L} \\
\multicolumn{2}{c}{(\textit{statement})\ s ::= v :=^{l} \delta(e_1, e_2) \mid \mathsf{loosen}(v) \mid \mathsf{tighten}(v)} \\
\multicolumn{2}{c}{\mid \mathsf{assume}(v) \mid s_1; s_2 \mid s_1 + s_2 \mid s^*}
\end{array}
$$

Figure 3.4: Language syntax.

$$
\begin{array}{rr}
(\textit{stack})\ \rho \in \mathbb{V} \to \mathbb{R} & (\textit{tainted set})\ T \subseteq \mathbb{V} \\
\multicolumn{2}{c}{(\textit{state})\ \omega ::= \langle s, \rho, T \rangle \mid \langle \rho, T \rangle \mid \mathsf{error} \mid \mathsf{halt}}
\end{array}
$$

Figure 3.5: Semantic domains.

semantics determines if a given set of operations is approximable. As this problem is undecidable, we develop a static analysis that conservatively infers the largest set of approximable operations in a given annotated program.

3.5.1 Core Language

Figure 3.4 shows the syntax of our core language. It supports real-valued data and control-flow constructs for sequential composition, branches, and loops. We elide conditionals in branches and loops, executing them nondeterministically and using the $\mathsf{assume}(v)$ construct that halts if $v \leq 0$.

We extend the language with annotations $\mathsf{loosen}(v)$ and $\mathsf{tighten}(v)$. These annotations arise from their source-level counterparts described in Section 3.3. Further, $\mathsf{tighten}(v)$ is implicitly added by the FlexJava compiler before each use of variable v in a conditional, an array index, a pointer dereference, or a program output. To statically identify operations that are approximable under the given annotations, each operation has a unique label l.

Example. We illustrate the above concepts for the program on the left below. For now, ignore the sets in annotations next to each line of the program.

```
                              L={1,2,5,6}     L={2,6}
 1: v1 := input();            {{v1}}          {{}}
 2: v2 := input();            {{v1,v2}}       {{v2}}
 3: tighten(v1);              {T}             {{v2}}
 4: while (v1 > 0) {          {T}             {{v2}}
 5:   v1 := f(v1);            {T}             {{v2}}
 6:   v2 := g(v2);            {T}             {{v2}}
 7:   tighten(v1);            {T}             {{v2}}
 8: }                         {T}             {{v2}}
 9: loosen(v2);               {T}             {{}}
10: tighten(v2);              {T}             {{}}
11: output(v2);               {T}             {{}}
```

The compiler introduces $\mathsf{tighten}(\mathtt{v1})$ on lines 3 and 7 to ensure that $\mathtt{v1} > 0$ executes precisely, and $\mathsf{tighten}(\mathtt{v2})$ on line 10 to ensure that the value of $\mathtt{v2}$ output on line 11 is precise. The programmer relaxes the accuracy of $\mathtt{v2}$ on line 9, which allows the operations writing to $\mathtt{v2}$ on lines 2 and 6 to be approximated without violating the $\mathsf{tighten}(\mathtt{v2})$ requirement on line 10. However, the operations writing to $\mathtt{v1}$ on lines 1 and 5 cannot be approximated as they would violate the $\mathsf{tighten}(\mathtt{v1})$ requirement on line 3 or 7, respectively. □

3.5.2 Concrete Semantics

We define a concrete semantics to formalize approximation safety for our language. Figure 3.5 shows the semantic domains. Each program state ω (except for special states error and halt described below) tracks a *tainted set* T of variables. A variable gets tainted if its value is affected by an approximate operation, and untainted if **loosen** is executed on it.

$$L \models \langle v :=^{l} \delta(e_1, e_2), \rho, T\rangle \rightsquigarrow \langle \rho[v \mapsto [[\delta(e_1, e_2)]](\rho)], T'\rangle$$
$$\text{where } T' = \begin{cases} T \cup \{v\} & \text{if } l \in L \text{ or } uses(e_1, e_2) \cap T \neq \emptyset \\ T \setminus \{v\} & \text{otherwise} \end{cases} \qquad \text{(ASGN)}$$

$$L \models \langle \mathsf{loosen}(v), \rho, T\rangle \rightsquigarrow \langle \rho, T \setminus \{v\}\rangle \qquad \text{(LOOSEN)}$$

$$L \models \langle \mathsf{tighten}(v), \rho, T\rangle \rightsquigarrow \langle \rho, T\rangle \quad [\text{if } v \notin T] \qquad \text{(TIGHTENPASS)}$$

$$L \models \langle \mathsf{tighten}(v), \rho, T\rangle \rightsquigarrow \mathsf{error} \quad [\text{if } v \in T] \qquad \text{(TIGHTENFAIL)}$$

$$L \models \langle \mathsf{assume}(v), \rho, T\rangle \rightsquigarrow \langle \rho, T\rangle \quad [\text{if } \rho(v) > 0] \qquad \text{(ASMPASS)}$$

$$L \models \langle \mathsf{assume}(v), \rho, T\rangle \rightsquigarrow \mathsf{halt} \quad [\text{if } \rho(v) \leq 0] \qquad \text{(ASMFAIL)}$$

Figure 3.6: Concrete semantics of approximation safety.

Figure 3.6 shows the semantics as a set of rules of the form:

$$L \models \langle s, \rho_1, T_1\rangle \rightsquigarrow \langle \rho_2, T_2\rangle \mid \mathsf{halt} \mid \mathsf{error}$$

It describes an execution of annotated program s when the set of approximated operations is L, starting with stack (i.e., valuation to variables) ρ_1 and tainted set T_1. The rules are similar to information flow tracking: approximated operations in L are *sources* (rule ASGN), $\mathsf{loosen}(v)$ are *sanitizers* (rule LOOSEN), and $\mathsf{tighten}(v)$ are *sinks* (rules TIGHTENPASS and TIGHTENFAIL). The execution ends in state error if some $\mathsf{tighten}(v)$ is executed when the tainted set contains v, as described by rule TIGHTENFAIL. The execution may also end in state halt, which is normal and occurs when $\mathsf{assume}(v)$ fails (i.e., $v \leq 0$), as described by rules ASMPASS and ASMFAIL. We omit the rules for compound statements and those that propagate error and halt, as they are relatively standard and do not affect the tainted set.

We now define approximation safety formally:

Defn 3.5.1 (Approximation safety) *A set of operations L in a program s is approximable if* $\forall \rho : L \models \langle s, \rho, \emptyset\rangle \not\rightsquigarrow \mathsf{error}$.

To maximize approximation, we seek as large a set of approximable operations

as possible. In fact, a unique largest set exists, as our semantics satisfies the property that if operation sets L_1 and L_2 are approximable, then so is $L_1 \cup L_2$.

Example. In the example program, the largest set of approximable operations is those on lines 2 and 6. Column `L={2,6}` shows the tainted set as per our semantics after each statement under this set of approximated operations. The error state is unreachable in any run as the tainted set at each $\texttt{tighten}(v)$ does not contain v. Hence, this set of operations is approximable. □

3.5.3 Static Analysis

The problem of determining if a given set of operations is approximable in a given annotated program even in our core language is undecidable. We present a novel static analysis that *conservatively* solves this problem, i.e., if the analysis deems a set of operations as approximable, then it is indeed approximable according to Defn. 3.5.1. Further, we apply an efficient algorithm that uses the analysis to automatically infer the largest set of approximable operations.

Our static analysis is shown in Figure 3.7. It *over-approxim-ates* the tainted sets that may arise at a program point in the concrete semantics by an abstract state D, a set each of whose elements is $\top$ or an *abstract tainted set* π of variables.

The analysis is a set of transfer functions of the form $F_L[s](D) = D'$, denoting that when the set of approximated operations is L, the annotated program s transforms abstract state D into abstract state D'. The element $\top$ arises in D' either if it already occurs in D or if s contains a $\texttt{tighten}(v)$ statement and an abstract tainted set incoming into that statement contains the variable v. Thus, the element $\top$ indicates a potential violation of approximation safety. In particular, an annotated program does not violate approximation safety if the analysis determines that, starting from input abstract state $\{\emptyset\}$, the output abstract state

$$
\begin{array}{rcl}
(\textit{abstract tainted set}) & \pi & \in \quad \Pi = 2^{\mathbb{V}} \\
(\textit{abstract state}) & D & \subseteq \quad \mathbb{D} = \Pi \cup \{\top\}
\end{array}
$$

$$
\begin{array}{rcl}
F_L[s] & : & 2^{\mathbb{D}} \to 2^{\mathbb{D}} \\
F_L[s_1\,;s_2](D) & = & (F_L[s_1] \circ F_L[s_2])(D) \\
F_L[s_1 + s_2](D) & = & F_L[s_1](D) \cup F_L[s_2](D) \\
F_L[s^*](D) & = & \mathsf{leastFix}\ \lambda D'.(D \cup F_L[s](D')) \\
F_L[t](D) & = & \{\, \mathsf{trans}_L[t](d) \mid d \in D \,\}
\end{array}
$$

for atomic statement t, where:

$$
\begin{array}{rcl}
\mathsf{trans}_L[t](\top) & = & \top \\
\mathsf{trans}_L[v :=^l \delta(e_1,e_2)](\pi) & = & \begin{cases} \pi \cup \{v\} \text{ if } l \in L \vee \\ \qquad\qquad uses(e_1,e_2) \cap \pi \neq \emptyset \\ \pi \setminus \{v\} \text{ otherwise} \end{cases} \\
\mathsf{trans}_L[\mathsf{tighten}(v)](\pi) & = & \begin{cases} \pi \text{ if } v \notin \pi \\ \top \text{ otherwise} \end{cases} \\
\mathsf{trans}_L[\mathsf{loosen}(v)](\pi) & = & \pi \setminus \{v\}
\end{array}
$$

Figure 3.7: Approximation safety analysis.

does not contain $\top$:

Theorem 3.5.2 (Soundness) *For each program s, if $\top \notin F_L[s](\{\emptyset\})$ then for each state ρ, $L \models \langle s, \rho, \emptyset \rangle \not\rightsquigarrow \mathsf{error}$.*

Example. For our example from Section 3.5.1, the columns on the right show the abstract state computed by the analysis after each statement, under the set of approximated operations indicated by the column header. For `L={1,2,5,6}`, the final abstract state contains $\top$, and indeed the operations on lines 1 and 5 are not approximable. But for `L={2,6}`, the final abstract state does not contain $\top$, proving that operations on lines 2 and 6 are approximable. □

Our static analysis has the useful property that for any annotated program, there exists a unique largest set of operations that it considers approximable.

Theorem 3.5.3 (Unique largest solution) $\exists\, L_{max} \subseteq \mathbb{L} : \top \notin F_{L_{max}}[s](\{\emptyset\}) \wedge (\top \notin F_L[s](\{\emptyset\}) \Rightarrow L \subseteq L_{max})$.

We use a standard algorithm [139] to infer this largest set of approximable operations. Starting with all operations approximated, it iteratively finds a largest

set of approximable operations which passes all the **tighten** checks in the program.

3.6 Evaluation

This section aims to answer the following research questions.

- **RQ1**: Can FlexJava significantly reduce the number of manual annotations?
- **RQ2**: Can FlexJava significantly reduce the programmer effort and annotation time?
- **RQ3**: Can FlexJava give significant speedup and energy gains with both fine- and coarse-grained approximation?

As the results of the evaluations show, FlexJava reduces the number of annotations (between 2× and 17×) compared to EnerJ, the leading approximation language. We also conduct a user study that shows from 6× to 12× reduction in annotation time compared to EnerJ. FlexJava; however, provides the same level of energy savings (from 7% to 38%) compared to EnerJ with fine-grained approximation. With coarse-grained approximation, FlexJava achieves 2.2× energy reduction and 1.8× speedup for under 10% quality loss.

Benchmarks and quality metrics. As Table 3.2 shows, we evaluate FlexJava using 10 Java programs. Eight are the EnerJ benchmarks [119]. We use two additional benchmarks, hessian and sobel. Five of these come from the SciMark2 suite. The rest are zxing, an Android bar code recognizer; jmeint, an algorithm to detect intersecting 3D triangles (part of the jMonkeyEngine game engine); sobel, an edge detection application based on the Sobel operator; and raytracer, a simple 3D ray tracer. To better study the scalability of our analy-

Table 3.2: Benchmarks, quality metrics, and results of safety analysis: analysis runtime and # of approximable data and operations.

	Description	Quality Metric	# of Lines		Analysis Runtime (sec)	# of Approximable Data		# of Approximated Operations	
			Bench	Library		Inferred	Potential	Inferred	Potential
sor	SciMark2 benchmark: scientific kernels	Avg entry difference	36	60K	6	8	14	133	282
smm		Avg normalized difference	38	60K	6	9	17	114	278
mc		Normalized difference	59	60K	4	7	13	129	184
fft		Avg entry difference	168	60K	11	11	14	226	485
lu		Avg entry difference	283	60K	15	12	19	201	600
sobel	Image edge detection	Avg pixel difference	163	284K	102	2	5	153	416
raytracer	3D image renderer	Avg pixel difference	174	214K	14	4	9	128	264
jmeint	jMonKeyEngine game: triangle intersection kernel	Percents of correct decisions	5,962	216K	296	71	71	832	943
hessian	Interest point detection in BoofCV library	Avg Euclidean distance	10,174	261K	6,228	73	119	663	4988
zxing	Bar code decoder for mobile phones	Percents of correct results	26,171	271K	12,722	996	1,053	2,673	8,454

sis, we added the hessian application from the BoofCV vision library with 10,174 lines of code. This application uses the Hessian affine region detector to find interesting points in an image. The code for this application uses Java generics that is not supported by the EnerJ compiler and simulator. However, our safety analysis supports Java generics and was able to analyze this application. Therefore, only for this specific application, our comparisons are limited to annotation effort and safety analysis. Table 3.2 also shows the application-specific quality metrics. We measure quality by comparing the output of the fully precise and the approximated versions of the program. For each benchmark, we use 10 representative input datasets such as 10 different images. The quality degradation is averaged over the input datasets.

3.6.1 RQ1: Number of Annotations

To answer RQ1, we compare the number of EnerJ annotations with FlexJava annotations. We use EnerJ as a point of comparison because it requires the minimum number of annotations among existing approximate languages [119, 117]. EnerJ requires programmers to annotate all the approximate data declarations using type qualifiers. Then, the EnerJ compiler infers the safe-to-approximate operations for fine-grained approximation. In contrast, our approximation safety analysis *infers* both approximate data and operations from a limited number of FlexJava annotations on the program or function outputs. We used the Chord program analysis platform [140] to implement our approximation safety analysis. Compared to EnerJ, our analysis infers at least as many number of safe-to-approximate data and operations with significantly fewer number of manual annotations.

Figure 3.8 shows the number of annotations with EnerJ and FlexJava. As Figure 3.8 illustrates, there is a significant reduction in the number of annota-

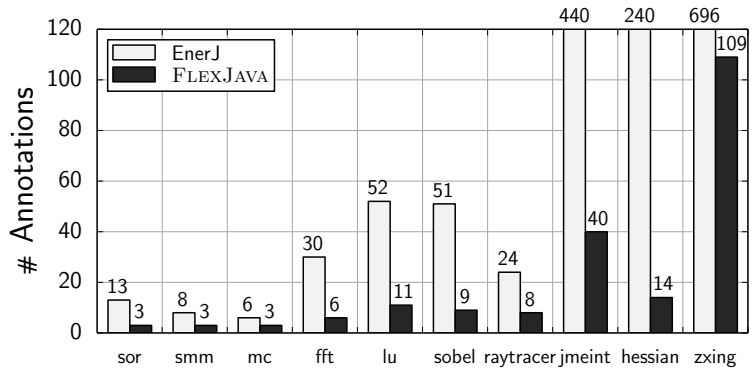


Figure 3.8: Number of annotations required to approximate the same set of data and operations using EnerJ and FlexJava.

tions with FlexJava. FlexJava requires between 2× (mc) to 17× (hessian) less annotations than EnerJ. The largest benchmark in our suite is zxing with 26,171 lines of code. It requires 696 annotations with EnerJ, 109 annotation with FlexJava. Thus, FlexJava reduces the number of annotations by a factor of 6×. The zxing benchmark needs several **loosen** annotations to mark its function outputs as approximable. Further, many condition variables are safe to approximate and such variables need to be annotated explicitly. Therefore, zxing requires a number of FlexJava annotations that is relatively large compared to all other benchmarks. These results confirm that FlexJava annotations and its approximation safety analysis can effectively reduce the number of manual annotations.

The results in Figure 3.8 are with no use of **loosen_invasive**. Using **loosen_invasive** only reduces the number of annotations with FlexJava. Moreover, in the evaluated benchmarks, there is no need for any manual **tighten** annotations. As described before, FlexJava's approximation safety analysis automatically inserts **tighten** annotations for the critical variables to ensure control flow and memory safety. The FlexJava highlighting tool was useful since it effectively visualizes the result of the automated approximation safety analysis.

Approximation safety analysis. In Table 3.2, columns "# of Lines" and "Analysis Runtime (sec)" report the number of lines in each program and the runtime of the approximation safety analysis. The analysis analyzes application code and reachable Java library (JDK) code uniformly although we report their sizes separately in the table. The analysis was performed using Oracle HotSpot JVM 1.6.0 on a Linux machine with 3.0 GHz quad-core processors and 64 GB memory.

The analysis runtime strongly correlates with the number of potentially approximable data and operations. The potential approximable elements include all the data declarations and all the operations that are not address calculations and jump or branch instructions in the byte code. The number of potential elements is presented in columns "# of Approximable Data-Potential" and "# of Approximable Operations-Potential", respectively. The analysis determines whether or not each of these elements is safe to approximate with respect to the programmer annotations. The number of all the potential approximable elements defines the search space of the analysis. Thus, the space of possible solutions that the approximation safety analysis explores for zxing is of size $2^{(1053+8454)}$. Automatically finding the largest set of approximable elements from this huge space justifies the 12,722 seconds (=3 hours and 32 minutes) of running time to analyze zxing. However, the analysis runtime is not exponential with respect to the number of potential elements. That is because in each iteration, the analysis eliminates at least one element from the potentials list.

Naturally, significantly reducing the number of manual annotations requires an automated analysis that takes some machine time. That is, the analysis is trading machine time for fewer annotations, potentially saving programmer time. Furthermore, we report the pessimistic runtime when all of the libraries and program codes are analyzed in a single compiler run without separate compilation. Separate compilation may reduce this runtime when precompiled approximate

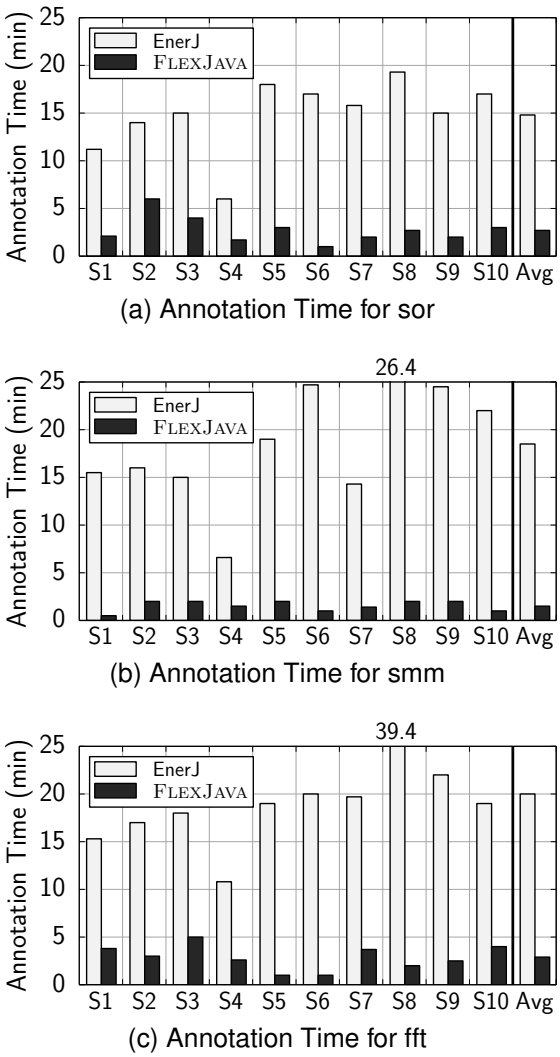


Figure 3.9: Annotation time with EnerJ and FlexJava for (a) sor, (b) smm, and (c) fft. The x-axes denotes the user study subjects.

libraries are available.

3.6.2 RQ2: Programmer Effort/Annotation Time

To answer RQ2, we conduct a user study involving ten programmers. The programmers are asked to annotate three programs with both languages. To avoid bias in our study toward FlexJava, we used three programs from the EnerJ benchmark suite [141]. The benchmarks are not large so that the subjects can understand their functionality before annotating them. As presented in Fig-

ure 3.9, we measure the annotation time with EnerJ and FlexJava and compare the results. The subjects are computer science graduate students who have prior background in Java programming but have no experience in approximate programming. We measured the annotation time using the following procedure.

First, we orally explain how to annotate the programs with FlexJava and EnerJ. Then, we demonstrate the annotation process on a simple benchmark, mc, and show the subjects how to use the tools for both languages. For this study, the subjects then annotate three of benchmarks, sor, smm, and fft, using both languages. Half of the subjects use EnerJ annotations first and the other half use FlexJava first. The measured time for EnerJ constitutes annotation plus compilation time. Whereas the measured time for FlexJava constitutes annotation time, plus the time for running the approximation safety analysis, plus the time for analyzing the analysis results using the source highlighting tool. We provide the unannotated application and a description of its algorithm for the subjects. We allow the subjects to review each application code prior to annotating it. Our current highlighting tool is enough to check whether or not the analyzed results are equivalent between the two languages.

Figure 3.9 shows the annotation time. On average the annotation time with FlexJava is 6×, 12×, 8× less than EnerJ for sor, smm, and fft, respectively. Although we demonstrate how the subjects can use the languages, they need time to gain experience while annotating the first program. Once the subjects acclimate to FlexJava with the first benchmark (sor), they spend proportionally less time annotating the next benchmark. The FlexJava annotation time for the second benchmark (smm) is typically lower than the first benchmark (sor). In contrast, the annotation time with EnerJ does not reduce beyond a certain point even after gaining experience. We believe that this is because EnerJ requires manually annotating all the approximate variable declarations and more. Using

FlexJava, sor and smm require three **loosen** annotation, but fft requires six. We believe that this explains why the time to annotate fft in FlexJava is greater than the time to annotate sor and smm. In summary, these results show that FlexJava significantly reduces programmer effort by providing intuitive language extensions and leveraging the automated approximation safety analysis.

3.6.3 RQ3: Energy Reduction and Speedup

To answer RQ3, we study energy gains and speedup of FlexJava with both fine- and coarse-grained approximation.

3.6.3.1 Fine-Grained Approximation

Tools and models. We modify and use the EnerJ open-source simulator [119] for error and energy measurements. The simulator provides the means to instrument Java programs based on the result of the analysis. It allows object creation and destruction in approximate memory space and approximating arithmetic and logic operations. The runtime simulator is a Java library that is invoked by the instrumentation. The simulator records memory-footprint and arithmetic-operation statistics while simultaneously injecting error to emulate approximate execution and measure error. The simulator uses the runtime statistics to estimate the amount of energy dissipated by the program. The error and energy measurements are based on the system models described in Table 3.1. The models and the simulator do not support performance measurements. We measured the error and energy usage of each application over ten runs and average the results.

Figure 3.10 shows the energy reduction and the output quality loss when the safe-to-approximate data and operations are approximated. These results match those of EnerJ [141]. As shown, the geometric mean of energy reduction

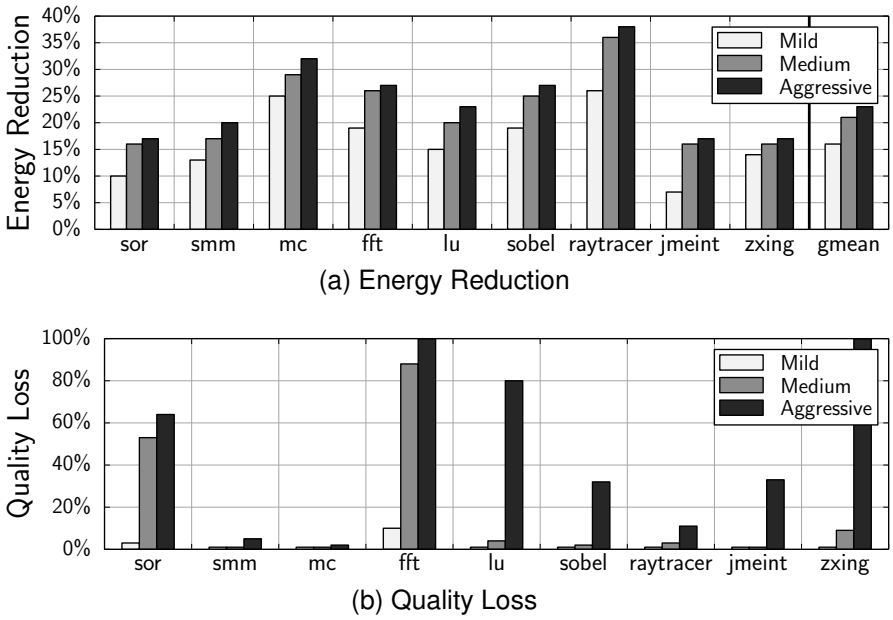


Figure 3.10: (a) Energy reduction and (b) quality loss when approximating all the safe-to-approximate data and operations.

ranges from 16% with the Mild hardware setting to 23% with the Aggressive hardware setting. The energy reduction is least for jmeint (7% with Mild) and highest for raytracer (38% with Aggressive). All the applications show low and acceptable output quality loss with the Mild setting. However, in most cases, there is a jump in quality degradation when the hardware setting is changed to Aggressive. If this level of quality is not acceptable (fft), then the application should dial down the hardware setting to Medium or Mild. FlexJava provides the same level of benefits and quality degradations as EnerJ while significantly reducing the number of manual annotations.

3.6.3.2 Coarse-Grained Approximation

To evaluate FlexJava's generality, we use the NPU coarse-grained approximation [128]. NPU can only be used to approximate the benchmarks fft, sobel, raytracer, and jmeint. Each benchmark has only one function that can be approximated with NPUs. Each of these functions can be delineated using a single

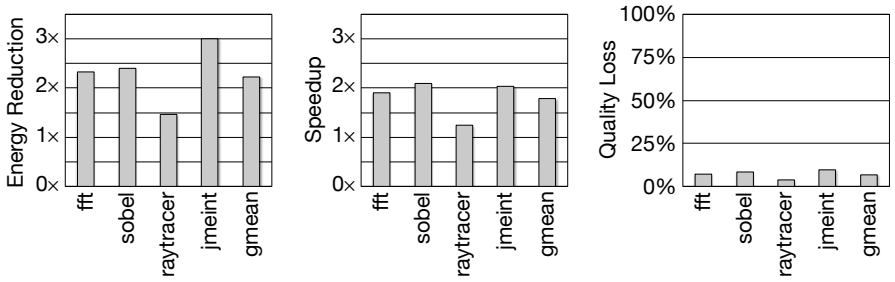


Figure 3.11: Speedup, energy reduction, and output quality loss when the approximate annotated functions using the NPU.

pair of **begin_loose–end_loose** annotation.

Tools and models. We measure the benefits of NPUs in conjunction with a modern Intel Nehalem (Core i7) processor. We use a source-to-source transformation that instruments the benchmarks' Java code to emit an event trace including memory accesses, branches, and arithmetic operations. This source-level instrumentation is unaffected by the JIT, garbage collection, or other VM-level systems. Using a trace-based simulator, we generate architectural event statistics. The architectural simulator includes a cache simulation. The simulation process outputs detailed statistics, including the cycle count, cache hit and miss counts, and the number of functional unit invocations. The trace-based CPU simulator is augmented with a cycle-accurate NPU simulator that also generates the statistics required for the NPU energy estimation. The resulting statistics are sent to a modified version of McPAT [142] to estimate the energy consumption of each execution. We model the energy consumption of an eight-processing-engine NPU using the results from CACTI 6.5 [143], McPAT [142], and [144].

Figure 3.11 shows the energy reduction, speedup, and quality loss with the NPU coarse-grained approximation. The baseline executes the precise version of the benchmark on the CPU without any NPU approximation. On average, the benchmarks see a 2.2× energy reduction and a 1.8× speedup. These benefits

come for less than 10% quality degradation across all the benchmarks, which is commensurate with other approximation techniques [120, 118, 122, 137] and prior NPU works [128, 6, 48]. The EnerJ system does not provide any coarse-grained approximation results for comparison.

These results demonstrate that coarse-grained approximation may have limited applicability but can provide higher benefits. Whereas, fine-grained approximation is more widely applicable with possibly lower gains. FlexJava supports both granularities as a general language to maximize opportunities for approximation in a wider range of applications.

3.7 Related Work

There is a growing body of work on language design, reasoning, analysis, transformations, and synthesis for approximate computing. These works can be characterized based on (1) static vs. dynamic, (2) approximation granularity, (3) automation, and (4) safety guarantees. To this end, FlexJava is a language accompanied with an automated static analysis that supports both fine- and coarse-grained approximation and provides formal safety guarantees. We discuss the related work with respect to these characteristics.

EnerJ [119] is an imperative programming language that statically infers the approximable operations from approximate type qualifiers on program variables. In EnerJ, all approximable variables must be explicitly annotated. EnerJ works at the granularity of instructions and provides safety but not quality guarantees. Rely [117] is another language that requires programmers to explicitly mark both variables and operations as approximate. Rely works at the granularity of instructions and symbolically verifies whether the quality requirements are satisfied for each function. To provide this guarantee, Rely requires the pro-

grammer to not only mark all variables and operations as approximate but also provide preconditions on the reliability and range of the data. Both EnerJ and Rely could be a backend for FlexJava when it automatically generates the approximate version of the program. Axilog [126] introduces a set of annotations for approximate hardware design in the Verilog hardware description language. Verilog does not support imperative programming constructs such as pointers, structured data, memory allocation, recursion, etc. The lack of these features results in fundamentally different semantics for safe approximation and annotation design.

Chisel [129] uses integer linear programming (ILP) formulation to optimize approximate computational kernels. A Chisel program consists of code written in an imperative language such as C and a kernel function written in Rely that will be optimized. Several works have focused on approximation at the granularity of functions or loops. Loop perforation [145, 116, 146] is an automated static technique that periodically skips loop iterations. Even though loop perforation provides statistical quality guarantees, the technique is not safe and perforated programs may crash. Green [130] provides a code-centric programming model for annotating loops for early termination and functions for approximate substitution. The programmer needs to provide the alternative implementation of the function. Green is also equipped with an online quality monitoring system that adjusts the level of approximation at runtime. Such runtime adjustments are feasible due to the coarse granularity of the approximation. FlexJava provides the necessary language extensions for supporting these coarse-grained approximation techniques as well as the fine-grained ones.

Similar to EnerJ, Uncertain<T> [147] is a type system for probabilistic programs that operate on uncertain data. It implements a Bayesian network semantics for computation on probabilistic data. Similarly, [148] uses Bayesian

networks and symbolic execution to verify probabilistic assertions.

3.8 Conclusion

Practical and automated programming models for approximation techniques are imperative to enabling their widespread applicability. This work described one such language model that leverages automated program analysis techniques for more effective approximate programming. The FlexJava language is designed to be intuitive and support essential aspects of modern software development: safety, modularity, generality, and scalability. We implemented FlexJava and its approximation safety analysis and evaluated its usability across different approximation techniques that deliver significant energy and performance benefits. The results suggest that FlexJava takes an effective and necessary step toward leveraging approximation in modern software development.

3.9 Hardware Description Lauguage (HDL) Support for Approximate Hardware Design

Similar to FlexJava's approach to provide modularity in approximate programming, we introduce Axilog to enable reusability in approximate hardware design for hardware description language. Axilog is a set of language annotations, that provides the necessary syntax and semantics for approximate hardware design and reuse in Verilog. Axilog enables the designer to relax the accuracy requirements in certain parts of the design, while keeping the critical parts strictly precise. Axilog is coupled with a approximation safety analysis that automatically infers the relaxable gates and connections from the designer's annotations. The analysis provides formal safety guarantees that approximation will

only affect the parts that the designer intended to approximate, referred to as relaxable elements. Finally, we describe a synthesis flow that approximates only the relaxable elements. Axilog enables applying approximation in the synthesis process while abstracting away the details of approximate synthesis from the designer. We evaluate Axilog, its analysis, and the synthesis flow using a diverse set of benchmark designs. The results show that the intuitive nature of the language extensions coupled with the automated analysis enables safe approximation of designs even with thousands of lines of code. Applying our approximate synthesis flow to these designs yields, on average, 54% energy savings and 1.9$\times$ area reduction with 10% output quality loss.

Chapter 4

UNDERSTANDING AND CONTROLLING QUALITY IN APPROXIMATE COMPUTING

4.1 Introduction

Power efficiency is a primary concern in modern systems. Battery capacity often limits mobile devices, while power consumption and cooling imposes budgetary constraints on data centers. Moreover, traditional CMOS scaling has slowed to a point that threatens the longstanding cadence of continuously improving performance [149, 150, 131]. Meanwhile, emerging workloads must manage ever-growing datasets with high responsiveness and availability to end users. Expert analyses show that in 2011, 1.8 zettabytes (1.8 trillion gigabytes) of information was created and replicated by all sources, with individual consumers responsible for 75% [151]. By 2020, the world's data centers will be responsible for managing 50$\times$ this staggering figure [151]. This level of demand for computing raises serious concerns about the capabilities of current computing systems to match emerging trends. Apropos these confluent challenges, a growing body of recent work seeks to exploit a common property of many emerging applications: tolerance to approximate computation. These techniques relax the traditional abstraction of full accuracy in data processing, storage, and retrieval, thus trading losses in output quality for improved performance and efficiency [118, 122, 120, 47, 6, 119, 4, 116, 117, 152, 153, 154].

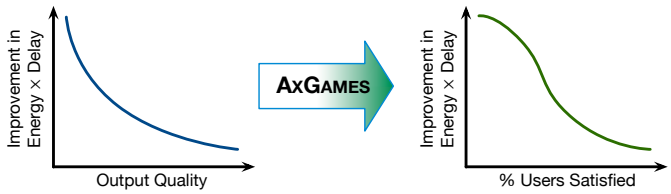


Figure 4.1: AxGames is a crowdsourcing solution that transforms the tradeoff between quality and energy-performance gains from approximation to the tradeoff between the gains and user satisfaction.

While these techniques provide promising gains, the effects of quality loss on the users are not well understood, leaving approximation techniques in a position of questionable utility. The challenge is determining the level of quality loss that the large majority of users deem acceptable. Discovering this level requires end users, who are not readily available during the development phase. Even after the application is deployed, frameworks that enable users to provide feedback on the quality loss are currently unavailable. To this end, we aim to develop a framework that methodically utilizes crowdsourcing to identify the desirable application output quality without exposing the details of approximation to the users. The objective is to aid the developers in identifying the acceptable level of quality loss and enable the crowd of users to directly help in determining this level. The crowdsourcing process needs to also be engaging and enjoyable enough to retain users. To address these challenges, we describe AxGames, a game-based crowdsourcing framework that statistically projects *user-driven* quality targets for approximate computing. As Figure 4.1 illustrates, AxGames changes the tradeoff between output quality and energy-performance gains to a tradeoff between the gains and the percentage of the users who are satisfied with the output.

AxGames comprises three web-based games that enable players to collectively identify the acceptable level of quality for the application in question. The games are designed to find a statistical consensus among the players on which

level of quality loss is acceptable. Finding the statistical consensus is imperative in ensuring that the majority of the application users will accept the quality loss caused by the approximation technique. The first game allows the users to express their perception about the quality of the approximated output without regard to the quality-cost tradeoff. The second game enables users to choose a level of quality while considering an abstract cost tradeoff. The third game adds an element of context by asking the players to answer a multiple-choice question about the approximated output. The users are given an incentive to select the lowest output quality that allows them to answer the question. All the three games involve betting, spending, losing, and winning virtual money. The virtual money is an abstract metaphor for compute resources (time, energy, storage) that need to be spent to achieve a higher quality output. The rewarding procedure in the games is designed to place players in competition with previous players. This strategy uses the overall group to act as a check mechanism for the feedback that is provided by the players. While the participants/users play, the games collect statistics about their choices.

We use the Clopper-Pearson exact method [155] to statistically project the acceptable level of accuracy based on the statistics collected by the games. These projections provide a statistical basis for the developers to decide which degree of approximation will provide a satisfactory experience for the users. Our analysis is impartial to the benefits of approximation and independent of the approximation technique that is utilized.

To evaluate our solution, we study seven applications that produce user perceptible outputs and cover a wide range of domains including image processing, optical character recognition, speech recognition, and audio processing. Humans are naturally tolerant to approximation; hence, many approximation techniques target these domains of applications. We recruit 700 partici-

pants/users through Amazon's Mechanical Turk to play the games. The study shows that level of acceptable quality changes significantly across applications. For instance, to satisfy 90% of users, the level of acceptable quality loss is 2% for one application and 26% for another. Moreover, the study shows that generally users have higher tolerance to approximation when exposed to the tradeoff between cost and quality. The users' tolerance is even higher when they consider a context. Moreover, the pattern with which the crowd responds to approximation takes significantly different shape and form depending on the class of applications. These results suggest the necessity of solutions that systematically explore the effect of approximation on the end user experience.

By introducing the AxGames framework, this work makes the following contributions:

1. **Crowdsourcing for approximate computing:** We develop a game-based crowdsourcing solution as an effective step towards enabling developers to systematically assess the effect of approximation from the user's perspective.
2. **Statistical inference:** We couple the crowdsourcing with statistical analysis to quantitatively translate raw data from the games to actionable results.
3. **Deployment:** Through deployment on Amazon's Mechanical Turk, we investigate the effectiveness of the proposed solution and show the necessity of the end user feedback by examining a diverse of real applications from different domains.

This study open a new axis, that of user experience, for the growing research in approximate computing. This work also sheds light on previously unexplored effects of approximation on the users. Moreover, it provides a development tool–rather unconventional–for the research community to better assess their innovative approximation techniques. Our tool is open source and is

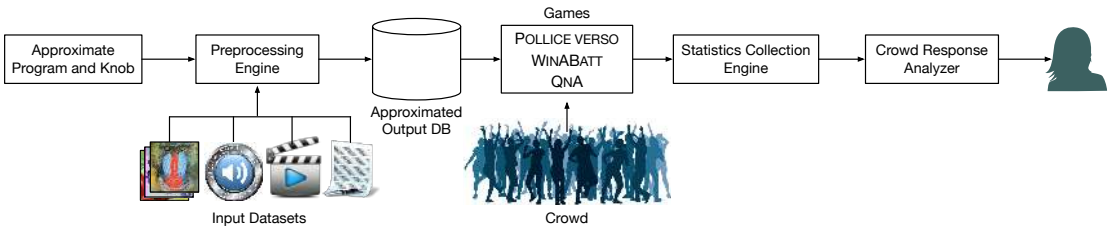


Figure 4.2: An overview of the AxGames crowdsourcing solution which determines the user-driven quality target for a given approximated application.

publicly available at http://act-lab.org/artifacts/axgames.

4.2 Overview

Figure 4.2 provides an illustration of AXGAMES's overall structure. AXGAMES is comprised of four major components: (1) approximated output database, (2) the three games namely POLLICE VERSO[1], WINABATT, and QNA; (3) the data collection engine; and (4) the crowd response analyzer. This section provides an overview of these components.

Approximated output database. The first step in using the AXGAMES solution is generating outputs of a given approximated application with varying degrees of quality loss. Note that AXGAMES is independent of the approximation technique and does not depend on how approximation is applied in the program. The developer provides a program which: (1) has an approximation knob to vary the degree of quality loss, and (2) can measure the quality loss for an approximated output[2]. For each input in the input dataset, the application is executed with different degrees of quality losses. A database records

[1] Wikipedia: "Pollice verso refers to the hand gesture or thumbs signal used by Ancient Roman crowds to pass judgment on a defeated gladiator."

[2] When developing an approximated program the developer needs to provide both the approximation knob and the quality measurement procedure [119, 47, 117, 116]. Therefore, in this regard, using AXGAMES does not require extra effort.

the approximated outputs, their degree of quality loss, and the setting of the approximation knob. We refer to this database as the approximated output database.

AxGames is designed to study a wide range of applications that generate outputs perceivable by humans through output devices such as a monitor or a speaker. Therefore, AxGames currently provides a large collection of images, audio, and text. This collection can be used by a wide variety of applications that span different domains to generate their own specific approximated output database. By playing the games, players collectively build a judgment regarding the acceptable quality for the collection of outputs. Additional information may be stored in the approximated output database. For instance, in the QnA game, players will need to answer one simple question for each approximated output. These questions are stored in the database as well. Section 4.3 discusses these questions and describes the three games in detail. Populating the approximated output database is performed offline to avoid unnecessary involvement of developers with the internals of gaming and crowdsourcing.

The three games. AxGames used the approximated output database as an input to its three different games. In all the games, a player is given an initial allowance of virtual money. The player's objective is to earn more money by guessing the statistical common ground among the previous players. In a sense, each player is playing with all of the past players and her guess affects the majority vote for the future players. As the crowd of gamers play the games, the players are *iteratively* converging to a statistical common ground. AxGames can then statistically infer the acceptable level of quality from the gamers' choices. Section 4.3 presents the details of the three games.

Statistics collection engine. As the users play, the games record the player choices and the game state in a database along with the player user IDs. AXGAMES uses this data to perform statistical projections about the percentage of users that deem a certain level of quality acceptable.

User response analyzer. After collecting the statistics from all the players, AXGAMES uses the Clopper-Pearson exact method [155] to calculate the binomial proportion confidence interval [156] for each level of quality loss. These intervals represent the percentage of users that deem a certain level of quality acceptable. Section 4.4 elaborates on the calculation and use of these intervals to recommend quality targets for the approximated applications.

4.3 The Three Games

AXGAMES includes three web-based games which aim to enable the crowd to iteratively converge to a statistical common ground. The players register with a unique user ID on the website to play the games without revealing personal information. Each user plays all three games independently and each game is played for 10 rounds. From the player's perspective, all three games revolve around betting, earning, spending, and losing virtual money. The score is the player's balance at the end of the game. We intentionally avoided exposing the direct relationship between virtual money and the computation cost to avoid biasing the gamers in choosing any level of quality. This relationship is a parameter in the games and can be exposed if desired. We also intended to make the games entertaining and enjoyable by using virtual money as the score and as a proxy for compute resources in two of the games. Our surveys show that 84% of the users were entertained when playing the games.

We devised the three games with different intuitions about inferring user-

driven acceptable level of quality through crowdsourcing. The first game, POLLICE VERSO, is a betting game. In each round, the player is presented with an approximated output and its corresponding precise version. The player is asked to guess whether or not the majority of other players thought that the approximated output is *good enough*. The player bets money on her guess and wins money back if the guess is correct[3]. This game aims to find a statistical common ground about the acceptability of an approximated output. However, POLLICE VERSO does not have any notion of tradeoff between quality and cost. To include the notion of cost, we designed WINABBATT which presents the player with a very low quality output and asks the player "How much would your spend to receive a better output?". The player can spend money to improve the output quality with a slider. The player wins money back depending on how close her choice of quality is to the previous players' selection. The objective of these two games is to find the statistical common ground while players judge the quality of the output in an abstract and context-insensitive manner. To provide some general context to the players, the third game QNA, presents the player with a very low quality output and asks the player a multiple choice question about that specific output. To answer the multiple-choice question, the player can improve the quality of the output by spending money with a slider. The player wins money back based on both the correctness of her answer and the closeness of her choice of quality to the previous players' selection. QNA gives incentive to the players to strike a balance between quality and cost while considering some context. The rest of this section describes these three games in further detail.

[3] POLLICE VERSO shares similarities with A/B testing [157]. However, A/B testing does not incorporate (1) games and betting (provided by all three games) (1) a sense of tradeoff (provided by WINABBATT) and (2) a sense of context (provided by QNA) to the users. As the statistical results show, users's tolerance to approximation increases when the tradeoff and/or context is added to the games.

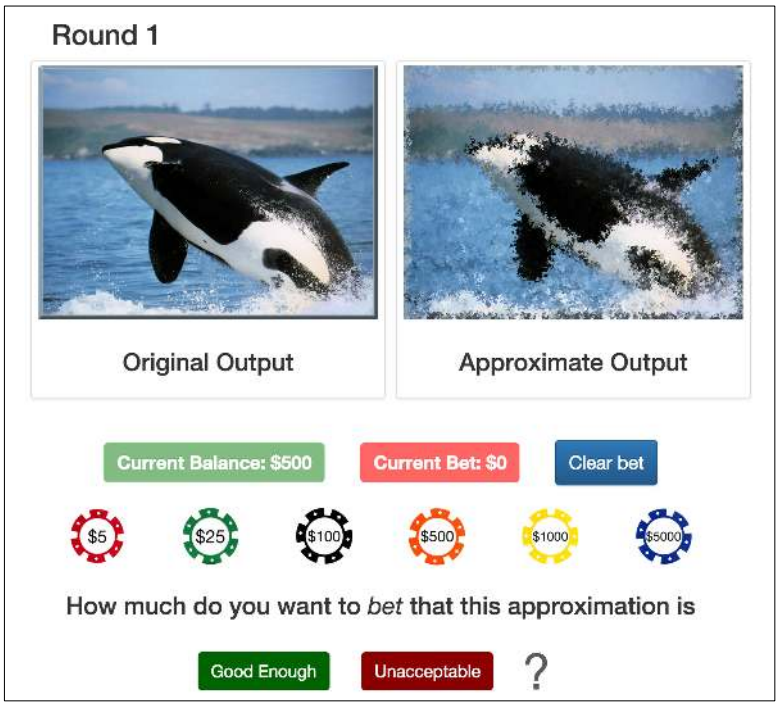


Figure 4.3: A round of the Pollice Verso game when deployed for an approximated implementation of the jpeg application.

4.3.1 Pollice Verso

As Figure 4.3 illustrates, POLLICE VERSO is a betting game that gives each player an initial allowance of $500 virtual money. By keeping track of players' bets, the game aims to infer the statistical common ground for the acceptable level of quality for a given application. The game randomly selects an approximated output o, and displays o along with its precise counterpart o^*. Internally, we represent each approximated output o with the following tuple:

$$(o, q, s, n, n_{GoodEnough}) \tag{4.1}$$

In Equation (4.1), q is the output quality; s is the setting of the approximation knob that led to this quality; n is the total number of past players that have played this particular output; and $n_{GoodEnough}$ is the number of players who thought the output is good enough. The last two parameters capture the history of the previous players' choices.

After displaying the outputs, a player is asked "How much do you want to **bet** that this approximation is Good Enough/Unacceptable?" Using the gaming chips shown in Figure 4.3, the player chooses to bet b amount of money on her answer. The player's choice c, is a binary decision.

Rewarding procedure. The player may win money or lose the bet depending on whether the past players agree with her choice. This rewarding strategy incentivizes the players to gradually come to a statistical common ground without directly interacting with each other. As Equation 4.2 shows, the winnings w, is a function of the player's choice c, the amount of bet b, and the past player's choices, captured by n and $n_{GoodEnough}$. Note that the values of n and $n_{GoodEnough}$ are updated after the player receives her reward. Therefore, the player's choice affects the winnings of future players.

$$w(c,b,n,n_{GoodEnough}) = b \cdot (reward(c,n,n_{GoodEnough}) - 1) \tag{4.2}$$

where

$$reward(c,n,n_{GoodEnough}) = \begin{cases} 2 \cdot f(c,n,n_{GoodEnough}) & \text{if } 0 \leq f \leq 0.5 \\ -7.8 \cdot f(c,n,n_{GoodEnough}) + 8.9 & \text{if } 0.5 < f \leq 1 \end{cases} \tag{4.3}$$

As Equation 4.2 shows, the player wins money proportional to the amount of bet b. The *reward* function (Equation 4.3 and Equation 4.4) defines this proportion based on whether or not the majority of previous players agree with the player's choice c. In Equation 4.3, the constants $(2,\ -7.8,\ 8.9)$ are picked such that the player loses all her bet in the worst case or quadruples her bet in the best case. Moreover, if the output is controversial, the loss is low and the gain is high. An output is controversial if $n_{GoodEnough}/n \text{approx} 0.5$. That is, almost half of the past players think the output is good enough and the other half thinks otherwise. In Equation 4.3, f captures the level of agreement between the

player's choice with the majority vote of the previous players as presented in Equation 4.4.

$$f(c,n,n_{GoodEnough}) = agreement(c,n,n_{GoodEnough}) = \begin{cases} n_{GoodEnough}/n & \text{if } c = Good\ Enough \\ 1 - n_{GoodEnough}/n & \text{if } c = Unacceptable \end{cases} \tag{4.4}$$

To enable players to make choices primarily based on their own perception, the reward function is hidden. Additionally, they play the game with no knowledge of the majority vote.

4.3.2 WinABatt

POLLICE VERSO enables users to perceptively judge the quality of an approximated output without regard to the tradeoff between quality and cost. To add this notion of tradeoff, we designed WINABATT as shown in Figure 4.4. The player starts with $100 of initial allowance and in each round, the game displays an approximated output at its lowest quality and asks the player: "How much would you spend to receive a **better** output?" The player is also given a slider with which she can adjust the output quality. The slider controls the quality and the cost associated with each quality level. Selecting a higher quality translates to spending more virtual money. Unlike POLLICE VERSO, the player's choices in WINABATT are no longer *yay/nay* binary decisions, and the player uses a continuous slider to choose a *level* of quality while considering its cost. If the game was naively designed, the player would always choose the lowest level of quality since it costs the least. However, in WINABATT, the player will be rewarded or penalized depending on how her choice of quality is close to the previous players. Hence, the player is also trying to guess the statistical common ground among the past players in a cost-conscious manner.

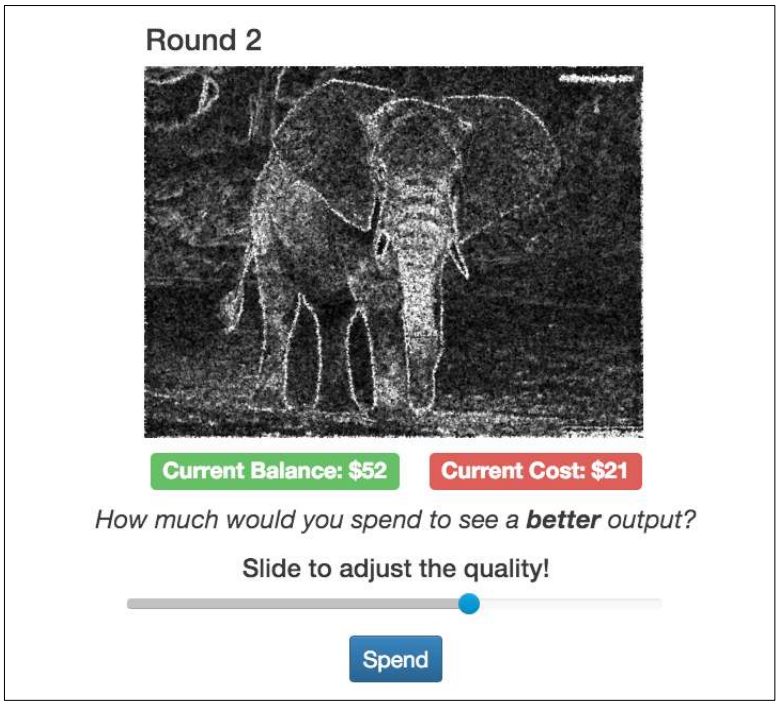


Figure 4.4: A round of the WinABatt game when deployed for an approximated implementation of the sobel application.

Rewarding procedure. To calculate the player's winnings and the statistical common ground, the game internally represents each output with the following tuple:

$$(\mathbb{O}, \mathbb{Q}_c, q_{MA}, n) \tag{4.5}$$

In Equation 4.5, $\mathbb{O}$ is the set of different approximated versions of an output; $\mathbb{Q}_c$ is the set of previous players' choice of quality, q_{MA} is the cumulative moving average of the past players' choice of quality; and n is the number of previous players that played the $\mathbb{O}$ set. The q_{MA} captures the statistical common ground among the past n players and is updated based on Equation 4.6 after the current player is rewarded.

$$q_{MA}^{(n+1)} = \frac{q_{MA}^{(n)} \cdot n + q_c}{n+1} \tag{4.6}$$

As shown in Equation 4.7, the player's winnings w, is a function of her choice of quality q_c and q_{MA}. As shown, the player's reward is deducted by her bet money b, which is the cost associated with her choice of quality q_c. This cost

function is linear to avoid bias towards any specific quality with \$5 for the lowest quality version (q_{min}) and \$30 for the highest quality version (q_{max}). The conditional part of Equation 4.7 is the reward that is determined by $f(q_c, q_{MA})$, which is presented in Equation 4.8 and captures how the player's choice of quality, q_c, is close to the choice of previous players' moving average, q_{MA}. The constants in Equation 4.7 is chosen such that the player's winnings, w, is between $-$\$35 and $+$\$35.

$$w(q_c, q_{MA}) = -b + \begin{cases} 5 \cdot f(q_c, q_{MA}) - 10 & \text{if } f \leq 10 \\ 40 = 5 \cdot 10 - 10 & \text{if } f > 10 \end{cases} \tag{4.7}$$

$$f(q_c, q_{MA}) = agreement(q_c, q_{MA}) = \left(\frac{|q_c - q_{MA}|}{q_{max} - q_{min}} \right)^{-1} \tag{4.8}$$

This rewarding procedure incentivizes the players to balance the cost and quality *while* guessing the past player's consensus.

4.3.3 QnA

While WINABATT provides an opportunity to the players to explore the tradeoff between quality and cost, they do so in an abstract and context-insensitive manner. To provide some context to the players, we design the QNA game. As Figure 4.5 illustrates, in each round, QNA displays an approximated output initially set to its lowest quality level, along with a slider, and a multiple-choice question about the output. The questions are in the form of "What can you find in this image? Sports car / SUV / Truck / Heavy equipment." The player needs to answer the question and can spend money to improve the quality using the slider. Similar to WINABATT, the initial allowance provided is \$100. In contrast to WINABATT, the slider cannot move backwards. This feature is to prevent the players from cheating, increasing the quality to answer the question, and then decrease the quality to minimize the cost. In other words, once the player

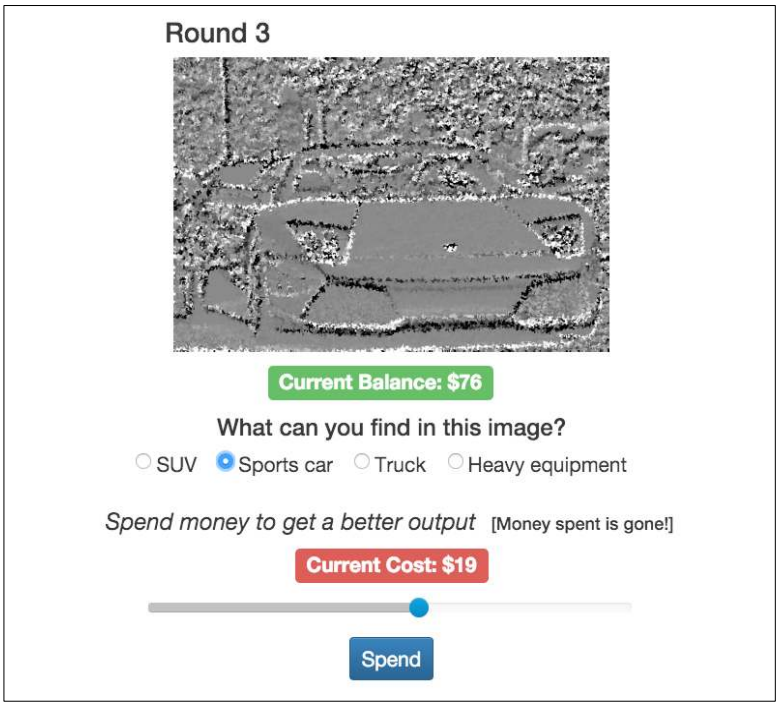


Figure 4.5: A round of the QnA game when deployed for an approximated implementation of the emboss application.

improves the quality, she cannot recover the cost of seeing the higher quality output.

Rewarding procedure. The winnings are calculated based on the rewarding procedure explained for WINABATT with the exception that the player also pays a $20 penalty for answering the question incorrectly. There is no extra reward for correct answers. The player wins money back depending on the correctness of her answer and the closeness of her choice of quality to the moving average of the previous players. QNA incentives the players to find a statistical common ground while balancing quality and cost with respect to some context about the output.

4.4 Statistical Analysis

As mentioned before, the games internally collect the player's choices and decisions for a series of outputs at different levels of quality. To enable the application developer to draw meaningful conclusions from this raw data, we devise a statistical framework that projects the user-driven quality target. Due to the large space of possible inputs and the diversity of users, it is practically infeasible to find a quality target that satisfies the entire population of the users for any arbitrary input. However, coupling the games with statistical analysis provides a pragmatic approach to determine the quality target that, with high confidence, satisfies the large majority of users.

4.4.1 Binomial Proportion Confidence Interval

We calculate the binomial proportion confidence interval [156] for each level of quality loss. Given the decisions of a *sample population* (players of the games), the binomial proportion confidence interval projects what percentage of the *statistical population* (all the users) are likely to deem a certain level of quality good enough (acceptable). After the players play the games, AxGames calculate this confidence interval for a range of quality losses which resulted from approximating the application-under-study. Based on the confidence interval, we can determine the level of quality loss that is highly likely to satisfy, for example, 90% of the statistical population[4] of the users.

AxGames leverages a commonly used method, the Clopper-Pearson exact method [155] to compute the binomial proportion confidence interval. We chose the Clopper-Pearson method as it has certain advantages over the other

[4]Statistical population is the entire pool from which a sample population is drawn. Here, the sample population are the gamers who are drawn from the entire pool of the application users. Thus, statistical population is the entire users.

available options [158, 159] such as, (1) higher accuracy as the number of samples becomes relatively large, and (2) it can calculate the confidence interval even when the opinion of the sample population is very skewed towards a decision. These features are important in our setup since the games provide a relatively large number of statistical samples, and the large majority of the players are highly likely to think that 1% quality loss is almost always acceptable and similarly a 50% quality loss is almost never good enough. Moreover, the Clopper-Pearson exact method calculates a conservative confidence interval that reduces the risk of being too aggressive when it comes to approximation. The binomial confidence interval is calculated based on a set of binary decisions from the sampled population. For example, in the case of the POLLICE VERSO game, a binary decision comes directly from the player's choice on whether or not an approximated output with the quality of q is good enough. Later in this section we describe how the WINABATT's and QNA's sliders are translated to binary decisions.

To calculate the confidence interval, we first need to calculate the sampled binomial proportion for each level of quality. The sampled binomial proportion is calculated for each approximated output by computing the fraction of votes that deem a level of quality good enough to the total number of votes. This sampled binomial proportion is calculated based on the $(n_{Votes}, n_{GoodEnough})^{(q)}$ pair, where n_{Votes} is the total number of decisions on outputs with the quality of q, and $n_{GoodEnough}$ is the number of decisions that deem these outputs good enough. This pair is calculated for each level of quality.

As Equation 4.9 shows, the Clopper-Pearson exact method computes the one-sided confidence interval of success rate, $\theta^{(q)}$, when the number of sample trials, n_{Votes}, and the number of successes among the trials, $n_{GoodEnough}$, are

measured for a sample of the population.

$$\frac{1}{1+\frac{n_{Votes}-n_{GoodEnough}+1}{n_{GoodEnough}\cdot \mathbf{F}[1-\alpha;2\cdot n_{GoodEnough},2\cdot(n_{Votes}-n_{GoodEnough}+1)]}} < \theta^{(q)} \tag{4.9}$$

In Equation 4.9, $\mathbf{F}$ is the F-critical value that is calculated based on the F-distribution [160]. The *discontinuous* nature of the binomial distribution precludes any interval with exact coverage for all values of n_{Votes} and $n_{GoodEnough}$ (all possible values of the binomial proportions). However, because of the relationship between the cumulative binomial distribution and the *continuous* F-distribution, we use the common alternative form of the binomial confidence interval that provides exact coverage for all population proportions. $\mathbf{F}[1-\alpha;d_1,d_2]$ is the $(1-\alpha)$ quintile of the F-distribution with d_1 and d_2 degrees of freedom. The $(1-\alpha)\cdot 100\%$ is degree of confidence on the interval. For instance, for 95% confidence interval, α is 0.05 and for 90% confidence interval, α is 0.10. The two degrees of freedom, d_1 and d_2, decide the shape of the F-distribution based on the collected statistics, n_{Votes} and $n_{GoodEnough}$ in our case.

To understand the meaning of $\theta^{(q)}$, we discuss an example deployment of the game that resulted in n_{Votes}=60 and $n_{GoodEnough}$=56 for quality level q=97%. From Equation 4.9, the lower limit of the 95% confidence interval, $\theta^{(97\%)}$, is 85.4%. That is, with 95% confidence, we can project that at least 85.4% of the users will deem 97% quality level acceptable. This projection is conservative because the Clopper-Pearson exact method calculates a conservative lower bound for the confidence interval. The degree of confidence is the probability of the projection being true. The projection based on 95% confidence interval is true with probability of 0.95.

For each level of quality, AXGAMES projects the fraction of user population (statistical population) that deems that level of quality acceptable. Using this

information, the developer can choose the level of quality that satisfies a target majority of users.

Translating a choice of quality to a set of binary decisions. Players in POLLICE VERSO make binary decision on the quality of an approximated output. These yay/nay decisions can be directly used in the Clopper-Pearson statistical analysis. In contrast, the players in WINABATT and QNA, choose a level of quality using a slider. To be able to use the Clopper-Pearson statistical analysis, each chosen level of quality needs to be translated to a series of binary decisions. Intuitively, when a player chooses the quality level of q_c to be good enough, she implies that any level of quality higher than q_c is also good enough. Because of the rewarding procedure, the player has the incentive to choose the lowest acceptable quality. Choosing higher quality translates to a higher cost. The choice of player also implies that lower quality outputs are not good enough, otherwise, she would have chosen a lower quality to pay a lower cost. Based on this intuition, we convert one chosen level of quality, q_c, which is in the range of (q_{min}, q_{max}) to $q_{max} - q_{low} + 1$ binary decisions (Equation 4.10).

$$decision(q) = \begin{cases} \text{“}GoodEnough\text{”} & \text{if } q \geq q_c \\ \text{“}Unacceptable\text{”} & \text{if } q < q_c \end{cases} \quad \forall q \in (q_{min}, q_{max}) \tag{4.10}$$

By using this conversion, the same method of statistical projection can be applied to all three games. Moreover, WINABATT and QNA provide larger number of decisions per round. We did not use this conversion in POLLICE VERSO to enlarge the number of decisions since the players do not have an opportunity to see a range of quality losses for each output. Whereas, WINABATT and QNA allows the players to see the same output at different quality levels using the slider.

4.5 Statistical Quality Guarantees for Approximate Acceleration

While the AXGAMES solution provides a way for the approximate application developers to understand the quality requirements of application users, controlling approximation techniques to meet the quality targets is still a remained challenge. Conventionally, an approximate accelerator replaces every invocation of a frequently executed region of code without considering the final quality degradation. However, there is a vast decision space in which each invocation can either be delegated to the accelerator—improving performance and efficiency–or run on the precise core—maintaining quality. To this end, as a follow-up work of AXGAMES, we propose MITHRA [7], a co-designed hardware-software solution, that navigates these tradeoffs to deliver high performance and efficiency while lowering the final quality loss. MITHRA seeks to identify whether each individual accelerator invocation will lead to an undesirable quality loss and, if so, directs the processor to run the original precise code.

This identification is cast as a binary classification task that requires a cohesive co-design of hardware and software. The hardware component performs the classification at runtime and exposes a knob to the software mechanism to control quality tradeoffs. The software tunes this knob by solving a *statistical optimization problem* that maximizes benefits from approximation while providing statistical guarantees that final quality level will be met with high confidence. The software uses this knob to tune and train the hardware classifiers. We devise two distinct hardware classifiers, one table-based and one neural network based. To understand the efficacy of these mechanisms, we compare them with an ideal, but infeasible design, *the oracle*. Results show that, with 95% confidence the table-based design can restrict the final output quality loss to 5%

Table 4.1: Applications and their quality metric.

	Description	Domain	Quality Metric
emboss	Embossing filter	Image Processing	Normalized Root Mean Square Error (NRMSE)
jpeg	Lossy compression		
mean	Blurring filter		
sobel	Edge detection		
audio-enc	Audiio encoder	Audio Processing	
ocr	Optial character recognition	Pattern Recognition	Text Similarity Ratio
speech2txt	Speech to text		

for 90% of unseen input sets while providing 2.5× speedup and 2.6× energy efficiency. The neural design shows similar speedup however, improves the efficiency by 13%. Compared to the table-based design, the oracle improves speedup by 26% and efficiency by 36%. These results show that MITHRA performs within a close range of the oracle and can effectively navigate the quality tradeoffs in approximate acceleration.

4.6 Evaluation

To evaluate the effectiveness of the AXGAMES crowdsourcing framework, we deploy the three games on the web for seven different applications. We use Amazon's Mechanical Turk to recruit a large number of users to play the games. Using the collected data through the games and our statistical analysis, we measure what level of quality is acceptable for the majority of users. We also study how the acceptable level of quality varies across different applications and how the statistical analysis effectively captures these trends.

4.6.1 Methodology

Applications. x As Table 4.1 shows, we examine AXGAMES using a wide range of applications from diverse domains that include image processing, audio processing, optical character recognition, and speech to text conversion.

AXGAMES is not limited to these applications and can be used with other applications that produce outputs perceptible by humans. As shown in Table 4.1, our set of programs includes four image processing applications. The emboss application is an image filter that replaces each pixel either by a highlight or a shadow. Applying this filter to an image usually results in an image resembling a paper or metal embossing of the original image. The jpeg application implements the JPEG image compression algorithm. The sobel application is an edge detection algorithm which employs the Sobel operator. The mean is a sliding-window spatial filter that replaces the center pixel with the average (mean) of the pixel values in the window and blurs the image. Additionally, we also evaluate audio-enc, an audio compression engine that compresses WAV files and transforms them into MP3 files [161] . The quality metric for the image processing and audio compression applications is the normalized root mean square error (NRMSE) which is calculated by comparing an approximated output with its precise counter part.

We also use two applications that recognize text and speech. The ocr application is an optical character recognition program that converts raster images of written text to characters. The speech2txt application is a speech recognition engine that converts speech audio files to text [162]. The quality metric for these two applications that produce text output is the text similarity ratio that is computed by comparing the original application output with the approximated application output. We use an open-source implementation of the Gestalt Pattern Matching algorithm [163] to measure the text similarity ratio. This algorithm assigns higher scores to the outputs that *look right* to a human reader. The set of applications have often been used to evaluate the benefits of approximation techniques in the approximate computing literature [117, 47, 164, 118, 10, 122].

Approximation techniques. For the image processing applications, we use a

variation of loop perforation [116] as the approximation technique. This technique skips computing some of the pixels and instead, copies the values from neighboring pixels. The rate at which the computation is skipped is the knob for controlling the quality. We refer to this technique as tiling. We chose tiling for image processing applications since it is a simple yet effective coarse-grained approximation technique. For the audio compression and pattern recognition applications, we use the same model used in previous fine-grained approximation works [119, 4, 117, 129, 120] that adds stochastic noise to the computation by leveraging voltage overscaling, bit-width reduction, and reducing the DRAM refresh rate. The rate and the magnitude of noise are the knobs for controlling approximation. We refer to this technique as fine-grained approximation. We use tiling and fine-grained approximation to examine AXGAMES, since they represent the two main categories of approximation techniques, coarse-grained and fine-grained, respectively. AXGAMES is general and is not limited to these techniques. This flexibility is inherent in the framework since it only needs a set of approximated outputs with varying degrees of quality loss. Naturally, if the approximation technique changes, the games need to be redeployed in order to understand the user response to the new technique.

Datasets. As part of the AXGAMES framework, we include the input dataset that contains 200 images, 200 audio files, 200 speech files, and 200 printed text files. We collect this dataset from open-source databases or public data archives such as ImageNet [165], Freesound [166], VoxForge [167], and New York Times TimeMachine [168]. The applications can use this collection to generate the approximated output database which is used in the three games. AXGAMES also assigns one multiple-choice question to each data element. These questions are used during the QNA game to provide some context to the gamers when they trade quality for cost. The questions are part of the frame-

work and do not need to be regenerated for other applications that produce similar outputs. Additionally, the approximated output database is generated offline by running the application-under-study with different settings of the approximation knob. The AXGAMES's requirement is to generate each output with varying degrees of quality loss. For our experiments, we generate 51 versions of each output, with quality loss ranging from 0% to 50%, and a step size of 1%.

Game deployment. We separately deploy the three games on the web for each of the seven applications. In each round, each game randomly selects an output from the approximated output database. In the POLLICE VERSO game, the players bet and vote on whether the quality of the approximated output is good enough. Thus, the game randomly picks a quality loss for the displayed output. The following ten quality losses are used for the selected approximated output: 1%, 3%, 5%, 7%, 10%, 15%, 20%, 30%, 40%, and 50%. The other two games use all 51 quality levels which are generated for each approximated output and allow the player to pick any of them. We have made AXGAMES open source in our artifact portal (http://act-lab.org/artifacts/axgames). The deployed games, which are used for evaluations, are also available through the same portal.

Crowdsourcing through Amazon's Mechanical Turk. To collect a reasonable amount of statistics from a diverse set of users, we leverage a crowdsourcing Internet marketplace, Amazon's Mechanical Turk [169]. The games for each application is played by 100 individual Mechanical Turk workers (turkers)[5]. For the seven applications, 700 turkers contributed to our study. Each turker plays the games once. We do not allow a worker to play the games more than once to avoid bias in statistics from a few heavy gamers. Each game has 10 rounds,

[5]We received Institutional Review Board (IRB) approval before deploying AXGAMES. The request was approved in three weeks.

our 100 turkers make a total of 1,000 binary decisions in POLLICE VERSO for each individual application. Therefore, each of the 10 discrete levels of quality in POLLICE VERSO receives 100 votes for each application. As discussed in Section 4.4, in each round of WINABATT and QNA, a player's choice of quality on the slider is translated to 51 binary decisions, which correspond to one of the 51 quality levels of an approximated output. Hence, for each of these two games the turkers make a total of 51,000 (= 100 players $\times$ 10 rounds $\times$ 51 binary decisions per round) binary decisions for each individual application . Thus, each of the 51 levels of quality receive 1000 votes in WINABATT and QNA for each application. This amount of information provides the grounds for making high confidence statistical projections on the acceptable level of quality for each application. We also conduct an optional survey at the end of the games to understand whether or not the games were entertaining. The survey results show that 84% of players were entertained.

Game initialization. The results are reported when the games are deployed with random initial votes for 10 imaginary gamers on each output. These random initial votes are not used in the statistical analysis. To understand the effect of initialization, in a separate experiment, we also asked 10 graduate students to play the games without regard to the rewards. We then extrapolated their choices across all the outputs with the same quality. We observed that both initialization strategies yield similar trends and therefore, we report the results with random initial values. This similarity is the result of recruiting large number of turkers and the fact that the human turkers ultimately make decisions based on their own perception of the approximated outputs.

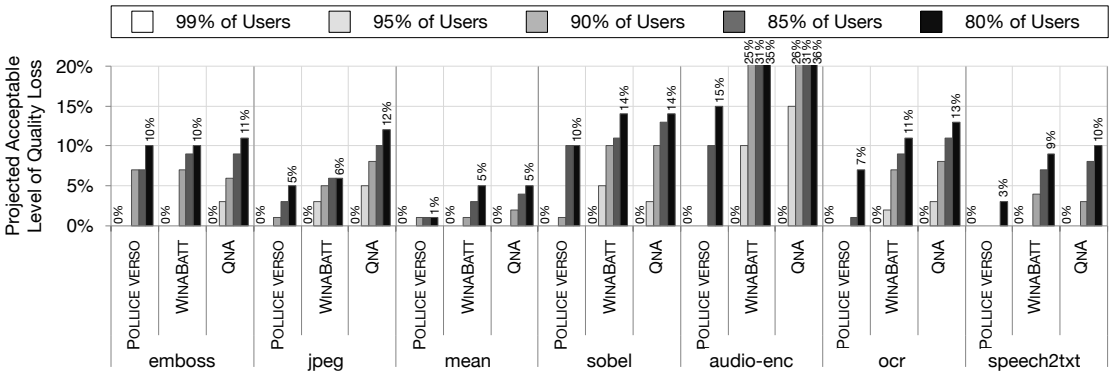


Figure 4.6: Projected acceptable level of quality loss with 95% confidence. These levels of quality are projected by our statistical analysis based on the game plays of 100 Mechanical Turk workers. Starting from left, each bar corresponds to the level that is projected to satisfy 99%, 95%, 90%, 80%, and 80% of the applications' users. These projections vary significantly across the applications.

4.6.2 Statistical Projections

Figure 4.6 shows the projected acceptable level of quality for each approximated application. The confidence level of these projections is 95%. As shown, each pair of (application, game) yields a set of projections. Starting from left, each bar corresponds to the level that satisfies 99%, 95%, 90%, 85%, and 80% of statistical population of the application users. The details of these projections are provided in Figure 4.7 and are discussed later in the section. As shown in Figure 4.6, for all the pairs of (application, game), the projected level of quality loss that satisfies more than 99% of the users is 0%. That is, for developers who aim to satisfy 99% of their users, the specific approximation techniques that are used in our evaluations (tiling and fine-grained approximation) are not a viable option. However, when the target is to satisfy a large majority of the users, starting from 90% of the users, there are opportunities to utilize these approximation techniques across all the image processing applications. Based on the statistics from POLLICE VERSO, if a developer chooses to satisfy 90% of the users, emboss can utilize tiling with 7% quality loss while jpeg, mean, and sobel

can utilize tiling with 1% quality loss. Based on POLLICE VERSO, audio-enc and ocr can be only approximated if target is to satisfy 85% or less percentage of the users. For speech2txt, only if the target is to satisfy 80% or less percentage of the users, approximation can be enabled given the statistics from POLLICE VERSO.

As Figure 4.6 depicts, based on the statistics from QNA, all the applications can be approximated if the target is to satisfy 90% of the users. For this target, the acceptable level of quality loss is 6% for emboss, 8% for jpeg, 2% for mean, 10% for sobel, 26% for audio-enc, 8% for ocr, and 3% for speech2txt. As these results show, there is a clear difference between the three games when they assess the user satisfaction even for one application. This difference emanates from the fact that WINABBATT and QNA provide an opportunity for the users to consider tradeoff and context, while POLLICE VERSO does not. We will discuss these differences in more detail later in this section.

User response varies significantly across applications. Another observation from these results is that the user-driven level of acceptable quality varies significantly across applications. Consider the four image processing applications that use the same approximation technique, tiling. Users show relatively higher tolerance to the tiling approximation for two of the applications, emboss and sobel. However, for other applications, especially for mean, the users show significantly lower tolerance. This significant variation in user response to the same approximation technique across different applications shows the necessity of solutions that statistically evaluate the acceptable level of quality. AXGAMES is one such solution, that effectively enables the users to provide statistical feedback to the developers who intend to leverage approximation techniques. With QNA, 90% of the users only accept 2% quality loss for mean

while they tolerate 26% for audio-enc. These results shed light on the significant variation of users response to quality loss from approximation for different applications. To this end, AxGames provides the grounds for researchers to statistically evaluate their innovations from the users' perspective.

Users show higher tolerance to approximation when they consider cost or context. From the results in Figure 4.6, it is evident that users show higher level of tolerance to the approximation when playing WinABatt and QnA. Whereas the level of tolerance to approximation is lower in Pollice verso. Intuitively, in these two games the players choose a level of acceptable quality in a cost-conscious manner while also considering the context in the case of QnA. In the case of Pollice verso, the players vote only based on their personal impression of the approximated output without a chance to explore the cost-quality tradeoff. On the other hand, the other two games, WinABatt and QnA, introduce a notion of tradeoff between cost and benefits of approximation. Moreover, the players' choice of quality in QnA is driven by their ability to answer the associated multiple choice question with the output. This analysis shows that these three games collectively provide a deeper understanding of the user' reaction to the approximation technique. The developer may choose to use the results from any of these games on her own discretion depending on her constraints on user experience and her target deployment environment.

4.6.3 Collected Statistics from the Games

The collected statistics from the games are presented in Figure 4.7. The bars represent the fraction of players who chose "Good Enough" for each level of quality loss. Our analysis uses the Clopper-Pearson method to computes the one-sided confidence interval for each level of quality loss. This one-sided

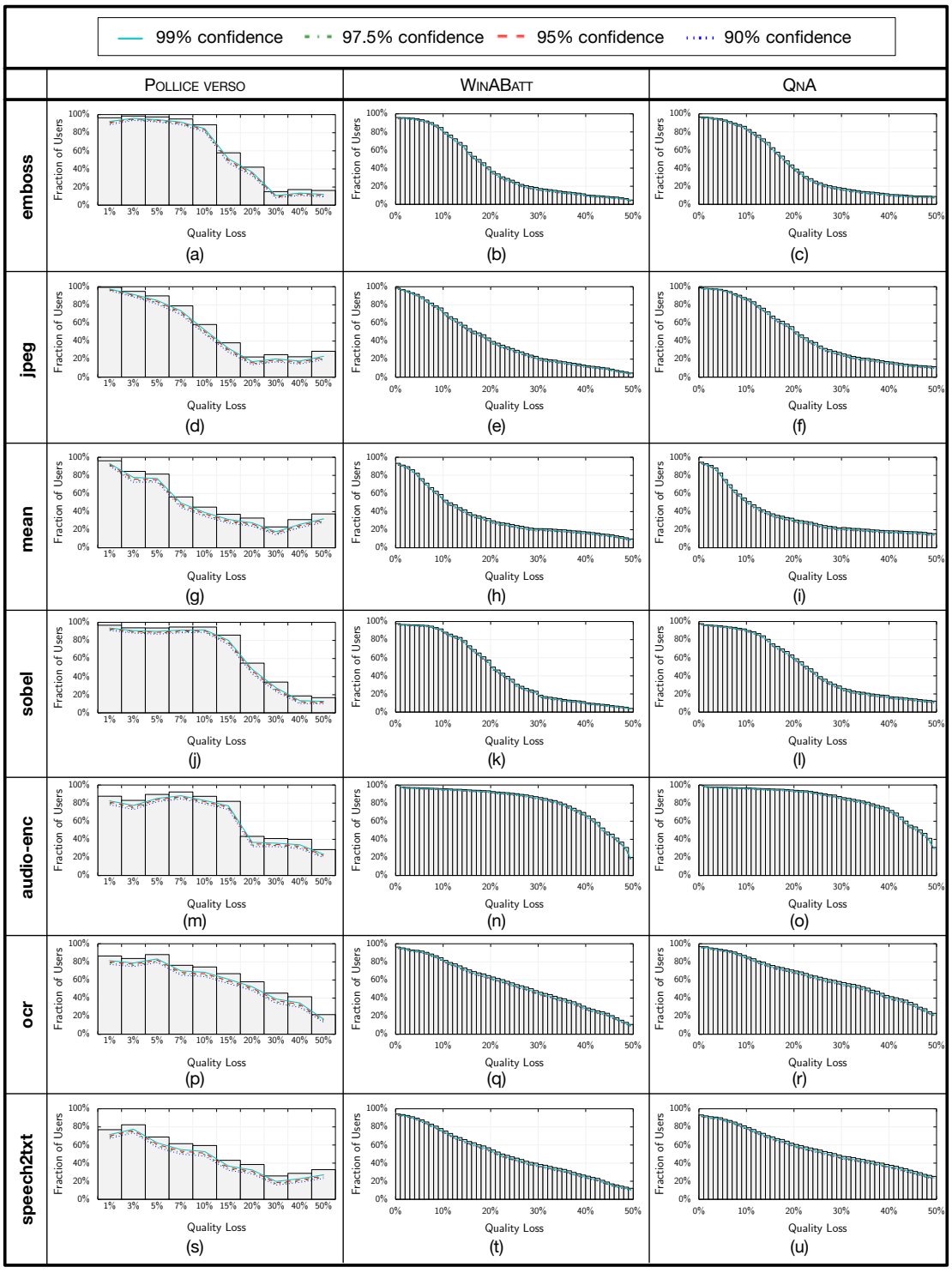
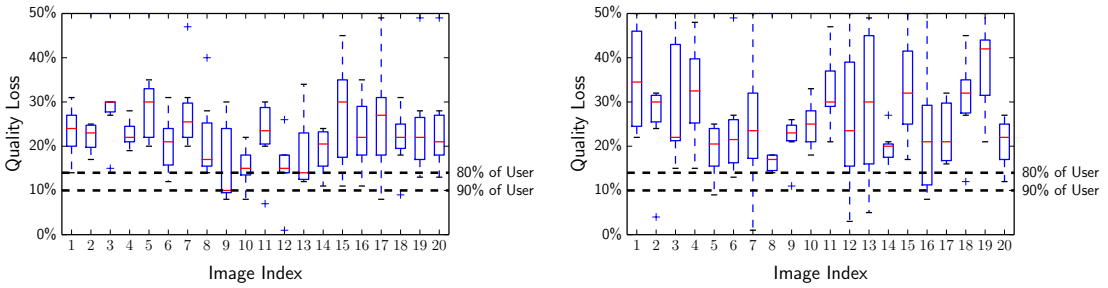


Figure 4.7: The collected statistics and the statistical projections. The bars show the fraction of players that selected a certain level of quality loss as "Good Enough." The lines represent the lower bound of the binomial confidence interval with different degrees of confidence, including 99%, 97.5%, 95%, and 90%.

lower-bound conservatively projects what fraction of the users will deem a particular level of quality loss as "Good Enough." The lines in Figure 4.7 represent the statistical projections with different levels of confidence, including 99%, 97.5%, 95%, and 90%. As shown, the projection lines fall below the collected statistics from the sampled population (the turkers who played the games). That is because the Clopper-Pearson confidence interval, by definition, covers the sampled statistics in a conservative manner. A point on a projection line with the (x, y) coordinates predicts that *at least* $y\%$ of the users will deem $x\%$ quality loss as "Good Enough." For instance, suppose that we want to use the statistical results from WINABBATT to find an acceptable level of quality for emboss. Let's target to find the quality loss level that provides satisfactory experience at least for 90% of the users with 95% confidence level. The quality loss is 7%, which is the x coordinate of the intersection point between the $y =$90% line and the red dashed projection line in Figure 4.7(b). The results in Figure 4.6 are obtained in this manner and also summarizes the statistical data given in Figure 4.7. As depicted, the results for POLLICE VERSO do not decrease monotonically whereas the results for WINABBATT and QNA do. That is because only for these two game, the statistical analysis converts the players' selected level to binary decisions for all the levels of quality loss (see Section 4.4).

From the results in Figure 4.7, we observe different patterns for the different classes of applications. For the image processing applications, the graphs have three regions, the top region where the majority of users agree on the low quality loss; the middle region where there is no clear consensus among the users; and the tail where the majority of users reject low quality outputs The quality level from which the middle region starts captures the point that the majority's opinion is shifting. This point of shift in opinion can be found by calculating changes in the derivative of the projection lines. This point of shift may be used



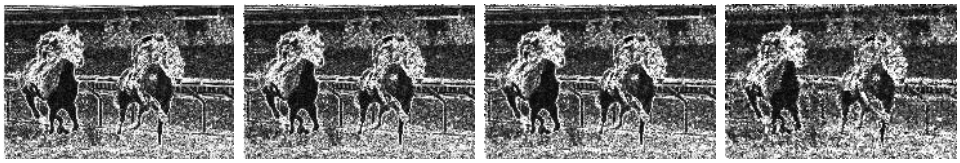
(a) Distribution of the quality choices in WINABATT

(b) Distribution of the quality choices in QNA

Figure 4.8: The box plot distribution of the players' choices of quality for sobel. The dashed horizontal lines show the projected acceptable level of quality loss that satisfies 80% and 90% of the users with 95% confidence (a) based on WinABatt and (b) QnA.

by the developers to choose the acceptable level of quality. AXGAMES provides the opportunity to find this point of shift for the developers. Instead of choosing the level of quality loss that certain percentage of users prefer, a developer can optimistically choose this point of shift in opinion.

The audio-enc application, which generates auditory output, shows a different pattern. The gamers show a significantly higher tolerance to the quality loss in the audio outputs. The majority of users tolerate the quality loss up to a significantly high level, after which the user satisfaction drastically declines. For the pattern recognition applications, ocr and speech2txt, the graphs show that the fraction of satisfied users almost linearly decreases as the quality loss increases. These two applications generate textual output. Unlike the other applications, there is not a clear point of shift in the crowd's opinion about the quality loss. These difference are significant and depend on the inherent characteristics of application, its output format, and the applied approximation techniques. Our experimental results show that the AXGAMES framework can effectively capture such patterns that need to be taken into account when approximation is employed.



(a) Precise output (b) Output with 10% quality loss. (c) Output with 14% quality loss. (d) Output with 30% quality loss.

Question: *What can you see in the image?* **Correct answer**: *Horse racing*
Wrong answers: *Bowling, Bridge, Buffalo wings*

Figure 4.9: Outputs from the edge detection filter, sobel, for image 13 in Figure 4.8b. The leftmost output is the precise version and the rest are the approximated outputs. The approximated outputs (b) and (c) have 10% and 14% of quality loss that satisfy 90% and 80% of the users, respectively. The output (d) has 30% of quality loss, which is the median of the votes for the image 13 from the QnA plays.

4.6.4 User Response Variations

We investigate the user response variation across the same approximated images in Figure 4.8. Figure 4.8 shows the distribution of the responses from the players for a randomly selected subset of the output images from sobel. The trends are similar for the other applications. This distribution is shown as a box plot. The bottom and top of each box represent the lower and upper quartiles, respectively, and the band near the middle of the box is the 50^{th} percentile (the median). The bottom whisker represents the lowest datum that is still within 1.5 inter quartile range (IQR) of the lower quartile. The top whisker denotes the highest datum that is still within 1.5 IQR of the upper quartile. The dashed horizontal lines show the projected acceptable levels of quality loss that satisfy 80% and 90% of the users with 95% confidence. It is even visually evident that these levels of quality cover their corresponding majority of users.

As expected, there is a large variation in the players choices. That is each player is providing her own personal judgment on the quality of the output, which is one of the main objectives of the games. Interestingly, the variation is higher in the QNA game. We investigate this higher variation in Figure 4.9

by showing the output image with the highest variation in Figure 4.8b, image 13. The associated question with this image is “What can you see in the image: Horse racing/Bowling/Bridge/Buffalo wings?”. This question can be answered even with high quality loss (30%) as shown Figure 4.9. However, it is understandable if a player chooses a lower quality loss to answer the question. Qualitatively, the context provided by the question allows a fraction of the users to choose levels of quality loss that are relatively higher. We speculate that approximation can be applied more aggressively if the output of this image processing is fed to a machine learning algorithm that performs scene detection or object recognition.

4.6.5 Changing the Tradeoff for Approximate Computing

The AxGames framework enables programmers to *change* the tradeoff between quality and performance-energy gains into the tradeoff between the users' satisfaction and the gains. We demonstrate this ability by measuring the speedup and energy savings for the image processing applications. A full investigating of the benefits of approximation is out of the scope of this work. This study is not to advocate approximation or show how much gains are possible; rather, it is to investigate the users' perspective on the output quality loss. For the performance and energy modeling, we use the same setup as the one used in [6]. We use the MARSSx86 x86-64 cycle-accurate simulator [170] and McPAT [142] for timing and energy modeling, respectively. The processor is modeled after a single-core Intel Nehalem (3.4 GHz with 0.9 V at 45 nm). The use statistics are based on QNA.

Figure 4.10 shows the improvement in energy-delay product when the output quality and the fraction of satisfied users change from 80% to 100%. The baseline is the application running on the processor without any approximation.

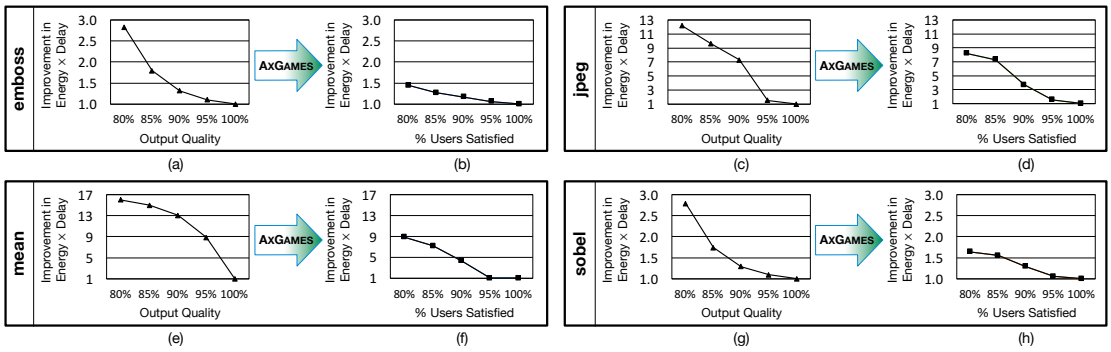


Figure 4.10: Improvement in energy-delay products vs. output quality (a, c, e, and g), and vs. % users satisfied. (b, d, f, and h). The results in (b), (d), (f), and (h) are based on the statistics collected from the QnA plays.

Figure 4.10(a, c, e, and g) represent the tradeoff between quality and the gains from approximation while Figure 4.10(b, d, f, and h) represent the tradeoff between the user satisfaction and the gains. All applications see a disparity in the energy-delay product improvement for the same level of quality loss and fraction of satisfied users. For instance, in Figure 4.10(e), mean sees 8.9× energy-delay product improvement with 95% quality. Even though this level of quality seems high, it only satisfies about 80% of users. For mean, as shown in Figure 4.10(f), to satisfy 95% of users, only 1.3× energy-delay product improvement can be achieved, which is significantly lower than the gains that can be achieved with the 95% quality. Although the results in Figure 4.10 are specific to the pair of (application, approximation technique), they show a clear change in the tradeoffs when user response is considered.

While many approximate techniques provide significant benefits, their utility cannot be established without understanding the users' perspective. Our framework and our analysis are impartial to approximation techniques and are intended to shed light on how application users react to approximation. The games and the statistical analysis provide an effective mean for developers to understand the user experience before employing approximation. In fact, the

developers can explore the tradeoffs and the benefits in a new light that considers users' perspective.

4.6.6 Discussion

Currently, AxGames is implemented for applications that directly produce human-perceivable outputs, such as image processing, audio processing, and pattern recognition applications. These programs represent a large body of applications that are designed for humans as the end users. These applications can benefit from approximation due to the inherent tolerance of humans to inexact results. In fact, many of the approximate computation research includes such applications [119, 117, 129, 116, 47, 6, 118, 122, 10]. Furthermore, these classes of applications, such as Instagram, Microsoft HoloLens, Qualcomm Vuforia, are gaining prominence as the computing services, more and more, aim to provide natural and targeted experiences for the end users. This trend has been amplified by prevalence of mobile devices and will grow in importance as wearable electronics is gaining traction and smart and interactive home/office environments are emerging. Moreover, even complex machine learning algorithms are being deployed in interactive data visualization [171] and analytics tools [172] that allow humans to interact with complex and large amounts of data. Besides the direct use of AxGames in these domains, the results of AxGames can be used as an upper bound for inaccuracy in cases where the approximate output is fed to a machine learning engine. Humans are the ultimate recognition engines.

As an end-to-end attempt toward crowdsourcing the target quality determination, our approach takes an initial and effective step toward enabling the end users to become a helping force in the development and deployment of approximation techniques.

4.7 Related Work

There has been a substantial amount of effort to leverage crowdsourcing tools and games to solve computationally difficult problems. However, there is a lack of solutions that leverage crowdsourcing or game design for approximate computing which statistically determine the level of quality considered acceptable by the majority of application users. We provide one such solution and lies at the intersection of (a) approximate computing, (b) crowdsourcing, and (c) game with a purpose.

Approximate computing. A growing body of recent work explores a variety of approximation techniques. These techniques include (a) approximate storage designs [173, 174] which trade data quality for reduced energy [173] and larger capacity [174], (b) voltage over-scaling [120, 175, 125, 176, 121, 117, 129], (c) computation and data sampling [116, 145, 177, 152], (d) loop early termination [130], (e) computation substitution [118, 130, 178, 179], (f) memoization [180, 122, 181], (g) limited fault recovery [124, 182, 183, 184, 145, 185, 186], (h) precision scaling [119, 127], (i) approximate circuit synthesis [187, 188, 189, 10, 190, 191, 192], and (j) neural acceleration [47, 6, 164]. Many of these solutions include applications which produce perceptible outputs for users. Although these techniques report promising benefits from approximate computing, they do not explore the acceptability of the quality loss from the users' perspective.

Crowdsourcing. The reCAPTCHA [193] project is a successful crowdsourcing application in production. This system uses the CAPTCHA, human Turing test, to classify text images from scanned books when other techniques fail. Automan [194] is an automatic crowd programming system which enables the integration of human computation in conventional programming languages

and rigorously studies scheduling, budgeting, and statistical quality control in programming with human computational resources. TurKit [195] provides a programming model to integrate human computation in JavaScript using templates which provide close integration with Mechanical Turk. TurKit also supports checkpointing and recovery. Russell et al. [196] developed a web-based tool that leverages crowdsourcing to identify objects and locations in images and annotate them. These research efforts are not concerned with quality loss in approximate computing.

Games with a purpose. Games with a purpose provide an opportunity to engage the crowd in entertaining applications while providing insight to researchers. Several prior works have successfully utilized game-based crowdsourcing to label random images and locate objects [197, 198, 199]. Furthermore, Dietl et al. [200] proposed to transform verification tasks into a puzzle games which can be solved by humans. The solution of the puzzle is then translated back and used for proving correctness and verification. Foldit [201] is an online multiplayer game whose players interact with protein structures while they compete and collaborate to optimize the computed energy. They discovered that players of their game are able to search the state space of proteins configurations faster than computational algorithms. We are inspired by these efforts and exclusively developed a solution that uses games with a purpose and crowdsourcing to statistically determine the users' acceptable level of quality for approximate computing.

4.8 Conclusion

Approximate computing is an emerging area that breaks the long-held fundamental abstraction of near-perfect accuracy in PL [202, 119, 117, 4, 147, 203],

OS [204], and Architecture [47, 120, 173, 118, 122, 6, 164]. While these techniques provide promising gains, they cause quality loss whose effects on the users are not well understood; leaving approximation in a position of questionable utility. Many of these inspiring studies argue that a certain quality loss may be acceptable without systematically considering the users' perspective. It is timely to systematically explore and study users' perspective on the effects of approximation. This work takes an effective initial step in this direction. This work provided an automated programming tool–rather unconventional–to methodically utilize crowdsourcing in identifying the desirable application output quality from the final users. This readily available tool provides a path for the research community to better assess their innovative approximation techniques. The framework enables developers to conveniently study user responses at scale and gain statistical confidence when deploying approximated applications. Our results from examining a variety of applications show the necessity of solutions such as AxGames since the crowd's response to approximation varies drastically across different applications. Moreover, when the users consider tradeoff and context, they tend of to be more tolerant to approximation. These results suggests that AxGames can add an unexplored, yet important, dimension to the research and development in approximate computing.

4.9 Other Work of This Author in Approximate Computing

In addition to the aforementioned works, I have been actively involved in many research projects in the field of approximate computing. We propose and develop several hardware approximation techniques, which target to achieve performance and efficiency while imposing a modest level of accuracy degradation.

As the first effort, we aim to tackle two fundamental memory bottlenecks:

limited off-chip bandwidth (bandwidth wall) and long access latency (memory wall). To achieve this goal, our approach exploits the inherent error resilience of a wide range of applications. We introduce an approximation technique, called Rollback-Free Value Prediction (RFVP) [205]. When certain safe-to-approximate load operations miss in the cache, RFVP predicts the requested values. However, RFVP does not check for or recover from load value mispredictions, hence, avoiding the high cost of pipeline flushes and re-executions. RFVP mitigates the memory wall by enabling the execution to continue without stalling for long-latency memory accesses. To mitigate the bandwidth wall, RFVP drops some fraction of load requests which miss in the cache after predicting their values. Dropping requests reduces memory bandwidth contention by removing them from the system. The drop rate is a knob to control the tradeoff between performance/energy efficiency and output quality.

We also explore to develop solutions–from circuit to compiler–that enable general-purpose use of limited-precision hardware to accelerate "approximable" code–code that can tolerate imprecise execution. We utilize an algorithmic transformation that automatically converts approximable regions of code from a von Neumann model to a neural model. While the prior work [128] aims to develop a digital Neural Processing Unit (NPU) [128], we propose its analog variation (ANPU) [6], which offers even larger performance and efficiency gains without significant increase in accuracy loss. In this work, we outline the challenges of taking an analog approach, including restricted-range value encoding, limited precision in computation, circuit inaccuracies, noise, and constraints on supported topologies. We address these limitations with a combination of circuit techniques, a hardware/software interface, neural-network training techniques, and compiler support. As the next step, we explore the integration of Neural Processing Unit (NPU) with GPUs, called NGPU [164]. Graphics Processing

Units (GPUs) can accelerate diverse classes of applications, such as recognition, gaming, data analytics, weather prediction, and multimedia. Many of these applications are amenable to approximate execution. This application characteristic provides an opportunity to improve GPU performance and efficiency. GPUs are, in a sense, many-core accelerators that exploit large degrees of data-level parallelism in the applications through the SIMT execution model. This project aims to harmoniously bring neural and GPU accelerators together without hindering SIMT execution or adding excessive hardware overhead. We introduce a low overhead neurally accelerated architecture for GPUs, dubbled NGPU, that enables scalable integration of neural accelerators for large number of GPU cores. This work also devises a mechanism that controls the tradeoff between the quality of results and the benefits from neural acceleration.

Chapter 5

FUTURE DIRECTIONS

Recent innovations in ML are set to revolutionize medicine, robotics, commerce, transportation, and many other aspects of our lives. Such transformative effects are predicated on providing (1) high-performance compute capabilities that enable learning of compute-intensive ML models, and (2) constantly advancing ML algorithms that can adopt to ever-changing application needs. Computer system and architecture community has taken the charge of fulfilling the first precondition. The advances in the realm of computer system and architecture have not only unleashed the capabilities of unsung ML algorithms, which used to be computationally infeasible for many practical problems, but also offered opportunities for further advances in the algorithms. However, current systems mostly rely on completely offloading intelligence to the cloud. This approach is not scalable in the era of Intelligent IoT, and raises privacy and security concerns. To this end, our future research will focus on enabling intelligence and learning on the edge. As the first step, we will devise an algorithm-defined specialized computing stack for accelerating ML and AI on edge devices. Then, we will develop hardware-assisted privacy and security solutions for intelligent edge devices.

5.1 Pushing Intelligence to the Edge

IT industry has reached to a point where the capabilities of ML are enough to be integrated with real-world applications. ML based services (e.g., Amazon Alexa) are expanding their capabilities and available on edge devices such as mobile phone. However, currently, the machine intelligence at the edge devices is still only active when they are connected to centralized, high performance cloud platforms. This system architecture is built upon the producer-consumer model that the high-performance, power-hungry cloud servers learn and execute the ML models, while the low-performance, energy-limited edge devices merely communicate the model inputs and outputs with the cloud. This separation is suboptimal since the data exchange between these compute platforms not only incurs significant intercommunication cost, but also raises privacy and security concerns when sensitive personal information is exchanged. To this end, we plan to explore solutions that push intelligence to the edge, where the edge devices have the learnt ML models in the local storage and perform the model inference. The goal is to obtain sufficient energy-efficiency and performance to enable inference at the edge while offering programmability to support diverse ML algorithms. Therefore, we plan to research reconfigurable accelerators for the energy-limited edge systems and to develop programming abstractions for the accelerators, building upon my most recent work [114].

5.2 Online Learning at the Edge

Our next step after realizing inference at the edge is to enable learning and adaptation, which is still left on the cloud platforms. The challenge is that it is infeasible for edge devices to completely take over the entire learning tasks from

remote servers due to the growing scale and complexity of modern learning models. As such, the objective will be to design learning system architectures for heterogeneous compute platforms–from high-performance cloud servers to power-constrained edge devices–which the learning tasks are split and distributed over. With my experience in building specialized computing stacks, we will develop a complete stack that provides high-level abstractions to unite the heterogeneous compute platforms as a single learning system. In developing the stack, we will leverage the online active learning algorithms, such as federated and few/one/zero shot learning, which are designed for learning in the decentralized and small data setup.

5.3 Private and Secure Learning

Although the effort of pushing intelligence towards the edge reduces the inter-platform communication, it cannot completely resolve the privacy and security concerns due to the unavoidable exchange of sensitive data between clouds and edge systems. Large internet companies such as Apple, Amazon, and Google run their own clouds, which collect enormous amount of sensitive personal data from edge devices to provide the ML based services and improve the ML model accuracy. Apple collects raw voice clips for Siri and Google collects geographic information to predict the traffic on their map service. While conventional cryptographic algorithms and more recently introduced blockchain technologies may meet the privacy and security demand, these solutions require enormous amount of compute power, which make them infeasible to be hosted on general-purpose systems in the energy-budgeted environment. To tackle this challenge, we will first develop the hardware-friendly security algorithms that can enable the private and secure learning at the end systems without

imposing significant hardware complexity. We will then devise programmable accelerators and their programming abstractions so that the acceleration systems not only offer efficiency to meet the performance and energy constraints of the edge systems, but also provides expressibility for a wide range of security algorithms.

REFERENCES

[1] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *MICRO*, 2017.

[2] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *HPCA*, 2016.

[3] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Misra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *MICRO*, 2016.

[4] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris, "Flexjava: Language support for safe and modular approximate programming," in *ESEC/FSE*, 2015.

[5] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards Crowdsourcing Quality Target Determination in Approximate Computing," in *ASPLOS*, 2016.

[6] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[7] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *ISCA*, 2016.

[8] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *PACT*, 2014.

[9] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural Acceleration for GPU Throughput Processors," in *MICRO*, 2015.

[10] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi,

H. Esmaeilzadeh, and K. Bazargan, “Axilog: Language support for approximate hardware design,” in *DATE*, 2015.

[11] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Misra, and H. Esmaeilzadeh, “DnnWeaver: From High-Level Deep Network Models to FPGA Acceleration,” in *CogArch*, 2016.

[12] D. Mahajan, K. Ramkrishnan, R. Jariwala, A. Yazdanbakhsh, J. Park, B. Thwaites, A. Nagendrakumar, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, “Axilog: Abstractions for Approximate Hardware Design and Reuse,” *IEEE MICRO Special issue on Alternative Computing Designs and Technologies*, 2015.

[13] A. Yazdanbakhsh, B. Thwaites, J. Park, and H. Esmaeilzadeh, “Methodical Approximate Hardware Design and Reuse,” in *WACAS*, 2014.

[14] A. Yazdanbakhsh, R. S. Amant, B. Thwaites, J. Park, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, “Toward General-Purpose Code Acceleration with Analog Computation,” in *WACAS*, 2014.

[15] J. Park, K. Ni, X. Zhang, H. Esmaeilzadeh, and M. Naik, “Expectation-Oriented Framework for Automating Approximate Programming,” in *WACAS*, 2014.

[16] *Apache Spark*, 2017. [Online]. Available: https://spark.apache.org/.

[17] *Apache Hadoop*, 2017. [Online]. Available: http://hadoop.apache.org/.

[18] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *ASPLOS*, 2014, pp. 269–284.

[19] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, “DaDianNao: A machine-learning supercomputer,” in *MICRO*, 2014.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “ShiDianNao: Shifting vision processing closer to the sensor,” in *ISCA*, 2015.

[21] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, “PuDianNao: A polyvalent machine learning accelerator,” in *ASPLOS*, 2015.

[22] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016. [Online]. Available: http://arxiv.org/abs/1602.01528.

[24] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.

[25] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.

[26] *Snickerdoodle: Affordable FPGA platform for powering everything robots, drones, and computer vision*, 2017. [Online]. Available: http://krtkl.com/.

[27] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, Jun. 2014.

[28] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[29] *Amazon EC2 F1 instances: Run custom FPGAs in the AWS cloud*, 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[30] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *SIGMOD*.

[31] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[32] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech dnns," in *ICASSP*, 2014.

[33] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[34] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. Jan, pp. 165–202, 2012.

[35] J. Langford, A. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[36] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. D. Walker, "Efficient large-scale distributed training of conditional maximum entropy models," in *NIPS*, 2009.

[37] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv:1602.06709 [cs]*, 2016.

[38] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *ICLR Workshop Track*, 2016.

[39] Intel Altera, *Arria 10 architecture*, 2017. [Online]. Available: https://www.altera.com/products/fpga/arria-serqies/arria-10/features.html.

[40] *Spark MLlib: Apache spark's scalable machine learning library*. [Online]. Available: http://spark.apache.org/mllib/.

[41] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[42] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.

[43] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016.

[44] Y. Ji, Y. Zhang, S. Li, and P. Chi, "NEUTRAMS: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.

[45] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *MICRO*, 2016.

[46] I. Stamoulias and E. S. Manolakos, “Parallel architectures for the knn classifier – design of soft ip cores and fpga implementations,” *ACM Transactions on Embedded Computer Systems*, vol. 13, no. 2, 22:1–22:21, Sep. 2013.

[47] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “”neural acceleration for general-purpose approximate programs”,” in *MICRO*, 2012.

[48] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, “SNNAP: Approximate computing on programmable socs via neural acceleration,” in *HPCA*, 2015.

[49] E. Manolakos and I. Stamoulias, “Ip-cores design for the knn classifier,” in *ISCAS*, 2010.

[50] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, “Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data,” in *AHS*, 2011.

[51] T. Maruyama, “Real-time k-means clustering for color images on reconfigurable hardware,” in *ICPR*, 2006.

[52] A. Filho, A. Frery, C. de Araujo, H. Alice, J. Cerqueira, J. Loureiro, M. de Lima, M. Oliveira, and M. Horta, “Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm,” in *SBCCI*, 2003.

[53] M. Papadonikolakis and C. Bouganis, “A heterogeneous fpga architecture for support vector machine training,” in *FCCM*, 2010.

[54] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. Graf, “A massively parallel fpga-based coprocessor for support vector machines,” in *FCCM*, 2009.

[55] A. Majumdar, S. Cadambi, and S. Chakradhar, “An energy-efficient heterogeneous system for embedded learning and classification,” *IEEE Embedded Systems Letters*, vol. 3, no. 1, pp. 42–45, 2011.

[56] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, “A massively parallel, energy efficient programmable accelerator for learning and classification,” *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 1, 6:1–6:30, 2012.

[57] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “Neuflow: A runtime reconfigurable dataflow processor for vision,” in *CVPRW*, 2011.

[58] A. Roldao and G. A. Constantinides, “A high throughput fpga-based floating point conjugate gradient implementation for dense matrices,” *ACM Transactions on Reconfigurable Technology System*, vol. 3, no. 1, Jan. 2010.

[59] G. Morris, V. Prasanna, and R. Anderson, “A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer,” in *FCCM*, 2006.

[60] D. DuBois, A. DuBois, T. Boorman, C. Connor, and S. Poole, “An implementation of the conjugate gradient algorithm on fpgas,” in *FCCM*, 2008.

[61] D. Kesler, B. Deka, and R. Kumar, “A hardware acceleration technique for gradient descent and conjugate gradient,” in *SASP*, 2011.

[62] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *FPGA*, 2015.

[63] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv:1408.5093*, 2014.

[64] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “Delite: A compiler architecture for performance-oriented embedded domain-specific languages,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, 134:1–134:25, Apr. 2014.

[65] D. C. Ku and G. De Micheli, *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, 1992.

[66] Xilinx INC, *Xilinx virtex-7 fpga vc709 connectivity kit*, 2014. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html.

[67] ——, *Virtex-7 fpga xt connectivity targeted reference design for the vc709 board*, 2014. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/vc709/2014_3/ug962-v7-vc709-xt-connectivity-trd-ug.pdf.

[68] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[69] *A variant of mnist dataset with 8 millions records*. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#mnist8m.

[70] J. P. Pinto, "Multilayer perceptron based hierarchical acoustic modeling for automatic speech recognition," PhD thesis, EPFL, 2010.

[71] B. Zhou, "High-frequency data and volatility in foreign-exchange rates," *Journal of Business & Economic Statistics*, vol. 14, no. 1, 2008.

[72] S Dhanya and R. V. Kumari, "Comparison of various texture classification methods using multiresolution analysis and linear regression modelling," *Springerplus*, vol. 5, no. 54, 2016.

[73] M. Segal, K. Dahlquist, and B. Conklin, "Regression approaches for microarray data analysis," *Journal of Computational Biology*, vol. 10, no. 6, 2003.

[74] D Singh, P Febbo, K Ross, D Jackson, J Manola, C Ladd, P Tamayo, A Renshaw, A. A. D, J Richie, E Lander, M Loda, P Kantoff, T Golub, and W Sellers, "Gene expression correlates of clinical prostate cancer behavior," *Cancer Cell*, vol. 1, no. 2, 2002.

[75] I. Cantador, P. Brusilovsky, and T. Kuflik, "Movielens dataset," in *HetRec*, 2011.

[76] Grouplens. (2017). Movielens dataset, [Online]. Available: http://grouplens.org/datasets/movielens/.

[77] *Netflix Prize Data Set*. [Online]. Available: http://www.netflixprize.com/.

[78] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggiott, and V. Vapnik, "Feature selection for svms," in *NIPS*, 2000.

[79] *Spark gpu and simd support*. [Online]. Available: https://github.com/kiszk/spark-gpu.

[80] R. Bordawekar, *Accelerating spark workloads using gpus*. [Online]. Available: https://www.oreilly.com/learning/accelerating-spark-workloads-using-gpus.

[81] *GPUEnabler*. [Online]. Available: https://github.com/ibmsparkgpu/gpuenabler.

[82] *CUDA-MLlib*. [Online]. Available: https://github.com/ibmsparkgpu/cuda-mllib.

[83] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *12th International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[84] *Caffe2*, 2017. [Online]. Available: https://github.com/caffe2/caffe2.

[85] *CUDA v8.0*, 2017. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[86] *cuDNN v7.0*, 2017. [Online]. Available: https://developer.nvidia.com/cudnn.

[87] *Wattsup .net meter.* 2017. [Online]. Available: http://www.wattsupmeters.com/.

[88] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

[89] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient mini-batch training for stochastic optimization," in *KDD*, 2014.

[90] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan, "Better mini-batch algorithms via accelerated gradient methods," in *NIPS*, 2011.

[91] M. T. c, A. Bijral, P. Richtárik, and N. Srebro, "Mini-batch primal and dual methods for svms," in *ICML*, 2013.

[92] O. Dekel, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," *Journal of Machine Learning Research*, vol. 13, no. 1, pp. 165–202, 2012.

[93] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu, "Sample size selection in optimization methods for machine learning," *Mathematical Programming*, vol. 134, no. 1, 2012.

[94] *TABLA source code*. [Online]. Available: http://www.act-lab.org/artifacts/tabla/.

[95] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, "Large-scale fpga-based convolutional networks," *Machine Learning on Very Large Data Sets*, 2011.

[96] C. Donninger, A. Kure, and U. Lorenz, "Parallel brutus: The first distributed, fpga accelerated chess program," in *IPDPS*, 2004.

[97] J. P. Walters, X. Meng, V. Chaudhary, T. Oliver, L. Y. Yeow, D. Nathan, B. Schmidt, and J. Landman, "Mpi-hmmer-boost: Distributed fpga acceleration," *Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 223–238, 2007.

[98] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating persistent neural networks at datacenter scale," in *HotChips*, 2017.

[99] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[100] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467 [cs]*, 2016.

[101] S. Cheng and J. Wawrzynek, "High Level Synthesis with a Dataflow Architectural Template," in *OLAF*, 2016.

[102] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *ISCA*, 2013.

[103] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan, "CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures," in *FPGA*, 2008.

[104] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "Fpmr: Mapreduce framework on fpga," in *FPGA*, 2010.

[105] J. Fowers, K. Ovtcharov, K. Strauss, E. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *FCCM*, 2014.

[106] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: An in-fabric memory architecture for fpga-based computing," in *FPGA*, 2011.

[107] M. King, A. Khan, A. Agarwal, O. Arcas, and Arvind, "Generating infrastructure for FPGA-accelerated applications," in *FPL*, 2013.

[108] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[109] I. Ouaiss, S. Govindarajan, V. Srinivasan, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures," *Lecture Notes in Computer Science*, vol. 1385-1388, pp. 31–36, 1999.

[110] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze fpga accelerator deployment at datacenter scale," in *SoCC*, 2016.

[111] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A mapreduce framework on opencl-based fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3547–3560, 2016.

[112] D. Diamantopoulos and C. Kachris, "High-level synthesizable dataflow mapreduce accelerator for fpga-coupled data centers," in *SAMOS*, 2015.

[113] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[114] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in *ISCA*, 2018.

[115] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *ISCA*, 2011.

[116] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” in *FSE*, 2011.

[117] M. Carbin, S. Misailovic, and M. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *OOPSLA*, 2013.

[118] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke, “Sage: Self-tuning approximation for graphics engines,” in *MICRO*, 2013.

[119] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, “EnerJ: Approximate data types for safe and general low-power computation,” in *PLDI*, 2011.

[120] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” in *ASPLOS*, 2012.

[121] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, “ERSA: Error resilient system architecture for probabilistic applications,” in *DATE*, 2010.

[122] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” in *ASPLOS*, 2014.

[123] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” *Communications of the ACM Research Highlights*, vol. 58, no. 1, pp. 105–115, Jan. 2015.

[124] M. de Kruijf, S. Nomura, and K. Sankaralingam, “Relax: An architectural framework for software recovery of hardware faults,” in *ISCA*, 2010.

[125] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, “Scalable stochastic processors,” in *DATE*, 2010.

[126] A Yazdanbakhsh, D Mahajan, B Thwaites, J Park, A Nagendrakumar, S Sethuraman, K Ramkrishnan, N Ravindran, R Jariwala, A Rahimi, H Esmailzadeh, and K Bazargan, “Axilog: Language support for approximate hardware design,” in *DATE*, 2015.

[127] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Quality programmable vector processors for approximate computing,” in *MICRO*, 2013.

[128] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *MICRO*, 2012.

[129] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard, “Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels,” in *OOPSLA*, 2014.

[130] W. Baek and T. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” in *PLDI*, 2010.

[131] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” in *ASPLOS*, 2010.

[132] G. Venkatesh, J. Sampson, N. Goulding, S. K. Venkata, S. Swanson, and M. Taylor, “QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores,” in *MICRO*, 2011.

[133] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, “Dynamically specialized datapaths for energy efficient computing,” in *HPCA*, 2011.

[134] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, “Conversion of control dependence to data dependence,” in *POPL*, 1983.

[135] L. Cardelli, “Program fragments, linking, and modularization,” in *POPL*, 1997.

[136] J Dean, C Chambers, and D Grove, “Selective specialization for object-oriented languages,” in *PLDI*, 1995.

[137] B. Grigorian, N. Farahpour, and G. Reinman, “BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing,” in *HPCA*, 2015.

[138] J. P. Banning, “An efficient way to find the side effects of procedure calls and the aliases of variables,” in *POPL*, 1979.

[139] X. Zhang, M. Naik, and H. Yang, “Finding optimum abstractions in parametric dataflow analysis,” in *PLDI*, 2013.

[140] M. Naik, *Chord*. [Online]. Available: http://jchord.googlecode.com/.

[141] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, *EnerJ*, http://sampa.cs.washington.edu/research/approximation/enerj.html.

[142] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[143] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[144] S Galal and M Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 913–922, 2011.

[145] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[146] S. Sidiroglou, S. Misailovic, H. Hoffman, and M. Rinard, "Probabilistically accurate program transformations," in *SAS*, 2011.

[147] J. Bornholt, T. Mytkowicz, and K. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *ASPLOS*, 2014.

[148] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *PLDI*, 2014.

[149] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[150] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, 2011.

[151] J. Gantz and D. Reinsel, *Extracting value from chaos*. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[152] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015.

[153] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *MICRO*, 2014.

[154] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO*, 2013.

[155] C. J. Clopper and E. S. Pearson, “The use of confidence or fiducial limits illustrated in the case of the binomial,” *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934.

[156] S. Wallis, “Binomial confidence intervals and contingency tests: Mathematical fundamentals and the evaluation of alternative methods,” *Journal of Quantitative Linguistics*, vol. 20, no. 3, pp. 178–208, 2013.

[157] *A/b testing*. [Online]. Available: https://en.wikipedia.org/wiki/A/B_testing.

[158] E. B. Wilson, “Probable inference, the law of succession, and statistical inference,” *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

[159] J. Sauro and J. R. Lewis, “Estimating completion rates from small samples using binomial confidence intervals: Comparisons and recommendations,” in *HFES*, 2005.

[160] M. H. DeGroot, *Probability and Statistics*. Chapman & Hall, 1974.

[161] *Lame mp3 encoder*. [Online]. Available: http://lame.sourceforge.net.

[162] *Speech recognition toolkit*. [Online]. Available: http://cmusphinx.sourceforge.net.

[163] J. W. Ratcliff and D. E. Metzener, “Pattern matching: The gestalt approach,” *Dr. Dobb’s Journal*, vol. 13, no. 7, p. 46, 1988.

[164] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lofti-Kamran, and H. Esmaeilzadeh, “Neural acceleration for gpu throughput processors,” in *MICRO*, 2015.

[165] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *CVPR*, 2009. [Online]. Available: http://image-net.org.

[166] *Freesound.org*. [Online]. Available: https://freesound.org.

[167] *Voxforge*. [Online]. Available: https://developer.nvidia.com/cuda-llvm-compiler.

[168] *The new york times timesmachine*. [Online]. Available: http://timesmachine.nytimes.com.

[169] *Amazon’s mechanical turk*. [Online]. Available: https://www.mturk.com.

[170] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSSx86: A full system simulator for x86 CPUs,” in *DAC*, 2011.

[171] J. Choo, C. Lee, H. Kim, H. Lee, B. L. Drake, and H. Park, “Apolo: Making sense of large network data by combining rich user interaction and machine learning,” in *VAST*, 2014.

[172] D. H. P. Chau, A. Kittur, J. I. Hong, and C. Faloutsos, “Apolo: Making sense of large network data by combining rich user interaction and machine learning,” in *CHI*, 2011.

[173] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn, “Flikker: Saving refresh-power in mobile devices through critical data partitioning,” in *ASPLOS*, 2011.

[174] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, “Approximate storage in solid-state memories,” in *MICRO*, 2013.

[175] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, “Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology,” in *DATE*, 2006.

[176] R. Hegde and N. Shanbhag, “Energy-efficient signal processing via algorithmic noise-tolerance,” in *ISLPED*, 1999.

[177] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, “Patterns and statistical analysis for understanding reduced resource computing,” in *Onward!*, 2010.

[178] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, “Petabricks: A language and compiler for algorithmic choice,” in *PLDI*, 2009.

[179] J. Sartori and R. Kumar, “Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications,” *IEEE Transactions on Multimedia*, vol. 15, no. 2, 2013.

[180] C. Alvarez, J. Corbal, and M. Valero, “Fuzzy memoization for floating-point multimedia applications,” *IEEE Trans. Comput.*, vol. 54, no. 7, 2005.

[181] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization,” in *ISCA*, 2014.

[182] X. Li and D. Yeung, “Application-level correctness and its impact on fault tolerance,” in *HPCA*, 2007.

[183] ——, “Exploiting application-level correctness for low-cost fault tolerance,” *J. Instruction-Level Parallelism*, 2008.

[184] M. de Kruijf and K. Sankaralingam, “Exploring the synergy of emerging workloads and silicon reliability trends,” in *SELSE*, 2009.

[185] Y. Fang, H. Li, and X. Li, “A fault criticality evaluation framework of digital systems for error tolerant video applications,” in *ATS*, 2011.

[186] V. Wong and M. Horowitz, “Soft error resilience of probabilistic inference applications,” in *SELSE*, 2006.

[187] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, “Aslan: Synthesis of approximate sequential circuits,” in *DATE*, 2014.

[188] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, “Salsa: Systematic logic synthesis of approximate circuits,” in *DAC*, 2012.

[189] J. Miao, A. Gerstlauer, and M. Orshansky, “Approximate logic synthesis under general error magnitude and frequency constraints,” in *ICCAD*, 2013.

[190] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, “ABACUS: A technique for automated behavioral synthesis of approximate computing circuits,” in *DATE*, 2014.

[191] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, “Synthesizing parsimonious inexact circuits through probabilistic design techniques,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, 93:1–93:26, May 2013.

[192] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, “Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling,” in *Proceedings of the 9th Conference on Computing Frontiers*, 2012.

[193] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum, “Recaptcha: Human-based character recognition via web security measures,” *Science*, vol. 321, no. 5895, pp. 1465–1468, 2008.

[194] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "Automan: A platform for integrating human-based and digital computation," in *OOPSLA*, 2012.

[195] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: Human computation algorithms on mechanical turk," in *UIST*, 2010.

[196] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, "Labelme: A database and web-based tool for image annotation," *International Journal of Computer Vision*, vol. 77, no. 1-3, pp. 157–173, 2008.

[197] L. Von Ahn, "Games with a purpose," *Computer*, vol. 39, no. 6, pp. 92–94, 2006.

[198] L. Von Ahn, R. Liu, and M. Blum, "Peekaboom: A game for locating objects in images," in *CHI*, 2006.

[199] L. Von Ahn and L. Dabbish, "Labeling images with a computer game," in *CHI*, 2004.

[200] W. Dietl, S. Dietzel, M. D. Ernst, N. Mote, B. Walker, S. Cooper, T. Pavlik, and Z. Popović, "Verification games: Making verification fun," in *FTfJP*, 2012.

[201] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, and F. players, "Predicting protein structures with a multiplayer online game," *Nature*, vol. 466, no. 7307, pp. 756–760, 2010.

[202] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ASPLOS*, 2011.

[203] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *ASPLOS*, 2015.

[204] P. Stanley-Marbell and M. Rinard, "Lax: Driver interfaces for approximate sensor device access," in *HotOS*, 2015.

[205] B. Thwaites, G. Pekhimenko, A. Yazdanbakhsh, J. Park, G. Mururu, H. Esmaeilzadeh, O. Mutlu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *PACT*, 2014.