© 2021 Hashim Sharif

APPROXHPVM: A RETARGETABLE COMPILER FRAMEWORK FOR ACCURACY-AWARE OPTIMIZATIONS

BY

HASHIM SHARIF

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair
Professor Sarita Adve
Assistant Professor Sasa Misailovic
Professor Saman Amarasinghe, MIT
Associate Professor Henry Hoffmann, University of Chicago

Abstract

With the increasing need for machine learning and data processing near the edge, software stacks and compilers must provide optimizations for alleviating the computational burden on low-end edge devices. Approximate computing can help bridge the gap between increasing computational demands and limited compute power on such devices. We present ApproxHPVM, a portable optimizing compiler and runtime system that enables flexible, optimized use of multiple software and hardware approximations in a unified easy-to-use framework.

ApproxHPVM uses a portable compiler IR and compiler analyses that are designed to enable accuracy-aware performance and energy tuning on heterogeneous systems with multiple compute units and approximation methods. ApproxHPVM automatically translates end-to-end application-level quality metrics into accuracy requirements for individual operations. ApproxHPVM uses a hardware-agnostic accuracy-tuning phase to do this translation that provides greater portability across heterogeneous hardware platforms.

ApproxHPVM incorporates three main components: (a) a compiler IR with hardware-agnostic approximation metrics, (b) a hardware-agnostic accuracy-tuning phase to identify error-tolerant computations, and (c) an accuracy-aware hardware scheduler that maps error-tolerant computations to approximate hardware components. As ApproxHPVM does not incorporate any hardware-specific knowledge as part of the IR, it can serve as a portable virtual ISA that can be shipped to all kinds of hardware platforms.

We evaluate ApproxHPVM on 9 benchmarks from the deep learning domain and 5 image processing benchmarks. Our results show that our framework can offload chunks of approximable computations to special-purpose accelerators that provide significant gains in performance and energy, while staying within user-specified application-level quality metrics with high probability. Across the 14 benchmarks, we observe from 1-9x performance speedups and 1.1-11.3x energy reduction for very small reductions in accuracy.

ApproxTuner extends ApproxHPVM with a flexible system for dynamic approximation tuning. The key contribution in ApproxTuner is a *novel three-phase approach to approximation-tuning that consists of development-time, install-time, and run-time phases.* Our approach decouples tuning hardware-independent and hardware-specific approximations, thus providing retargetability across devices. To enable efficient autotuning of approximation choices, we present a novel accuracy-aware tuning technique called *predictive approximation-tuning.* It can optimize the application during development-time and can also refine the optimization with (previously unknown) hardware-specific approximations at install-time.

We evaluate ApproxTuner across 11 benchmarks from deep learning and image processing domains. For the evaluated convolutional neural networks, we show that using only hardware-independent approximation choices provides a mean speedup of 2.2x (max 2.7x) on GPU, and 1.4x mean speedup (max 1.9x) on the CPU, while staying within 2 percentage points of inference accuracy loss. For two different accuracy-prediction models, our predictive tuning strategy speeds up tuning by 13.7x and 17.9x compared to conventional empirical tuning while achieving comparable benefits.

Dedicated to my parents (Shahid Sharif & Lalarukh Shahid), sister (Zainab) & my life partner, Hareem Dar, for their unconditional love and support.

ACKNOWLEDGMENTS

I owe a debt of gratitude to people who have helped me in my Ph.D. journey. First, I would like to thank my Ph.D. advisor Professor Vikram Adve, who has been my mentor throughout these years. Vikram has been inspirational to me in many ways, not only as a role model in research, but also as a great person. Vikram's mentorship has taught me the importance of always aiming high, being resilient and persistent, and to not be shy of continuously challenging yourself. His long term vision on Compilers and Systems research has helped me learn how to better position my ideas and work in the context of the broader challenges and opportunities faced by the community. Despite his busy schedule, he has always been available to discuss research and career goals, and always taken a keen interest in my progression as a researcher. Through thick and thin, Vikram has been extremely positive and supportive, which made it easier to cope with the challenges of a Ph.D. program. For these, I will be ever grateful to Vikram.

I would like to say a special thanks to Dr. Sasa Misailovic and Dr. Sarita Adve who have also played a significant role in my development as a researcher. Sasa has been ever available to discuss research ideas and spent significant amount of time with me to help me refine these further. Sasa has played an important role in pushing me to continuously improve my work and encouraging me to make it more widely impactful. I have greatly learned from his expertise in Approximate Computing and Programming Languages. Sarita has also played a key role in my research by asking the hard questions that forced me to think more deeply about my research directions. Her drive for top quality research and her ability to connect ideas from different research areas has helped me broaden my own thought process. I am extremely grateful to both Sasa and Sarita for being wonderful mentors and collaborators through this journey.

I am particularly grateful to people who believed in my abilities early in my career and pushed me to pursue a Ph.D. program. My friend Waqas Iftikhar was instrumental in encouraging me to pursue research and grad studies and gave me the confidence that I can make it to a top school. The detailed conversations about career goals with Waqas helped me a great deal in carving out a pathway to a research career. Waqas also encouraged me to join the Lahore University of Management Sciences (LUMS) for a research internship which played a major role in preparing me for grad school. At LUMS, Dr. Fareed Zaffar was my research mentor from whom I learnt a great deal about research, academia, and the academic publication process. Dr. Fareed spent a great deal of time with me preparing

me for grad school and helping me build a research profile by involving me in multiple very interesting research projects. Post the internship and throughout my Ph.D., Dr. Fareed has continued to offer his support and guidance for which I am very grateful. My collaborator Dr. Ashish from SRI International has also been a very positive influence. I have learnt a great deal from Ashish about Systems research and continue to do so. I am very grateful for the positive encouragement and guidance he has always provided.

My research was not possible without the contributions and the support of my highly talented collaborators. These include Yifan Zhao, Akash Kothari, Maria Kotsifakou, Muhammad Huzaifa, Prakalp Srivastava, Abdul Rafae Noor, Adel Ejjeh, Peter Pao-Huang, Nathan Zhao, Arun Narenthiran, and Dr. Girish Chaudhry. To Yifan, Akash, and Maria: I thoroughly enjoyed our time together working on the HPVM project. Throughout our journey on the ApproxHPVM and ApproxTuner work, we took all challenges head-on and worked together truly as a team. Huzaifa, you went above and beyond in helping me with Computer Architecture related queries. Not only were these conversations very helpful, it was always fun catching up.

I am also extremely grateful to friends and relatives that have been very supportive of me taking up grad school and always believed in me to perform well. These include Shehroze Farooqi, my friend who is also pursuing his Ph.D. at the University of Iowa, and has always been a positive influence. Talal Anwar, my close friend from undergrad has been more than supportive and helpful whenever I have turned to him for help. I am extremely grateful to both of them. I also owe a thanks to Haris, my cousin who always believed in me, my Aunt, Shaheena, who has helped me in many ways. Abubakar, my cousin who pursued a degree in Computer Science was the first to encourage me to take up CS, and always offered very sound career advice in my early years. I owe a special thanks to my wife's parents, Bushra Dar and Ishfaq Dar, for being pillars of support through this journey, and standing by me through thick and thin. The positive encouragement, guidance, and words of wisdom have helped me both academically and in my personal life.

I am more than thankful to the University of Illinois at Urbana-Champaign (UIUC) Staff that has been more than willing to help me out with any academic and non-academic queries. Kathy Runck, Viveka Kudaligama, and Maggie Chappell have went above and beyond in their assistance and support. Glen Rundblom at Engineering IT support has helped me and my colleagues out in the most stressful times; whenever we required urgent IT assistance close to paper deadlines. For these, I am very grateful.

For anything I have become, the excellent education I was fortunate to receive, and for the opportunities I was able to secure was never possible without the unconditional love, support, encouragement, and guidance of my parents, Shahid Sharif, and Lalarukh Shahid.

My father Shahid is my hero and my inspiration. From him, I learnt the values of hard work, making smart decisions in life, integrity and honesty, and the virtue of going out of the way to help friends, family, and colleagues. From early years, my father's love for his medical profession instilled in me the importance of being true to your profession and having a good work ethic. Throughout my Ph.D., he has been the most supportive, helping me to be positive through the toughest of times, and always reminding me to be positive. My mother has been a similar positive influence, always taking my mind off the negatives and helping me focus on the task at hand. She has always believed in my abilities as a student and trusted me to succeed through challenging times. Zainab, my sister, has been a huge influence of my professional career. She has been extremely supportive of my graduate studies and always believed in me. Her love, care, and support has been instrumental in me achieving better things. I am also very grateful to my paternal grandparents, who are sadly no longer with me, for the great love, care and support they offered in my early years of education. This thesis would have made them very proud.

In the journey of my degree, I met Hareem Dar, my life partner, companion, and best friend. I feel extremely luck to have met her and extremely fortunate to have her as part of my life. I have learned a number of positive virtues from her, including the importance of being resilient, hard working, forgiving, and not losing sight of the most important things. I always seek her advice on career and non-career related things, and always receive useful guidance. Through thick and thin, she has stood beside me, steady as a rock. Without the positivity she brings to my life, I could not have imagined completing this degree. I was more than lucky to have had Hareem on my side as part of this wonderful journey.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION . . . 1
1.1 Approximate Systems: Motivation . . . 1
1.2 Challenges in Approximation Tuning . . . 4
1.3 Limitations of State of the Art . . . 7
1.4 Problem Statement . . . 8
1.5 Contributions . . . 8

CHAPTER 2 RELATED WORK . . . 18
2.1 Approximation-Aware Programming Languages . . . 18
2.2 Systems for Offline Autotuning . . . 20
2.3 Approximation-Driven Adaptive Systems . . . 21
2.4 Compilers for Machine Learning . . . 23
2.5 Compilers for Heterogeneous Systems . . . 23
2.6 Approximation Techniques . . . 25
2.7 Analytical Techniques for Approximation Tuning . . . 27

CHAPTER 3 CHALLENGES IN APPROXIMATION TUNING . . . 28
3.1 Diverse Range of Heterogeneous Systems . . . 28
3.2 Source Code Portability . . . 29
3.3 Object Code Portability . . . 30
3.4 Manual Tuning is Challenging . . . 30
3.5 Large Tradeoff Space Combining Multiple Approximations . . . 31
3.6 High Cost of Empirical Autotuning . . . 31
3.7 Optimization Choice Depends on Runtime Conditions . . . 32
3.8 Application-Specific End-to-End Effects of Approximation . . . 32

CHAPTER 4 APPROXHPVM: A PORTABLE COMPILER IR FOR ACCURACY-AWARE OPTIMIZATIONS . . . 34
4.1 Introduction . . . 34
4.2 ApproxHPVM Internal Representation and System Workflow . . . 36
4.3 Accuracy-Aware Mapping and Optimization . . . 42
4.4 Methodology . . . 47
4.5 Evaluation . . . 53
4.6 Conclusion . . . 63

CHAPTER 5 APPROXTUNER: COMPILER AND RUNTIME SYSTEM FOR ADAPTIVE APPROXIMATIONS . . . 64
5.1 Introduction . . . 64
5.2 ApproxTuner Overview . . . 67
5.3 Development-Time Tuning . . . 72

5.4 Install-Time Tuning 76
5.5 Runtime Approximation Tuning 77
5.6 Evaluation Methodology 79
5.7 Evaluation 82
5.8 Exploratory Study: Tuning for Pruned Models 90
5.9 Conclusion 91

CHAPTER 6 APPROXROBOTICS: THE COST AND ACCURACY TRADE-OFF FOR SMALL MOBILE ROBOTS 93
6.1 Introduction 93
6.2 Background: Robot System Design 95
6.3 Approximation: Model Pruning 99
6.4 Experimental Methodology 99
6.5 Evaluation 101
6.6 Discussion and Conclusions 109

CHAPTER 7 IMPLICATIONS FOR CURRENT PRACTICE 110

CHAPTER 8 FUTURE WORK 113
8.1 Support for More Application Domains 113
8.2 Translation to Approximate Hardware Accelerators 113
8.3 Automatic Generation of Approximate Kernels 114
8.4 Domain-Specific Approximation Techniques 114
8.5 Extending Approximation-Tuning with Model Retraining Support. 115

REFERENCES 116

CHAPTER 1: INTRODUCTION

1.1 APPROXIMATE SYSTEMS: MOTIVATION

With the slowdown of Moore's Law and the end of Dennard scaling, the gap between hardware performance and the computational demands of software applications is widening [1]. A recent trend that contributes to this gap is *edge computing*; a computing paradigm that brings computation closer to where data is being gathered, to improve response times, reduce network bandwidth usage, and reduce cloud costs [2, 3, 4]. Some examples of edge devices include mobile phones, IoT devices, small autonomous robots, and drones among many others. In contrast to high budget cloud environments (that use high-end servers), edge computing has tighter constraints on the cost of deployed hardware [5]. Also, edge devices are battery powered; hence power- and energy-efficiency is a key consideration. Design characteristics that enable low-cost, low-power hardware include: fewer compute cores, lower clock frequencies, and smaller local and global memories. These design characteristics also cause the hardware to be compute and memory constrained.

A wide range of different kinds of computations are being deployed on the edge including image and video processing, object classification, speech recognition, facial recognition, data analytics, among others [6, 7, 8]. These applications are very compute-intensive, which often renders it completely infeasible to run such computations on resource-constrained edge hardware devices. An example of one such compute-intensive edge computing application are visually-guided autonomous navigation systems that use deep neural networks (DNNs) for object detection, object classification, and regression tasks [9, 10]. The latest neural network architectures provide reasonable end-to-end prediction accuracy which makes them suitable for use with autonomous navigation controllers, but these require computing millions of operations for each data input image [11, 12, 13], which results in high execution times, and high power and energy usage.

Many computations in these edge computing application domains are inherently approximate, in the sense that the input data are often derived from noisy sensors and output results are often probabilistic, e.g., for object classification or facial recognition, or also noisy, e.g., for image and audio streams. Such computations can often *tolerate small errors that introduce some acceptable degradation in the quality of the output result* [14]. This fault tolerance property of applications has lead to the adoption of *approximate computing techniques*; software and hardware optimizations that trade-off small amounts of accuracy (or result quality) for gains in performance and/or energy.

Approximation Technique	Reference Systems
Load Value Approximation	Miguel et al. [15], Sutherland et al. [16]
Approximate Writes	Fang et al. [17]
Voltage Scaling	Chippa et al. [18]
Compute Precision Tuning	Gonzalez et al. [19] Duben et al. [20]
DRAM Refresh Rate Reduction	Jung et al. [21], Liu et al. [22]
Analog Computation	Srivastava et al. [23], Li et al. [24], Amant et al. [1]
InExact Hardware	Kang et al. [25], Kulkarni et al. [26]
Neural Processing Units	Esmailzadeh et al. [27, 28]

Table 1.1: Various hardware approximation techniques proposed by different studies.

Approximation Technique	Reference Systems
Task Skipping	Goiri et al. [29], Raha et al. [30]
Loop Perforation	Hoffman et al. [31], Sidiroglou et al. [32]
Input Sampling	Samadi et al. [33]
Barrier Elision	Misailovic et al. [34]
Function Substitution	Zhu et al. [35], Ansel et al. [36]
Approximate Memoization	Keramidas et al. [37], Rahimi et al. [38]
Approximate Parallelization	Misailovic et al.[39]
Dynamic Knobs	Hoffman et al. [40, 41], Xu et al. [42]
DNN weight pruning	Han et al. [43], Han et al. [44], LeCun et al. [45]
Low Rank Factorization	Han et al. [44]
Perforated Convolutions	Figurnov et al. [46]
DNN Quantization	Han et al. [43] Sakr et al. [47], Bulat et al. [48]
DNN Weight Sharing	Han et al. [43]
Lossy Compression	Samadi et al. [49]

Table 1.2: Various software approximation techniques proposed by different studies.

Approximation techniques have been proposed at several different layers of the system stack ranging from low-level hardware primitives for approximation, approximate functions in software libraries, and compiler optimizations. At the hardware architecture level, approximations have been applied in many different components: floating-point units, caches, DRAM, and specialized analog and digital accelerators [23, 1, 27]. The use of *specialized accelerators* is becoming widespread with hardware vendors incorporating these as part of commercial System on Chips (SoCs). Specialized accelerators are being used in widely varied domains domains including computer vision [50, 51, 52], natural language processing [53, 54, 55], and graphics among others. These accelerators provide orders-of-magnitude energy and performance improvements over general-purpose hardware. Some popular commercial accelerators (at the time of writing this document) include Google Tensor Process-

ing Units (TPUs) [50], Nvidia NVDLA [51], and Intel Movidius [52]. These accelerators, Nvidia GPUs (commonly used for machine learning), and also a few CPUs support reduced precision in FP16 (16-bit floating point) and INT8 (8-bit Integers) as hardware-supported approximations. Table 1.1 lists a few hardware approximations.

Approximate techniques are similarly diverse at the software level. These include *function substitution* (use of approximate function variants), *loop perforation* [56, 32] (skipping loop iterations), *barrier elision* [57] (speculatively skipping barriers), *reduction sampling* [35] (reduction using subset of inputs), among many others. In the deep learning domain, domain-specific optimizations such as *weight pruning* [45, 43] (removing less important weights) and *low-rank factorization* [58] (reducing the rank of tensor computations) are gaining popularity given their ability to significantly compress deep neural network (DNN) model weights (up to 50x [43]) with minimal accuracy impact. Software approximations have the advantage of being applicable across different hardware devices without needing any special hardware support. Moreover, software and hardware approximations can be potentially combined (as we show in our work) to gain even higher performance benefits. Table 1.2 lists a few software approximations.

Tolerance to errors varies highly across applications and different application components. Not all application kernels (or operations) are resilient in the face of errors. Some kernels can tolerate significant inaccuracy with minimal impact on the end-to-end application quality, while even small errors in some kernels can result in unacceptable quality degradation. The choice of approximation is also important since different approximation types introduce different kinds of errors (e.g., Gaussian, Uniform, Random). This behavior motivates the need for analyses that identify the error resiliency of different application kernels and select which operators/kernels to approximate and what kinds of approximations to apply.

There is need for *easy-to-use systems* that facilitate end-users in using approximations for their applications. Many prior compiler and runtime systems include support for approximations [59, 60, 61, 62, 49, 33, 44, 31, 63, 36, 40, 41, 57, 56, 64]. In a typical usage scenario, end-users provide *high-level specifications* for desired quality of service (QoS) and performance constraints. A *quality-of-service (QoS) constraint* is a bounded change in a (usually domain-specific) quality metric, such as a numerical reduction in inference accuracy for a machine learning model, or a reduction in peak signal-to-noise ratio (PSNR) for an image processing computation. High-level specifications are important since they relieve users from having deep insights on low-level application kernels and the impact of different approximations on individual kernels.

Many prior systems have focused on the problem of *approximation tuning* [59, 60, 61, 62, 44, 49, 33, 36]; an optimization problem that maximizes performance and/or energy benefits

(possibly other objectives), while satisfying some high-level quality constraint (e.g., inference accuracy in machine learning models). The optimization problem includes selecting for each application operation an *approximation knob*; a discrete-valued parameter of an approximation method that can be modified to control the quality, energy, and run time (e.g., rate of loop perforation). The approximation tuning phases identifies candidate *configurations*. A *configuration* is an assignment of zero or more approximation choices to every operation in the program, together with a setting of the knobs for all the assigned approximations. The *search space* for optimizations is the set of all possible configurations.

Previous systems for approximation tuning have significant limitations including: i) some systems require significant user-guidance [61, 62, 65, 64], ii) some do not exploit hardware-specific approximation choices nor translate code to heterogeneous compute units [59, 60, 61, 36, 49, 33, 65, 31, 63], iii) many systems support limited approximation techniques [56, 34, 31, 63, 46, 35, 66, 32, 67], iv) lack support for combining different approximation techniques [31, 63, 46, 35, 66, 68, 69, 32, 67], v) many systems have no support for dynamically changing approximation knobs [36, 59, 60, 61, 62, 67, 33]. Our work fundamentally improves over prior work by addressing the aforementioned limitations. This dissertation proposes a retargetable compiler and runtime system that supports abstractions for approximate computing, incorporates a wide range of approximations types, supports combining different approximations in a single program, and includes a flexible and easy-to-use approximation tuning framework. The several challenges that arise in the design of our system are detailed next in Section 1.2, and limitations of prior systems are detailed in Section 1.3.

1.2 CHALLENGES IN APPROXIMATION TUNING

In practice, a realistic application (e.g., a neural network or a combination of an image processing pipeline and an image classification network) can make use of multiple approximation techniques for different computations in the code, each with its own approximation knobs that must be tuned, to achieve the best results. For example, our work shows that for the ResNet-18 convolutional neural network, which contains 22 tensor operations, the best combination (considering accuracy-performance tradeoff) is to use three different approximations with different parameter settings in different operations. A major open challenge is *how to select, configure, and tune the parameters for combinations of one or more approximation techniques*, while meeting *end-to-end requirements* on energy, latency, and accuracy. Several specific challenges arise in meeting this goal. These challenges are detailed in Section 3 and briefly explained here:

1.2.1 Diverse Range of Heterogeneous Systems

"Heterogeneous systems" refer to hardware platforms composed of multiple compute units with different microarchitectures that communicate over some shared interface (e.g., common global shared memory). One real-world example are mobile SoCs that include CPUs for general purpose tasks, GPUs for graphics and machine learning, and additional specialized accelerators for audio/video decoding, computer vision tasks, and DSP among other tasks. This diversity in hardware facilitates performance and power efficiency but introduces challenges for compilers that must target different ISAs, optimize code for each kind of processor, and maximize local and overall hardware utilization.

Compute units of different hardware types (e.g., CPUs, GPUs, accelerators) and across different generations of the same hardware type (e.g, different generations of Nvidia GPUs) provide different approximation options and also differing accuracy-vs-performance tradeoffs [70]. From the approximation-tuning perspective, domain-specific accelerators are particularly interesting since they support hardware-specific approximations (e.g. analog compute accelerators [23], low-precision ML accelerators [50, 51, 52]) that can offer orders-of-magnitude performance and energy improvements. To maximize overall performance and energy improvements, a compiler framework must be able to map to these efficient hardware primitives.

1.2.2 Source and Object Code Portability

Since software applications (especially for mobile phones and IoT devices) are expected to cater to a range of different devices, *software portability is an important requirement*, not just at the source-code level but also the ability to *ship* software that can execute on a wide range of systems. Applications for both desktop and mobile (e.g., smartphone or tablet) systems are almost always shipped by application teams to end-users in a form that can execute on multiple system configurations (e.g., with different vector ISAs or GPUs). GPUs, for example, provide virtual instructions sets, e.g., PTX [71] or HSAIL [72], to enable software to be shipped as "virtual object code" that is translated to particular hardware instances only on the end-user's system. A critical goal for real-world use of such approaches is to enable software to be shipped as *portable* virtual object code, while deferring the hardware-specific aspects of accuracy-performance-energy optimizations to be performed after shipping [73] (e.g., on the end-user's device or on servers in an app store).

1.2.3 Large Search Space and High-cost of Empirical Tuning

The variety of software and hardware approximations with many accuracy-performance trade-offs introduce a large search space of possible configurations that is hard to navigate efficiently. In our work, for ResNet-50 with 53 convolution layers and 63 approximation knobs for each convolution layer, the total search space includes a total of $7e^{91}$ unique configurations. Since an exhaustive search over this large search space is completely infeasible, heuristic alternatives are employed to intelligently navigate a fraction of the search space. One such approach is *autotuning*: an optimization that uses heuristic search techniques (e.g., genetic algorithms, hill climbers, random search) to navigate a subset of configurations, and uses a (developer-specified) fitness function to compute the cost/profitability for each evaluated configuration [74, 36, 75]. *Empirical autotuning* uses empirical evaluations (i.e., running the program binary) to measure the attributes of interest such as quality of service (QoS) measurements, (e.g., inference error in CNNs) throughout, latency, and energy usage among possible others. We find that approximation-tuning via empirical tuning takes days (on a server-class machine) for some benchmarks - 1.5 days for VGG16 and 11 days for ResNet50. These long empirical tuning times motivate the need for efficient tuning.

1.2.4 Optimization Choice Depends on Run-time Conditions

Dynamic optimization capabilities are important in deployments where runtime conditions are likely to keep changing. Few examples of such changing conditions are:

- **Varying system workloads.** Increasing system load can cause applications to be unresponsive or miss deadlines.
- **Low power modes.** Low-power modes usually involve power gating compute cores and reducing processor and memory clock frequencies [76, 77], leading to overall system slowdowns.
- **Changing QoS requirements.** The nature of the operating environments can impact QoS (Quality of Service) requirements. For instance, a autonomous robot navigating in a dark environment may require a higher accuracy mode compared to a robot navigating in daylight.

1.3 LIMITATIONS OF STATE OF THE ART

Current programming environments (languages, compilers, libraries, frameworks) both in production use and in the research literature, have several limitations in supporting flexible use of approximation techniques for performance and efficiency:

- **Not easy to use.** Many systems (EnerJ [61], ACCEPT [62], DECAF [65], Chisel [59], Rely [60], Petabricks [36], ApproxNet [64]) are not easy to use since they either require users to port their applications to new programming languages (Chisel [59] and Rely [60]), require source code changes (ApproxNet [64]), or require programmer-guided annotations to separate approximate and precise data and instructions (EnerJ [61], ACCEPT [62], DECAF [65]). Ideally, an approximation tuning system should only require from the user a high-level end-to-end quality specification and the unmodified program (in source or IR form), to generate optimized versions.

- **Support limited range of approximations.** Many systems support a very limited range of known approximation techniques, typically one or two approximation techniques [56, 34, 31, 63, 46, 35, 66, 78, 68, 69, 32, 67]. Some systems (PowerDial [40] and JouleGuard [41]) require that the knobs for controlling performance-quality trade-offs are exposed by the application developers as configuration options (e.g., command-line inputs) that are tuned by a runtime system, and do not support any software, algorithmic, or hardware approximations.

- **Do not combine different approximation choices.** Combining different approximation choices is attractive since it allows for choosing a combination of knobs (per operation/sub-computation) that maximizes performance and energy benefits, while staying within the accuracy constraint. Most systems do not support combining multiple approximation techniques in a single application, which requires tuning approximation parameters for different computational kernels in an application [56, 34, 31, 63, 46, 35, 66, 68, 69, 32, 67].

- **Lack the capability to target heterogeneous systems.** Most such systems (with exception of ACCEPT [62]) do not exploit heterogeneous hardware compute units that offer a variety of approximation choices with potential of providing significant speedups and energy reductions [59, 60, 61, 36, 49, 33, 65, 31, 63, 44, 64, 40, 41]. The TVM system can compile machine learning workloads for a variety of heterogeneous compute devices (CPUs, GPUs, and accelerators), but has no support for targeting accelerator-specific approximations, and doesn't have the ability to tune approximation choices.

- **Do not support source and object code portability.** None of the systems for accuracy-aware tuning [62, 61, 65, 59, 60, 63, 31, 49, 33, 36] uses or builds on top of a portable compiler IR representation that can be used as a format for shipping application code (with approximation information included).

- **Lack of support for dynamic approximation tuning.** Many systems (Petabricks [36], ACCEPT [62], Chisel [59], Rely [60], EnerJ [61], Paraprox [33], TVM [67]) do not provide any support for dynamically adapting approximation choices under changing resource constraints, such as changing workloads, battery capacity, or varying time budgets for inference decisions. Systems that support runtime tuning (PowerDial [40], JouleGuard [41] MCDNN [44], ApproxNet [64], SpeedGuard [31]) do not support any of the other key requirements in this list, including diverse approximation techniques, high-level specification, software portability, and heterogeneous hardware targets.

1.4 PROBLEM STATEMENT

Given the diversity of approximation techniques and the varying error tolerance across applications, *determining a mapping of approximations to computational kernels* while achieving an acceptable trade-off between application-level accuracy and performance or energy is an open research problem. Different kernels within an application are amenable to different kinds of approximation techniques often with varying approximation knob settings [33]. To make it easy to use approximations for delivering performance and energy improvements, there is a need for a framework that automatically discovers an optimal mapping of approximation knobs to program operations.

Moreover, application developers and end users cannot be expected to specify error tolerances in terms of the system-level parameters required by the various approximation techniques, or even know about many of them: *we need automated mapping strategies that can translate application-level specifications (e.g., tolerable classification error in a machine learning application) to* system-level *parameters* (e.g., neural network parameter precision or circuit-level voltage swings).

1.5 CONTRIBUTIONS

To develop a retargetable, portable, efficient, and easy-to-use compiler framework for approximate computing, I adopted a 2-step strategy.

My first work on ApproxHPVM focuses on developing an Intermediate Representation (IR) that includes portable abstractions for approximation metrics, and an accompanying compiler framework that intelligently allocates approximation budget to individual sub-computations in a programs, generates code for approximate accelerators, and can map to hardware-specific parameters and knobs that control approximation.

My second work on ApproxTuner extends ApproxHPVM with an efficient approximation tuning framework that includes a novel 3-phase approximation-tuning approach, supports a wider range of approximation choices, and adds support for dynamic adaptation of approximation knobs.

I also perform a preliminary study that characterizes the flexibility for approximations in a real-world visual perception system used for autonomous robot navigation. Our study shows that this system has much room for inaccuracy, allowing approximations to deliver significant performance improvements. These observations reinforce that approximation-tuning systems like ApproxHPVM and ApproxTuner are very applicable in real-world deployments.

1.5.1 ApproxHPVM: A Portable Compiler IR for Accuracy-Aware Optimizations

Existing systems for accuracy-aware optimizations do not provide a fully automated framework that is able to target multiple heterogeneous devices with diverse approximation choices. We propose **ApproxHPVM**, a unified compiler IR and framework that provides ease of programming and object-code portability – and does in a fully automatic manner:

- Programmers only have to specify *application-level, end-to-end error* tolerance constraints, and ApproxHPVM can use this information to optimize and schedule programs on a heterogeneous system containing multiple approximation techniques; and
- ApproxHPVM enables *software portability* by using a hardware-agnostic, accuracy-aware compiler IR and virtual ISA, and by partitioning the accuracy-energy-performance optimizations into a hardware-agnostic stage and a hardware-specific stage, where software can be shipped between the two stages.

The ApproxHPVM compiler is developed as an extension of Heterogeneous Parallel Virtual Machine (HPVM) [79], a retargetable compiler infrastructure and portable virtual ISA for heterogeneous parallel systems. HPVM itself is built on LLVM [80], and can use LLVM compiler passes and code generators for individual tasks. Using HPVM allows us to target diverse heterogeneous parallel systems, and facilitates developing an accuracy-aware IR that

serves as a fully self-contained, portable virtual ISA that can be shipped and mapped to a variety of hardware configurations.

ApproxHPVM takes as input a program written in Keras (a high-level library for designing neural networks and tensor-based programs) or HPVM-C (extensions for describing HPVM dataflow graphs using C functions), and *end-to-end quality metrics* that quantify the acceptable difference between approximate and non-approximate outputs. Our custom frontends compile code in Keras or HPVM-C to the ApproxHPVM Intermediate Representation (IR). The compiler performs optimization steps on the ApproxHPVM IR that map individual approximable computations within the program to specific hardware components and specific chosen approximation techniques. The approximation mappings are chosen such that they minimize execution time and maximize energy savings, while satisfying the given end-to-end constraints with high probability. To our knowledge, ApproxHPVM is the first system to support the combination of these capabilities: a) full automation from end-to-end application-level quality specifications, b) support for using multiple approximation techniques in a single program, c) ability to exploit hardware-specific approximations and targeting heterogeneous systems, and d) supporting object code portability (using a portable IR representation).

ApproxHPVM solves three key challenges to achieve these goals:

- For applications with multiple approximable computations, it automatically translates end-to-end error specifications to *individual error budgets* and bounds per approximable computation (e.g, a tensor operation), while statistically guaranteeing with high probability that the end-to-end specifications are satisfied

- It automatically determines how to map approximable computations to a variety of compute units and multiple approximation mechanisms, including efficient special-purpose accelerators designed to provide improved performance with lower accuracy guarantees.

- It optionally provides object code portability by *decoupling the approximation mapping and compilation problem* into a hardware-independent autotuning phase (at development-time) and a subsequent hardware-dependent mapping phase (at install-time).

The portability is optional because it does not always come for free: the optimization choices may sometimes be suboptimal compared to a single, end-to-end and hardware-specific strategy (we show this in our Evaluation). ApproxHPVM supports either strategy, and so the unified, hardware-specific strategy can be used when portability is not a requirement.

ApproxHPVM solves these challenges in a domain-specific manner, through a number of key features. ApproxHPVM extends HPVM with a set of approximable domain-specific operations as part of the IR, which enables the compiler to identify approximable computations, and also to define *hardware-independent but domain-specific approximation metrics* as attributes of those operations. ApproxHPVM is the *first compiler* to support approximation metrics at the IR level. Currently, ApproxHPVM (and ApproxTuner) supports tensor computations which are general enough to support a number of important application domains such as neural networks and image processing. As part of future work, we believe our framework can be extended to an even wider range of application domains.

For assigning per-operation approximation metrics, our custom autotuner uses randomized artificial error injection to measure the sensitivity of individual tensor operations and uses it to translate end-to-end specifications to approximation metrics at an operation-level. The error injection phase is hardware-agnostic and inserts errors of varying magnitude (chosen by the autotuner) sampled from standard error distributions (e.g., Gaussian). The hardware-specific approximation selection phase at install-time uses these approximation metrics and uses a simple lookup table (per approximation method per IR operation) to perform the translation very efficiently.

Specifically we make the following contributions:

Retargetable Compiler IR and Virtual ISA with Approximation Metrics: We show how to capture hardware-agnostic approximation metrics in a parallel compiler IR, while preserving retargetability across a wide range of heterogeneous parallel hardware. Moreover, the IR can serve as a hardware-agnostic virtual ISA, and so software can be shipped between the two optimization stages to achieve virtual object code portability for approximate computing applications.

Hardware-agnostic Accuracy Tuning: Given an end-to-end user-provided quality metric (e.g., reduced inference accuracy or PSNR for images), our hardware-independent accuracy tuner computes the corresponding accuracy requirements for individual IR operations that can satisfy the end-to-end goal.

Accuracy-aware Hardware Scheduling: The second stage maps individual tensor operations to specific target compute units and to specific approximation options within those compute units, by taking into account the error tolerance of operations and the accuracy guarantees provided by the target compute unit.

Evaluation on Target Platform: We evaluate 9 DNN benchmarks and 5 image processing filters, using two different accuracy thresholds for each: 1% and 2% decreases in inference accuracy for the DNNs, and 20dB and 30dB loss of PSNR for the image processing filters. We use the NVIDIA Jetson TX2 mobile SoC [81] and extend it by adding a sim-

ulated version of a (fully programmable) Machine Learning accelerator called PROMISE, which has shown to provide orders-of-magnitude energy and throughput benefits for vector dot-product operations [23]. Our results show that ApproxHPVM can successfully assign different tensor operations to different compute units (GPU or PROMISE) with different approximation options, achieving speedups of 1-9x and energy reductions of 1.1-11.3x compared to using FP32 precision, while statistically guaranteeing the specified accuracy metrics with 95% probability.

ApproxHPVM is published and presented at OOPSLA'19.

1.5.2 ApproxTuner: A Compiler and Runtime System for Adaptive Approximations

The second phase of my work extends ApproxHPVM with an efficient approximation-tuning framework called *ApproxTuner*. ApproxTuner adds to ApproxHPVM: a) a novel 3-phase strategy for selecting and tuning approximations, b) support for software algorithmic approximations, and c) capability to adapt approximation knobs at runtime.

The opportunities and challenges that motivate our ApproxTuner work are:

- *Empirical autotuning is slow.* Empirical evaluation of the program binary to measure performance and accuracy can be prohibitively expensive on edge systems, requiring weeks or months for realistic kernels, and also undesirable on cloud systems where energy and/or monetary costs can become overwhelming. Even for medium-sized programs, empirical tuning usually requires thousands of such expensive evaluations [75]. We find that empirical tuning supported in ApproxHPVM takes days (on a server-class machine) for some benchmarks - 1.5 days for VGG16 and 11 days for ResNet50.

- *Not all hardware platforms support approximation knobs.* Software algorithmic approximations present an important opportunity since these do not need special hardware support. ApproxHPVM only supported approximations exposed as hardware-specific knobs (FP16 and PROMISE), and had no support for software algorithmic approximations. To maximize performance benefits, a system should exploit both hardware-specific and hardware-independent software approximations.

- *Changing runtime conditions affect optimization choices.* Satisfying end-to-end requirements (accuracy, performance, energy) may depend on many run-time conditions, e.g., the load of the system, the state of the battery, the inputs to the computation, or varying application demands during execution. To meet its requirements, the application may need to adapt to the changing conditions and reconfigure during run time.

ApproxTuner is able to exploit both hardware-specific and hardware-independent approximations, preserve portability of source and object code, and provide runtime adaption capabilities by decomposing the optimization process into three stages: development-time, install-time and run-time. ApproxTuner takes as input a program and a QoS constraint, and generates a set of possible configurations that maximize a hardware-agnostic performance metric and produce QoS values within the constraint. The generated configurations are organized as a trade-off curve (with QoS and performance as the axes) At install time, the system refines this curve using hardware-specific optimizations and performance measurements, then uses the refined curve for the best static choices of approximations and parameter settings. The final tradeoff curve is included with the program binary. At run time, the tradeoff curve is used to dynamically change approximation settings based on run-time conditions (e.g., system slowdowns).

ApproxTuner is extensible to a wide range of software and hardware approximations. This work evaluates five approximations – three software (hardware-independent) techniques implemented for CPUs and GPUs - including, perforated convolutions (skip computing subset of outputs), input sampling for convolutions (using subset of input values), and reduction sampling, and two hardware-specific approximations (also supported in ApproxHPVM) - FP16 and analog computation on PROMISE.

We find that the approximation metrics used in ApproxHPVM are not suitable for tuning the software approximations used in ApproxTuner. This is because software approximations (perforation and sampling) cannot be easily captured with simple attributes such as L1 or L2 norms of output tensors. We discover that L1 and L2 norms work well with approximations that introduce noise that is independent of the inputs; this includes Gaussian random noise injected by analog computation in PROMISE, floating-point rounding errors due to FP16, and random bit flips. The software approximations are highly sensitive to the inputs. For instance, what spatial values in an output tensor get skipped while doing a *perforated convolution* (approximating some output elements) can have a major impact on the application accuracy. Accordingly, in ApproxTuner, we do not tune with hardware-agnostic approximation metrics, instead directly invoke the approximation techniques as part of our autotuning workflow.

To address the challenge of reducing the execution time of approximation tuning, ApproxTuner introduces a novel technique, *predictive approximation-tuning*. Predictive approximation-tuning uses one-time error profiles of individual approximations, together with error composition models for tensor-based applications, to predict end-to-end application accuracy[1].

[1]The work on accuracy prediction models is lead by Yifan Zhao (yifanz16@illinois.edu) and details on the models will also appear in his Thesis.

Our approach also facilitates distributed tuning since the error profile collection can happen at multiple client devices in a distributed manner with autotuning performed on a centralized server. This makes install-time tuning (with hardware-specific approximations) feasible which can otherwise be prohibitively expensive to do on a single resource-constrained edge device.

In summary, our contributions are:

- A system that combines a wide range of existing *hardware and software approximations*, supports diverse heterogeneous systems, and provides an easy-to-use programming interface for accuracy-aware tuning. We show that different kinds of approximations and approximation knobs are suited for different applications and also across sub-computations in the same application.

- A novel *three-phase accuracy-aware tuning technique* that provides performance portability, retargetability to compute units with hardware-specific approximation knobs, and dynamic tuning. It splits tuning into: 1) selection of hardware-independent approximations at *development-time*, 2) mapping to hardware-specific approximations at *install-time*, and 3) a fast approximation selection at *runtime*.

- *Predictive approximation-tuning*, a novel technique that speeds up both development-time and install-time analyses, achieving 14x faster tuning compared to empirical autotuning, while identifying similar-quality configurations.

- We propose *distributed predictive tuning*, that enables efficient selection of hardware-specific approximations at install-time.

- Our evaluation on 11 benchmarks (10 CNNs and 1 combined CNN + image processing benchmark) shows:

  Generic Approximations. Exploiting generic hardware-independent approximations, ApproxHPVM achieves geometric mean speedup of 2.2x and energy reduction of 2.1x on GPU, with merely 2 percentage points of drop in inference accuracy. On CPU, we observe a geometric mean speedup of 1.4x and energy reduction of 1.4x.

  Hardware Approximations. At install time, mapping tensor operations to PROMISE, an analog compute accelerator, ApproxHPVM provides geometric mean speedup of 4.5x across the benchmarks.

  Runtime Adaptation for Approximations. ApproxHPVM can dynamically tune approximation knobs to counteract system slowdowns imposed by runtime conditions such as low-power modes.

  ApproxTuner is accepted and presented at PPoPP'21.

1.5.3 ApproxRobotics: The Cost and Accuracy Tradeoff for Small Mobile Robots

My work on ApproxHPVM and ApproxTuner, and other related works [59, 60, 61, 65, 36, 49, 33, 44, 64, 67, 42, 82] are mostly focused on tuning application sub-components (e.g., neural networks, image and video processing filters) in isolation and independent of the larger application context. ApproxHPVM and ApproxTuner allow users to specify maximum tolerable degradation in accuracy (or quality of service), but it remains unclear on how much accuracy loss is acceptable in real-world deployments. I believe that the tolerable accuracy degradation is highly domain-, application- and context-specific. I envision that the choice of reasonable accuracy thresholds (to use for tuning purposes) will be guided by studies that characterize the flexibility for approximations in real-world applications. We perform one such study for a real-world autonomous navigation system used in agriculture robots[2]

We present an empirical study of tradeoffs, which highlights the significant role that computational approximations can play in enabling low-cost visually guided autonomous robots. Autonomous robots are increasingly reliant on visual information for perception, planning, and control [83, 84, 85, 9, 10, 86, 87]. Visual data can be very high dimensional, yet, with advances in deep learning, we are now seeing many applications that are able to extract actionable information through this data. A major challenge is that inference with these deep learning models is highly computationally expensive to run, especially on hardware devices with limited compute and memory resources. Large robots such as autonomous cars can afford to have much larger computational payloads, since these are powered by carbon fuels or large batteries, have sophisticated cooling systems, the cost of compute hardware can be kept to a small fraction of the total cost. In contrast, small battery-powered autonomous robots such as those used for agriculture, mining, or remote area exploration, have much tighter size, weight, and power constraints. Furthermore, cost-sensitive fields like digital agriculture [88] impose stringent cost constraints as well. For such robots, optimizing the computational requirements for visual navigation can be crucial.

Hence, a key open question is how the computational requirements for visual navigation can be optimized to use low cost hardware, in small battery-operated mobile robots. Answering this question can provide robot system designers with the understanding necessary to make optimal hardware choices. Indeed, conservative choices for compute hardware can be typically traced back to unclear computational requirements of the task-specific software

[2]This work is done jointly and equally with Yifan Zhao (yifanz16@illinois.edu), and will also appear in his Thesis. The software setup for the neural networks and the Extended Kalman Filter was done by Arun Narenthiran (av7@illinois.edu), and the high-level navigation controllers are developed by Mateus Valverde Gasparino (mvalve2@illinois.edu).

stack. Even more unclear to robot system designers is whether there is room to reduce computational demands by using software optimizations.

In this paper, we provide an empirical study that sheds light on the cost and accuracy tradeoff for small mobile robots. A key insight in this work is that, *in the context of feedback control systems, it may be possible to relax the accuracy of neural networks models for vision (which are usually the most computationally expensive part of a navigation system), without significantly hurting navigation robustness.* In particular, we show that approximate, pruned neural network models, which can trade off accuracy for speed, can still provide sufficient task accuracy when used in robust closed-loop autonomous systems. This enables robot designers to safely relax some accuracy constraints, and therefore select lower-cost hardware, without expecting to lose task performance quality, even when performing demanding real-world visual inference tasks such as autonomous navigation in agricultural fields.

In this work, we evaluate these research questions in the context of a small mobile agricultural robot, *TerraSentia* (obtained from EarthSense [89]), a production agbot that is used for autonomous navigation through corn fields for high-throughput phenotyping and a variety of production agriculture tasks. Our results, however, are applicable to any battery operated small robots that rely on feedback control driven by visual inference for task execution. The key goal of our work is to investigate *to what extent can approximations be used to trade off model accuracy for performance improvements (in particular, increased frames-per-second) in the neural-network models used in small autonomous robots.* These speedups can then facilitate using lower-cost compute hardware, or performing additional computations on the same hardware, or combinations of the two. The primary task under consideration is visually-guided autonomous navigation between crop rows using a state-of-the-art visual guidance system, CropFollow [90]. CropFollow uses a (low-cost) front-facing monocular RGB camera with a pair of convolutional neural networks (CNNs) for predicting the robot heading and the distance from crop rows, which is then used to perform autonomous row navigation.

To optimize the CNN models, we use a popular technique, *structured weight pruning* [91, 92, 93], which compresses CNN models by dropping a subset of convolution filters that have relatively small weights. There is a lot of recent work on network pruning [93, 94, 95, 43, 96, 97, 98], but it remains unclear 1. if it is feasible to apply pruning to CNNs used in the context of a larger real-world application, especially in the context of closed-loop control 2. is it acceptable to relax some accuracy to gain additional performance while avoiding observable impact on the end-to-end quality of the application, i.e. without losing control robustness.

Our focus on end-to-end robustness is in contrast with most existing approaches to neural network pruning, which are aimed at retaining the computational accuracy. This conserva-

tive approach to pruning limits the achievable computational gains as it does not leverage the inherent robustness of closed-loop autonomous systems.

On the other hand, with our approach of trading off accuracy without losing robustness (measured as crashes that needed manual interventions), we are able to drive far more aggressive computational performance improvements, ranging up to 15x (on CPU), with close to a 2x increase in inference error. *These performance speedups allow us to perform the entire navigation pipeline, including two CNNs, Bayesian sensor fusion and Model Predictive Control on a low-end $35 Raspberry Pi4 [99].*This compares with the $876 Intel NUC [100] used on commercial TerraSentia robots, and with the $59 Jetson Nano, the cheapest device we found to deliver necessary performance without approximations. Moreover, the Pi4 requires 30% lower peak power than the Jetson Nano (7W vs 10W).

To evaluate if our optimizations facilitate running multiple tasks on a single resource-constrained Raspberry Pi4, we also apply our pruning approach to *corn stand counting*: a video analytics task for counting corn stands. We show that by only slightly relaxing requirements for the accuracy of the final result counts enables stand counting to run in real-time, concurrently with the full navigation pipeline.

Specifically, our contributions are:

- We perform the first empirical study to characterize the impact of neural network model pruning on the end-to-end navigation quality of an autonomous robot with visually guided feedback control.

- We show that pruning the convolutional neural network models (CNNs) used in the visual perception system helps provide the minimal required FPS from the vision models (for crash-free navigation) on a $35 Raspberry Pi4, making it a feasible choice for compute hardware.

- We find that the CNN-based autonomous navigation control in the evaluated agbot is robust to infrequent mispredictions. We also identify pruning settings that introduce large prediction errors that greatly impact the navigation quality, resulting in crashes.

- By relaxing accuracy constraints, we show that multiple compute-intensive tasks including the navigation pipeline and 2 instances of stand counting can run on a shared compute-constrained Raspberry Pi4.

CHAPTER 2: RELATED WORK

This section presents related work in the areas of accuracy tuning, approximation-driven adaptive systems, compilers for machine learning, compilers for heterogeneous systems, programming languages supporting primitives for approximate computing, software- and hardware-based approximations, and analytical techniques for approximation tuning. Table 2.1 compares the capabilities of ApproxHPVM and ApproxTuner to other state-of-the-art systems. The capabilities compared include: support for different kinds of approximations, ease of use (requiring no source code modifications), retargetability across hardware architectures, support for portable object code, and approximation-tuning capabilities (runtime and static tuning).

2.1 APPROXIMATION-AWARE PROGRAMMING LANGUAGES

Rely [60] is an imperative programming language that allows developers to specify reliability specifications. Rely allows developers to allocate program variables in unreliable memories and use unreliable arithmetic and logical operations. Rely also includes analysis that verify the correctness of the developer-specified reliability specifications. The analysis computes the reliability of a program path as the probability that all individual operations on the path execute reliably. Chisel [59] extends the Rely programming language with specifications for accuracy in addition to reliability (supported by Rely). The accuracy specifications quantify the maximum acceptable difference between approximate and exact results, as opposed to reliability specifications that only capture the probability of a fully correct result. Chisel requires developers to provide (non-approximate) implementations of kernels in the Chisel programming language including desired reliability and accuracy specifications at the function level. These specifications are used by the compiler to automatically select instructions to approximate and data to map to unrealiable memories. The limitations of these systems include: a) require programmers to rewrite their applications in Chisel/Rely, b) do not exploit hardware-specific approximation knobs nor support mapping to approximate hardware accelerators, c) the approximation-tuning phase can choose low-level approximate instructions (and memory allocations) but doesn't support replacing high-level algorithms with approximate variants.

The EnerJ [61] type system extends the Java programming language with type qualifiers that distinguish *precise* and *approximate* data types. The type qualifiers are used by the compiler and runtime to chose energy energy-efficient approximate mechanisms (instructions,

Table 2.1: Comparing capabilities of ApproxHPVM and ApproxTuner to most closely related state-of-the-art systems.

	Approximations				**Programmability**			**Tuning Strategies**	
	Algorithm Approx	*Accelerator -specific Approx*	*Multi Domain Approx*	*Model Approx*	*No Code Changes*	*Retarget*	*Portable Object Code*	*Runtime Approx Tuning*	*Predictive Tuning*
ApproxTuner	✓	✓	✓	✗	✓	✓	✓	✓	✓
ApproxHPVM [102]	✗	✓	✓	✗	✓	✓	✓	✗	✗
Rely [60]	✗	✗	✓	✗	✗	✗	✗	✗	✗
Chisel [59]	✗	✗	✓	✗	✗	✗	✗	✗	✗
TVM [67]	✗	✗	✗.	✓	✓	✓	✓	✗	✗
ACCEPT [62]	✓	✓	✓	✗	✗	✓	✗	✗	✗
PowerDial [40]	✓	✗	✓	✗	✗	✗	✗	✓	✗
SAGE [49]	✓	✗	✓	✗	✗	✗	✗	✓	✗
SpeedPress [32]	✓	✗	✓	✗	✓	✗	✗	✓	✗
PetaBricks [36]	✓	✗	✓	✗	✗	✗	✗	✗	✗
DeepCompression [43]	✗	✗	✗	✓	✓	✗	✗	✗	✗

functions, memory allocations) that maintain correctness of the data marked as precise, and only a best effort translation for data marked approximate. DECAF [65] generalizes the EnerJ type system by allowing developers to specify expected probabilities of correctness with type declarations. Like Chisel and Rely, EnerJ and DECAF have no support for mapping to approximate hardware accelerators. Also, EnerJ and DECAF approximate low-level instructions but do not support algorithmic approximations. ACCEPT [62] is a compiler framework that uses developer annotations to guide approximation choices. ACCEPT extends C and C++ to incorporate an `APPROX` keyword that programmers use to annotate types (inspired by EnerJ). It uses autotuning to automatically find optimal low-level approximation parameters. ACCEPT supports an automatic neural acceleration transform (proposed by Esmaeilzadeh et al. [101]) that selects regions of code to map to an FPGA-based neural network accelerator. It profiles regions of code to extract pairs of inputs and corresponding outputs, and trains a neural network to mimic the code. ACCEPT supports only supports this very specific kind of hardware approximation and does not include general support for using hardware-specific approximations. All these three systems require developer-guided code changes to add type declarations.

These systems proposed a source code level approach for specifying approximation metrics. ApproxHPVM is the first to introduce the idea of quantifiable accuracy metrics at the IR level. Incorporating approximation metrics at the IR level provides a portable mechanism for preserving approximation metrics post-compilation and shipping these for use with later compiler optimizations at install-time. Making approximation attributes a first-class citizen in the compiler workflow facilitates the interaction of the accuracy-aware IR with various front-end languages *and* hardware-specific features and approximations (especially in heterogeneous systems).

2.2 SYSTEMS FOR OFFLINE AUTOTUNING

The *Petabricks* language provides programmers the flexibility to describe multiple algorithms for solving a problem and specify how the different components fit together [36, 73, 103]. This allows the Petabricks compiler and runtime to create and *autotune* a hybrid algorithm that optimizes a user-desired goal. For instance, users can specify multiple algorithm implementations of a matrix multiply operation (within some large application) each with varying accuracy and performance characteristics, and allow the autotuner to search for optimal configurations. Its auto-tuner uses heuristic techniques to navigate through a search space of alternative program implementations. ApproxHPVM also has the ability to tune for multiple algorithm choices and it does so without requiring any source code changes, while Petabricks requires developers to port their code to the Petabricks programming language. Petabricks doesn't allow users to exploit hardware-specific mechanisms in the alternate algorithm implementations. In contrast, ApproxHPVM has the ability to automatically select low-level hardware-specific knobs, while requiring from users only a high-level specification for desired quality of service.

Paraprox [33] is a system that identifies common data parallel patterns in code and automatically generates tuned approximate implementations for these patterns. Paraprox supports six different data parallel patterns including Map, Scatter, Scan, Reduce, Stencil and Partition. Paraprox requires users to provide baseline implementations of kernels in OpenCL or CUDA. It uses a source-to-source transformation to create approximate variants which are compiled to binaries using traditional GPU (nvcc) and CPU compilers (gcc). Since Paraprox does not generate low-level object code, it is fundamentally limited in its capability to exploit low-level hardware knobs. The system supports CPUs (through OpenCL) and GPUs (through CUDA) but doesn't support accelerators.

Misailovic et al. [56, 32] present an offline autotuner that automatically detects which loops to approximate. The autotuning includes two phases: 1) perforating expensive program loops with varying perforation levels (chosen by tuner) and attempting to maximize speedups with a certain accuracy threshold, 2) correctness testing; tests to validate that the transformed binary has correct behaviour. This includes checks for possible memory leaks and incorrect outputs (with respect to some set of known correct outputs). This system supports loop perforation as the only approximation technique and doesn't serve as a generic approximation-tuning system that supports algorithmic and/or hardware approximations.

OpenTuner [75] is a framework for building domain-specific autotuners. OpenTuner allows developers to specify the search space of possible configurations, a fitness function for ranking configurations, ability to chose from a set of available search techniques, and also

allows specifying custom search techniques. The core concept used by OpenTuner is the use of *ensembles* of search techniques. Multiple search techniques (hill climbers, genetic algorithms, random search etc.) run in parallel and collaboratively share results using a common database. Techniques that find better configurations/results are allocated higher time budgets while techniques performing poorly are allocated lesser time (or completely disabled). OpenTuner has shown to succeed with search spaces as large as 10^{3600} possible configurations. OpenTuner is a generic library with reusable primitives but in itself is not a framework for approximation-tuning, neither does it include any compiler support for code generation or selection of low-level hardware-specific attributes. In my ApproxHPVM and ApproxTuner work, I use the OpenTuner library to develop custom transforms for approximation tuning.

2.3 APPROXIMATION-DRIVEN ADAPTIVE SYSTEMS

PowerDial [40] uses a runtime and compiler approach to dynamically vary program accuracy to adapt application behaviour in response to power or load fluctuations. Compiler transforms identify program variables that host control values; values that affect performance and accuracy (e.g., number of max iterations for a convergence loop). The compiler passes insert callbacks that allow the PowerDial control system to directly update these variables based on control decisions. Some limitations include: a) it requires the application to expose control parameters as command-line inputs, b) it assumes that the program has a main control loop that includes computations that have performance dependent on the control variables. *JouleGuard* [41] is a similar runtime control system but also provides guarantees for energy consumption.

SAGE [49] is a runtime system for approximation-tuning. It includes two phases: a) an offline phase that generates multiple versions of target kernels with varying levels of accuracy, b) a runtime phase that uses a greedy algorithm to tune the per-kernel approximation parameters to achieve high performance, while staying within user-specified accuracy thresholds. As the behaviour of approximate kernels (accuracy and performance) can change at runtime, SAGE performs (infrequent) periodic recalibration that checks the output quality and performance and accordingly retunes the parameters. Green [104] is a similar runtime approximation-tuning framework that monitors application QoS and includes recalibration steps to ensure QoS constraints are met. Unlike SAGE, Green requires user to provide approximate kernel implementations or annotate their code using C/C++ extensions.

The *SpeedPress* [31] compiler systems takes standard C or C++ code and automatically applies loop perforation. SpeedPress uses profiling to measure accuracy and performance tradeoffs for varying perforation levels applied to different program loops. The accompanying

runtime system called *SpeedGuard* uses application heartbeats to monitor the application performance, and if it detects a slowdown, it dynamically updates the levels of loop perforation. *Sculptor* [63] is a similar system that dynamically varies the loop start and perforation rates. Sculptor is more sophisticated as it intelligently chooses which instructions to skip in different loop iterations (SpeedPress drops all instructions in a skipped iteration). The core idea is to skip instructions that are less important to application accuracy.

MCDNN [44] is a runtime adaptation system for deep learning models. It uses a catalog of models, each with varying speedup and accuracy tradeoffs. Based on changing performance and accuracy constraints, MCDNN switches the model used. To ensure that model switching cost is low, MCDNN requires the catalog of models to be in main memory. For memory-constrained edge systems, it is impractical to host multiple neural network models given DNNs can have millions of parameters and can occupy gigabytes of memory. In view of these limitations, Xu et al. presented *ApproxNet* [64]; a network architecture that includes a novel spatial pyramid pooling layer (SPP) that allows for early exits (effectively skipping convolution layers). Based on changing performance and accuracy requirements, this architecture allows for dynamically varying the number of layers used; with deeper network configurations providing higher accuracy but also higher execution times. The major drawback with ApproxNet is the need for training the model from scratch which currently (on high-end GPUs) takes days on large datasets such as ImageNet. In contrast, ApproxTuner does dynamic adaptations for accuracy and performance without requiring any model/-source changes (i.e., works with pretrained models). Moreover, ApproxTuner maintains a single model (one set of model weights) in memory and is hence practical for edge systems.

The *SiblingRivalry* [105] system uses and extends the Petabricks Compiler system to support online autotuning that can adapt to changing runtime conditions. For instance, varying machine load (common in cloud and data center environments) can substantially degrade application performance, rendering the offline autotuned program as suboptimal. SiblingRivalry takes the approach of allocating a fraction of available compute resources for online autotuning, with an autotuned program variation running in parallel with the normal program execution. If the autotuned program variation performs better, it replaces the current version. In this way, the autotuning progressively improves program performance and allows for adapting to new changing runtime conditions. ApproxTuner is able to adapt to runtime conditions by picking one of the configurations discovered in the offline autotuning phase, and hence imposes no additional overheads at runtime.

Many of these systems require changes to original source code (PowerDial, JouleGuard, Green, ApproxNet, SiblingRivalry), some are very domain-specific (MCDNN, ApproxNet), and some of these only support one or few approximation techniques (SpeedPress, Sculptor,

PowerDial, JouleGuard). None of these systems provides retargetability to heterogeneous compute units, and none of these exploits hardware-specific approximations.

2.4 COMPILERS FOR MACHINE LEARNING

TVM [67] is an end-to-end compiler framework that supports the compilation and optimization of machine learning workloads for multiple hardware targets, including CPUs, GPUs, FPGAs, and accelerators, and supports multiple frontends including Tensorflow, Pytorch, Keras and MXNet. TVM includes these key features: a) supports a *tensor expression language* that makes it convenient for developers to add new tensor operations; the compiler generates efficient low-level hardware-specific code for these high-level tensor operations, b) program optimizations that find optimized tensor operator implementations using an ML-based cost model, and c) graph rewriting optimizations that use the data-flow graph of the computation. TVM provides object code portability since programs are shipped in a portable (and differentiable) IR called *Relay*. TVM has a few limitations including: a) only supports one approximation technique; precision tuning for FP16 (on GPUs), b) has no support for algorithmic approximations, c) lacks support for approximation-tuning, and d) has no support for dynamic adaptation.

XLA [106] from Google (now part of the MLIR [107] framework), and Glow [108] from Facebook are also end-to-end compiler frameworks for machine learning that support multiple language frontends and hardware backends. Both these compilers include many domain-specific optimizations such as operator fusion, common sub-expression elimination (specific to tensor operations), constant propagation (specific to tensor operations) among many others. Like TVM, these systems only support precision tuning (to FP16), do not support algorithmic approximations, and have no support for approximation tuning.

Like TVM, XLA, and Glow, ApproxHPVM IR includes domain-specific information about tensor operations at the IR level. While our current implementation does not support domain-specific (semantics preserving) optimizations such as operator fusion, this support can be easily added in future versions. All the above systems are specific to machine learning, while the ApproxHPVM framework is meant as a general-purpose compiler for a wider range of tensor-based programs. Unlike these systems, ApproxHPVM system supports a wide range of algorithmic and hardware-specific approximations in a unified framework.

2.5 COMPILERS FOR HETEROGENEOUS SYSTEMS

HeteroIR [109] is a hardware-agnostic intermediate representation that includes abstractions for common high-level accelerator operations used across heterogeneous systems. The

IR supports optimizations such as tiling and memory transfer reductions, and later the compiler translates the code to architecture-specific backend models (CUDA or OpenCL).

Delite [110] is a compiler framework that facilitates the development of new domain-specific languages (DSLs). Delite allows developers to create custom DSLs embedded as part of Scala (a general-purpose programming language), and supports compilation to heterogeneous devices. Delite includes a suite of common optimizations that can be used by all DSLs. This includes standard compiler optimizations such as common subexpression elimination (CSE), dead code elimination (DCE), and code motion. Delite also allows developers to add new domain-specific optimizations. The Delite runtime supports execution of DSL operators on heterogeneous compute units including CPUs, GPUs, and accelerators.

Multi-level Intermediate Representation (MLIR) [107] is a compiler framework that facilitates the development of new intermediate representations (called dialects in MLIR), transformations and optimizations over these intermediate representations, and translation across representations operating at different abstraction levels. For instance, Tensorflow is a high-level domain-specific dialect in MLIR (among others), which is translated down to the LLVM IR dialect (including other dialects in the pipeline), and eventually down to hardware-specific dialects for CPUs, GPUs, and accelerators.

HPVM [79] is a compiler framework designed to address the performance and portability challenges of heterogeneous parallel systems. At its core is the HPVM IR which is a parallel program representation that uses hierarchical dataflow graphs to capture a diverse range of coarse- and fine-grain data and task parallelism including pipeline parallelism, nested parallelism, and SPMD-style (single program, multiple data) data parallelism. An HPVM program consists of a set of one or more distinct dataflow graphs, which describe the computationally heavy part of the program that is to be mapped to accelerators, and host code that can initiate the execution and wait for the completion of the dataflow graphs. Nodes in the HPVM dataflow graph (DFG) represent units of computation, and edges between nodes describe explicit data transfer requirements between nodes. These abstractions allow HPVM to compile from a single program representation in HPVM IR to diverse parallel hardware targets such as multicore CPUs, vector instructions, and GPUs. ApproxHPVM builds on and extends HPVM with approximate computing abstractions, runtime support for software and hardware-specific approximation techniques, an approximation tuning framework, and new code generation backends that target approximate accelerators.

While these compiler frameworks provide useful abstractions for targeting heterogeneous systems, they do not support abstractions for approximate computing, and do not target approximate accelerators or hardware-specific approximation choices.

2.6 APPROXIMATION TECHNIQUES

2.6.1 Approximate Hardware Accelerators

Recently, many machine learning accelerators are being designed in industry and academia. The core idea that enables these accelerators to be more energy-efficient than general-purpose processors is that they exploit the common computational and communication patterns found in ML applications. Popular commercial accelerators at the time of writing this document include Google TPUs [50], Google Coral (TPUs for edge) [111], Nvidia NVDLA [51], and Intel Movidius [52] among others. Both cloud and edge TPUs (Google Coral), and NVDLA provide support for high-throughput INT8 and FP16 reduced precision operations as available hardware approximations knobs. Intel Movidius includes support for high throughput tensor operations in FP16.

Research studies also include many ML-specific accelerator designs. The DianNao project proposed small footprint accelerators that provided high-throughput machine learning computations. Esmaeilzadeh et al [27, 1] proposed the neural processing unit (NPU) that uses analog computation circuitry to accelerate neural network computations. Moreover, their work showed that general purpose applications can be algorithmically transformed into a neural network computation, thereby facilitating execution on an NPU. Eldrige et al [112] proposed neural-network based accelerators for approximating the results of commonly-used mathematical functions including `cos`, `sin`, `exp`, `log`, and `pow`. They train the accelerator on a dataset of inputs and corresponding output values and then use neural network approximation to accelerate the computations at runtime. Their results show speedup over the `glibc` implementations.

The PROMISE accelerator [23] employs in-memory, low signal-to-noise ratio (SNR) analog computation on the bit lines of an SRAM array to perform faster and energy efficient matrix operations, including convolutions, dot-products, vector adds, and others. PROMISE consumes 3.4-5.5x less energy and has 1.4-3.4x higher throughput than application-specific custom digital accelerators, which are themselves known to be orders of magnitude better in terms of energy-delay product than CPUs and GPUs. The PROMISE accelerator instruction set has a parameter `swing` voltage, which controls the bit-line voltage swing in the accelerator and allows a trade-off between accuracy and energy. The `swing` parameter can take up to seven different values giving seven choices for approximation tuning. Higher voltage modes imply higher energy-consumption (higher dynamic power) and lower computational error, while lower voltage leads to lower energy-consumption but also higher computational error. In our ApproxHPVM and ApproxTuner work, we use PROMISE as an example of a

domain-specific accelerator that provides approximation knobs. We believe our framework is more broadly applicable to the wide range of emerging approximate accelerator platforms.

2.6.2 Software Approximations

Many studies have proposed novel software approximation techniques that reduce execution time and/or energy. Some of these include task skipping [113, 114, 115], loop perforation [56, 32, 116], approximate function substitution [104, 73, 35, 33], dynamic knobs [40] (dynamically changing function version), reduction sampling [35, 33, 117], tuning floating-point operations [19, 118], and approximate parallelization [39, 34, 33, 119]. These techniques have been shown to work well across a variety of application domains resilient to small errors. ApproxHPVM is developed with the goal of supporting various kinds of approximations in one unified framework. We allow developers to add custom approximations in our framework and leverage the same approximation-tuning components that apply to currently-supported approximations.

2.6.3 Domain-specific approximations for Machine Learning

LeCun et al. originally proposed *DNN weight pruning*; a technique for pruning unimportant connections from neural networks [45]. LeCun et al. presented an analytical approach to remove features with low saliency and using network retraining to fine-tune the weights and recover the accuracy loss (due to pruning). Their technique prunes parameters using the second-order derivative of the objective function (e.g., mean squared error) with respect to the parameters.

More recently, Han et al. proposed the *Deep Compression* [43] that includes support for a) pruning DNN weights with low magnitude (i.e., close to 0), b) quantizing to lower bit representations, and c) Huffman encoding for compressing weights. The authors show that pruning alone reduces the number of connections by 9X to 13X; quantization reduces the number of bits that represent each connection from 32 to 5. The optimizations work together to reduce the storage requirement of neural networks by 35X to 49X without any noticeable accuracy degradation. The performance and energy improvements range from 3x to 7x.

Pruning arbitrary DNN weights leads to non-structured random connectivity resulting in irregular memory accesses that are particularly inefficient on architectures specialized for parallel processing (e.g., GPUs). Due to this, unstructured weight pruning can even lead to slowdowns [120, 121, 122, 123]. Wen et al. propose a alternate pruning approach called *structured sparsity* [120]; technique that prunes groups of weights (consecutive bytes in

memory). Structured pruning that removes entire output channels from convolution filters has shown to provide speedups of up to 5x on CPUs and 3x on GPUs, with no loss in accuracy [120].

While these machine-learning specific optimizations are not currently supported in the ApproxHPVM framework, our system can work in combination with these existing techniques to achieve even higher speedups and energy reductions. We also performed a Preliminary experiment (Section 6.3) that shows ApproxTuner can also apply well to pruned networks (i.e., the performance improvements add up).

2.7 ANALYTICAL TECHNIQUES FOR APPROXIMATION TUNING

Sakr et al. proposed an analytical technique [47] to determine numerical precision assignment for individual DNN layers, while ensuring that end-to-end accuracy stays within user-specified bounds. The key idea is to leverage weight and activation gradients (obtained from standard back-propagation) since they capture the sensitivity of the final output with respect to the individual weights and activation tensors. Their technique assigns lower bit-width precision to activations and/or weights with lower gradient values and higher precision to activations and/or weights with higher gradient values. Lin et al. proposed a similar analytical method to estimate the error impact of different precision levels [124], and uses it to quantize layer weights and activations. Both these methods retrain the neural network to fine-tune the weights and reduce the accuracy degradation due to quantization. Srivastava et al [23] used Sakr's method [47] to analytically compute the minimum voltage swing levels for each DNN layer computation running on the analog-compute accelerator PROMISE. In future work, we aim to reduce the need for binary invocations and simulations (as part of autotuning) by using a combination of analytical modelling and empirical techniques.

CHAPTER 3: CHALLENGES IN APPROXIMATION TUNING

The impact on the quality of service (QoS) of a program can be arbitrarily high if approximation-selection is done naively. This is because some operations (or subcomputations) are error-tolerant while others are very sensitive to perturbations, and hence must not be approximated. Mapping the less error-tolerant operations to approximations can result in significant degradation of the end-to-end application result quality. Therefore there is a need for an intelligent approximation-mapping phase that chooses which operations to approximate and also tunes the level of approximation for each operator (e.g., low, medium, high approximation). Moreover, different kinds of approximations (e.g, loop perforation, barrier elision, function substitution) can have very different accuracy and performance impact for a particular operation/computation and hence the approximation choice must also be tuned at a per-operation basis.

In practice, a realistic application (e.g., a neural network or a combination of an image processing pipeline and an image classification network) can make use of multiple approximation techniques for different computations in the code, each with its own parameters that must be tuned, to achieve the best results. For example, our work shows that for the ResNet-18 convolutional neural network, which contains 22 tensor operations, the best combination (considering accuracy-performance tradeoff) is to use three different approximations with different parameter settings in different operations. A major open challenge is *how to select, configure, and tune the parameters for combinations of one or more approximation techniques*, while meeting *end-to-end requirements* on energy, latency, and accuracy. Several specific challenges arise in meeting this goal:

3.1 DIVERSE RANGE OF HETEROGENEOUS SYSTEMS

"Heterogeneous systems" refer to hardware platforms composed of multiple compute units with different microarchitectures that communicate over some shared interface (e.g., common global shared memory). One real-world example are mobile SoCs that include CPUs for general purpose tasks, GPUs for graphics and machine learning, and additional specialized accelerators for audio/video decoding, computer vision tasks, and DSP among other tasks. Different types of compute units are optimized for different goals, e.g., CPUs are suitable for sequential processing tasks, while GPUs are more optimized for parallel tasks (provide higher throughput). This diversity in hardware facilitates performance and power efficiency

but introduces challenges for compilers that must target different ISAs, optimize code for each kind of processor, and maximize local and overall hardware utilization.

Compute units of different hardware types (e.g., CPUs, GPUs, accelerators) and across different generations of the same hardware type (e.g, different generations of Nvidia GPUs) provide different approximation options and also differing accuracy-vs-performance tradeoffs [70]. For instance, Nvidia Volta GPUs with tensor cores [125] provide support for high-throughput INT8 tensor operations (e.g. matrix multiplication), while Pascal generation GPUs only support FP16 [126]. The performance tradeoffs for a single approximation choice also vary across different generations of the same hardware type, e.g., FP16 performance greatly varies across Nvidia GPUs with some providing speedups as high as 2x, while others only having only functional support for FP16 with throughput even lower than FP32 [127].

From the approximation-tuning perspective, domain-specific accelerators are particularly interesting since they support hardware-specific approximations (e.g. analog compute accelerators [23], low-precision ML accelerators [50, 51, 52]) that can offer orders-of-magnitude performance and energy improvements. To maximize overall performance and energy improvements, a compiler framework must be able to map to these efficient hardware primitives.

3.2 SOURCE CODE PORTABILITY

Today application developers write code in a variety of different languages including but not limited to C/C++, C#, Python, Java, Rust, Swift, Bash, Julia, Matlab, R, Ruby among many others. To facilitate ease of programming and productivity, domain-specific extensions such as Tensorflow [128], Keras [129], Pytorch [130], MxNet [131] are commonly used for developing machine learning codes. For parallel programming, OpenMP [132], CUDA [133], OpenACC [134], Intel TBB [135], Data Parallel C++ [136] are currently popular.

Porting an existing application to use a different programming language, adding language extensions, or adding developer annotations to facilitate optimizations is generally undesirable (though sometimes unavoidable) since: a) it requires a significant time investment from software developers, and b) can introduce bugs. Given these challenges, an important goal for compilers and libraries is to support *source code portability*; using the same unmodified (or only slightly modified) sources across different hardware types, Additionally, compiler frameworks should abstract hardware-specific details and automatically choose appropriate hardware-specific knobs and approximations without requiring source-code annotations/changes.

3.3 OBJECT CODE PORTABILITY

Since software applications (especially for mobile phones and IoT devices) are expected to cater to a range of different devices, *software portability is an important requirement*, not just at the source-code level but also the ability to *ship* software that can execute on a wide range of systems. Applications for both desktop and mobile (e.g., smartphone or tablet) systems are almost always shipped by application teams to end-users in a form that can execute on multiple system configurations (e.g., with different vector ISAs or GPUs). GPUs, for example, provide virtual instructions sets, e.g., PTX [71] or HSAIL [72], to enable software to be shipped as "virtual object code" that is translated to particular hardware instances only on the end-user's system. A critical goal for real-world use of such approaches is to enable software to be shipped as *portable* virtual object code, while deferring the hardware-specific aspects of accuracy-performance-energy optimizations to be performed after shipping [73] (e.g., on the end-user's device or on servers in an app store).

3.4 MANUAL TUNING IS CHALLENGING

Manually tuning approximation knobs for an application is challenging since it requires developers to have insights on the accuracy and performance impact of different approximation knobs, and identify regions of code that are amenable or susceptible to approximation. Source-level annotations for specifying parallel regions is a commonly-employed strategy since developers usually have insights on which regions of code can be parallelized without violating program semantics. For instance, programmers can annotate parallel loops using OpenMP [132] extensions that declare the absence of any loop-carried dependencies. Unlike extensions for parallelism, it is not immediately clear from the source code which sections of code are more error-tolerant and what approximation choices are likely to provide highest benefits. This is because: a) approximation impact is often input-dependent and must be viewed with respect to some representative data inputs, b) approximation impact can vary across invocations due to presence of stochastic operators (e.g., random noise), c) the performance benefits of approximation knobs greatly vary across operations (even operators of the same type) depending on input parameters, e.g., for a convolution operation these are: strides, padding, dimensions of input and weight tensors among others.

Manual tuning is also hard since realistic programs usually have many tunable components which creates a large space of possible settings. This challenge is detailed next.

3.5 LARGE TRADEOFF SPACE COMBINING MULTIPLE APPROXIMATIONS

The variety of software and hardware approximations with many accuracy-performance and accuracy-energy trade-offs introduce a large search space of possible configurations spanning both software and hardware choices. A configuration setting (selected by compiler/runtime) is an assignment of an approximation knob setting for each operation/computation, and each operation can be mapped to multiple approximation choices. This results in an *exponential search space*, specifically K^N where `K` is number of unique approximation knobs (per-operation) and `N` is the operation count. For realistic programs, this results in a very large search space that is hard to navigate efficiently. In our work, for ResNet-50 with 53 convolution layers and 63 approximation knobs for each convolution layer, the total search space includes a total of $7e^{91}$ unique configurations. The available approximation knobs can include choice of precision (FP32, FP16, INT8) and alternate algorithm implementations for common high-level operations. For instance, convolution operators in neural networks can be replaced with more efficient implementations that use a subset of inputs (known as *input sampling*).

3.6 HIGH COST OF EMPIRICAL AUTOTUNING

As described above, the search space of configurations can be large enough that an performing exhaustive search is infeasible. A heuristic search that navigates a fraction of a large search space is a feasible alternative. For code-generation and optimization, compilers often employ *autotuning*: an optimization that uses heuristic search techniques (e.g., genetic algorithms, hill climbers, random search) to navigate a subset of configurations, and uses a (developer-specified) fitness function to assign a score that indicates the cost/profitability for each evaluated configuration [74, 36, 75]. *Empirical autotuning* is a variant of autotuning that uses empirical measurements (i.e., running the program binary) to determine the cost and profitability of a configuration point in the search space. For approximation-tuning, the attributes of interest are quality of service (QoS) measurements such as inference error in convolutional neural networks (CNNs), and Peak Signal to Noise Ratio (PSNR) for image filters, throughout, latency, and energy usage among possible others. Empirically measuring these quantities can be expensive, especially on low-end edge systems. For instance, running ResNet-50 on Nvidia Tegra TX2 [81] (a popular edge device) takes several minutes to process a batch of 10K images. Tuning over large search spaces can require thousands of iterations [75], and given that each iteration can consume multiple minutes/seconds, tuning can require weeks, even months for realistic kernels. Such high tuning cost is completely

infeasible for the edge and also undesirable on cloud systems where energy and/or monetary costs can become overwhelming. In our work, we find that approximation-tuning via empirical tuning takes days (on a server-class machine) for some benchmarks - 1.5 days for VGG16 and 11 days for ResNet50. These long tuning times motivate the need for efficient tuning.

3.7 OPTIMIZATION CHOICE DEPENDS ON RUNTIME CONDITIONS

Optimization choices can be made ahead-of-time (before code generation) or done dynamically at runtime. Static optimization is suitable for scenarios where conditions mostly remain constant (e.g., a non-interactive batch processing task on the cloud) and re-optimizing parameters is rarely required. Dynamic optimization capabilities are important in deployments where runtime conditions are likely to keep changing. Few examples of such changing conditions are:

- **Varying system workloads.** Increasing system load can cause applications to be unresponsive or miss deadlines (important to meet for real-time tasks). These slowdowns can be counteracted with dynamically increasing approximation levels to help achieve the desired level of application performance (with some accuracy loss).
- **Low power modes.** Battery powered devices (e.g., mobile phones, IoT devices) activate low-power modes when the battery is running low. Low-power modes involve power gating some of the available compute cores (CPU and/or GPU), and reducing processor clock frequencies to reduce dynamic power usage [76, 77] (among other settings). These settings impose system slowdowns.
- **Changing QoS requirements.** The nature of the operating environments can also impact the required level of QoS (Quality of Service). For instance, a autonomous robot navigating in a dark environment may require a higher accuracy mode compared to a robot navigating in daylight.

3.8 APPLICATION-SPECIFIC END-TO-END EFFECTS OF APPROXIMATION

Most studies in approximate computing literature use high-level aggregate metrics for measuring application QoS. For instance, systems that tune machine learning programs use the loss in *inference accuracy* as the metric to tune, while systems for image processing benchmarks use *PSNR* or *SSIM* (Structural Similarity Index Measure). *These high-level metrics*

serve as a relative measure of quality degradation but do not quantify how the end-to-end applications goals are affected by the loss in accuracy. Realistic programs usually include multiple kernels and sub-computations from different application-domains, e.g, CNNs, statistical computations, probabilistic programs, low-level control code among many other kinds of tasks. For instance, a CNN used for perception tasks is one of the many components in an autonomous robot navigation system that potentially includes MPC (Model Predictive Controllers), LIDAR processing code, FFT and Viterbi processing components among others. How the accuracy degradation in one component affects the other interacting components, and how it impacts the end-to-end application goals (e.g., user-experience) are important considerations. I believe that this is an open research question and I plan to investigate this further as part of future work.

CHAPTER 4: APPROXHPVM: A PORTABLE COMPILER IR FOR ACCURACY-AWARE OPTIMIZATIONS

4.1 INTRODUCTION

We propose **ApproxHPVM**, a unified compiler IR and framework that provides ease of programming and object-code portability – and does in a fully automatic manner:

- Programmers only have to specify *application-level*, *end-to-end* error tolerance constraints, and ApproxHPVM can use this information to optimize and schedule programs on a heterogeneous system containing multiple approximation techniques; and
- ApproxHPVM enables software portability by using a hardware-agnostic, accuracy-aware compiler IR and virtual ISA, and by partitioning the accuracy-energy-performance optimizations into a hardware-agnostic stage and a hardware-specific stage, where software can be shipped between the two stages.

The ApproxHPVM system takes as input a program compiled to the ApproxHPVM Intermediate Representation (IR), and end-to-end quality metrics that quantify the acceptable difference between approximate and non-approximate outputs. It generates final code that maps individual approximable computations within the program to specific hardware components and specific chosen approximation techniques, while satisfying end-to-end constraints with high probability and attempting to minimize execution time and maximize energy savings under those constraints. To our knowledge, *no previous system achieves both* full automation from end-to-end application-level quality specifications, *and* support for multiple approximation mechanisms (on one or more heterogeneous compute units). Moreover, previous systems do not provide object code portability.

ApproxHPVM solves three key technical challenges to achieve these goals:

- For applications with multiple approximable computations, it automatically translates end-to-end error specifications to individual error specifications and bounds per approximable computation, while statistically guaranteeing with high probability that the end-to-end specifications are satisfied.
- It automatically determines how to map approximable computations to a variety of compute units and multiple approximation mechanisms, including efficient special-purpose accelerators designed to provide improved performance with lower accuracy guarantees.

- It optionally provides object code portability by decoupling the overall mapping and compilation problem into a hardware-independent autotuning stage and a subsequent hardware-dependent mapping stage.

The portability is optional because it does not always come for free: the optimization choices may sometimes be suboptimal compared to a single, end-to-end and hardware-specific strategy, as we show in our experiments. ApproxHPVM supports either strategy, and so the unified, hardware-specific strategy can be used when portability is not a requirement. An additional benefit of the two-stage mapping strategy is that the autotuning can be very slow, while the second, hardware-specific stage is extremely fast, essentially just a small number of table lookups.

ApproxHPVM solves these challenges in a domain-specific manner, through a number of key features. The ApproxHPVM Intermediate Representation (IR) is an extension of Heterogeneous Parallel Virtual Machine (HPVM), a retargetable compiler infrastructure and portable virtual ISA for heterogeneous parallel systems [79]. HPVM itself is built on LLVM [80], and can use LLVM compiler passes and code generators for individual tasks. These design choices allow ApproxHPVM to target diverse heterogeneous parallel systems, and also to serve as a *fully self-contained*, portable virtual ISA that can be shipped and mapped to a variety of hardware configurations. ApproxHPVM defines a set of approximable domain-specific operations as part of the IR, which enables the compiler to identify approximable computations, and also to define hardware-independent but domain-specific error metrics as attributes of those operations. The initial domain supported in our work is tensor computations, which are general enough to support a number of important application domains such as neural networks and image processing. (Although this approach focuses on domain-specific operations, our design and general strategy allow the specifications to be extended to generic low-level instructions.) It uses an autotuner with randomized error injection to translate end-to-end specifications to individual error bounds per approximable computation in a hardware-*independent* manner, while satisfying end-to-end application metrics. It uses a simple lookup table per approximation method per IR operation to perform the second-stage hardware-*dependent* manner very fast.

Specifically, we make the following key contributions:

Retargetable Compiler IR and Virtual ISA with Approximation Metrics: We show how to capture hardware-agnostic approximation metrics in a parallel compiler IR, while preserving retargetability across a wide range of heterogeneous parallel hardware. Moreover, the IR can serve as a hardware-agnostic virtual ISA, and so software can be

shipped between the two optimization stages to achieve virtual object code portability for approximate computing applications.

Hardware-agnostic Accuracy Tuning: Given an end-to-end user-provided quality metric (e.g., reduced inference accuracy or PSNR for images), our hardware-independent accuracy tuner computes the corresponding accuracy requirements for individual IR operations that can satisfy the end-to-end goal. In this way, programmers need not understand the details of approximation techniques in the underlying system.

Accuracy-aware Hardware Scheduling: The second stage maps individual tensor operations to specific target compute units and to specific approximation options within those compute units, by taking into account the error tolerance of operations and the accuracy guarantees provided by the target compute unit. This mapping is a fast table-lookup, trained using offline accuracy profiling of kernels running on the hardware.

Evaluation on Target Platform: To evaluate the efficacy of ApproxHPVM, we study 9 DNN benchmarks and 5 image processing filters, using two different accuracy thresholds for each: 1% and 2% decreases in inference accuracy for the DNNs, and 20dB and 30dB loss of PSNR for the image processing filters. We use the NVIDIA Jetson TX2 mobile SoC [137], which has 8GB of shared memory between ARM cores and an NVIDIA Pascal GPU. We extend the platform by adding a simulated version of a (fully programmable) Machine Learning accelerator called PROMISE, which has previously been shown to provide orders of magnitude energy and throughput benefits for a wide range of vector dot-product operations commonly used in ML kernels [23]. The combined platform provides 9 hardware settings to trade-off energy and accuracy for each tensor operation: FP32 or FP16 on the GPU and 7 voltage swing levels on PROMISE. Executing all operations on the GPU with FP32 precision is considered the exact case. Our results show that ApproxHPVM can successfully assign different tensor operations to different compute units (GPU or PROMISE) with different approximation options, achieving speedups of 1-9x and energy reductions of 1.1-11.3x, while statistically guaranteeing the specified accuracy metrics with 95% probability.

4.2 ApproxHPVM INTERNAL REPRESENTATION AND SYSTEM WORKFLOW

Figure 4.1 shows the overall ApproxHPVM workflow. The primary input is a program written using high-level abstractions of the Keras library [138], a popular open-source library for deep neural networks on TensorFlow. Our frontend translates a Keras source program to the ApproxHPVM IR. The second input is a programmer-specified end-to-end quality threshold, a domain-dependent parameter. For the neural network domain, we use the

acceptable loss in final classification accuracy and for image processing pipelines, we use desired PSNR of the approximated output.

ApproxHPVM's overall goal is to map the computations of the program to the compute units on a target system, along with selected approximation parameter values on each compute unit, so that the program outputs satisfy the specified end-to-end accuracy. We decompose this mapping problem into a hardware-agnostic first stage and a hardware-specific second stage.

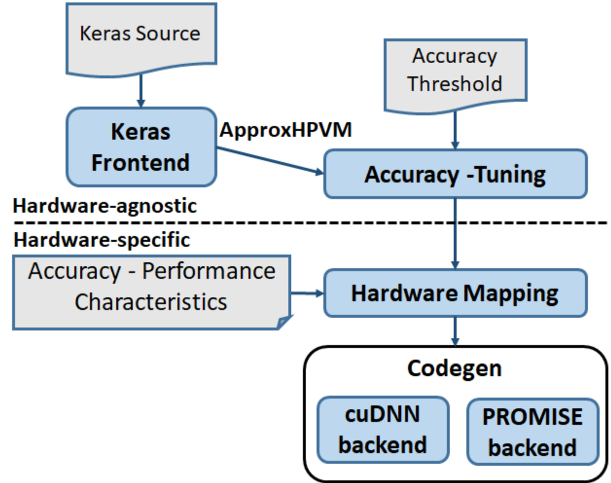


Figure 4.1: ApproxHPVM System Workflow

The hardware-agnostic accuracy-tuning phase takes an end-to-end quality threshold and computes the error tolerance for individual ApproxHPVM operations, adding these requirements in the IR. This phase guarantees that if these error tolerances for individual operations are (independently) satisfied, then the end-to-end accuracy specification will also be satisfied with some high probability, e.g., 95%. The output of this stage is hardware-agnostic ApproxHPVM code, which is legal LLVM and can optionally be used as a virtual instruction set to ship the code as "virtual object code" to one or more targets [80]. For each target, a (static) accuracy-aware hardware mapping phase chooses which compute units should execute each tensor operation, and optimizes any approximation parameters available on each compute unit to minimize energy and/or maximize performance, while satisfying the *individual accuracy constraints on each operation*. Finally, the code generation phase leverages the hardware-specific backends to generate code for each compute unit. In our work, we build a) a GPU backend that targets the cuDNN and cuBLAS libraries, which are optimized for high-level tensor operations, and b) a PROMISE backend that targets a library that performs optimized tensor computations on the PROMISE hardware simulator. The GPU can use FP32 or FP16 values for the network weights and bias values, where FP32 is considered exact. PROMISE can only use 8-bit integers, and offers a choice of seven voltage values to further trade off accuracy for energy (see Section 4.3.2).

ApproxHPVM is inspired by and builds on HPVM [79], a dataflow graph compiler IR for heterogeneous parallel hardware. We extend the HPVM IR to support execution of basic linear algebra tensor computations and to specify accuracy metrics for each operation. We

Table 4.1: Tensor intrinsics in the ApproxHPVM representation.

Tensor Intrinsic	***Description***
i8* **@hpvm.tensor.mul**(i8* lhs, i8* rhs)	Performs a matrix multiply operation on the input tensors.
i8* **@hpvm.tensor.conv**(i8* input, i8* filter, i32 stride, i32 padding)	Applies a convolution filter on input tensor with given stride and padding.
i8* **@hpvm.tensor.add**(i8* lhs, i8* rhs)	Element-wise addition on input tensors.
i8* **@hpvm.tensor.reduce_window**(i8* input, i32 reduction_type, i32 window_size)	Performs a (configurable) reduction operation over a specified window size on the input tensor.
i8* **@hpvm.tensor.relu**(i8* input)	Element-wise relu activation function.
i8* **@hpvm.tensor.clipped.relu**(i8* input)	Element-wise clipped relu activation function.
i8* **@hpvm.tensor.tanh**(i8* input)	Element-wise tanh activation function.
i8* **@hpvm.tensor.map**(i8* function, i8* input1, i8* input2, ...)	Zips multiple equal-shaped tensors and applies function element-wise.
i8* **@hpvm.tensor.reduce**(i8* function, i8* input, i32 axis)	Performs a reduction operation along an axis of the input tensor.

first briefly discuss the HPVM IR in the next subsection, and then describe our extensions to it.

4.2.1 Background: HPVM Dataflow Graph

HPVM [79] is a framework designed to address the performance and portability challenges of heterogeneous parallel systems. At its core is the HPVM IR which is a parallel program representation that uses hierarchical dataflow graphs to capture a diverse range of coarse- and fine-grain data and task parallelism including pipeline parallelism, nested parallelism, and SPMD-style (single program, multiple data) data parallelism. We showed that these abstractions allow HPVM to compile from a single program representation in HPVM IR to diverse parallel hardware targets such as multicore CPUs, vector instructions, and GPUs. ApproxHPVM leverages the existing infrastructure of HPVM and extends it to compile to our heterogeneous approximate computing platform.

An HPVM program consists of a set of one or more distinct dataflow graphs, which describe the computationally heavy part of the program that is to be mapped to accelerators, and host code that can initiate the execution and wait for the completion of the dataflow graphs. Nodes in the HPVM dataflow graph (DFG) represent units of computation, and edges between nodes describe explicit data transfer requirements between nodes. Each DFG node can be instantiated multiple times at runtime, effectively enabling its computation to be performed multiple times. The dynamic instances of a DFG must be independent, i.e., safe to execute in parallel. Different nodes can access the same shared memory locations by passing pointers along edges, which is important for modern heterogeneous systems that

support cache-coherent global and partial shared memory. A node can begin execution once it receives a data item on every one of its input edges.

The HPVM DFG is hierarchical, i.e., a node can itself contain an entire DFG. Such nodes are called internal nodes, while other nodes are leaf nodes. Computations in leaf nodes are represented by ordinary LLVM scalar and vector instructions, and can include loops, function calls, and memory accesses. The `@hpvm.createNode` instruction is used to create a node in the HPVM DFG, and the `@hpvm.createEdge` is used to connect an output of a node to an input of another node in HPVM. The `@hpvm.bind.input` instruction is used to map an incoming edge of an internal node to the input of a node in the internal DFG of this node. `@hpvm.bind.output` instructions serve a similar purpose for outgoing edges.

The execution of a DFG is initiated by a "launch" operation in host code, and is asynchronous by default. The host can block to wait for outputs from a DFG, if desired.

4.2.2 Tensor Operations in ApproxHPVM

Domain-specific languages such as Tensorflow and Pytorch allow for improved programmer productivity and have gained wide-spread adoption. Accordingly, compilers such as XLA for TensorFlow [106] and TVM for MxNet [67] are beginning to support efficient mapping of high-level domain-specific abstractions to heterogeneous parallel compute units including CPUs, GPUs, FPGAs, and special-purpose accelerators, and to run-time libraries like cuDNN or cuBLAS.

A general-purpose parallel IR such as HPVM translates high-level operations into generic low-level LLVM instructions. However, such early lowering of domain-specific operations can result in loss of important semantic information that may be needed by a back end to target run-time libraries or domain-specific accelerators. Reconstructing the higher-level semantics after lowering is generally very difficult and sometimes infeasible.

Instead, we choose to incorporate high-level but broadly applicable operations into HPVM IR directly. In this work, we extend the HPVM IR representation with linear algebra tensor operations that allow for naturally expressing tensor-based applications. Tensors are used in a wide range of important domains, including mechanics, electromagnetics, theoretical physics, quantum computing, image processing and machine learning. For instance, convolutional neural networks may be expressed using generic linear-algebra operations. This design choice provides two essential benefits: a) it enables efficient mapping of tensor operations to special purpose hardware and highly optimized target-specific runtime libraries, such as cuDNN for GPUs, and b) it allows approximation analyses to leverage domain-specific infor-

```
define i8* @tensorConvNode(i8* %input, i8* %filter) {
  %result = call i8* @tensor.conv(i8* %input, i8* %filter, i32* %strides,
      i32* %padding)
  return i8* %result
}

define i8* @tensorAddNode(i8* %input, i8* %bias_weights) {
  %result = call i8* @tensor.add(i8* %input, i8* %bias_weights)
  return i8* %result
}

define i8* @tensorReluNode(i8* %input) {
  %result = call i8* @tensor.relu(i8* %input)
  return i8* %result
}

define void @DFG_root(i8* %W, i8* %X, i8* %B) { ; DFG Root node
  ; Creating DFG nodes
  %nodeConv = call i8* @hpvm.createNode(i8* @tensorConvNode)
  %nodeAdd = call i8* @hpvm.createNode(i8* @tensorAddNode)
  %nodeRelu = call i8* @hpvm.createNode(i8* @tensorReluNode)
  ; Creating data-flow edges between different DFG nodes
  call void @hpvm.createEdge(i8* %nodeConv, i8* %nodeAdd, 1, 0, 0, 0)
  call void @hpvm.createEdge(i8* %nodeAdd, i8* %nodeRelu, 1, 0, 0, 0)
  ; Binding the parent input to inputs of the leaf nodes
  call void @hpvm.bind.input(i8* %nodeConv, 0, 0, 0)
  call void @hpvm.bind.input(i8* %nodConv, 1, 1, 0)
  call void @hpvm.bind.input(i8* %nodeAdd, 2, 1, 0)
  ; Binding final DFG node output to parent node output
  call void @hpvm.bind.output(i8* %nodeRelu, 0, 0, 0)
}
```

Figure 4.2: Convolution layer sub-operations represented as ApproxHPVM tensor intrinsics. The dataflow nodes are connected through explicit dataflow edges using HPVM intrinsics.

mation, because the approximation properties, parameters, and analysis techniques usually are determined by properties of the domain.

Table 4.1 presents the list of tensor intrinsics introduced in ApproxHPVM. The tensor operations in ApproxHPVM are represented as calls to LLVM intrinsic functions (the same approach used by HPVM). The intrinsic calls appear to existing LLVM passes as calls to unknown external functions, so existing passes remain correct. For applications where all data-parallelism occurs via the tensor operations, the dataflow graph is only used to capture pipelined and task parallelism across nodes, while data-parallelism is captured by the tensor operation(s) within individual nodes.

Figure 4.2 presents a single neural network convolution layer encoded in ApproxHPVM. The encoding uses three tensor intrinsics: @tensor.conv, @tensor.add, and @tensor.relu. The *DFG_root* function is the root of the dataflow graph, and would be invoked by host code. The root node is an internal graph node, which creates the leaf nodes *tensorConvNode*, *tensorAddNode* and *tensorReluNode* (using *hpvm.createNode* calls) and connects the nodes through dataflow edges (using *hpvm.createEdge* calls). The leaf nodes invoke the tensor intrinsics to perform tensor computations on the input tensors. The output of the last node in the dataflow graph is connected to the output of the root node and is returned back to the caller.

4.2.3 Approximation Metrics in the IR

The second key feature of ApproxHPVM is the use of hardware-independent approximation metrics that quantify the accuracy of unreliable and approximate computations. We attach error metrics, defined below, as additional arguments to high-level tensor operations. Our design allows the specifications to be added to generic low-level instructions, but we do not use that in this work. To express the (allowable) difference between approximate and exact tensor outputs, we use vector distance metrics:

- Relative L_1 error: $L_1^e = \frac{L_1(A-G)}{L_1(G)}$ where $L_1(X) = \sum_i |x_i|$

  The numerator captures the sum of absolute differences between the approximate tensor output A and the golden tensor output G. The denominator is the L_1 norm of the golden output tensor, so that the ratio is the relative error.

- Relative L_2 error: $L_2^e = \frac{L_2(A-G)}{L_2(G)}$ where $L_2(X) = \sqrt{\sum_i x_i^2}$

  This is similar to the L_1^e norm, except that the numerator represents the Euclidean distance and the denominator uses the L_2 norm.

Note that the relative L_1 error and relative L_2 error are non-negative and lie in the the range $[0, +\infty)$. Figure 4.3 shows how the approximation metrics are represented in the compiler IR. The two approximation parameters for each tensor operation are attached as additional arguments to the respective intrinsic functions. While our current system only uses the two metrics described, our implementation and analyses can be easily extended to include additional approximation metrics.

```
define i8* @tensorConvNode(i8* %input, i8* %filter) {
  %result = call i8* @tensor.conv(i8* %input, i8* %filter, i32* %strides,
      i32* %padding, float %relative_l1, float %relative_l2)
  return i8* %result
}

define i8* @tensorAddNode(i8* %input, i8* %bias_tensor) {
  %result = call i8* @tensor.add(i8* %input, i8* %bias_tensor, float
      %relative_l1, float %relative_l2)
  return i8* %result
}

define i8* @tensorReluNode(i8* %input) {
  %result = call i8* @tensor.relu(i8* %input, float %relative_l1, float
      %relative_l2)
  return i8* %result
}
```

Figure 4.3: Tensor intrinsics annotated with accuracy metrics. The accuracy metrics L_1^e and L_2^e are passed as parameters to the intrinsic calls.

4.3 ACCURACY-AWARE MAPPING AND OPTIMIZATION

In this section, we describe the accuracy-aware mapping of computations to hardware compute units in the ApproxHPVM system. ApproxHPVM uses a hardware-agnostic accuracy tuning phase (Section 4.3.1) to determine per-operation accuracy requirements and an efficient accuracy-aware scheduler (Section 4.3.2) that maps the approximable components to hardware compute units and hardware-level system parameters.

4.3.1 Hardware-Agnostic Accuracy Tuning

The goal of hardware-independent accuracy tuning is to compute the accuracy requirements (represented by the L_1^e and L_2^e defined earlier) for each operation so that, *if the individual requirements are satisfied*, the user-provided end-to-end quality metric is met. For instance, a user may specify an acceptable classification accuracy degradation of 1%, allowing the tuner to lower the accuracy constraints on a tensor multiply operation by 10%. By computing the individual accuracy constraints, the tuner enables the hardware scheduler to map *individual* tensor operations to approximate hardware independently. This independence goal is a compromise: better energy efficiency or performance or both might be achieved if two or more operations were considered together in the second stage, but that would require a combinatorial optimization problem across all operations, compute units,

and approximation choices. Using independent decisions allows a much faster decision problem in the second stage.

Figure 4.4 describes the overall workflow of the accuracy-tuning phase. The heart of the accuracy-tuner is an autotuning search that uses statistical error injection to model potential run-time errors and directly executes the program on a standard GPU to measure the end-to-end accuracy vs. the expected ("golden") output. If the hardware target was known, the autotuner could skip the (artificial) error injection and instead execute the program on the target with a selected mapping and selected approximation settings to estimate the error. Instead, the autotuner uses a hardware-agnostic error model and objective function to perform the search. Since our tuner uses statistical error injection to validate the accuracy constraints, the autotuner enforces the accuracy threshold to be met with a certain tunable success rate (fixed at 95% in our experiments).

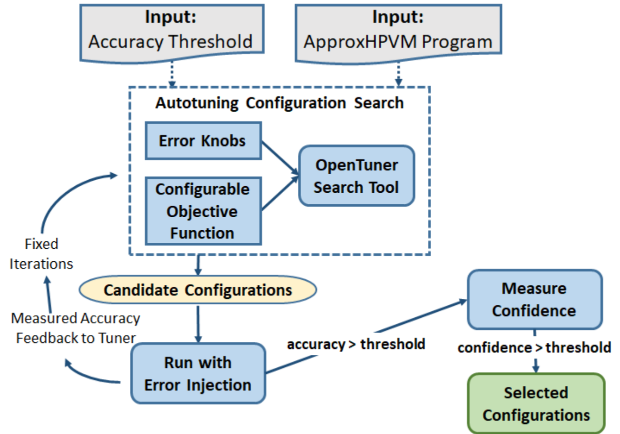


Figure 4.4: Hardware-agnostic accuracy-tuning workflow.

Autotuning framework. Considering realistic applications with multiple tunable operations, the size of the search space makes exhaustive search intractable. To enable efficient search, we use OpenTuner [75], an extensible framework for building domain-specific autotuners. OpenTuner allows users to configure a domain-specific search space and specify a custom objective function. Prior work has shown that OpenTuner provides promising results with enormous search spaces, exceeding 10^{3600} possible configurations. Leveraging OpenTuner, we build our custom accuracy tuner that tunes the error knob for each tensor operation while minimizing an objective function. The objective functions we use are described below. In our experiments, we are able to extract high-quality configurations while searching through only a small subset of the full search space. For our experiments, we run OpenTuner for a total of 1000 iterations, where each iteration generates a unique configuration.

Inputs. The accuracy-tuner takes as input an end-to-end accuracy threshold T, and the target program compiled to ApproxHPVM, and generates a set of configurations, defined below.

Error Injection. The accuracy tuner works by injecting errors into the outputs of individual tensor operations and predicting their impact on end-to-end accuracy. The key

```
Configuration: {
    hpvm.tensor.mul: 5,
    hpvm.tensor.add: 6,
    hpvm.tensor.tanh: 4
}
```

Figure 4.5: Example configuration output of the hardware-agnostic autotuner. The numeric value corresponds to error budget assigned to each tensor operation.

to making our decomposed strategy work is to do this analysis in a hardware-independent manner. We achieve this by using a simple, hardware-agnostic error model, where errors in the outputs of tensor values X[i] are injected as: $X[i] = X[i] \times (1 + E \times \mathcal{N}(0, 1))$. The parameter E provides a simple, linear error model optimized by the autotuner, producing hardware-agnostic error values that can be mapped by the back-ends to hardware-specific approximation choices.

In our analysis, we choose the value of E from 1 to 15, increasing linearly, thereby linearly increasing the L_1^e and L_2^e metrics. In our experiments, we tune the values of the L_1 error norm ranging from 0.5% to 40%.

Search Space and Configurations. A configuration in the autotuning search consists of a value of the error parameter E assigned to each of the tensor operations in the target program. By selecting this value at each operation, the autotuner controls the magnitude of error injected into each tensor operation. For instance, one configuration for the code in example 4.2 is shown in Figure 4.5.

For every configuration generated by the accuracy tuner, the final accuracy is empirically evaluated by running the program with the tuned level of error injection. If the measured end-to-end accuracy is below the threshold, the configuration is rejected. Otherwise, the configuration is saved as a candidate configuration.

Measuring Success Rate. Since we used statistical error injection to evaluate candidate configurations, our end-to-end "guarantee" can be probabilistic, at best. Consistent with prior work in optimistic parallelization [39], we use statistical testing to determine the probabilistic guarantee provided by each candidate configuration. The statistical accuracy test runs each candidate configuration with additional random error injection trials, where the magnitude of error is determined by the selected error knobs. We treat each run as a Bernoulli trial which succeeds if the execution satisfies the user-defined accuracy threshold T and fails otherwise. For measuring the success rate $R_{success}$, we execute each configuration for 100 runs and accept a configuration if the statistical accuracy test has a minimum success rate of 95%.

Hardware-independent objective functions. All remaining candidate configurations satisfy the end-to-end accuracy threshold with a minimum success rate R_{min}, and can be ranked to achieve our goal of maximizing energy efficiency and performance. We use a hardware-independent objective function to do so, using operation count as a proxy for execution time, and assuming that higher allowable errors yield better energy efficiency. Thus, we heuristically compute a cost function C_{Total} of a candidate configuration as:

$$C_{Total}(config) = \sum_{i=0}^{N} C(op(i), E(i)) \tag{4.1}$$

The total cost of a configuration is defined as the sum of the cost of each operation at the selected error knob. The individual operation costs must increase with execution time and decrease as allowable error increases. We include three alternative objective functions, where we use the error knob E as a proxy for error:

$$C_1(op, E) = \frac{N_c(op)}{\log E} \qquad C_2(op, E) = \frac{N_c(op)}{E} \qquad C_3(op, E) = \frac{N_c(op)}{E^2} \tag{4.2}$$

Here, $N_c(op)$ computes the total count of multiplication and add operations performed as part of the higher-level tensor operation, *op*. Note that the more expensive operations (higher $N_c(op)$) are likely to prefer a higher error value, which prefers scheduling these operations for more approximate hardware, in the hope of achieving higher overall benefits. The autotuner generates configurations once for each of the objective functions. We ship the IR with the top 10 configurations for each of the three objective functions, allowing the hardware scheduler to select the best performing configuration for the specific deployment.

4.3.2 Accuracy-Aware Scheduling

Given an application in ApproxHPVM along with error norms L_1^e and L_2^e for each tensor operation in the ApproxHPVM dataflow graph, the goal is to choose the right hardware setting for each operation. We envision that multiple software and hardware approximate computing techniques will be available as a choice for each operation. The scheduler attempts to find a configuration that maximizes energy efficiency and performance while meeting the individual accuracy constraints per operation.

Accuracy-aware scheduling presents these challenges: **(C1)** given error metrics, selecting a hardware knob corresponding to each operation. **(C2)** Maximizing energy and/or performance based on an objective function. **(C3)** Incurring low runtime cost, thereby enabling dynamic scheduling.

Approximate Computing Hardware: In this work, we map and compile tensor operations onto two hardware compute units: an NVIDIA GPU and a programmable mixed-signal accelerator for machine learning called PROMISE [23]. Computations are offloaded to an NVIDIA GPU using the cuDNN library, which supports both 32-bit (FP32) and 16-bit floating point (FP16) operations. FP16 computation reduces execution time and energy by 1.5-4x compared to FP32, at the cost of reduced accuracy [139, 140].

The PROMISE accelerator employs in-memory, low signal-to-noise ratio (SNR) analog computation on the bit lines of an SRAM array to perform faster and energy efficient matrix operations, including convolutions, dot-products, vector adds, and others. As shown in [23], PROMISE consumes 3.4-5.5x less energy and has 1.4-3.4x higher throughput than application-specific custom digital accelerators, which are themselves known to be orders of magnitude better in terms of energy-delay product than NVIDIA GPUs. The PROMISE accelerator instruction set has a parameter `swing` voltage, which controls the bit-line voltage swing in the accelerator and allows a trade-off between accuracy and energy. The `swing` parameter can take up to seven different values giving us seven choices for the PROMISE hardware, denoted in this paper as P1, P2, ..., P7, in increasing order of voltage and decreasing error.

For our hardware platform including a GPU and PROMISE, we have 9 different choices (FP16, FP32 on GPU and P1-P7 on PROMISE) for mapping each tensor operation. Figure 4.6 shows the speedup, energy reduction, and accuracy of 3 hardware settings – P1, P7, and FP16. These are measured for a matrix multiplication of matrix M_1 of size $5000 \times K$ and matrix M_2 of size $K \times 256$, where $K \in \{2^8, 2^9, \ldots, 2^{15}\}$. The matrices are initialized with random values from uniform distribution $\mathcal{U}(0, 1)$.

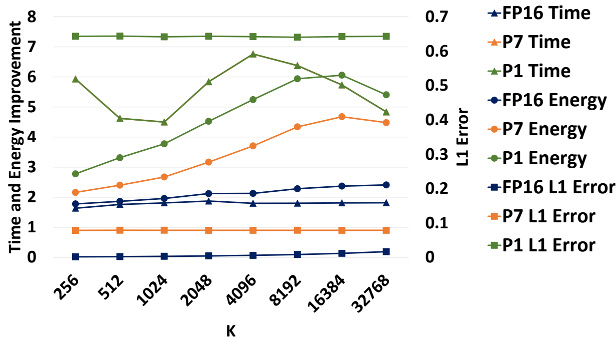


Figure 4.6: Time and energy improvement, and L_1^e for the following hardware knobs: FP16, P7, and P1. PROMISE is faster and less accurate than FP16, which is faster and less accurate than FP32. P1 and P7 Time curves overlap since execution time is constant across different swing values.

For readability, we do not show curves for P2-P6, which follow the same trends as P1 and P7. The left Y-axis shows speedup and energy reduction over FP32. The right Y-axis depicts error in the computation by showing L_1^e of the matrix multiplication for each hardware setting.

The graph shows that the L_1^e of different hardware settings remains constant for different values of K. FP16 is most accurate followed by P7 and P1 in that order. FP16 is slower than P7 and P1 for all K and also consumes more energy than P7 and P1, except for an anomaly for $K = 256, 512$. As the swing voltage level decreases in PROMISE, the energy consumption reduces, hence P1 has lower energy than P7. However, the execution time remains constant across the different swing values in PROMISE, hence P7 and P1 time curves overlap.

Mapping L_1^e and L_2^e metrics to Hardware Settings:

We generated similar graphs to Figure 4.6 for all ApproxHPVM tensor operations for each hardware setting FP16, P7, P6, ... P1. These operations include tensor multiplication, addition, convolution, activations (tanh, relu, clipped relu), and window reductions (max-pooling, avg-pooling, min-pooling). We used this data to find the maximum L_1^e and L_2^e constraints tolerable by each hardware setting for each operation. We observed that the L_1^e and L_2^e metrics for each hardware setting had very little variation across different tensor sizes, thereby serving as a useful metric for measuring errors in tensor operations. Our backend maps a tensor operation to the least accurate hardware setting that meets the L_1^e and L_2^e constraints of the operation. Since the mapping from individual operation L1 error and L2 error to hardware knobs is merely a table lookup operation, hardware scheduling is an inexpensive step. This makes our hardware specific mapper very lightweight, which in the future can be used for dynamic scheduling or for SoC design space exploration. Moreover, our approach is extensible to other hardware compute units since it merely requires adding a mapping from the hardware-agnostic approximation metrics to the hardware-specific approximation knobs of the target hardware.

4.4 METHODOLOGY

4.4.1 Platform

For our experiments, we assume a modern System-on-Chip (SoC) architecture with CPUs, GPUs, and accelerators that communicate via main memory. The specific system we model is an NVIDIA Jetson TX2 developer kit [137], augmented with the PROMISE programmable machine learning accelerator [23]. PROMISE does not exist as real hardware, and we instead obtained the PROMISE simulator from its authors and extended it with a memory timing and energy model.

To model the overall system, one approach would be to use a cycle-accurate integrated CPU-GPU-PROMISE simulator, but this is impractical due to several prohibitive limitations of current state-of-the-art GPU simulators such as GPGPU-Sim. First, they do not support

Table 4.2: System parameters for TX2 and PROMISE.

TX2 Parameters	
CPU Cores	6
GPU Cores	2
GPU Frequency	1.12 GHz
DRAM Size	8 GB
DRAM Bandwidth	58.4 GB/s peak; 33 GB/s sustained
DRAM Energy	20 pJ/bit
PROMISE Parameters	
Banks	256×16 KB
Frequency	1 GHz

dynamic linking of libraries such as cuDNN and cuBLAS. Moreover, they do not support newer PTX instructions required by these libraries. Second, regardless of library support, simulator execution is orders of magnitude slower than real hardware, which makes running real world DNNs and realistic data sets infeasible.

Instead, we opted for a split approach to model the SoC. We ran the GPU tensor operations on the real GPU and the PROMISE tensor operations on the PROMISE simulator. Since all communication between different system agents occurs via main memory, reads/writes to/from main memory sufficiently model communication between the CPU, GPU, and PROMISE. For instance, if a particular layer executes on the GPU and the next layer executes on PROMISE, we just assume that PROMISE obtains all the required data from main memory. Therefore, this approach accurately models the behavior of a modern SoC architecture.

For our GPU experiments, we used an NVIDIA Jetson TX2 developer kit [137]. This board contains the NVIDIA Tegra TX2 SoC [81], that contains a Pascal-family GPU with 2 Streaming Multiprocessors (SMs), each with 128 CUDA cores (FP32 ALUs). The board has the same system architecture as our target SoC. Table 5.2 lists the relevant characteristics of both Tegra TX2 and the PROMISE simulator. Finally, due to our split approach, the functional and timing aspects of our experiments were split as well.

4.4.2 Functional Experiments

To verify the functional correctness of our generated binaries and to measure the end-to-end accuracy of each network with different configurations, we used the GPU in tandem with PROMISE's functional simulator. If a layer was mapped to the GPU, the corresponding tensor operations were executed on the GPU. If a layer was mapped on PROMISE, it was

offloaded to PROMISE's functional simulator. Consequently, the final result was the same as it would be if these operations were all executed on a real SoC containing both a GPU and PROMISE. Since the PROMISE simulator adds Gaussian random error to each run, we use statistical testing to measure the fraction of program runs that satisfy the end-to-end quality metric - we call this $R_{success}$. We ran each configuration 200 times to obtain the mean and standard deviation of the classification accuracy, and $R_{success}$ of the configuration.

4.4.3 Timing Experiments

GPU. To measure the execution time and energy of tensor operations on the GPU, we built a performance and energy profiling tool. While an application is running, the tool continuously reads GPU and DRAM power from Jetson's voltage rails via an I2C interface [141] at 1 KHz (1 ms period). Furthermore, it associates each GPU tensor operation with a begin and end timestamp pair. Once the application has finished execution, execution time is calculated by simply taking the difference between the begin and end timestamp of the tensor operation. Then, energy is calculated by integrating the power readings using 1 ms timesteps.

We used this tool to obtain per-tensor operation time and energy for both FP32 and FP16 for each benchmark. To obtain reliable results for each operation, we did 100 runs per benchmark, and used the average time and energy. The coefficient of variation was less than 1% after 100 runs. Instead of rerunning an operation on the GPU each time we ran a configuration, we collected these results once per benchmark and tabulated them. Then, whenever a particular tensor operation or network layer was mapped to the GPU, we obtained the required values from this lookup table.

PROMISE. Using the functional simulator obtained from the authors of PROMISE, we built a timing and energy model for PROMISE. Since the compute and memory access pattern of PROMISE is known *a priori* based on the operation being performed, a cycle-accurate simulator is not required and analytically computing both time and energy is sufficient. This analytical model first calculates the mapping of input matrices to PROMISE's banks, and then computes the time and energy of 1) loading the data from main memory, 2) performing the computation, and 3) writing data back to main memory. We extended the baseline PROMISE design with a programmable DMA engine (pDMA) [142, 143]. PROMISE operates on INT8 data and requires a data layout transformation, both of which are handled by pDMA. All the required data is loaded into PROMISE before starting the computation.

For the compute model, we used the pipeline parameters obtained from the authors of PROMISE [144]. For the main memory model, we empirically measured peak sustained

Table 4.3: Description of Evaluated Benchmarks.

(a) DNN Benchmarks, corresponding datasets, layer count, and classification accuracy with FP32 baseline.

Network	Dataset	Layers	Accuracy
FC-4	MNIST	4	93.72%
LeNet	MNIST	4	98.7%
AlexNet	CIFAR-10	6	79.16%
AlexNet v2	CIFAR-10	7	85.09%
ResNet-18	CIFAR-10	22	89.44%
VGG-16-10	CIFAR-10	15	89.41%
VGG-16-100	CIFAR-100	15	66.19%
MobileNet	CIFAR-10	28	83.69%
Shallow MobileNet	CIFAR-10	14	88.4%

(b) Image Processing Benchmarks and corresponding datasets. The Description shows the composition of filters that forms the particular image pipeline.

Filter	Dataset	Description
GEO	Caltech 101	Gaussian-Emboss-Outline
GSM	Caltech 101	Gaussian-Sharpen-MotionBlur
GEOM	Caltech 101	Gaussian-Emboss-Outline-MotionBlur
GEMO	Caltech 101	Gaussian-Emboss-MotionBlur-Outline
GSME	Caltech 101	Gaussian-Sharpen-MotionBlur-Emboss

bandwidth and energy per bit on our Jetson TX2 development board to ensure that both PROMISE and the GPU used the same memory system. The DRAM energy reported by PROMISE and the energy measured on Jetson TX2 was highly correlated, validating our model.

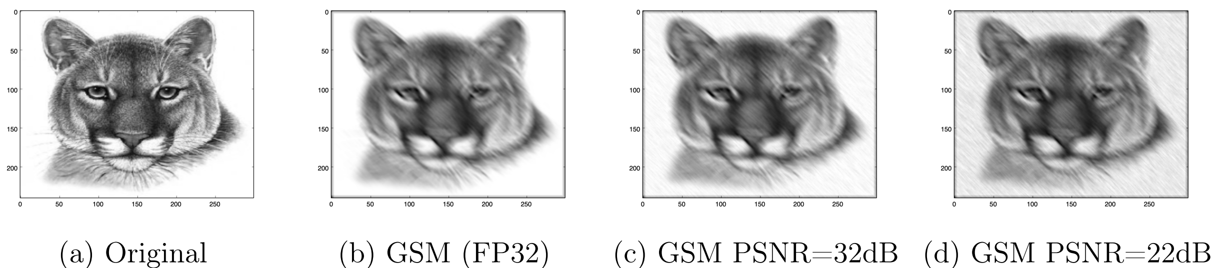
Integration. Similar to the functional experiments, we obtained the total time and energy for a network by summing the time and energy of each layer. If the layer was scheduled on PROMISE, PROMISE's timing and energy simulator was invoked to get the time and energy. If the layer was scheduled on the GPU, a lookup was performed on the FP32/FP16 time and energy tables that were generated after profiling. If consecutive operations required a different precision, quantization was performed and its time and energy overhead was added to the total. PROMISE performed quantization internally while a CUDA kernel performed quantization for the GPU.

4.4.4 Benchmarks

Our evaluation includes 9 DNN benchmarks and 5 image processing pipelines, detailed in Table 5.1 and Table 4.3b, respectively.

DNN Benchmarks. We include a range of different convolutional neural networks for 3 different datasets: MNIST [145], CIFAR-10, and CIFAR-100 [146]. The MNIST dataset includes 60K grey-scale images of handwritten digits 0 through 9. The CIFAR-10 dataset contains 60K $3 \times 32 \times 32$ sized color images belonging to 10 classes, 6K images per class. CIFAR-100 includes 60K $3 \times 32 \times 32$ sized color images belonging to 100 distinct classes, with 600 images belonging to each class. For each of the three datasets, the dataset is divided into 50K images for training and 10K for inference. The inference set is divided equally into calibration and validation sets (5K each). The calibration set is used for the autotuning phase that identifies approximable computations, and the validation set is used for evaluating the performance, energy, and accuracy of each autotuned configuration (combination of hardware knobs). We use popular DNN benchmarks including LeNet[147], AlexNet [148] (reference implementation [149]), ResNet-18 [12], VGG-16 [11] (reference implementation [150]), MobileNet [151], and Shallow MobileNet [151]. We trained VGG-16 for both CIFAR-10 and CIFAR-100 since it has been shown to provide relatively good end-to-end accuracy on both the datasets [150]. We also created a variant of Alexnet (called Alexnet v2) that includes an extra convolution layer (a total of 6 convolution layers) and provides approximately 6% higher end-to-end accuracy. We include MobileNet which is an efficient DNN model with respect to both performance and model size. We also include a shallow version of the original MobileNet architecture (similar to the shallow model proposed in the original work) called Shallow MobileNet that includes 14 layers as opposed to 28 layers in the full MobileNet model. Shallow MobileNet provides approximately 5% higher accuracy on CIFAR-10 compared to the full MobileNet model, because for CIFAR-10, which involves small images and only 10 classes, the larger network is prone to overfitting. We also include a 4 layer fully-connected DNN, called FC-4, trained on the MNIST dataset.

Image Processing Benchmarks. We also include 5 convolution-based image processing benchmarks (Table 4.3b). We construct these benchmarks by including different combinations of commonly-used image filters: Gaussian (G), Emboss (E), Outline (O), MotionBlur (M), and Sharpen (S). At the IR level, the filters are represented as tensor convolutions, with the exception of Emboss which is a convolution followed by a bias add operation. To evaluate the filters, we used the Caltech 101 dataset [152] that includes a set of 9145 images. The dataset includes a mix of small and large images, so we resized all the images to 240×300 pixels to allow running the filters on a batch of images. We converted the color images to



(a) Original (b) GSM (FP32) (c) GSM PSNR=32dB (d) GSM PSNR=22dB

Figure 4.7: Sample output from GSM (Gaussian-Sharpen-MotionBlur) benchmark. 4.7a: Original image; 4.7b: GSM baseline output (FP32 without approximation); 4.7c, 4.7d: GSM output approximated at $PSNR_{30}$ and $PSNR_{20}$ respectively.

grey-scale since our cuDNN-based backend does not support convolution on separate RGB channels. For evaluation, we split the images into two sets of 4572 images for calibration and validation. The calibration set is used by the autotuning step, and the validation set is used to evaluate the average PSNR and violation rate of each configuration provided by the autotuner.

4.4.5 Quality Metrics

For the DNN benchmarks, we studied an accuracy loss of 1% ($Loss_{1\%}$) and 2% ($Loss_{2\%}$). $Loss_{1\%}$ refers to an accuracy degradation of 1% with respect to the baseline and $Loss_{2\%}$ refers to an accuracy degradation of 2% compared to the baseline. The baseline uses FP32 for all computations with no approximation.

For the image processing benchmarks, we use PSNR to quantify the error in the output of the processed image in comparison to the baseline. We use two PSNR loss thresholds of 30db ($PSNR_{30}$) and 20db ($PSNR_{20}$). (Quality loss of about 20-25dB is considered to be acceptable in lossy situations, such as wireless transmission [153, 154].) To illustrate the visual impact, Figure 4.7 shows the impact of such losses between for the output of the GSM pipeline applied to a sample image, at "exact" (FP32 precision on the GPU), and with additional losses of $PSNR_{30}$, and $PSNR_{20}$ due to approximations. The GSM pipeline introduces noticeable blur *without approximations*. PSNR 32.2 dB only causes a small perceptible difference in the image, while reducing PSNR to 22.3 dB results in an observable visual difference, but still acceptable in many situations. While autotuning these filter pipelines, we also measure the violation rate that quantifies the fraction of images that do not meet the target PSNR. Consistent with prior work in approximating video filters [42], we use 5% as an acceptable threshold for the violation rate. Similar to the DNN benchmarks, the baseline uses FP32 for all filter computations.

4.5 EVALUATION

This section presents an evaluation of ApproxHPVM. Our evaluation seeks to answer the following research questions:

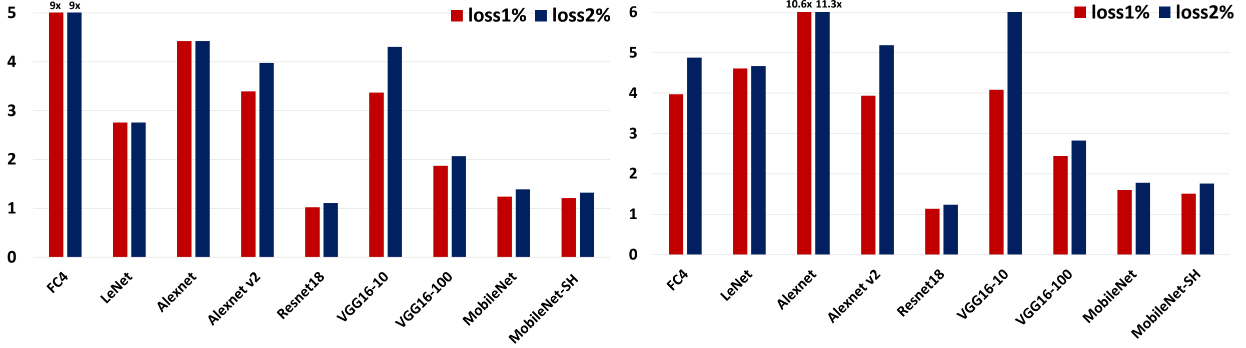
1. What are the performance and energy benefits provided by the ApproxHPVM framework, which uses only application-level end-to-end error tolerance specifications?
2. Does increased error threshold allow for increased performance and energy benefits?
3. Can ApproxHPVM techniques apply to different end-to-end quality metrics (PSNR, Accuracy)?
4. How does two-stage autotuning using a hardware-agnostic first stage compare against direct hardware-specific autotuning?
5. How is performance affected by changes in hardware configuration?

4.5.1 Performance and Energy Evaluation

DNN Benchmarks. Figure 4.8 shows the aggregate results for all nine DNN benchmarks for $Loss_{1\%}$ and $Loss_{2\%}$ experiments.

For each network, we report the results for the best performing configuration with respect to the energy-delay (ED) product. The configuration is a set of hardware knobs that control the level of approximation. In our system, the knobs for approximation are FP16 (16-bit FP), and the 7 distinct swing voltage levels of the PROMISE accelerator. To give more insight into how our accuracy-aware scheduler maps DNN layers to hardware, Figure 4.8c shows the best configuration selected by the ApproxHPVM autotuner and hardware mapper. Each entry shows the number of DNN layers mapped to each distinct hardware setting. For instance, for $Loss_{1\%}$, ApproxHPVM maps 1 layer in LeNet to FP16, 1 layer to P7 (swing voltage level 7), and 2 layers to P4 (swing voltage level 4). In Section 4.5.2 we compare how well the configurations attained by the hardware-agnostic tuner perform in comparison to the optimal (in scenarios where determining the optimal is feasible) and against hardware-specific autotuning (where computing optimal is not feasible).

Figure 4.8 shows that nearly all configurations given by the ApproxHPVM framework improve upon the FP32 baseline. The performance improvement ranges from 1.02x (ResNet $Loss_{1\%}$) to 9x (FC-4 $Loss_{2\%}$). The energy reduction ranges from 1.14x (ResNet $Loss_{1\%}$) to 11.3x (AlexNet $Loss_{2\%}$). Most networks obtain from 1.5x–4x (1.5x–5x) improvements in



(a) Speedup

(b) Energy Reduction

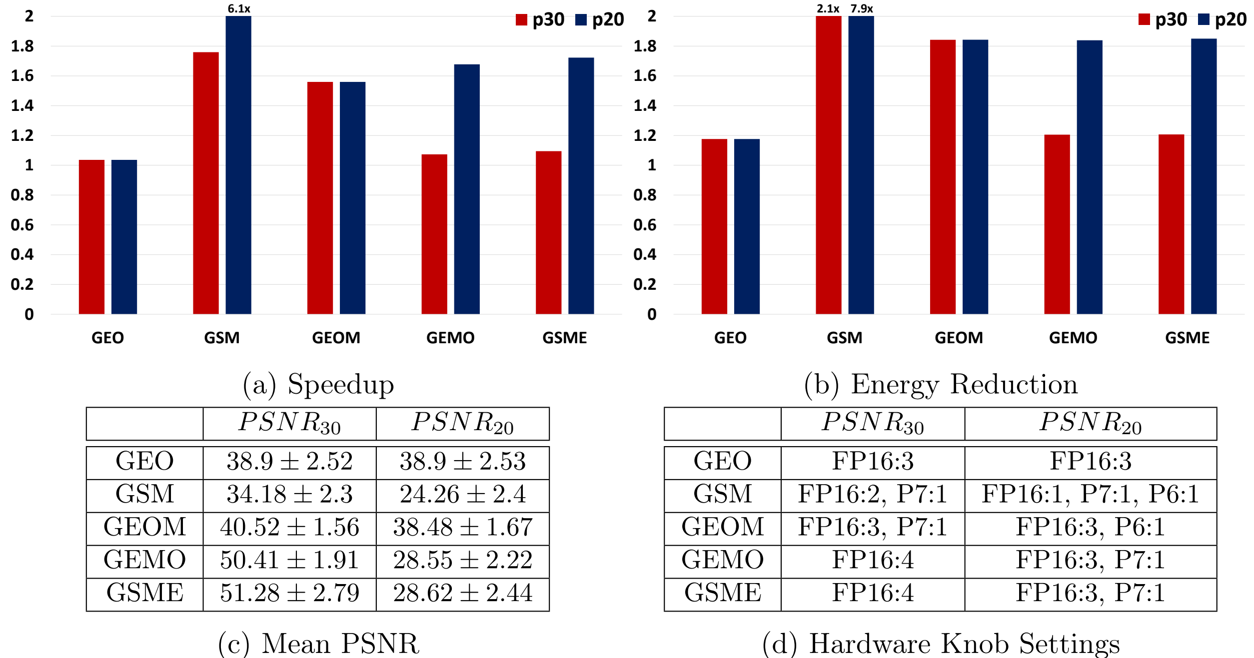
	Mean Classification Accuracy			Hardware Knob Settings	
	FP32	$Loss_{1\%}$	$Loss_{2\%}$	$Loss_{1\%}$	$Loss_{2\%}$
FC-4	93.72	93.47 ± 0.2	92.41 ± 0.2	P7:3, P6:1	P4:3, P5:1
LeNet	98.7	98.28 ± 0.1	98.26 ± 0.1	FP16:1, P4:2, P7:1	FP16:1, P4:3, P5:1
AlexNet	79.16	78.51 ± 0.2	78.43 ± 0.2	FP16:1, P6:3, P7:2	FP16:1, P7:1, P6:1, P4:3
AlexNet v2	85.09	84.52 ± 0.1	84.45 ± 0.1	FP32:3, P7:4	FP32:2, FP16:1, P7:2, P6:2
ResNet-18	89.44	88.84 ± 0.1	88.54 ± 0.1	FP32:1,FP16:18,P7:3	FP32:1, FP16:16, P7:5
VGG-16-10	89.41	88.64 ± 0.1	87.77 ± 0.2	FP32:4,FP16:4,P7:7	FP32:1,FP16:4,P7:2,P6:1,P5:2,P4:5
VGG-16-100	66.19	65.97 ± 0.2	64.97 ± 0.1	FP32:1,FP16:9,P7:5	FP32:1, FP16:8, P7:3, P6:3
MobileNet	83.69	83.05 ± 0.1	82.35 ± 0.1	FP16:25, P7:3	FP16:22, P7:6
MobileNet-SH	88.4	88.08 ± 0.1	86.74 ± 0.2	FP16:12, P7:2	FP32:1, FP16:9, P7:3, P6:1

(c) Mean Classification Accuracy and Hardware Knob Settings

Figure 4.8: Speedup and energy reduction (over baseline) of all nine DNNs for $Loss_{1\%}$ and $Loss_{2\%}$ experiments (higher is better). 4.8a: Speedup. 4.8b: Energy reduction. 4.8c: Mean accuracy $\pm$ standard deviation, and hardware knob settings showing the number of layers mapped to each type of hardware knob (for both $Loss_{1\%}$ and $Loss_{2\%}$) where FP32: 32-bit floating point on GPU; FP16: 16-bit floating point on GPU; Px: PROMISE with swing x.

performance (energy). Figure 4.8c shows the mean and standard deviation of the end-to-end accuracy for the different DNNs for both $Loss_{1\%}$ and $Loss_{2\%}$ experiments. The final accuracy of each configuration is within the corresponding allowable accuracy loss threshold of 1% and 2%.

We observe most of the DNNs to be amenable to using the approximation mechanisms for multiple layers. Figure 4.8c shows that a number of layers in the DNNs are often mapped to the PROMISE accelerator, which provides significant performance and energy improvements over the GPU. AlexNet obtains the highest energy benefit (11.3x energy reduction) since 5 of the total 6 layers are mapped to PROMISE in both $Loss_{1\%}$ and $Loss_{2\%}$ experiments. Note that when moving from $Loss_{1\%}$ to $Loss_{2\%}$, more layers in AlexNet could utilize lower PROMISE voltage levels (that provide higher benefits), thereby providing an extra 6% energy reduction. For VGG-16-10, the difference is more significant with 48% extra energy reduction when moving from $Loss_{1\%}$ to $Loss_{2\%}$. For MobileNet and Shallow MobileNet, we observe energy reductions ranging from 1.5x to 1.78x and performance improvements rang-



(a) Speedup

(b) Energy Reduction

	$PSNR_{30}$	$PSNR_{20}$
GEO	38.9 ± 2.52	38.9 ± 2.53
GSM	34.18 ± 2.3	24.26 ± 2.4
GEOM	40.52 ± 1.56	38.48 ± 1.67
GEMO	50.41 ± 1.91	28.55 ± 2.22
GSME	51.28 ± 2.79	28.62 ± 2.44

(c) Mean PSNR

	$PSNR_{30}$	$PSNR_{20}$
GEO	FP16:3	FP16:3
GSM	FP16:2, P7:1	FP16:1, P7:1, P6:1
GEOM	FP16:3, P7:1	FP16:3, P6:1
GEMO	FP16:4	FP16:3, P7:1
GSME	FP16:4	FP16:3, P7:1

(d) Hardware Knob Settings

Figure 4.9: Speedup and energy reduction (over baseline) of all 5 image processing benchmarks for $PSNR_{30}$ (p30) and $PSNR_{20}$ (p20) thresholds. 4.9a: Speedup. 4.9b: Energy reduction. 4.9c: Mean PSNR $\pm$ standard deviation. 4.9d: Hardware knob settings shows the number of convolution layers mapped to each type of hardware knob (for both $PSNR_{30}$ and $PSNR_{20}$) where FP32: 32-bit floating point on GPU; FP16: 16-bit floating point on GPU; Px: PROMISE with swing x.

ing from 1.21x to 1.39x. Note that the MobileNet DNN has been specifically optimized for improved performance, and Shallow MobileNet achieves even better performance while also preserving high accuracy. Our results show that by exploiting approximations we can achieve further gains even for such optimized models. For ResNet-18, hardware-agnostic tuning only provides marginal performance (up to 1.1x) and energy gains (up to 1.2x) for both $Loss_{1\%}$ and $Loss_{2\%}$. In Section 4.5.2, we show that for ResNet, the hardware-specific tuner also provides small improvements, showing that this DNN architecture is not very amenable to the approximation choices offered by our hardware platform. Other approximation techniques may achieve better gains for ResNet.

Image Processing Benchmarks. Figure 4.9 shows the aggregate results for all 5 image processing benchmarks. Figures 4.9a, 4.9b, 4.9c show the performance improvement, energy reduction, and mean PSNR of $PSNR_{30}$ and $PSNR_{20}$ experiments.

Figure 4.9 shows that nearly all configurations given by the ApproxHPVM framework achieve performance and energy benefits. The performance improvement ranges from 1.04x (GEO $PSNR_{30}$) to 6.1x (GSM $PSNR_{20}$). The energy reduction ranges from 1.2x (GEO

$PSNR_{30}$) to 7.9x (GSM $PSNR_{20}$). Note that for GSM, we see a further energy reduction of 3.7x and performance improvement of 3.5x when reducing the quality metric from $PSNR_{30}$ to $PSNR_{20}$, as the autotuner and mapper are able to offload the Gaussian and MotionBlur filters to PROMISE. We see similar trends for the GEOM, GEMO, and GSME benchmarks when reducing the quality threshold to $PSNR_{20}$ (Figure 4.9d).

The FP16 computations also provide both improved performance and energy efficiency though not as significant as the PROMISE accelerator. For instance, for the GEO benchmark the autotuner could not map any operation to PROMISE for either $PSNR_{30}$ or $PSNR_{20}$ but we still achieve a small 4% performance and 18% energy improvement with FP16. For the GEO filter benchmark, we do not observe any benefits when moving from $PSNR_{30}$ to $PSNR_{20}$ since none of the filters could be mapped to a precision level lower than FP16 i.e mapping any one filter (of the total 3) to PROMISE would produce images below $PSNR_{20}$.

Note that the selected hardware knobs in 4.8c and 4.9d vary across DNNs with differing approximation settings for PROMISE swing levels and GPU precision. This reinforces the need for accuracy-tuning on a per-DNN basis since each DNN has different error-tolerance characteristics.

AlexNet Layer-wise Analysis. To gain more insight into the benefits observed by ApproxHPVM, we perform a layer-wise analysis of the $Loss_{2\%}$ AlexNet configuration. Figure 4.10 shows the layer-wise breakdown of performance and energy improvement for this configuration. The autotuner identifies that Conv1 cannot be run on PROMISE because it is highly error prone and mapping it to PROMISE results in an unacceptable accuracy loss. Therefore, Conv1 is mapped to FP16, which only provides a 1.3x performance improvement and a 2.1x energy reduction. For the other five layers, ApproxHPVM identifies these as being error-tolerant and maps them to PROMISE. The speedup ranges from 5x (FC1) to 11x (Conv2), and the energy reduction ranges from 5.4x (FC1) to 30x (Conv2) for these five layers. Compared to the performance improvement, the energy reduction is higher due to the fact that convolution layers are

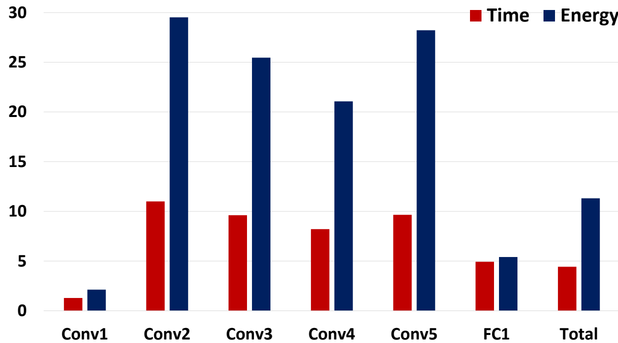


Figure 4.10: Speedup and energy reduction over baseline for all six layers of AlexNet ($Loss_{2\%}$). Conv1 cannot be mapped to PROMISE and is executed using FP16. It thus observes the smallest benefit and also significantly reduces the overall time and energy improvement.

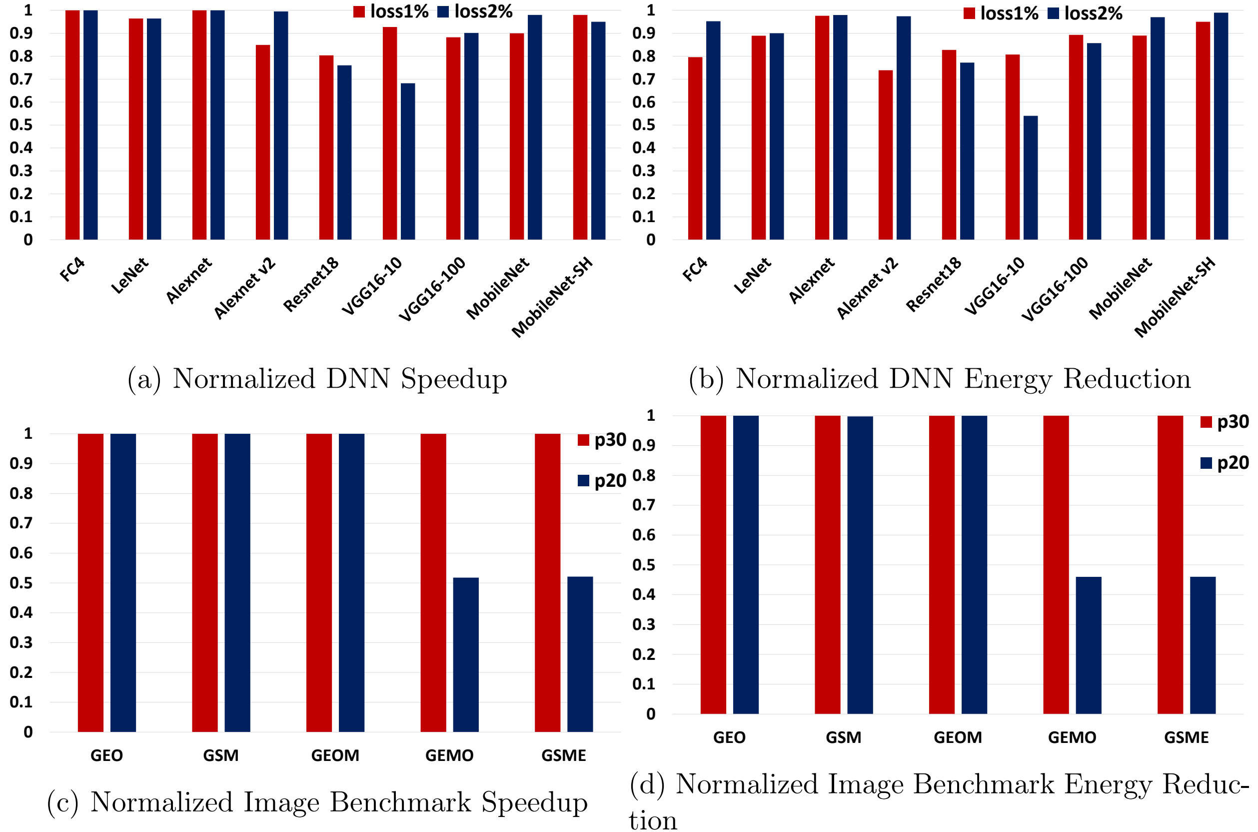
typically memory bound, and costly memory accesses constitute most of the total energy in FP32. The high data locality provided by the specialized storage of PROMISE drastically reduces that cost. Overall, ApproxHPVM achieves a speedup of 4.4x and an energy reduction of 11.3x for AlexNet with only a 2% loss in accuracy.

Statistical Accuracy Tests. For all benchmarks, we measured the success rate $R_{Success}$ of our configurations by measuring the fraction of program runs where the measured end-to-end metric (accuracy degradation or PSNR violation rate) satisfies the programmer-specified threshold. For configurations generated by the autotuner, 94% configurations passed the statistical accuracy test by achieving $R_{success} > 95\%$ on the target hardware (PROMISE+GPU). This shows that our hardware-agnostic approach yields configurations which benefit from approximation and yet remain within the programmer-specified constraint when evaluated on the target hardware platform.

To evaluate the effectiveness of our hardware-agnostic autotuning approach, we compare against hardware-specific autotuning. While hardware-agnostic autotuning provides several benefits including portability and facilitating efficient dynamic scheduling, it can potentially lead to sub-optimal mappings. In the hardware-agnostic autotuning phase, error budgets are allocated to individual tensor operations without knowledge of the approximation choices offered by a specific hardware platform. This can potentially waste error budgets when the autotuner allocates an error budget to an operation that cannot be approximated on the target hardware. To conduct the hardware-specific autotuning experiment, we use OpenTuner (as in the hardware-agnostic tuner) to directly search over the set of hardware knobs that maximize performance and energy while satisfying the end-to-end quality metric. For our target platform, these hardware knobs include FP32, FP16, and the 7 levels of PROMISE, providing a total of 9 hardware knobs for each operation. For a fair comparison with hardware-agnostic tuning, we use the same number of Autotuner iterations - 1000. For FC-4, LeNet, and the 5 image benchmarks, we found the configuration search space to be tractable for exhaustive search and hence compare against exhaustive search. The performance and energy of hardware-agnostic (HA) normalized to hardware-specific (HS) is shown for both DNN and image processing benchmarks in Figures 4.11a, 4.11b, 4.11c, and 4.11d.

4.5.2 Hardware-Agnostic vs Hardware-Specific Tuning

For the DNN benchmarks, we observe that on average hardware-agnostic tuning is within 10% performance and 15% energy of hardware-specific tuning. For VGG16-10 $Loss_2$, we observe a significant difference of 32% in performance and 46% in energy. The difference occurs since hardware-specific tuning is able to map the most expensive convolution layers to



(a) Normalized DNN Speedup

(b) Normalized DNN Energy Reduction

(c) Normalized Image Benchmark Speedup

(d) Normalized Image Benchmark Energy Reduction

Figure 4.11: Speedup and Energy reduction of hardware-agnostic autotuning compared to hardware-specific autotuning. All bars are normalized to the corresponding best hardware-specific autotuning configuration.

PROMISE while the hardware-agnostic tuner was able to map fewer layers to the accelerator. The particular reason for the difference is that the hardware-agnostic tuner allocates error budgets to tensor operations without knowing the approximation mechanisms on the target hardware. For instance, the hardware tuner would allocate error budgets to *tensorRelu* and *tensorPooling* operations unaware that the PROMISE accelerator does not add error to these operations (since they are executed in the digital domain). The wasted error budget can sometimes result in missed opportunities for utilizing that error budget elsewhere. However, as majority of the results show, the difference between hardware agnostic and hardware-specific tuning is not significant in most cases. Interestingly, for FC-4 and LeNet, hardware-agnostic is within 5% and 10%, respectively, of the energy reduction achievable by the *optimal* configuration (given by the exhaustive search).

ResNet-18 only achieves a small performance improvement of up to 1.1x and energy reduction of up to 1.2x with hardware-agnostic tuning, and up to 1.5x performance improvement and 1.6x energy reduction with hardware-specific tuning. As Figure 4.8c shows, for

ResNet-18 most layers were found to be less error-tolerant by the autotuner and could not be mapped to PROMISE. Note that in an attempt to maximize opportunities for approximation, the hardware-agnostic tuner maps a number of ResNet layers to FP16. However, for certain layers we observe that FP16 provides slightly worse performance (and energy) compared to FP32, thereby neutralizing any benefits provided by FP16 computation. We found this to be an anomaly, since in general computations provide both performance and energy improvements on FP16.

For the image processing benchmarks, we compare against exhaustive search for all 5 benchmarks since the search space is tractable. Among the 5 different types of filters uses in the benchmarks, only Gaussian and MotionBlur can be offloaded to PROMISE because of the minimum vector length (64) imposed by PROMISE. Hence, the other 3 filters, Emboss, Sharpen, and Outline, have two hardware choices, FP16 and FP32. In 8 of the 10 total experiments ($PSNR_{30}$ and $PSNR_{20}$ for each image processing benchmark), the performance improvement matches that of the optimal configuration determined by exhaustive search. For energy, 5 of the 10 hardware-agnostic results match the optimal, while 3 are within **0.03%** of the optimal. For GEMO $PSNR_{20}$ and GSME $PSNR_{20}$, we see a significant difference in both performance (48%) and energy (54%) when compared to the optimal configuration. The large difference is observed because of a single sub-optimal decision where the hardware-agnostic tuner maps the first Gaussian filter to FP16, whereas the hardware-specific tuner maps it to PROMISE. The hardware-agnostic tuner is unaware that PROMISE cannot map small vector sizes to PROMISE and allocates error budget to the other filters (Emboss, Outline, Sharpen), thereby reducing the error budget that could be allocated to the Gaussian filter. Though hardware-agnostic tuning is sub-optimal in this specific example, the fact that it can distribute error budgets independent of the hardware makes it a more flexible choice when considering a variety of hardware platforms that may have very different characteristics.

Overall, our results show that hardware-agnostic autotuning performs reasonably well compared to hardware-specific autotuning and exhaustive search. We believe that hardware-agnostic autotuning is more flexible since it allows for shipping code with hardware-independent approximation metrics, which can in turn be used by a wide variety of hardware devices. Shipping application programs tuned for each unique hardware platform is infeasible in practice. Moreover, hardware-specific autotuning will be infeasible in scenarios where the hardware tuning takes an excessively long time, for instance design-space exploration in FPGA synthesis. The proposed hardware-agnostic approach also enables flexible dynamic scheduling where the target device can be chosen at runtime given the error tolerance of an operation and the accuracy guarantee provided by the target compute unit. In scenarios

where hardware platform details are known and hardware tuning is feasible in practice, the ApproxHPVM also facilitates such hardware-specific autotuning.

Non-optimality of hardware-agnostic tuning. The hardware-agnostic accuracy-tuning approach is suboptimal since error budgets allocated in the tuning phase may not be fully utilized when mapping to approximation knobs in hardware. For instance, the hardware-agnostic tuner may allocate an error budget to a tensor operation for which no approximate version is present on the target hardware platform. More generally, a tensor operation may only be able to use a fraction of its error budget, leaving the rest unused. In theory, wasted error budgets in one operation can be reapportioned to other operations that can utilize the error budget. Currently, we don't support such error reapportioning because the second (hardware-specific) mapping stage selects an approximation choice independently for each operation.

Another mode in which hardware-agnostic tuning is sub-optimal is that multiple IR operations with individual error budgets are merged into a single hardware operation in the back-end code generator, which requires assigning a single approximation option to all those operations. This in turn forces conservative choice of approximation that satisfies the error budget of all such operations. For example, in our hardware mapping phase for the PROMISE accelerator, multiple tensor operations are sometimes mapped to a single PROMISE operation. When selecting the voltage swing for the PROMISE operation, we make the conservative choice of choosing the least error budget allocated to each of the individual tensor operations. Such conservative choices waste the error budget for some of the operations. Notice that the hardware-specific autotuner can be constrained to avoid this problem because it can take into account the actual mapping of IR operations to hardware operations, while selecting the approximation choices. As part of future work, we will study techniques for composing error budgets allocated to individual tensor operations. Analyses for composing error budgets should, in turn, allow for more precise selection of hardware knobs.

4.5.3 Autotuning Times

We built our autotuner by leveraging the interface provided by the OpenTuner framework [75]. We ran our autotuning experiments on an NVIDIA V100 GPU with 5120 cores and 16GB HBM2 global memory. Both the cuDNN-based runtime (for running FP32, FP16) and the PROMISE simulator leverage the parallelism offered by the GPU. For both the hardware-agnostic (HA) and hardware-specific (HS) tuning experiments, we use 1000 iterations of the autotuner (searching over 1000 points in the search space). The tuning times

Table 4.4: Hardware-agnostic (HA) and Hardware-specific (HS) autotuning times (in hours).

Benchmark	**HA**	**HS**
FC-4	1.4	5.11
LeNet	4.4	5.9
AlexNet	16.5	16.8
AlexNet v2	17.6	15.2
ResNet-18	13.7	15.3
VGG-16-10	32.1	31
VGG-16-100	20.8	24.4
MobileNet	16.2	11.3
MobileNet-SH	11.4	8.2
GEOM	12.6	5.9
GEMO	8.4	3.8
GSME	11.2	4.9
GEO	7.7	0.3
GSM	10.8	0.7

in hours for hardware-agnostic and hardware-specific experiments for each benchmark are included in Table 4.4. Note that for FC-4, Lenet-5, and the five image filter benchmarks GEO, GSM, GEOM, GEMO, GSME, the hardware-specific phase does a fully exhaustive search since the search space is tractable for these benchmarks. The HS autotuning times for these benchmarks include the time for performing the exhaustive search. The HS tuning times for the image benchmarks are low given that the search space is small. Since only the Gaussian and MotionBlur filter could be mapped to PROMISE, the other filters can only map to 2 hardware choices - FP32 and FP16. For instance, for the GSM (Gaussian-Sharpen-MotionBlur) filter, HS exhaustive search only needs to search through $9 \times 2 \times 9 = 162$ unique configurations, as opposed to 1000 iterations in the HA tuner. Hence for the image filter benchmarks, the HS tuning times are lower than in HA tuning. While exhaustive search was possible in this scenario, for a system with more approximation choices for each operation (more accelerator knobs, perforation, sampling etc.), such exhaustive search through all combinations may not be feasible. For the DNNs, the HA and HS times are mostly similar since both autotuner runs are assigned equal iterations.

4.5.4 Hardware Sensitivity

To validate the benefits of ApproxHPVM across different hardware characteristics, we study the impact of pDMA and number of PROMISE banks on performance and energy.

pDMA: In deep learning, GEMM-based convolutions maps convolution $X \circledast W$ into a product of two matrices P_X and P_W, known as patch matrices. Using GEMM for convolution is desirable because GEMM is typically a highly optimized operation, and both

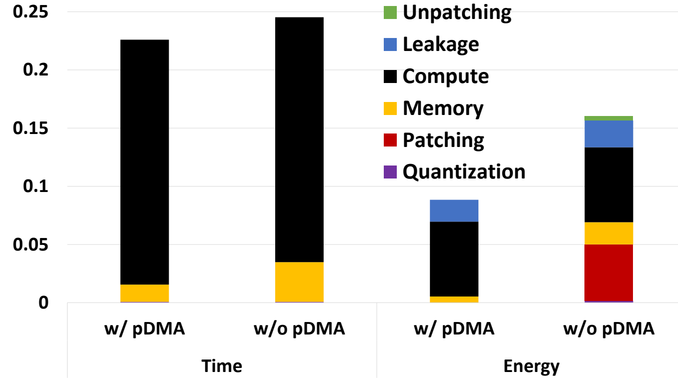


Figure 4.12: Normalized execution time and energy for AlexNet with and without pDMA. Without pDMA, PROMISE relies on the GPU to perform quantization, patching, and unpatching; these overheads reduce the performance and energy improvement over FP32.

NVIDIA's cuDNN library [155] and PROMISE perform GEMM-based convolution. The overhead of GEMM-based convolution consists of two data layout transformations: "patching" to generate matrices P_X and P_W, and "unpatching" to convert the GEMM's output to the application's desired format.

In cuDNN, patching and unpatching are done in on-chip memory to minimize this overhead [155]. In PROMISE, we can either use the pDMA scheme described in Section 4.4.3 or rely on the GPU to perform patching and unpatching. Similarly, quantization to/from INT8 can be performed either by pDMA or by the GPU before/after PROMISE's execution. In order to compare these two choices, we implemented CUDA kernels for patching, unpatching, and quantization, and compared their performance and energy to pDMA. We pipelined patching/unpatching with PROMISE's execution to maximize performance.

Figure 4.12 shows execution time and energy, normalized to FP32, for the $Loss_{2\%}$ AlexNet configuration with and without pDMA. While pipelining minimizes the time overhead, the entire energy cost of the GPU kernels is still incurred. Moreover, the increased data movement (the patch matrix is 121x larger than the input matrix in Conv2) causes both time and energy to increase further. Nonetheless, PROMISE without pDMA still achieves a 4.1x speedup and 6.3x energy reduction compared to FP32. While the benefits are higher with pDMA (4.4x performance and 11.3x energy), these results show our approach is effective regardless of the method used.

#Banks:

We performed a scaling study of the number of PROMISE banks to establish the suitability of a 256 bank configuration. Figure 4.13 shows the execution time and energy of FC4 as the number of banks is increased, as well as the area overhead associated with the increasing

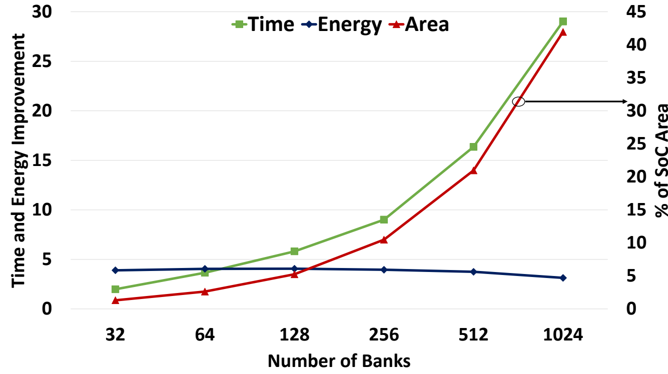


Figure 4.13: Speedup and energy reduction over FP32, and area for FC4 vs the number of PROMISE banks. The SoC contains 4B transistors.

number of banks. Using 256 banks, PROMISE strikes a balance between performance, energy, and area – it only consumes 10% of a 4B transistor SoC's area and still significantly outperforms FP32.

4.6 CONCLUSION

In this paper, we introduced ApproxHPVM, a compiler IR that introduces hardware-agnostic accuracy metrics that are decoupled from hardware-specific information. We augment ApproxHPVM with an accuracy-tuning analysis that lowers the accuracy requirements of IR operations given an end-to-end quality metric, while the hardware scheduling phase uses the extracted constraints to map to different approximation choices. Our results show that ApproxHPVM provides promising results on a heterogeneous target platform with multiple hardware compute units. Across 14 benchmarks in the deep learning and image processing domains, we observe performance speedups ranging from 1-9x and energy reductions ranging from 1.1-11.3x. As ApproxHPVM does not include hardware-specific information at the IR level, we envision ApproxHPVM to be extensible to a wide range of approximate computing hardware. Moreover, we believe that the hardware-independent accuracy constraints can also be satisfied by software-only techniques for approximation.

CHAPTER 5: APPROXTUNER: COMPILER AND RUNTIME SYSTEM FOR ADAPTIVE APPROXIMATIONS

5.1 INTRODUCTION

With the ubiquitous deployment of machine-learning and big data processing workloads [156], improvements are increasingly important for optimizing these workloads. A wide range of different kinds of computations are being deployed on both the cloud and the edge, including image processing, object classification, speech recognition, and face recognition [6, 7, 8]. These applications are compute-intensive, which often renders it infeasible to run such computations on resource-constrained edge computing systems, and also drives up operational costs in data centers [157, 158, 159].

Many computations in these domains are inherently approximate, in the sense that the input data are often derived from noisy sensors and output results are often probabilistic, e.g., for object classification or facial recognition, or also noisy, e.g., for image and audio streams. Such computations can *trade off small amounts of result quality for improved performance and efficiency* [14]. Previous research has presented many individual domain-specific and system-level techniques for trading accuracy for performance. For instance, reduced precision models are widespread in deep learning [160, 161, 162]. Recent specialized accelerators incorporate hardware-specific approximation knobs that provide orders-of-magnitude improvements in the performance and energy benefits achieved in exchange for relaxing accuracy [163, 23, 50, 164, 165, 101]. Such techniques provide an important opportunity to improve performance and/or energy of applications for emerging heterogeneous parallel systems.

In practice, a realistic application (e.g., a neural network or a combination of an image processing pipeline and an image classification network) can make use of multiple approximation techniques for different computations in the code, each with its own parameters that must be tuned, to achieve the best results. For example, our experiments show that for the ResNet-18 network, which contains 22 tensor operations, the best combination is to use three different approximations with different parameter settings in different operations. A major open challenge is *how programmers can select, configure, and tune the parameters for combinations of one or more approximation techniques*, while meeting *end-to-end requirements* on energy, latency, and accuracy. Several specific challenges arise in meeting this goal:

Large Trade-off Space Combining Multiple Approximations. The variety of software and hardware approximations with many accuracy-performance and accuracy-energy trade-

offs induce a large search space spanning both software and hardware choices, as large as $7e{+}91$ in our benchmarks. The optimization tasks include selecting the versions of individual data structures and algorithms (e.g., number representation and tensor operators in CNNs), mapping each computation to a specific compute unit, and tuning various *knobs* – parameters that impact the performance and accuracy of the application.

High cost of measuring accuracy and performance. A heuristic search for optimal approximation choices requires determining attributes such as quality of service (e.g., inference error or PSNR), throughput, and energy. Empirically measuring these quantities by running approximate program versions is prohibitively expensive on edge systems, requiring weeks or months for realistic kernels, and undesirable on cloud systems where energy and/or monetary costs can become overwhelming. We find that empirical tuning can take days (on a server-class machine) for some of our benchmarks - 1.5 days for VGG16 and 11 days for ResNet50. These high tuning times motivate the need for efficient tuning.

Diverse range of heterogeneous systems. The diversity of hardware compute units [70] means that different ones provide different approximation options and differing accuracy-vs-performance trade-offs. Domain-specific accelerators that support hardware-specific approximations (e.g. analog compute accelerators [23], low-precision ML accelerators [50, 51, 52]) can offer orders-of-magnitude performance and energy improvements. Moreover, for some domains (e.g., smartphones or tablets), software portability is essential since modern applications must execute on widely varying system configurations [166], but different systems may require very different sets of approximation choices.

Optimization choice depends on run-time conditions. Satisfying end-to-end requirements may depend on many run-time conditions, e.g., the load of the system, the state of the battery, the inputs to the computation, or varying application demands during execution. To meet its requirements, the application may need to adapt to the changing conditions and reconfigure during run time.

5.1.1 ApproxTuner System

We present **ApproxTuner**, an automatic framework for accuracy-aware optimization of applications, given user-provided high-level end-to-end quality specifications. It addresses all of the challenges above, *and is the first and only system to handle all of these.*

ApproxTuner tackles the last two challenges above — hardware-specific, yet portable, tuning and run-time adaptation — by decomposing the optimization process into three stages: development-time, install-time and run-time. The system selects hardware-independent approximations and creates a Pareto-optimal trade-off curve for them at development time. At

install time, the system refines this curve using hardware-specific optimizations and performance measurements, then uses the refined curve for the best static choices of approximations and parameter settings. The final Pareto curve is included with the program binary. At run time, the system optionally adapts these choices based on run-time conditions, using the final Pareto curve to keep the overheads of run-time adaptation very low.

To address the first two challenges – efficiently navigating the large trade-off space and efficient performance and quality estimation – ApproxTuner introduces a novel technique, *predictive approximation-tuning*. Predictive approximation-tuning uses one-time error profiles of individual approximations, together with error composition models for tensor-based applications, to predict end-to-end application accuracy[1]. Our approach also facilitates distributed tuning since the error profile collection can happen at multiple client devices in a distributed manner with autotuning performed on a centralized server. This makes install-time tuning (with hardware-specific approximations) feasible which can otherwise be prohibitively expensive to do on a single resource-constrained edge device.

5.1.2 Contributions

In summary, our contributions are:

- A system that combines a wide range of existing *hardware and software approximations*, supports diverse heterogeneous systems, and provides an easy-to-use programming interface for accuracy-aware tuning. We show that different kinds of approximations and approximation knobs are suited for different applications and also across sub-computations in the same application.
- A novel *three-phase accuracy-aware tuning technique* that provides performance portability, retargetability to compute units with hardware-specific approximation knobs, and dynamic tuning. It splits tuning into: 1) selection of hardware-independent approximations at *development-time*, 2) mapping to hardware-specific approximations at *install-time*, and 3) a fast approximation selection at *runtime*.
- *Predictive approximation-tuning*, a novel technique that speeds up both development-time and install-time analyses. For two different accuracy-prediction models, our predictive tuning strategy speeds up tuning by 13.7x and 17.9x compared to conventional empirical tuning while achieving comparable benefits.

[1] The work on accuracy prediction models is lead by Yifan Zhao (yifanz16@illinois.edu) and details on the models will also appear in his Thesis.

- Our evaluation on 11 benchmarks (10 CNNs and 1 combined CNN + image processing benchmark) shows:

  Generic Approximations. Exploiting generic hardware-independent approximations, ApproxTuner achieves geometric mean speedup of 2.2x and energy reduction of 2.1x on GPU, with merely 2 percentage points of drop in inference accuracy. On CPU, we observe a geometric mean speedup of 1.4x and energy reduction of 1.4x.

  Hardware Approximations. At install time, mapping tensor operations to PROMISE, an analog compute accelerator, ApproxTuner provides geometric mean speedup of 4.5x across the benchmarks.

  Runtime Adaptation for Approximations. ApproxTuner can dynamically tune approximation knobs to counteract system slowdowns imposed by runtime conditions such as low-power modes.

5.2 ApproxTuner OVERVIEW

Figure 5.1 shows the high-level workflow for ApproxTuner. ApproxTuner builds on the HPVM and ApproxHPVM compiler systems [79, 102], which are briefly described below. ApproxTuner takes as input programs written in Keras or PyTorch for convolutional neural networks, or CNNs, or an extension of C that can be compiled to HPVM [79] (for other tensor-based programs), and compiles them to the ApproxHPVM internal representation (IR) [102]. ApproxHPVM manages the compilation to various compute units (Section 5.2.1).

ApproxTuner optimizes the computations in this IR using three phases: 1) development-time, 2) install-time, and 3) run-time (Section 5.2.2). ApproxTuner's goal is to select combinations of software and hardware approximations (detailed in Section 5.2.3) that maximize performance and energy benefits.

5.2.1 Preliminaries and Terminology

ApproxHPVM. HPVM IR is a dataflow graph-based parallel program representation that captures coarse- and fine-grain data and task parallelism. HPVM provides a portable IR and a retargetable compiler framework that can target diverse compute units including CPUs, GPUs and FPGAs.

ApproxHPVM is an extension of HPVM [79] with added support for tensor operations, and limited support for accuracy-aware tuning (see Section 2). The tensor operations rep-

resent data parallel computations to support application domains such as CNNs and image processing.

ApproxHPVM also adds additional backends to the HPVM framework, including one for cuDNN on NVIDIA GPUs and one for a programmable analog in-memory compute accelerator called PROMISE [23].

This work focuses on approximations for a set of predefined tensor operations included in ApproxHPVM, such as convolutions, matrix multiplication, ReLU, map, and reduce. The full list of supported intrinsics is listed in Table 4.1.

These operations are the units of scheduling and approximation in ApproxTuner, where a schedule is a mapping of tensor operations to compute units in the target system.

Hardware-independent approximations for an operation are those whose impact on the quality of a program is fixed, regardless of the hardware used to execute the operation. Some approximations, like the use of (IEEE) FP16 instead of FP32 may have hardware-independent semantics (when available) and yet may be implemented in hardware for efficiency.

Other approximations are called *hardware-specific*. The impact on energy and performance will usually be hardware-dependent for both kinds of approximations.

Quality of Service. A quality-of-service (QoS) metric is a (usually domain-specific) metric over the quality of some computation, such as the inference accuracy of a machine learning model, or the peak signal-to-noise ratio (PSNR) of an image processing application. For a tensor-based program, its QoS metric function $QoS : T_{\text{out}} \times T_{\text{gold}} \to \mathbb{R}$ takes the output tensor T_{out} and the gold tensor T_{gold}, and produces a scalar value.

A *QoS constraint* is a constraint over the QoS level of an application. As QoS is conventionally a higher-better value, we assume a QoS constraint is always a lower bound; the opposite case can be treated similarly.

Knobs, Configurations, and Tradeoff Curves. ApproxTuner supports approximation methods implemented in software or hardware. Software techniques may be algorithmic or system-level (Section 5.2.3). An *approximation knob* is a discrete-valued parameter of an approximation method that can be modified to control the quality, energy, and run time.

Each tensor operation may be assigned an approximation choice or none. A *configuration* is a map $Config : op \to \text{Int}$ that assigns an approximation knob value to every tensor operation in the program. Zero value denotes no approximation. The *search space* is the set of all possible configurations.

A tradeoff point is a triple $(QoS, Perf, config)$, which records the quality-of-service and the performance (e.g., latency, throughput, or energy) of the configuration (on a representative input set). The set of all tradeoff points, denoted $\mathcal{S}$, represents the *tradeoff space*. To

compare tradeoff points, we define a *dominance* relation, $\preccurlyeq$, in the usual way [167]: a point $s_1 = (QoS_1, Perf_1, config_1)$ is dominated by another point $s_2 = (QoS_2, Perf_2, config_2)$ iff it has both lower QoS and worse performance. Formally, $s_1 \preccurlyeq s_2$ iff $QoS_1 \leq QoS_2$ and $Perf_1 \leq Perf_2$. Strict dominance, $s_1 \prec s_2$, is defined as dominance when the two points are not equal.

A search algorithm explores a subset of the tradeoff space $S \subseteq \mathcal{S}$ to find desirable tradeoff points. One way to describe desirable points is through *Pareto* sets. The Pareto set of a tradeoff set S is its subset consisting of non-dominated points:

$$PS\,(S) = \{\, s \mid s \in S \ \wedge \ \forall s' \in S \ .\ s \not\prec s' \,\} \tag{5.1}$$

This set defines the *tradeoff curve*, which contains linear segments between the points in the Pareto set. These points thus have the best tradeoffs that the search algorithm was able to find. We will also use a relaxed version, PS_ε, which includes tradeoff points that are *close* to (i.e., within an ε distance from) the points in the Pareto set:

$$PS_\varepsilon\,(S) = \{\, s \mid s \in S \ \wedge \ \exists s^* \in PS(S)\ .\ dist(s, s^*) \leq \varepsilon \,\} \tag{5.2}$$

Here, *dist* as the usual Euclidean distance in the tradeoff space.

Different search algorithms may explore different subsets of the tradeoff space, e.g., S_1 and S_2. To numerically characterize a Pareto set, $PS(S_1)$ or $PS(S_2)$, researchers use a variety of quality indicators [167]. In this paper we use a *hypervolume indicator* that measures the size of the dominated subspace. It corresponds to the area-under-curve (AUC) of the tradeoff curve. To compare $PS(S_1)$ and $PS(S_2)$, we compute the difference between their hypervolume indicators on a desired interval between QoS_{min} and QoS_{max}.

5.2.2 Overview of Three Stages of ApproxTuner

Our goal is to automatically select approximation knobs that minimize energy and/or maximize performance for a given program or its component, while satisfying some user-specified end-to-end QoS constraint. We refer to this task as *approximation-tuning* and do it in three stages:

The development-time stage (Section 5.3) computes a tradeoff curve using only hardware-independent approximations. ApproxTuner takes as input a program and a QoS constraint, and generates a set of possible configurations, S_0, that maximize a hardware-agnostic performance metric and produce QoS values within the constraint. These configurations are

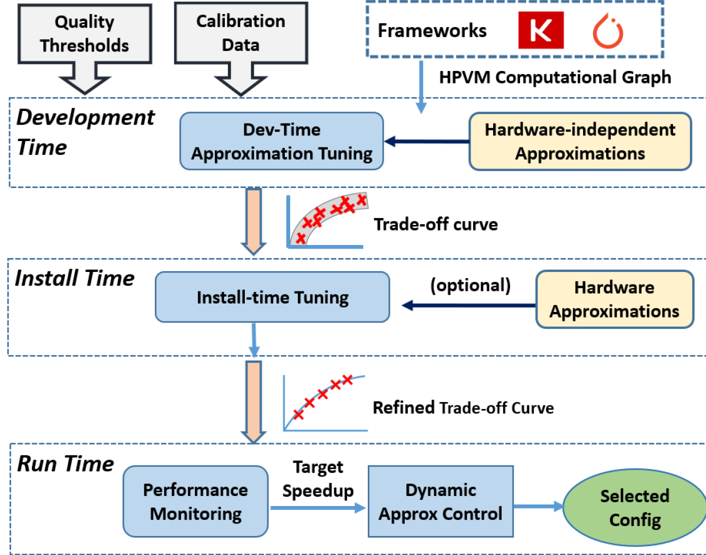


Figure 5.1: ApproxTuner workflow.

used to create the relaxed tradeoff curve, $PS_{\varepsilon}(S_0)$ (much smaller than S_0) which is shipped with the application.

The install-time stage (Section 5.4) uses the development-time trade-off curves and measures the actual performance of each configuration in those trade-off curves on the target hardware. Next, we create a refined trade-off curve, $PS(S')$, where S' is the PS_{ε} curve from development-time updated with real performance measurements. This stage can be run any time that the target hardware is known.

If hardware-specific approximations (not known at development time) are available, which may also change the QoS metrics, the install-time stage performs a new autotuning step that includes the hardware approximations

The run-time stage (Section 5.5), takes the program's final trade-off curves from the install-time phase and uses them for dynamic approximation tuning. The system can track various metrics (e.g. load, power, and frequency variations) and provide feedback to the dynamic control, which computes a target speedup (and configuration) to maintain the required level of performance.

5.2.3 Approximation Methods

ApproxTuner is extensible to a wide range of software and hardware approximations. This work evaluates five approximations below – the first three are software (hardware-independent) techniques implemented for CPUs and GPUs; the fourth is a previously proposed experimental hardware accelerator representing a hardware-specific approximation; and the

fifth (reduced floating point precision, IEEE FP16) has hardware-independent semantics. We summarize these below:

Filter sampling for convolutions. Li et al. [91] proposed an approximation technique that compresses convolution filters by removing feature maps (i.e. channels) that have relatively low L1-norms (sum of absolute values). Based on this, we implement our own variant of filter sampling that supports dynamic knobs for approximation. Our implementation prunes an equal fraction of filter elements across all feature maps with a fixed stride. It has 9 knob values spread across i) sampling rates for filter sampling: 50% (skip 1 out of 2), 33% (skip 1 out of 3), and 25% (skip 1 out of 4) and ii) the initial offset at which elements are skipped; this has a noticeable impact on overall accuracy, as different offsets align with more or less important filter elements

Perforated convolutions. Figurnov et al. [168] proposed *perforated convolutions*; an algorithmic approximation that computes a subset of the convolution output tensor, and interpolates the missing values using nearest neighbor averaging of computed tensor elements. Our implementation skips rows and columns of the tensor operation outputs at a regular stride, then interpolates the missing output elements. It has 18 knob values: i) skipping rows or columns, ii) *skip rate*: 50%, 33%, and 25%, and iii) initial offset.

Reduction sampling. Zhu et al. [35] proposed *reduction sampling*; an algorithmic approximation that computes a reduction operation using a subset of inputs. Our implementation supports 3 knob values for sampling ratio: 50%, 40%, and 25%. For reductions like average, sum, or multiply, we scale the result by an appropriate constant.

PROMISE, an approximate analog accelerator. [23]. Our implementation considers PROMISE for tensor convolutions and matrix multiplications. PROMISE is an analog chip and its voltage swings introduce statistical (normally distributed) errors in the output values. The knob values are 7 different voltage levels (P1-P7), in increasing order of voltage (energy) and decreasing error. No mode in PROMISE produces exact results; all voltage levels introduce some errors. Srivastava et al. [23] show that PROMISE consumes 3.4-5.5x less energy and has 1.4-3.4x higher throughput compared to custom non-programmable digital accelerators.

IEEE FP16. Our implementation has FP16 versions of all tensor operations, including the two approximate algorithms above. FP16 can be used or not for an operation; there is no additional knob value. Although FP16 requires hardware support, its semantics (impact on QoS) is hardware-independent and hence can be accounted for in the development stage. Since FP16 availability is not guaranteed, we produce two trade-off curves at development time, one for FP16 and one for FP32, then use the appropriate one at install time.

The above set offers many choices for approximation. For each convolution operation, ApproxTuner offers 9 knobs for filter sampling, 18 knobs for perforation, and for each perforation/sampling knob, both FP32 and FP16 are supported. Convolution operations can also be mapped to an FP32-only (considered most-accurate) or FP16-only variant - adding 2 more knobs. PROMISE hardware offers 7 knobs. Currently, we do not combine perforation, sampling, and PROMISE at the same operation. In total, this results in $63 = (9 * 2 + 18 * 2 + 2 + 7)$ knobs for each convolution operation, $8 = 3 * 2 + 2$ knobs for each reduction, and 2 choices for every other tensor operation.

5.3 DEVELOPMENT-TIME TUNING

At development time, we tune the application for hardware-independent approximations. We propose *predictive approximation tuning*, which collects per-operation performance and accuracy profiles prior to the search, and then uses compositional models to predict QoS and performance to guide the autotuning search. This approach does not invoke the program binary for every autotuning iteration. In contrast, conventional *empirical autotuning* evaluates a configuration by actually running the program binary (e.g., CNN inference) which can be expensive, especially when many autotuning iterations are required to explore large search spaces.

5.3.1 Overview of Predictive Tuning

Algorithm 5.1 describes our predictive approximation tuning in the function `PredictiveTuner`. It consists of five steps:

- **Profile collection**: Lines 12-15 in Algorithm 5.1 collects a QoS profile for the given application *per operation*, *per knob setting* (§ 5.3.2).
- **QoS predictor refinement**: Lines 18-20 refines a parameter α of the *QoS prediction model* (§ 5.3.3) used in the following autotuning, so that the predictor fits better to the program to be tuned.
- **Autotuning**: Lines 23-30 heuristically explore the configuration space in a fixed number of iterations. The QoS prediction model and *performance prediction model* (§ 5.3.4) direct this search towards better configurations. We use the OpenTuner tool [75] which contains a number of algorithms for exploring large configuration spaces.
- **Tradeoff curve construction**: Line 33 selects autotuning configurations that are in or close to the Pareto set (§ 5.3.5).

Algorithm 5.1: Predictive Approximation Tuning

```
1  Inputs:
2    • P: target program
3    • C: calibration inputs for profiling
4    • K: knobs that apply to each operator in P
5    • QoS_min: minimal acceptable QoS
6    • nCalibrate: number of calibration runs used to tune α
7    • nIters: number of autotuning iterations
8    • ε1, ε2: maximum distances of a configuration to the Pareto set
9  Output: A trade-off curve for P
10 Function PredictiveTuner(P, C, K, QoS_min, nCalibrate, nIters, ε1, ε2)
11     // Step 1: collect QoS Profile
12     map qosProfiles;
13     foreach (op, knob) ∈ K do
14         (ΔQ, ΔT) = gatherQoSProfile(P, C, op, knob);
15         qosProfiles[(op, knob)] = (ΔQ, ΔT);
16
17     // Step 2: initialize and tune predictor to find coefficient α
18     autotuner = AutoTuner (P, K, nIters, QoS_min);
19     predictor = Predictor (qosProfiles);
20     α = predictor.calibrate (autotuner, nCalibrate);
21
22     // Step 3: Autotune with QoS and Perf. prediction Models
23     set candidateConfigs;
24     while autotuner.continueTuning() do
25         config = autotuner.nextConfig();
26         predQoS = predictor.calculateQoS (config, α);
27         predPerf = predictor.calculatePerf (config);
28         autotuner.setConfigFitness (config, predQoS, predPerf);
29         if predQoS > QoS_min then
30             candidateConfigs ∪= (predQoS, predPerf, config);
31
32     // Step 4: Take configs within ε distance of the tradeoff curve
33     paretoConfigs = PS_ε1 (candidateConfigs);
34
35     // Step 5: Filtering invalid configurations at the end of tuning
36     set filteredConfigs;
37     foreach (predQoS, predPerf, config) ∈ paretoConfigs do
38         realQoS = measureRealQoS (P, C, config);
39         if realQoS > QoS_min then
40             filteredConfigs ∪= (realQoS, predPerf, config);
41     return PS_ε2(filteredConfigs);
```

- **QoS validation**: Lines 36-40 empirically measure the QoS of configurations in previous step, and selects configurations with measured QoS greater than threshold.

5.3.2 Gathering QoS Profiles

QoS profiles are gathered for each unique pair of *tensor operation* and *approximation knobs*. Algorithm 5.1 infers (Line 13) a list of such $(op, knob)$ pairs from the input K, which is a mapping from each tensor operation in program P to the set of knobs applicable to it. The profiles are collected by running the entire program (with calibration inputs) but we approximate a single operator at a time.

The QoS profile consists of: 1) an end-to-end QoS metric, e.g., classification accuracy in CNNs or mean square error (see Section 5.6), and 2) final raw tensor output of the application, e.g., for CNNs, output of the softmax operation. The profiles are stored as two tables Q and T. Q maps $(op, knob)$ to the corresponding end-to-end QoS. T maps $(op, knob)$ to the raw tensor output T_{out}. We also measure and store the QoS and raw tensor output of the *baseline* version, which has no approximations, denoted as QoS_{base} and T_{base}, respectively.

5.3.3 Models for QoS Prediction

We propose and evaluate two different error composition models Π_1 and Π_2, for a program P:

$$\Pi_1(\mathit{config}) = QoS\left(T_{\text{base}} + \alpha \cdot \sum_{op \in P} \Delta T(op, knob),\ T_{\text{gold}}\right) \tag{5.3}$$

$$\Pi_2(\mathit{config}) = QoS_{\text{base}} + \alpha \cdot \sum_{op \in P} \Delta Q(op, knob) \tag{5.4}$$

where knob $= \mathit{config}(\text{op})$, α is the coefficient to be refined, and

$$\Delta T(op, knob) = T(op, knob) - T_{\text{base}}, \tag{5.5}$$

$$\Delta Q(op, knob) = Q(op, knob) - QoS_{\text{base}}, \tag{5.6}$$

The model Π_1 computes the QoS of a configuration by 1) summing the errors in the end-to-end raw tensor outputs for each $(op, knob)$ pair in the configuration *config*, 2) adding this sum to the baseline raw tensor output (T_{base}), and then 3) computing the QoS over this

summation. The model captures how approximations affect the errors in the raw output (for a single approximation) and sums the effects of all, before applying the QoS function. This allows Π_1 to work well with classification accuracy as the QoS metric, since accuracy is computed as an argmax probability of all candidate classes (essentially the raw tensor output), and is hence tolerant to errors in the raw outputs as long as the probability of the (correct) predicted class(es) stays the highest.

The model Π_2 is a coarser-grained model that does not examine individual tensor outputs. Instead, it uses QoS profile table, Q, to compute the QoS loss of a configuration by summing over the end-to-end QoS loss for each $(op, knob)$ pair in *config*. Π_2 (which only sums scalar losses) is computationally less expensive than Π_1 (which sums raw tensors), but is also relatively less precise, as shown in our evaluation.

Predictor Calibration using Regression. Both Π_1 and Π_2 can be viewed as linear regression models with a single coefficient α that scales the errors of each error profile. This coefficient allows the predictor to adapt to specific error propagation within the application. On Line 20, `predictor.calibrate` evaluates the real QoS of a small number of configurations (e.g., 50 are sufficient in our experiments), and updates α so that the predicted QoS fits the observed QoS better.

Scope of the Predictive Models. We have applied our predictive models to fixed DAGs of tensor operations in the context of CNNs and to one image processing benchmark, but they can be applied to other tensor domains. For these models to work, the program must meet the following criteria:

- The control flow must be deterministic and input-independent. This ensures that the error profile of different knobs captures the error behavior of the same control flow, and can be composed.
- The operators should have no side-effects.
- For predictor Π_1, the shapes of raw tensor outputs (i.e., number of dimensions and extents of each dimension) must match, since these outputs are summed up in the model. This requires that the output tensor shape is input-independent and fixed.

5.3.4 Models for Performance Prediction

To guide the tuner, we use a simple hardware-agnostic performance prediction model. As a proxy for execution time, we use the count of compute and memory operations, computed analytically for each tensor op with closed-form expressions using input tensor sizes, weight tensor sizes, strides, padding, etc. This calculation has negligible cost.

The total execution cost of a configuration is the sum of the cost for each operation with the knob the tuner selected,

$$C_{\text{Total}}(\mathit{config}) = \sum_{i=0}^{N} C(\mathit{config.op}[i],\ \mathit{config.knob}[i]). \tag{5.7}$$

We assume the operation count is reduced by a factor that is proportional to the approximation level (e.g, 50% vs 25% perforation). Thus, we estimate execution time of running operation *op* with approximation knob *knob* by:

$$C(op, knob) = \frac{N_m(op)}{R_m(knob)} + \frac{N_c(op)}{R_c(knob)}, \tag{5.8}$$

where N_c and N_m are the analytically-computed number of compute and memory operations, respectively, for the baseline (non-approximate) version of *op*. R_m and R_c are the corresponding reduction factors and are specific to the selected approximation knob. E.g., for FP16 50% *filter sampling*, $R_m = 4$ since the operation loads 2× fewer bytes due to FP16 and performs 2× fewer loads due to sampling, and has $R_c = 2$ since it skips half the computations.

The number of operations does not perfectly reflect actual speedup, as other factors change with the size of computation, such as cache friendliness. However, for the same operator, an approximation that reduces more operations is likely faster than one that reduces fewer operations. Therefore, this performance predictor *ranks* configurations correctly by their speedup, which suffices for autotuning purposes.

5.3.5 Configuration Filtering

Autotuning often discovers many `candidateConfigs`. To reduce the overhead of empirical validation, we filter away configurations that are not in or close to the Pareto set. On Line 33, points in the Pareto set or with distance to the Pareto set less than ε_1 are kept (by Equation 5.2). Similarly, an ε_2 controls the configuration filtering on Line 41. ε_1 and ε_2 are user-selected thresholds that control the quality and the space of trade-off curve and the time of our three-stage tuning.

5.4 INSTALL-TIME TUNING

The install-time tuning phase takes the tradeoff curve from the development-time tuning (PS_ε), together with the same calibration inputs for profiling (C), hardware-specific

knobs (K) for each operator on the edge device, the number of edge devices n_{edge}, and the other parameters from the development-time tuning. This step refines the shipped trade-off curves with real performance measurements and creates a new trade-off curve. Optionally, distributed predictive tuning is invoked to further optimize the program by exploiting hardware-specific approximations supported on the target platform. Distributed predictive tuning divides the tuning burden across multiple participating edge devices.

Software-only knobs. In this case, all steps are done on the edge-device. It runs the configuration from the input trade-off curve PS_{ε} on the inputs from C. Similarly to Lines 36-40 from Algorithm 5.1, it measures *both* the real QoS and performance (development-time stage collected only QoS), and filters the candidate configurations that do not satisfy the thresholds. Finally, for the resulting filtered set S' it constructs the final tradeoff curve $PS(S')$.

Hardware-specific knobs. We distribute predictive tuning across the server and edge-devices in three main steps:

- This phase is distributed across edge-devices. Each device gathers profiles for $|C|/n_{edge}$ calibration inputs. For hardware-specific approximations in K, the devices collect the QoS profiles as in Lines 12-15 from Algorithm 5.1.
- The edge-devices send the QoS profiles to a centralized server. It merges the profiles – taking the mean of ΔQ (change of QoS) in the profile, while concatenating the ΔT (change of tensor output) together. It then runs the predictive tuning as in Lines 18-30 from Algorithm 5.1. Because the approximation choices cannot be decoupled (as we found in initial prototype experiments), we cannot simply reuse the curves from development-time, but instead perform a fresh autotuning step that combines software and hardware approximations and constructs new trade-off curves.
- The server sends the configurations to the edge-devices. Each edge device validates an equal fraction of the total configurations and filters the configurations by measuring both the real QoS and performance (similar to Lines 36-40 in Algorithm 5.1).
- The server receives the filtered configurations from each of the participating edge devices and computes the final trade-off curve, $PS(S_1 \cup S_2 \cup ... \cup S_n)$, where S_i are configurations returned by edge device i; these are configurations with real QoS higher than the user-specified QoS threshold. This is the final curve that the server sends back to the devices.

5.5 RUNTIME APPROXIMATION TUNING

A key capability of ApproxTuner is the ability to adapt approximation settings at runtime to meet application goals such as throughput or responsiveness, in the face of changing

system conditions such as load variation, frequency scaling or voltage scaling, or changing application demands. Our runtime control assumes that the program is running in isolation on the target hardware. Significant prior work has addressed the problem of multi-tenancy (e.g., [169]) and can be incorporated in our approach.

ApproxTuner allows users to specify a desired target for performance and/or energy usage, and then uses the trade-off curve (built at install-time) to select configurations that allow for meeting these goals. Runtime conditions (e.g., lowering processor frequency) can impose system slowdowns which may cause the application performance to fall below the desired target. In these scenarios, the dynamic tuner switches configurations to choose a different point from the performance-accuracy trade-off space.

Performance is measured for each *invocation*, which is one execution of the target code, e.g., the entire CNN or entire image-processing pipeline (for one batch of images). A system monitor measures the execution time over a (configurable) sliding window of most recent batch N executions ($k-N, \ldots, k-2, k-1$). If the average performance of the sliding window executions falls below the desired target, the dynamic tuner is invoked. In this case, ApproxTuner chooses a configuration from the trade-off curve that achieves the target performance level. The runtime tuner switches configurations simply by using different approximation knob settings, which are parameters to the operations for each tensor method (e.g., perforation and sampling rates), and hence has negligible overhead.

One challenge is that there may not be an exact point matching the desired speedup in the trade-off curve, so our system allows the user to select between two policies for achieving the target speedup:

1. **Enforce Required Speedup in each Invocation.** It picks from the trade-off curve a configuration that provides speedup equal to or higher than the desired speedup. Picking a point from the curve is a $O(\log(|PS|)$ operation, as it is implemented as binary search over the trade-off curve stored as a sorted array data structure in memory.
2. **Achieve Average Target Performance over Time.** It probabilistically selects between two configurations, with their performance the closest below and above the required performance over a period of time (on average). The probabilities of selecting each of these two points, p_1 and p_2, are calculated such that $p_1 \cdot Perf_1 + p_2 \cdot Perf_2 = Perf_{target}$, as in [35]. For instance, if required speedup is 1.3x and the closest points on the curve provide 1.2x and 1.5x speedup, the configurations are randomly selected between invocations, with respective probabilities $2/3$, and $1/3$ so that the average speedup is 1.3x.

Policy 1 is better suited for hard or soft real-time systems, where computations must meet deadlines. Policy 2 is a better choice when application throughput is a goal.

Datasets. We use: MNIST [145] CIFAR-10 [146] and the ImageNet dataset ILSVRC 2012 [170]. CIFAR-10 and MNIST have 60K images (50K train set and 10K test set). For ImageNet, we use 10K randomly sampled images (from 200 randomly selected classes) from its 50K validation set. We divide each test set into equal-sized calibration set (for auto-tuning) and test sets (for evaluation).
Benchmarks. We use several CNNs (Table 5.1) and an image processing benchmark that combines a CNN (AlexNet2) with the Canny edge detection pipeline).

Table 5.1: CNN benchmarks, their datasets, layer count, classification accuracy with FP32 baseline, and size of auto-tuning search space.

Network	**Dataset**	**Layers**	**Accuracy**	**Search Space**
AlexNet [148]	CIFAR-10	6	79.16%	5e+8
AlexNet [148]	ImageNet	8	55.86%	5e+8
AlexNet2	CIFAR-10	7	85.09%	2e+10
ResNet-18 [12]	CIFAR-10	22	89.44%	3e+22
ResNet-50 [12]	ImageNet	54	74.16%	7e+91
VGG-16 [11]	CIFAR-10	15	89.41%	3e+22
VGG-16 [11]	ImageNet	15	72.88%	3e+22
MobileNet [151]	CIFAR-10	28	83.69%	1e+26
LeNet [171]	MNIST	4	98.70%	3e+3

5.6.1 Quality Metrics

For CNNs, we measure accuracy degradation with respect to the baseline, denoted $\Delta QoS_{x\%}$ for a degradation of $x\%$. For the image processing benchmark, we use average PSNR, between the output images x and ground truth images x_0, given by:

$$PSNR(x, x_0) := -10 \ \log_{10} \sum_i (x[i] - x_0[i])^2 \tag{5.9}$$

($PSNR_y$ denotes a PSNR of y.) A higher PSNR implies better image quality. The predictive models use the mean square error (exponential of PSNR) as the QoS metric.
Baseline: For our baseline, we map all computations to FP32 with no approximations.

5.6.2 Implementation

Our tensor library (targeted through our compiler backends) uses cuDNN for most tensor operators, but cannot use it for convolutions because it is proprietary and we cannot modify it to implement custom algorithms for perforation/sampling. Instead, we developed a

Table 5.2: System parameters for the Edge Device. NVIDIA Tegra TX2 board including the PROMISE accelerator on chip.

Tegra TX2	
CPU Cores	6
GPU SMs	2
CUDA Cores	256
GPU Frequency	1.12 GHz
DRAM Size	8 GB

PROMISE	
Memory Banks	256×16 KB
Frequency	1 GHz

hand-optimized convolution operator using CUDA, and optimized using cuBLAS, memory coalescing, and tuning hardware utilization, thread divergence, and scratchpad usage. Our CPU implementations vectorize tensor processing loops using OpenMP.

Our implementation is within 10% of PyTorch's production CUDA-based backend on average. CuDNN is much faster (2.8X on average), since it is able to use proprietary, device-specific tuning. Since perforation and sampling reduce both the compute and memory operations, we expect them to give similar speedups even with the proprietary cuDNN routines.

5.6.3 Hardware Setup

Client Device Setup.

The client device we use for our experiments (Table 5.2) is the NVIDIA Jetson Tegra TX2 developer board [137], commonly used in edge applications such as robotics and small autonomous vehicles [172, 173, 174].

We model an SoC that adds to the TX2 a simulated PROMISE accelerator for machine learning [23]. GPU, CPU, and PROMISE communicate through global shared memory. We use a split approach for profiling. We measure performance and power via direct execution on the GPU and CPU. Our profiler continuously reads GPU, CPU and DRAM power from Jetson's voltage rails via an I2C interface [175] at 1 KHz (1 ms period). Energy is calculated by integrating the power readings using 1 ms timesteps. To model PROMISE, we use the functional simulator and the validated timing and energy model [23] obtained from its authors.

Server Setup. We use a server-class machine for development-time tuning and for coordinating install-time distributed predictive tuning. It includes two NVIDIA GeForce 1080Ti GPUs each with 3584 CUDA cores and 11GB of global memory, 20 Intel Xeon cores (2.40GHz) and 65GB RAM.

Table 5.3: System Parameters for Server Device used for development-time autotuning.

Server Setup			
NVIDIA GeForce GPU		**Intel Xeon CPU**	
GPU Cores	3584	CPU cores	20
GPU Frequency	1.12 GHz	CPU Frequency	2.40GHz
GPU SMs	136	DRAM Size	65 GB
Global Memory Size	11 GB		

5.6.4 Autotuning Setup

For autotuning search, we use the OpenTuner [75] library. For both empirical and predictive tuning, we use the default OpenTuner setting that uses an ensemble of search techniques including Torczon hillclimbers, variants of Nelder-Mead search, a number of evolutionary mutation techniques, and random search.

For each QoS threshold, we run the tuner for a maximum of 30K iterations. We declare convergence if tuning result doesn't improve over 1K consecutive iterations. The iterations required per QoS threshold varies across benchmarks, from 1K (LeNet) to 28K (ResNet-50). Predictive and Empirical tuning convergence rates are similar across all benchmarks with an average 8.7K iterations, and 8.2K iterations, respectively.

Selecting Configurations for Shipping. Before QoS validation, we select configurations that lie within an ε_1 distance to the Pareto set (Line 33 of Algorithm 5.1); after autotuning, we select and ship configurations within ε_2 distance to the Pareto set (Line 41). These distance thresholds $\varepsilon_{1,2}$ are computed per-benchmark to limit the maximum number of configurations validated and shipped. We chose $\varepsilon_{1,2}$ so that at most 50 configurations are validated and shipped. For our benchmarks, this reduces the number of points by 87x.

Distributed Predictive Tuning Setup. We emulate a setting with 100 edge devices and a single-server coordinator. Lacking 100 TX2 boards, we measure performance on 1 (out of 100) distributed invocations on the actual TX2 hardware, for 1/100 of the total calibration inputs. Error profiling and accuracy validation use functional execution on the server.

Runtime Approximation Tuning. On the Tegra TX2, we vary GPU frequency to mimic low-power execution modes, using 12 different frequencies from 1.3Ghz to 319Mhz. The performance goal given to the runtime is to maintain the level of performance offered at the highest frequency mode (1.3Ghz). The frequency is updated after a batch of inputs has been processed and before the next batch starts. The frequency change is applied instantaneously.

For reliable measurements, we run 200 batches of 500 images each; we divide the 5K test set into 10 batches and have 20 such runs. We average the processing time and accuracy across batches. The experiments use Control strategy 2 from Section 5.5, with a sliding window size of 1 batch execution (500 images). When frequency is reduced at the end of

batch n, we measure the imposed slowdown at end of batch $n+1$, compute the required speedup to meet the target performance, and ApproxTuner picks a new configuration from the trade-off curve. Batch $n+2$ is then executed with new approximation configuration. The overhead of the runtime system to switch between configurations is negligible.

5.7 EVALUATION

We experimentally show benefits of ApproxTuner. We analyze each of 3-stage in Section 5.7.1, (development-time), Section 5.7.4 (install-time), and Section 5.7.5 (runtime). We characterize tuned approximations in Section 5.7.2. We show predictive tuning in Section 5.7.3. We demonstrate composite tuning (with multiple QoS metrics) in Section 5.7.6.

5.7.1 Performance and Energy Improvements

Improvements for GPUs. Figures 5.2a and 5.2b show the performance and energy benefits achieved on the Tegra's GPU. The X-axis presents the benchmarks, and the Y-axes represent improvements over the FP32 baseline. We show the results for three levels of specified accuracy reduction in percentage points: $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, $\Delta QoS_{3\%}$. These tuning results have only hardware-independent approximations (FP16, perforation, sampling). The improvements are reported after trying both predictors Π_1 and Π_2, and choosing the best result (we compare the two predictors in Section 5.7.3).

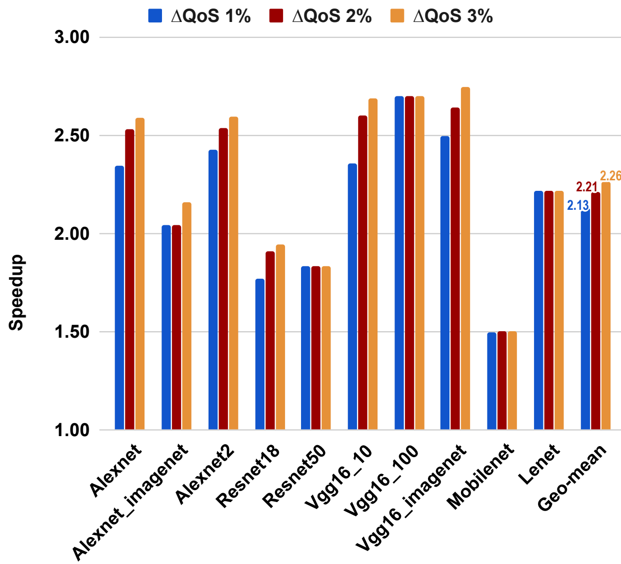
For $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, and $\Delta QoS_{3\%}$, the mean speedups are 2.13x, 2.21x, and 2.26x. The maximum speedup achieved is 2.75x for VGG-16-ImageNet at $\Delta QoS_{3\%}$. On average, FP16 alone provides 1.63x speedup, and moreover, has little effect on accuracy. Sampling and perforation together give an additional 1.4x speedup, on top of FP16 (e.g., total average speedups of 2.26x for $\Delta QoS_{3\%}$).

Figures 5.2a and 5.2b show that increasing loss threshold from 1%, to 2%, and 3%, provides higher improvements in six out of ten benchmarks, since it allows the tuner to gradually apply more aggressive approximation levels. Four networks (VGG16-100, ResNet50, MobileNet, LeNet), do not show gradual improvement, because more aggressive sampling and perforation of most layers increase the quality loss immediately beyond 3%.

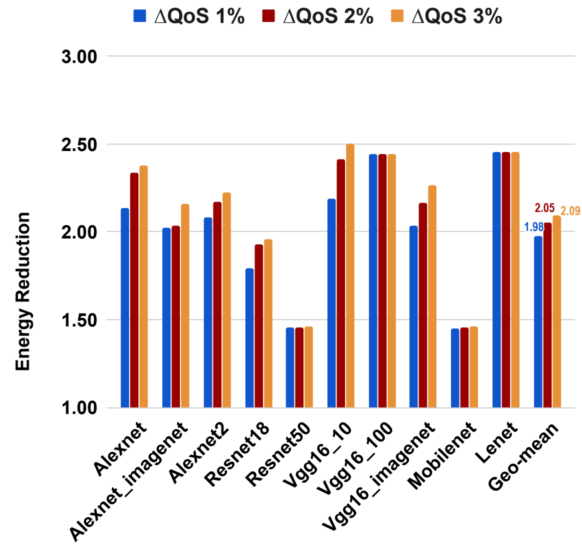
Energy reductions (Fig. 5.2b) are loosely correlated with performance increases. For $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, and $\Delta QoS_{3\%}$, the mean energy reductions are 1.98x, 2.05x and 2.09x.

Improvements for CPUs.

The mean speedups for CPUs for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$ and $\Delta QoS_{3\%}$ are 1.31x, 1.38x and 1.42x. Figure 5.4a shows the speedups for individual benchmarks. The maximum speedup



(a)



(b)

Figure 5.2: (a) Speedups and (b) Energy reductions achieved on GPU using generic approximations for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, $\Delta QoS_{3\%}$.

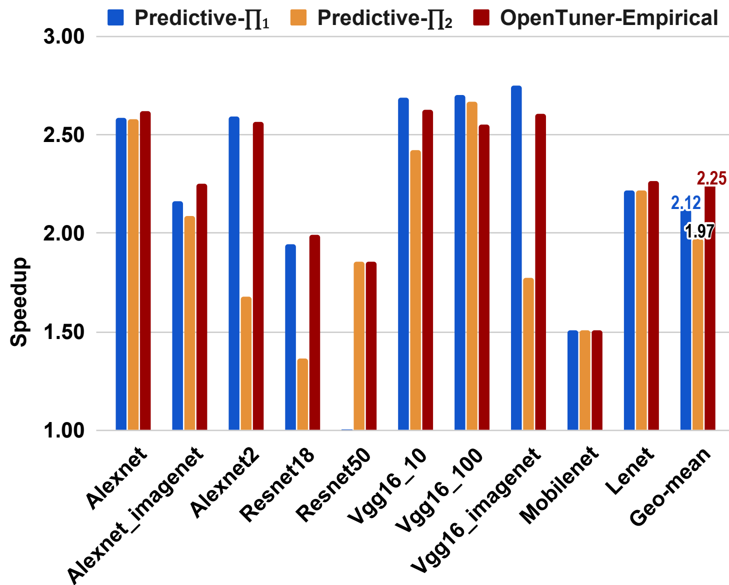


Figure 5.3: Speedups on GPU with predictive and empirical tuning at development time for $\Delta QoS_{3\%}$.

is 1.89x for VGG16-CIFAR10. The energy benefits (across thresholds) are quite similar. Figure 5.4b shows the speedups for individual benchmarks. The benefits on the CPU are significantly lower than on the GPU (though still valuable) since the ARM CPUs on the

Jetson TX2 board do not support FP16, and so the performance and energy benefits are due only to sampling and perforation. This particularly affects MobileNet and ResNet-50, which are not amenable to sampling or perforation.

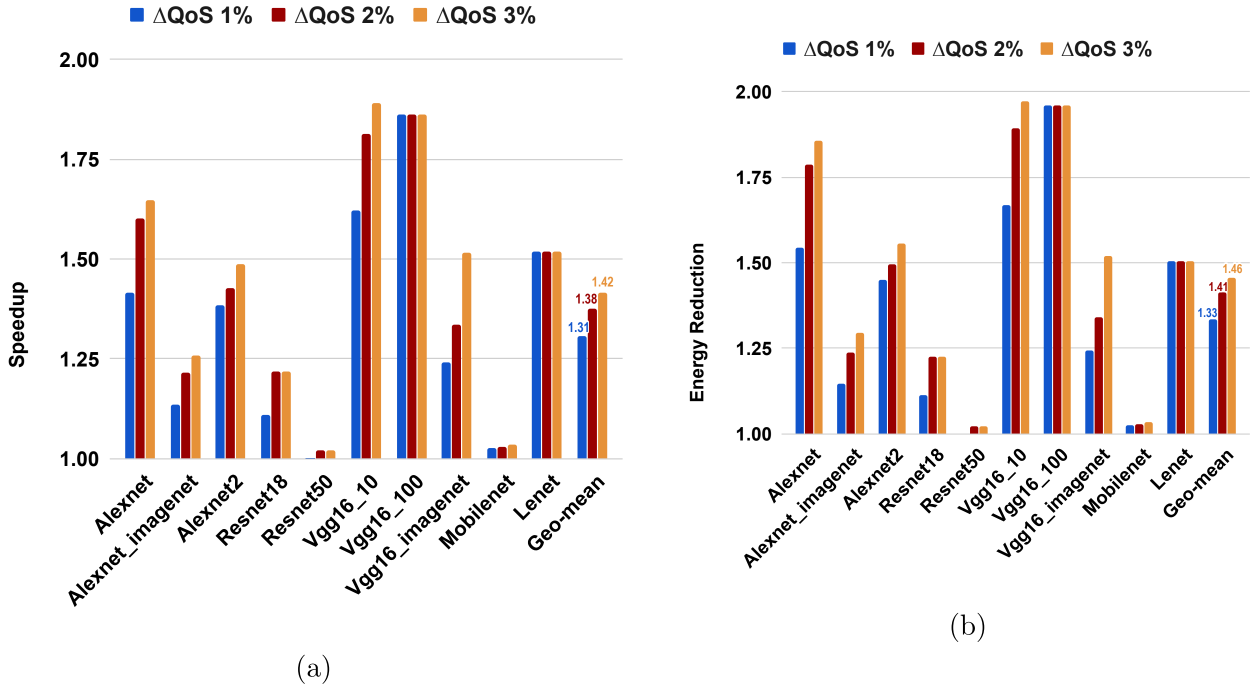


Figure 5.4: (a) Speedups and (b) Energy reductions achieved on CPU using generic approximations for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, $\Delta QoS_{3\%}$.

5.7.2 Characterizing Approximations

Table 5.4 shows the best performing GPU configurations found by ApproxTuner when the QoS constraint is set to 3 percentage point drop in inference accuracy. These show the number of tensor operations that are assigned to the supported approximation knob settings (or assigned no approximation).

General Trends. We find that the first few layers in the CNNs are relatively less suitable for approximations compared to the later layers. For 4 of the 10 CNN benchmarks, the first layer is not mapped to any of the approximation knobs (other than FP16). In ResNet-18 and MobileNet, the first 3 layers are not mapped to perforation or sampling. *These insights show the importance of combining different approximations and the importance of tuning the choices of combinations to balance accuracy vs. performance gains.*

We summarize some interesting insights for some representative CNNs. The other CNNs show similar behaviors to the ones mentioned below (for example, AlexNet2-CIFAR10 behaves similarly to AlexNet-CIFAR10.)

LeNet: We find LeNet to be highly approximable. The two convolution layers in LeNet can map to a variety of different approximation knobs (both perforation and sampling) with reasonable accuracy impact (below the threshold).

AlexNet-CIFAR10: None of the layers in AlexNet (across configurations) map to the perforated convolutions approximation, while all layers (in most configurations) are amenable to filter sampling. We also find that certain convolution operators (in layers 3, 4 and 5) have relatively less room for approximations since only 25% sampling could be mapped to these layers - higher levels of sampling result in high accuracy losses.

ResNet18: Across all configurations, 7 of the 21 convolution layers are not mapped to any approximation. Interestingly, 4 of the 21 layers are only mapped to 33% perforation and all 4 perforations start at different start offsets. Such observations confirm the hypothesis that varying start offsets with perforation (and sampling) combine well together.

VGG16-100: We find that 3 layers in VGG16-100 can only be mapped to column-based perforation while row-based perforation leads to high accuracy loss. This shows the value of having row vs column perforation as a knob exposed to the tuner.

MobileNet: For the best configuration on GPU, only 8 layers (out of 28) could be mapped to approximation techniques without significant accuracy loss, which is why MobileNet has the least performance improvement (1.50× speedup) across benchmarks. Interestingly, we find that certain layers in MobileNet (in layers 5, 9, 10) map well to column perforation but have high accuracy loss with row perforation.

VGG-16-ImageNet: It is much more amenable to filter sampling than perforation (similar to AlexNet). Convolution layers 6, 9, and 10 can map to all knobs of sampling.

ResNet50-ImageNet: Across all configurations, at most 13 convolution (from a total of 53) are mapped to any approximation (excluding FP16), with 10 convolution operators mapped to 25% perforation. As a result, ResNet50-ImageNet achieves a relatively low speedup of 1.77×, a large portion of which comes from FP16.

5.7.3 Predictive vs Empirical Tuning

Comparing Speedups. We compare our predictive approximation tuning with empirical tuning, both using OpenTuner system [75]. Figure 5.3 illustrates results for the maximum bound of 3%. It shows that predictors Π_1 and Π_2 provide (geometric) mean speedups of 2.12x and 1.97x, compared with 2.25x for empirical tuning. For ResNet-50, Π_1 was not

Table 5.4: Approximation knobs for top performing GPU configuration (maximum speedup) for $\Delta QoS_{3\%}$.

Benchmark	Occurrences of Approximation Knobs
LeNet-5	samp-50%:1 perf-50%:1 FP16:2
AlexNet-CIFAR10	FP16:2 samp-50%:3 samp-25%:1
AlexNet2	FP16:3 perf-50%:1 samp-50%:2 perf-33%:1
VGG-16-10	FP16:4 perf-50%:3 perf-33%:2 samp-50%:6
VGG-16-100	FP16:4 perf-50%:2 samp-50%:8 perf-33%:1
ResNet-18	FP16:13 perf-50%:6 perf-33%:2 samp-25%:1
MobileNet	FP16:20 perf-50%:3 perf-33%:3 perf-25%:2
AlexNet-ImageNet	FP16:2 perf-50%:1 perf-25%:3
VGG-16-ImageNet	FP16:8 perf-50%:1 samp-50%:7
ResNet50-ImageNet	FP16:38 perf-50%:1 perf-33%:4 perf-25%:10

usable because it required too much memory. Moreover, using Π_1 *or* Π_2, whichever is best, for each network, gives a mean speedup of 2.26x, which matches empirical tuning. For most benchmarks, Π_1 effectiveness is similar to empirical tuning; Π_2 provides lower speedups because it systematically underestimates accuracy loss for some benchmarks, and therefore chooses configurations for those benchmarks that are later removed in the accuracy validation phase.

Effectiveness of Predictors. Figure 5.5 shows the AUC for each benchmark. The mean AUC across benchmarks for Π_1 is 3.37 which is close to empirical tuning an AUC 3.4. Across benchmarks, Π_2 gives a much lower AUC of 2.79 which shows that Π_2 is less effective at finding high quality configurations. These values do not include ResNet-50 since Π_1 could not be applied to ResNet-50 due to memory constraints.

Autotuning Times. Table 5.5 shows the autotuning time for predictive tuning compared to empirical tuning using OpenTuner. Predictive-p1 and Predictive-p2 are on average 13.75x and 17.9x faster than empirical tuning. Π_1 calculations are significantly slower than Π_2's on large tensors, e.g., 3.8x and 6.7x slower on VGG16-ImageNet and AlexNet-ImageNet, respectively. This shows the importance of having both predictors: for large models and data sets, Π_2 is more likely to be feasible and gives good speedups with reasonably good accuracy, while Π_1 is usually more precise but requires more memory.

5.7.4 Install-time tuning

We evaluate the efficacy of our install-tuning strategy in exploiting the low voltage knobs in the PROMISE accelerator that trade accuracy for energy savings (Section 5.2.3).

Energy Reductions using PROMISE accelerator. Figure 5.6 shows the energy reductions achieved with install-time predictive distributed tuning compared to empirical tuning.

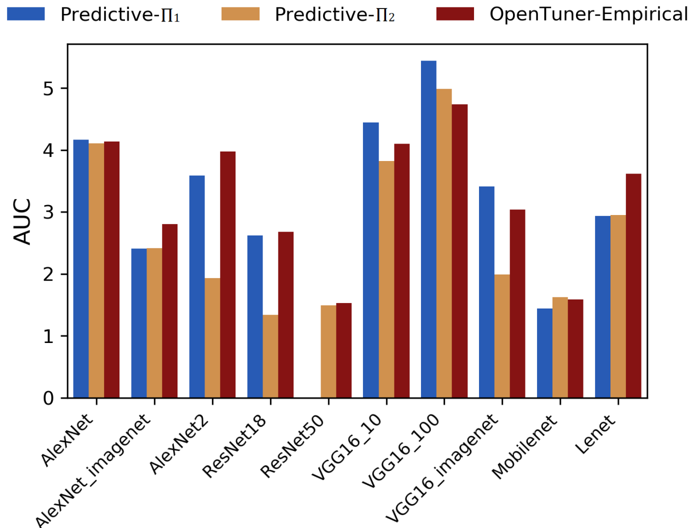


Figure 5.5: Area under the tradeoff curve (AUC) obtained through predictive and empirical autotuning. Each tradeoff curve is ΔQoS against speedup, and the area is computed between $\Delta QoS = 0$ and $\Delta QoS = 3$.

Table 5.5: Predictive Tuning times compared to Empirical (in minutes). "P1-red" and "P2-red" are speedups compared to Empirical.

Benchmark	Empirical	Pred-P1	Pred-P2	P1-red	P2-red
LeNet	11.4	1.0	1.1	11.21x	10.61x
AlexNet-CIFAR10	418.7	9.9	11.7	42.27x	35.80x
AlexNet2	133.9	11.1	12.4	12.09x	10.78x
VGG-16-10	1015.3	36.2	25.8	28.06x	39.35x
VGG-16-100	494.6	30.8	25.2	16.03x	19.64x
ResNet18	373.1	27.3	38.4	13.68x	9.72x
MobileNet	772.7	58.7	38.4	13.17x	20.10x
AlexNet-ImageNet	661.1	240.1	25.5	2.75x	25.93x
VGG-16-ImageNet	1937.1	334.9	143.5	5.78x	13.50x
ResNet50-ImageNet	16272.9	—	1042.6	—	15.61x
Geo-mean				**13.75x**	**17.90x**

The energy reductions are achieved by mapping tensor operations to the PROMISE accelerator and lowering the analog read swing voltages to further lower energy use at the cost of increased error. On average, using predictor Π_1 provides 4.5x energy reduction, compared to empirical that provides 4.8x reduction. Using predictor Π_2 provides reduction of 3.6x. It is lower than Π_1 due to higher prediction error, which leads to missing optimal configurations during search space exploration.

5.7.5 Runtime Approximation Tuning

GPU frequency has a significant impact on the average system power for Tegra Tx2 (similar for other edge devices). In other words, reducing GPU frequency helps lower overall

power consumption. Figure 5.8 shows how the GPU, DDR memory, and total system (SYS) average power (over 10 runs) varies with increasing frequency. These measurements are gathered for ResNet18-CIFAR10 (others have similar trends). While DDR power only slightly increases (DDR frequency is kept constant), the GPU power increases 7x when frequency increases from 318MHz to 1300MHz. Overall, average power increases by 1.9x with frequency increasing from 319MHz to 1300MHz.

Due to space constraints, we show 3 of the evaluated CNNs in Figure 5.7 (others show similar behavior). The X-axis presents the different frequencies, which we vary over time. The left Y-axis presents the *normalized execution time*, relative to time taken at the highest frequency level (1.3Ghz in our experiment). The right Y-axis presents the model accuracy (in %).

As we reduce the frequency, the baseline configuration (blue lines in Figure 5.7) slows down substantially, while accuracy remains unaffected. ApproxTuner's dynamic tuning counteracts the slowdown and maintains the original average batch processing time (orange lines), while gracefully degrading the inference accuracy (yellow lines, right Y-axis).

The experiment demonstrates that the amount of slowdown tolerated depends on the permissible QoS threshold. For instance, for ResNet18, a slowdown of 1.45x (at 675Mhz) can be counteracted with a accuracy drop of 0.33 percentage points, but as much as 1.75x (at

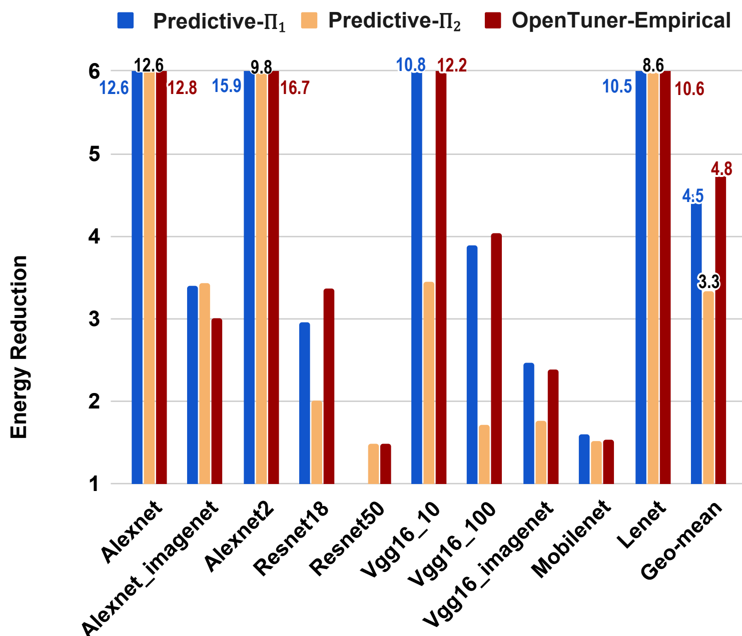


Figure 5.6: Energy reductions on GPU + PROMISE with install-time distributed predictive tuning (p_1, p_2) and empirical tuning for $\Delta QoS_{3\%}$.

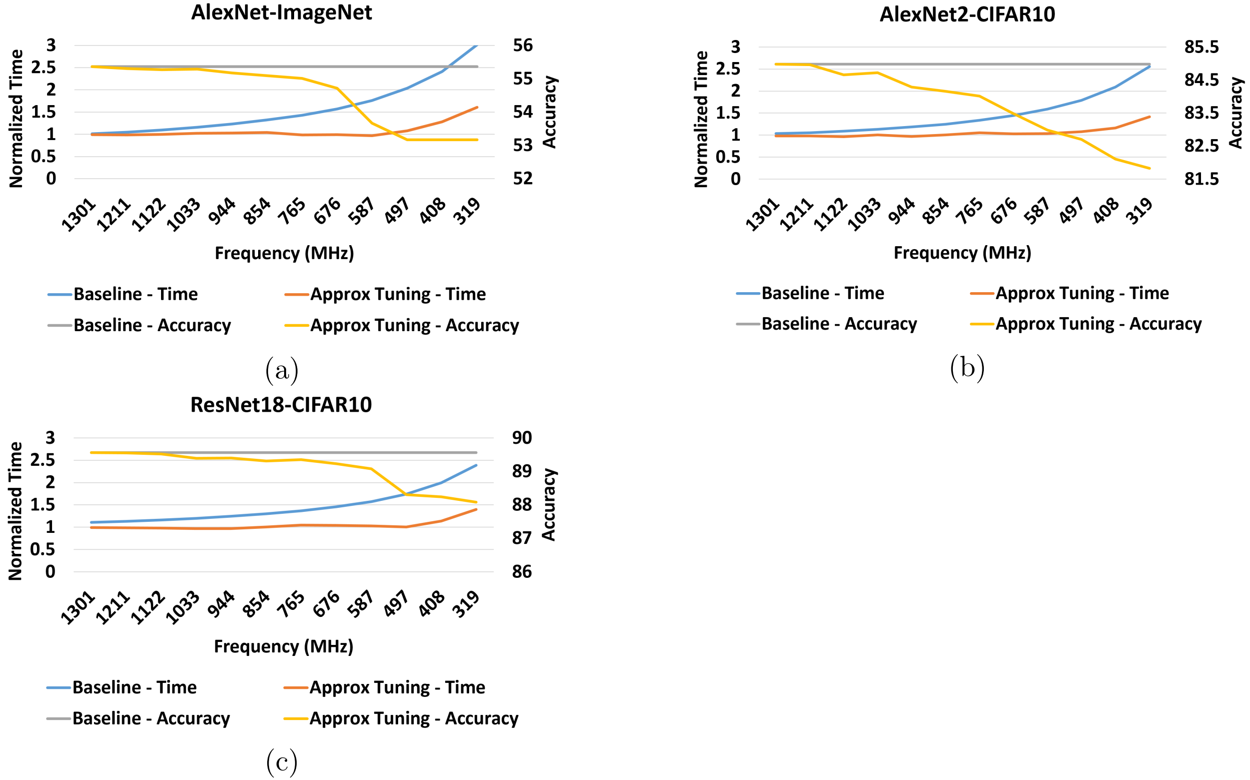


Figure 5.7: The figures a), b) and c) show our runtime approximation tuning trades off accuracy to maintain the same level of responsiveness when frequency levels are reduced. The times on y-axis are normalized with respect to performance achievable at highest frequency (1.3Ghz). Without dynamic approximations (the blue line), applications slows down.

497 MHz) can be compensated with an accuracy drop of 1.25 points. At 497MHz, there is a 1.72x reduction in average power consumed (Figure 5.8). Similarly, for AlexNet-ImageNet and AlexNet2-CIFAR10, frequency can be reduced up to 586MHz (with 1.7 and 1.9 points of accuracy loss), while maintaining performance. This reduces average power consumption by 1.66x and 1.62x, respectively (power-frequency graphs not shown for AlexNet and AlexNet2).

5.7.6 Combining CNNs and Image Processing

We experiment with a combined CNN and Image processing benchmark to evaluate: a) can ApproxTuner tune for multiple QoS metrics? and b) do the predictive tuning strategies apply beyond CNNs?

The application consists of a CNN (AlexNet2 on CIFAR-10) to classify images to one of target classes, and only images belonging to specific classes are forwarded for Canny edge detection. QoS is a pair *(PSNR, accuracy),* where PSNR measures the quality of edge detection and accuracy measures if the correct images were sent to the image filter.

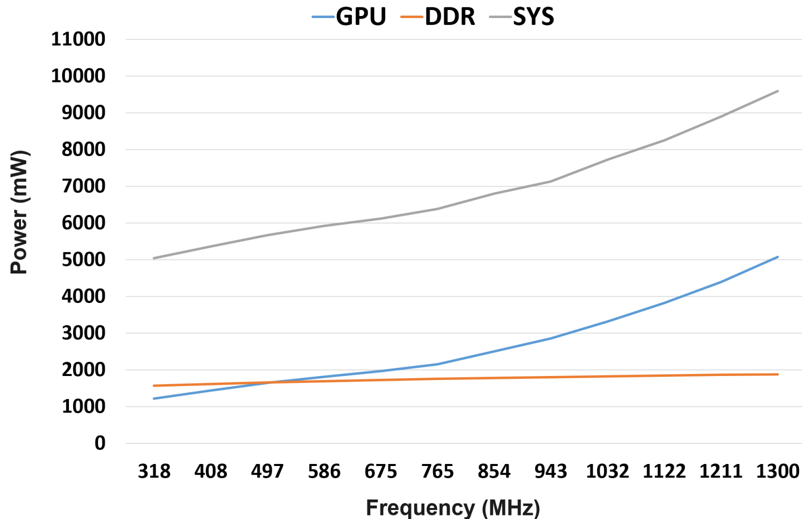


Figure 5.8: GPU (blue line), DDR (orange line), and overall system (grey line) power across frequency levels.

Figure 5.9 presents the best configurations achieved on GPU with each QoS pairs. Nine QoS pairs are evaluated with three different values of PSNR and three different accuracy values. The Y-axis shows speedup compared to FP32 baseline. As either threshold loosens (PSNR decreases or accuracy loss increases), speedup increases, since the tuner finds more opportunities for approximation.

For all pairs *(PSNR, accuracy),* predictive and empirical tuning find configurations with comparable speedups, while predictive is 20.1× faster. In five of the nine pairs, predictive tuning gives slightly higher speedups. Our inspection shows that this is the effect of the non-deterministic nature of OpenTuner (i.e., randomness in search).

5.8 EXPLORATORY STUDY: TUNING FOR PRUNED MODELS

Model compression techniques such as pruning [97] and quantization [176, 177] have grown popular. The primary goal of DNN pruning is to reduce the model size and it is shown that it doesn't always offer performance improvements [120, 121, 122, 123]. Although adding pruning (and quantization) to ApproxTuner is beyond the scope of this paper, we want to investigate if approximations intended for speedup (such as perforation and sampling) can be applied to pruned models with acceptable accuracy loss.

We perform this experiment in a Pytorch framework with our supported approximations emulated (using artificial error injections) and applied to pruned models, using empirical tuning. We consider perforation (for convolutions) and not filter sampling since the latter is

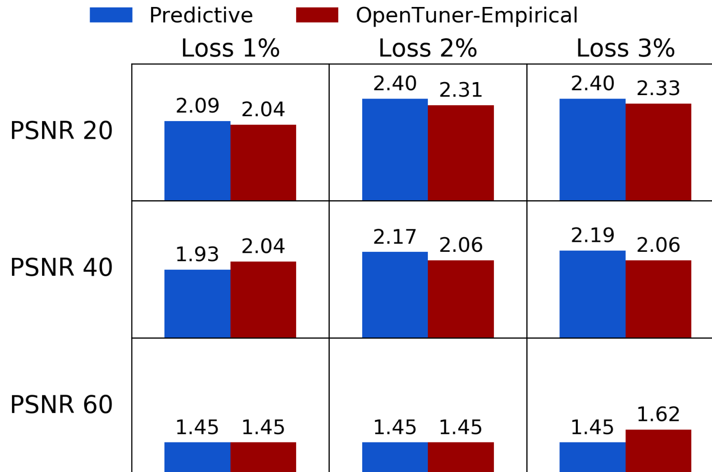


Figure 5.9: Speedups achieved on GPU for a grid of accuracy (horizontal) and PSNR (vertical) thresholds.

similar to pruning (skipping connections). We are currently unable to do this in ApproxTuner since it does not support sparse tensors required by pruned models. We consider MobileNet, VGG16, and ResNet18 (CIFAR-10) pruned up to 95.6%, 95.6%, and 91.4%, respectively, using learning rate rewinding [92].

Figure 5.10 shows the results of optimizing pruned models. Pruning itself causes a small drop in accuracy (<1%). The X-axis shows *additional* drop in accuracy incurred by approximations (perforation). Y-axis shows the resulting reduction in multiply-accumulate operations (MACs). We report MACs as a proxy measurement for speedup; we were unable to measure actual execution times since ApproxTuner does not have efficient sparse kernel implementations.

For less than <1% additional loss in accuracy, across three benchmarks we observe noticeable reduction in MACs (over the pruned model). The configurations for MobileNet model give up to 1.3x reduction in MACs, similar for VGG16, and 1.2x reduction for ResNet18. The key takeaways are that (a) approximation techniques (other than feature sampling), can be applied to pruned models without unacceptable loss of accuracy, and (b) that these techniques have at least the potential to give valuable additional speedups.

5.9 CONCLUSION

We proposed ApproxTuner, a compiler and runtime system that uses a 3-phase tuning approach including development-time, install-time, and runtime tuning. ApproxTuner uses performance and accuracy prediction heuristics to tune the program at development-time

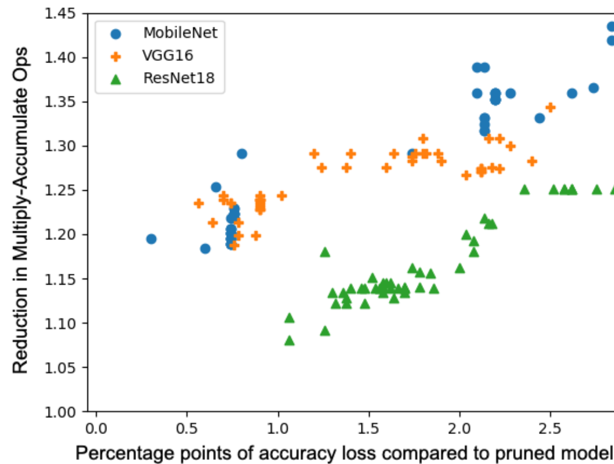


Figure 5.10: Results of tuning pruned models.

and generates a tradeoff curve, it refines this tradeoff curve with performance measurements and hardware-specific approximations at install-time, and uses this tradeoff curve at runtime to switch configurations efficiently in response to changing runtime conditions. Across 11 benchmarks, ApproxTuner delivers a geometric mean performance improvement of 2.1x on the GPU, and 1.3x on the CPU, with only 1 percentage point drop in accuracy. Dynamic tuning capabilities allow ApproxTuner to adapt application performance to changing runtime conditions. Overall, ApproxTuner provides a generic approximation-tuning framework that is extensible to a wide range of software and hardware approximations, for important application domains such as neural networks and image processing. Our future work includes extending ApproxTuner to other domains and applying it with an even broader of algorithmic optimizations.

CHAPTER 6: APPROXROBOTICS: THE COST AND ACCURACY TRADEOFF FOR SMALL MOBILE ROBOTS

6.1 INTRODUCTION

This paper presents an empirical study of tradeoffs, which highlights the significant role that computational approximations can play in enabling low-cost visually guided autonomous robots. Autonomous robots are increasingly reliant on visual information for perception, planning, and control [83, 84, 85, 9, 10, 86, 87]. Visual data can be very high dimensional, yet, with advances in deep learning, we are now seeing many applications that are able to extract actionable information through this data. A major challenge is that inference with these deep learning models is highly computationally expensive to run, especially on hardware devices with limited compute and memory resources. Large robots such as autonomous cars can afford to have much larger computational payloads, since these are powered by carbon fuels or large batteries, have sophisticated cooling systems, the cost of compute hardware can be kept to a small fraction of the total cost. In contrast, small battery-powered autonomous robots such as those used for agriculture, mining, or remote area exploration, have much tighter size, weight, and power constraints. Furthermore, cost-sensitive fields like digital agriculture [88] impose stringent cost constraints as well. For such robots, optimizing the computational requirements for visual navigation can be crucial.

Hence, a key open question is how the computational requirements for visual navigation can be optimized to use low cost hardware, in small battery-operated mobile robots. Answering this question can provide robot system designers with the understanding necessary to make optimal hardware choices. Indeed, conservative choices for compute hardware can be typically traced back to unclear computational requirements of the task-specific software stack. Even more unclear to robot system designers is whether there is room to reduce computational demands by using software optimizations.

In this paper, we provide an empirical study that sheds light on the cost and accuracy tradeoff for small mobile robots. A key insight in this work is that, *in the context of feedback control systems, it may be possible to relax the accuracy of neural networks models for vision (which are usually the most computationally expensive part of a navigation system), without significantly hurting navigation robustness*. In particular, we show that approximate, pruned neural network models, which can trade off accuracy for speed, can still provide sufficient task accuracy when used in robust closed-loop autonomous systems. This enables robot designers to safely relax some accuracy constraints, and therefore select lower-cost hardware, without

expecting to lose task performance quality, even when performing demanding real-world visual inference tasks such as autonomous navigation in agricultural fields.

In this work, we evaluate these research questions in the context of a small mobile agricultural robot, *TerraSentia* (obtained from EarthSense [89]), a production agbot that is used for autonomous navigation through corn fields for high-throughput phenotyping and a variety of production agriculture tasks. Our results, however, are applicable to any battery operated small robots that rely on feedback control driven by visual inference for task execution. The key goal of our work is to investigate *to what extent can approximations be used to trade off model accuracy for performance improvements (in particular, increased frames-per-second) in the neural-network models used in small autonomous robots.* These speedups can then facilitate using lower-cost compute hardware, or performing additional computations on the same hardware, or combinations of the two. The primary task under consideration is visually-guided autonomous navigation between crop rows using a state-of-the-art visual guidance system, CropFollow [90]. CropFollow uses a (low-cost) front-facing monocular RGB camera with a pair of convolutional neural networks (CNNs) for predicting the robot heading and the distance from crop rows, which is then used to perform autonomous row navigation.

To optimize the CNN models, we use a popular technique, *structured weight pruning* [91, 92, 93], which compresses CNN models by dropping a subset of convolution filters that have relatively small weights. There is a lot of recent work on network pruning [93, 94, 95, 43, 96, 97, 98], but it remains unclear 1. if it is feasible to apply pruning to CNNs used in the context of a larger real-world application, especially in the context of closed-loop control 2. is it acceptable to relax some accuracy to gain additional performance while avoiding observable impact on the end-to-end quality of the application, i.e. without losing control robustness. Our focus on end-to-end robustness is in contrast with most existing approaches to neural network pruning, which are aimed at retaining the computational accuracy. This conservative approach to pruning limits the achievable computational gains as it does not leverage the inherent robustness of closed-loop autonomous systems.

On the other hand, with our approach of trading off accuracy without losing robustness (measured as crashes that needed manual interventions), we are able to drive far more aggressive computational performance improvements, ranging up to 15x (on CPU), with close to a 2x increase in inference error. *These performance speedups allow us to perform the entire navigation pipeline, including two CNNs, Bayesian sensor fusion and Model Predictive Control on a low-end $35 Raspberry Pi4 [99].*This compares with the $876 Intel NUC [100] used on commercial TerraSentia robots, and with the $59 Jetson Nano, the cheapest device we found to deliver necessary performance without approximations. Moreover, the Pi4 requires 30% lower peak power than the Jetson Nano (7W vs 10W).

In realistic deployment scenarios, these robots may perform additional tasks such as real-time data analytics [178], automatic weed removal [179, 178], and automated berry picking [180] among others.

To evaluate if our optimizations facilitate running multiple tasks on a single resource-constrained Raspberry Pi4, we also apply our pruning approach to *corn stand counting*: a video analytics task for counting corn stands. We show that by only slightly relaxing requirements for the accuracy of the final result counts enables stand counting to run in real-time, concurrently with the full navigation pipeline.

Specifically, our contributions are:

- We perform the first empirical study to characterize the impact of neural network model pruning on the end-to-end navigation quality of an autonomous robot with visually guided feedback control.

- We show that pruning the convolutional neural network models (CNNs) used in the visual perception system helps provide the minimal required FPS from the vision models (for crash-free navigation) on a $35 Raspberry Pi4, making it a feasible choice for compute hardware.

- We find that the CNN-based autonomous navigation control in the evaluated agbot is robust to infrequent mispredictions. We also identify pruning settings that introduce large prediction errors that greatly impact the navigation quality, resulting in crashes.

- By relaxing accuracy constraints, we show that multiple compute-intensive tasks including the navigation pipeline and 2 instances of stand counting can run on a shared compute-constrained Raspberry Pi4.

6.2 BACKGROUND: ROBOT SYSTEM DESIGN

6.2.1 Hardware Setup.

The TerraSentia robot is a compact, light weight robot designed to maneuver between crop rows for commodity crops such as corn, and soybean. It is 55.88 cm long, 30.48 cm wide and 33.02 cm tall and weighs about 13.6 kg.

In the configuration used for our study, it used a forward-facing camera with 720p resolution at 30 fps for row following using the algorithm in [90]. It also has two identical side cameras on the left and right which are used for high-throughput phenotyping. An

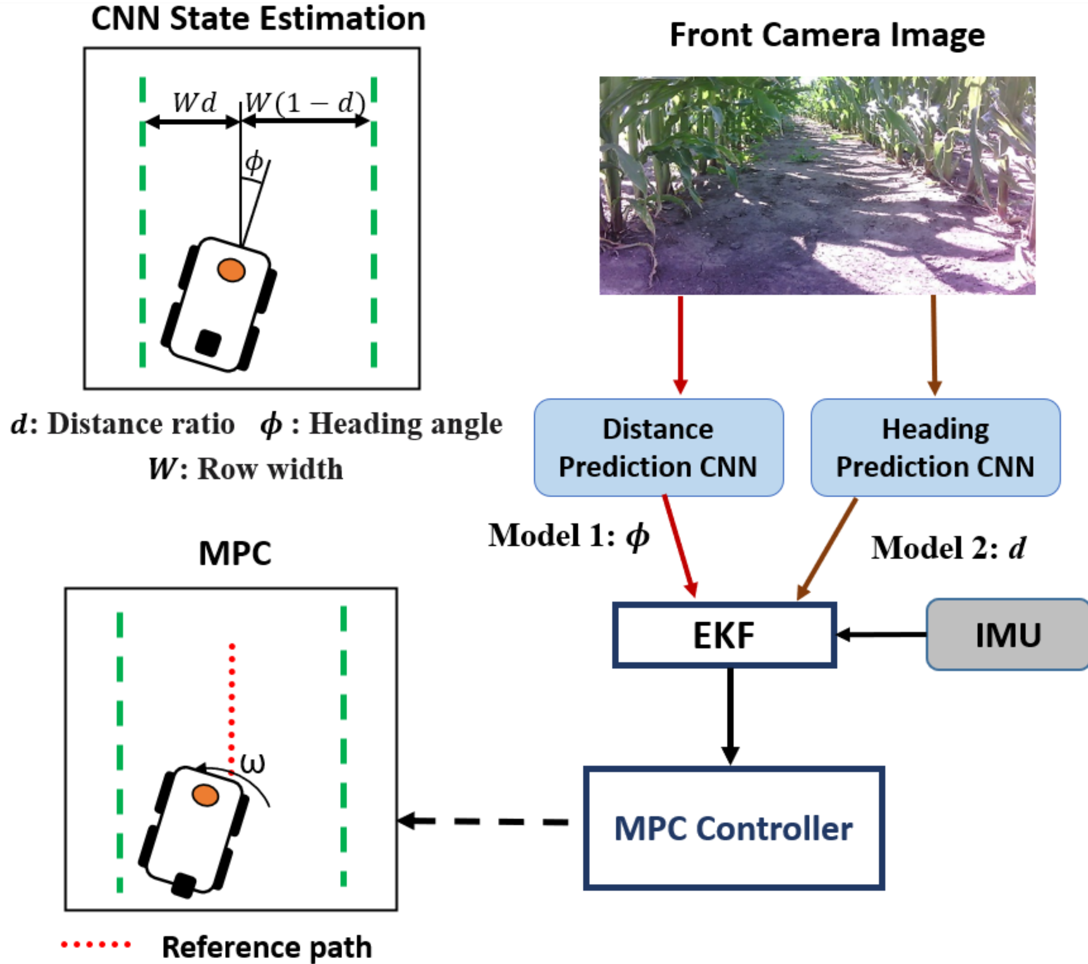


Figure 6.1: Workflow of the CropFollow [90] Navigation Pipeline. The front camera images feed into CNNs for distance and heading predictions. The CNN predictions are fused with IMU measurements using an Extended Kalman Filter (EKF). The fused state estimations are used by the MPC for computing angular velocity commands.

embedded 6 DoF Inertial Measurement Unit (IMU) gathers real-time measurements of angular velocity and acceleration. The robot also hosts two 2D horizontal-scanning LIDARs (one scanning parallel to the ground in the front and one perpendicular in the back) that are not used in this work. The TerraSentia comes standard with an Intel NUC 10i7FNH as the primary computer, and a Raspberry Pi3 as a secondary (controls) computer. The Pi3 on the standard robot runs lower-level control logic, such as the speed and the angular rate controller. There are a myriad of other digital processors onboard which are delegated to lower-level control or data-collection tasks that are not relevant here. The Intel NUC is used for the compute-intensive tasks, and in particular CNN models used with the visual navigation pipeline [90] run on the integrated GPU. In addition, it runs the high-level MPC

controller and real-time data processing workloads. An Intel NUC 10i7FNH of the same specification as on the robot costs $876 at the time of release [181]. In our work, we want to replace the Intel NUC on the robot with a single Raspberry Pi4 board that costs $35. The Raspberry Pi4 includes a quad core cortex-A72 (ARM v8) with max CPU frequency of 1.5 GHz with 2 GB of main memory. The Pi4 has an integrated GPU that is dedicated for graphics-only.

6.2.2 Autonomous Navigation Software Setup.

Figure 6.1 shows the workflow of the vision-based navigation pipeline [90].

Perception Module. We utilize the CropFollow visual navigation system developed in [90] as the primary perception module. The perception module takes as input 320×240 RGB images (resized from 720P resolution) and outputs robot heading θ and relative distance d in the crop row using a CNN for heading prediction and another CNN for distance prediction. The relative distance d is the ratio of the left distance (from center of robot to left crop row) to the total distance between two rows, i.e., $d = dL/(dL + dR)$. CropFollow uses a ResNet-18 backbone pretrained on ImageNet and fine-tuned on labelled crop data, and has been extensively field validated [90]. In CropFollow, the last layer in both models is modified to output a continuous value for regression task. We make no architectural modifications to CropFollow other than neural network approximations described in Section 6.3, below.

Our goal in this work is to study the effect of network optimization on the robustness of the perception and control system. For this purpose, we replaced the default ResNet-18 in CropFollow with the SqueezeNet-v1.1 model [182] since it has 4.4× fewer parameters and 3.3× fewer floating point operations (FLOPs) than ResNet-18. SqueezeNet-v1.1 also has a lower execution time (3.3× faster) and lower accuracy (14% higher error in heading prediction and 11% higher error in distance prediction) compared to ResNet-18. This makes SqueezeNet an interesting trade-off point.

IMU Fusion with Extended Kalman Filter. An Extended Kalman Filter (EKF) is used to fuse CNN predictions with inertial measurements from IMU. The Kalman filter reduces the effect of uncertainties from a single source (vision or IMU), thereby reducing the probability of abrupt control variations (e.g., a sudden large turn). The Kalman filter takes as input a) CNN distance prediction, b) CNN heading prediction, c) robot's linear speed from the wheel encoders, d) angular speed from the IMU, and e) the robot state at the previous time step s_{k-1}, to compute the current state s_k. This state includes a heading estimate and a distance estimate.

Model Predictive Controller. A non-linear Model Predictive Controller (MPC) receives the EKF estimates for heading and distance and solves a constrained optimization problem (with curvature radius as constraints) to compute angular velocity commands that keep the robot following the reference path. Since we evaluate "row following," the reference path is a straight line through the center of the crop lane.

6.2.3 Real-time Stand Counting Task.

Real-time analytics tasks in digital agriculture are relevant for crop breeders, and also farmers that need to closely monitor crop health and crop yield [183]. Performing these tasks in real-time has the advantage of 1. delivering results faster to users, 2. reducing data transfer and storage costs for large video/data recordings to be analyzed offline, and 3. reduced cloud and server costs for analyzing the data [2]. One such task we evaluate is corn stand counting: using a stream of images from side cameras to count the number of corn stands in each crop row. This process involves object detection, which is extremely expensive for resource constrained systems [184] and challenging to execute in real-time. We evaluate two algorithmic variants for corn stand counting with the goal of achieving reasonable accuracy, while being able to perform this task in real-time.

The first algorithm `standc-track` uses an object detector for detecting individual corn stands in image frames, and an object tracker that keeps track of specific corn stand instances across frames to avoid duplicate counting of the same corn stand. For each image frame, the detector returns bounding boxes around corn stands, which are sent to the tracker to update its internal state. This approach is similar to other methods for object tracking [185, 186] and can be used in other similar tasks that require real-time object tracking [187].

We also evaluate a relatively inexpensive algorithm, `standc-stride`, which uses the assumption that the robot travels at a constant speed. This is a reasonable assumption when the robot is in autonomous mode in which it maintains constant speed. Using this constant robot speed known ahead of time, the algorithm selects frames from the image stream at a *stride* such that the corn stands are not repeated in these frames, nor are any stands missed. Therefore, the overall stand count is the sum of counts (given by object detector) over these frames. This algorithm applies object detection to much fewer frames compared to the other variant.

In both variants, we use a Single Shot MultiBox Detector (SSD) as the object detector [188], with a MobileNet-v2 [189] backbone. For object tracking, we use SORT [190], which is the fastest object tracking algorithm in MOT Challenge 20 [191].

6.3 APPROXIMATION: MODEL PRUNING

In the literature, many different generic or domain-specific approximation techniques have been proposed for optimizing neural network computations with (potentially) small loss in inference accuracy. These include perforated convolutions [192] (skipping and interpolating some output computations), weight pruning (skipping some features in the convolutional filters) [45, 43, 91, 92], and integer quantization [43] to name a few. In this work, we apply weight pruning to the convolutional neural networks used in the navigation pipeline and corn stand counting task.

Weight pruning can be categorized into two major types: 1. *unstructured pruning* [45, 43] which removes individual weights that are deemed less important to the overall computational accuracy (e.g, low-magnitude values), and 2. *structured pruning* [91, 193] which removes groups of contiguous weights, for instance, entire filters and channels from convolution layer weights. Unstructured pruning provides much higher reduction in model sizes (up to 13× in prior work [43]) than structured pruning (up to 4.5× in prior work [193]) since it allows removing weights at a finer granularity. However, prior work [121] has shown that unstructured pruning introduces unpredictable sparsity in the underlying tensor computations (e.g., convolutions, matrix multiplications), which can lead to slowdown on highly parallel architectures not well suited to irregular computational patterns, such as GPUs. In contrast, structured pruning preserves dense computation patterns, which better utilizes parallel hardware and thus often delivers higher speedups.

In this work, we adopt the structured pruning technique proposed by Li et al. [91]. We start by training the network for a fixed number of epochs. Then, we repeatedly 1. remove a fraction of filters with lowest L1-norm values from each convolution layer, and 2. retrain for a fixed (smaller) number of epochs to recover some accuracy. While [91] is a one-shot pruning technique, we use it iteratively to obtain multiple models with different performance-accuracy tradeoffs.

We apply this structured pruning approach to heading and distance prediction models (Section 6.2.2), and the object detection model (Section 6.2.3) used in corn stand counting. In these models, convolutions are the most computation-intensive operations, hence, pruning filters improves performance.

6.4 EXPERIMENTAL METHODOLOGY

Datasets and Model Training. Both ResNet-18 and SqueezeNet-v1.1 for heading and distance prediction are pretrained on ImageNet and fine-tuned on 25K corn images with

heading and distance ratio labels generated from manually annotated vanishing lines [90]. 20K images are used for training and 5K images for validation. The learning rate used for both models is 1e-4 (with AdamW optimizer).

Heading and distance prediction are *regression* tasks, and they use the *mean-squared error* l_2 as the loss function in training. We also report the *95-percentile* of *mean-absolute error* l_1 on the validation set: 95% of the validation set images have a mean-absolute error below this value. For these scalar regression tasks, l_1 and l_2 are defined as:

$$l_1(pr, gt) := |pr - gt|, \quad l_2(pr, gt) := (pr - gt)^2 \tag{6.1}$$

where pr is the predicted value and gt is the groundtruth.

The object detector SSD-MobileNet-V2 is pretrained on the COCO [194] dataset and fine-tuned on 1K corn-only images with manually labelled bounding boxes. With a 4-to-1 split, 800 images are used for training and 200 for validation. In addition, common data augmentation techniques are applied. The learning rate used this model is 2e-4.

In the evaluation of stand counting algorithm (as opposed to evaluation of object detector), we use a dataset of 60 videos, each labelled with the total stand count (a number). Both datasets are captured using the side cameras during navigation runs of 0.5 m/s. This dataset contains the same kind of corn (early-season corn) as the object detector is trained on.

For all 3 models, the initial training (baseline models) lasts 100 epochs, and for each prune step, additional 10 retraining epochs are performed with the same learning rate as initial training.

Model Pruning. We apply the pruning technique in Section 6.3 with 20 iterations. Each pruning iteration removes 20% filters with lowest L1-norm independently for each layer.

Due to skip connections in ResNet and SqueezeNet, often fewer than 20% filters get skipped – skip connections require that filters at non-matching output indices be retained (to avoid mismatch in output dimensions). Each of the 20 pruning iterations generates a pruned model that is progressively more pruned and has fewer filters than the previous iteration. We refer to the output model of each pruning iteration as a *prune level* and label these from 1 to 20 inclusive (0 is the unpruned baseline). Higher prune level means fewer parameters and floating-point operations (FLOPs) but higher error.

We use the same prune level for both heading and distance models when evaluating the navigation quality. This is because both models run concurrently with their collective output fed into the EKF, hence, one model running slower than the other slows down the pipeline.

Hardware Setup. We use a Raspberry Pi4 (Section 6.2.1) to evaluate whether pruning enables the use of low-cost hardware. To understand the impact of FPS (image frames

processed per second) changes, we need a device that provides high-enough FPS to allow artificially varying the FPS. For this, we use a NVIDIA Jetson Nano (4GB memory) [195] that provides approximately 25 FPS on the baseline ResNet-18 models for heading and distance prediction.

Field Experiments Setup. We performed our experiments on production corn fields in late season (fully-grown corn). Each navigation run covers 360 meters over four different corn rows, each 90 meters in length. Each run varies between 6 minutes to 12 minutes depending on the robot speed. We evaluate 5 navigation speeds: 0.5 m/s, 0.6 m/s, 0.8 m/s, 1.0 m/s, and 1.2 m/s.

In all the runs, (at least) one human observer follows the robot and measures the number of *crashes*, which is when the robot crashes into the corn due to row-following algorithm failure and requires human intervention. The number of crashes in a run decides its *navigation quality*. Correspondingly a *crash-free run* is a run with strictly 0 crashes.

Offline Analyses. To facilitate our offline analysis (done after field experiments), we recorded experimental data as *rosbags* [196] for these *rostopics*: front camera outputs, heading and distance CNN outputs, EKF outputs, and MPC outputs.

Frameworks and Toolchains. All the DNN models are trained and pruned in PyTorch before being exported to ONNX [197], a standard intermediate representation for neural networks. For executing the models on the Raspberry Pi4 CPU, we use the ONNX runtime [198] with all optimizations enabled. For model execution on Jetson Nano, we use the NVIDIA TensorRT compiler [199] which automatically applies optimizations including quantization to FP16, and layer fusion, among others.

The Kalman filter (EKF) and MPC, are developed as ROS components in C++ and Python for the CropFollow work, and their code is not modified in our work.

6.5 EVALUATION

We evaluate these research questions: **RQ1:** Does pruning reduce the latency of CNN predictions and improve the FPS (frames processed per second). **RQ2.** What are the minimum FPS requirements from the vision models for crash-free navigation, and does pruning help satisfy these minimum FPS requirements on the Raspberry Pi4? **RQ3.** How do ResNet-18 and SqueezeNet pruned models compare in terms of delivering crash-free navigation? **RQ4.** Can pruning object detection models used in corn stand counting enable it to execute real-time, concurrently with navigation? **RQ5.** How does prediction error and FPS impact navigation quality?

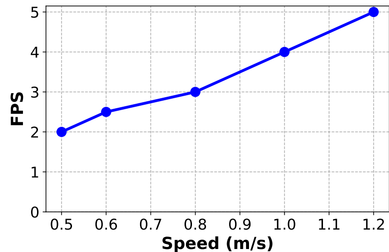


Figure 6.2: Minimal required FPS of the vision DNNs for different navigation speeds.

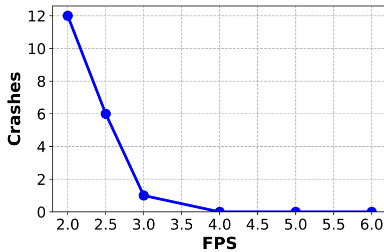


Figure 6.3: Impact of increasing FPS on navigation stability at 1.0 m/s measured by crashes, using baseline ResNet-18 heading and distance models.

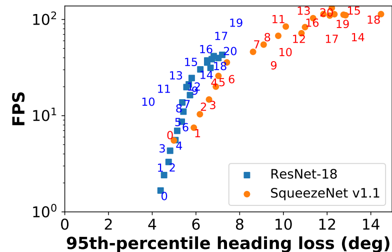


Figure 6.4: 95th percentile loss (in degree) and FPS of ResNet-18 and SqueezeNet heading models at different prune levels (annotated on each point).

We begin with (i) establishing the minimal FPS requirements of the vision models (heading and distance) necessary for robust navigation at different navigation speeds, and (ii) evaluate the accuracy-vs-FPS tradeoff for the individual neural networks at various pruning levels.

6.5.1 Establishing Minimal FPS Requirements

For a model FPS f, where navigation is crash-free, we declare f to be the *minimal FPS* if all lower FPS values lead to crashes. In practice, we check 0.5 FPS below f to confirm f as the minimal FPS and consider this to be sufficient precision. Figure 6.2 shows how the minimal FPS of the heading and distance models (baseline ResNet-18 models) varies with increasing robot speed. At 0.5 m/s, 2 FPS was sufficient for crash-free navigation. At the highest speed of 1.2 m/s, the minimal FPS is 5. Overall, minimal FPS increases with increasing navigation speed. The reason is that higher speeds require faster control decisions from the MPC, which requires faster heading and distance estimates from the EKF, subsequently related to the FPS of the CNN predictions (Figure 6.1).

Figure 6.3 shows how increasing FPS improves the navigation quality. This experiment is performed on the Jetson Nano with the baseline ResNet-18 models and navigation speed set to 1.0 m/s. At an FPS of 2 (lower than minimal FPS), the robot had 12 crashes over the 360-meter run. Increasing the FPS to 3 reduces the number of crashes to 1. With FPS greater than or equal to 4, the navigation stabilized and resulted in no crashes.

6.5.2 Evaluating Prune Levels on Validation Set

Figure 6.4 shows the tradeoff space of prune levels (level 0 for baseline) for the heading prediction model; distance prediction models show a similar trend. The tradeoff is

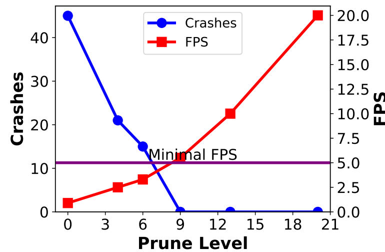


Figure 6.5: ResNet-18 performance (FPS) and navigation quality (crashes) across different prune levels.

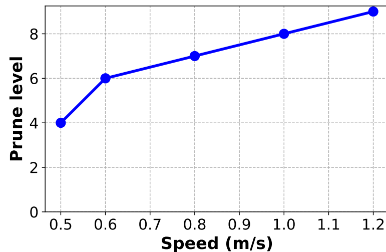


Figure 6.6: Minimal prune level to achieve the required FPS for different navigation speeds.

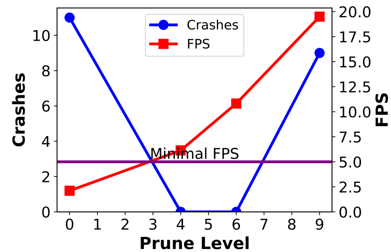


Figure 6.7: SqueezeNet performance (FPS) and navigation quality (crashes) across different prune levels.

between performance measured in FPS (number of images processed per second) and the *95th-percentile loss* (defined in Section 6.4) on validation set. Losses for the heading model are measured in degrees. Therefore, points to the upper left are better tradeoff points. The performance is measured when running the models on the Raspberry Pi4 in isolation (no other task running) – the results when running the complete CropFollow algorithm (including both DNN models, EKF and MPC) is shown later.

SqueezeNet is both smaller and less accurate than ResNet-18, thus covering a different part of the tradeoff space. The baseline version of heading SqueezeNet has 95th-percentile loss 14% higher than baseline heading ResNet-18, and baseline distance SqueezeNet (not shown here) has 11% higher loss than distance ResNet-18. On the other hand, baseline SqueezeNet is 3.3× faster than baseline ResNet-18 (5.5 FPS compared to 1.7).

Figure 6.4 shows that pruning provides significant speedups in FPS. With each higher prune level (each point is labeled with the prune level in the figure) the loss and FPS both increase. At the highest prune level of 20, ResNet-18 provides 43 FPS (a 25× increase from 1.7 FPS of baseline), while SqueezeNet provides 132 FPS (24× from 5.5 FPS). When considering relatively lower accuracy loss, ResNet-18 provides a better tradeoff as these points are more to the above and left than the SqueezeNet ones. For instance, ResNet-18 heading model at prune level 20 has higher FPS (43 FPS) while also having slightly lower loss (7.2 degrees) than SqueezeNet prune level 6 (36 FPS and 7.4 degrees). This shows, somewhat surprisingly, that pruning the larger, more accurate (but initially more expensive) model can achieve better frame rates *and* better accuracy than the smaller model.

6.5.3 Using Pruned Models to Achieve Minimal FPS

We now answer the second research question: whether pruning the vision models makes it possible to deploy the navigation pipeline on a single Raspberry Pi4 (including EKF,

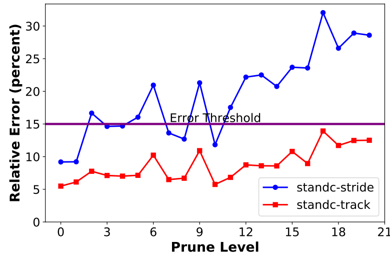


Figure 6.8: Relative error (in percent) of two stand counting algorithms, measured at their respective minimal FPS.

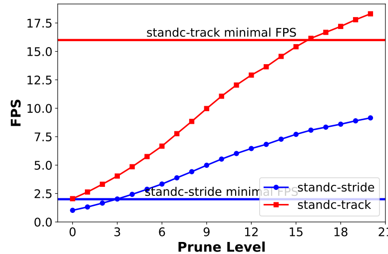


Figure 6.9: FPS of detector in two stand counting algorithms. FPS of `stand-stride` is measured with 2 instances running, while FPS of `stand-track` is measured with 1 instance.

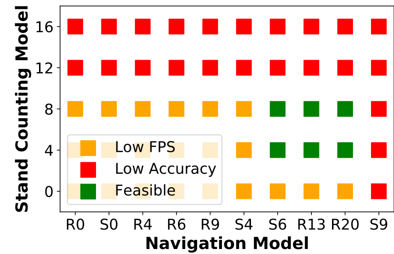


Figure 6.10: Prune levels of vision model in navigation and detector in stand counting. Rx represents ResNet-18 level x; S for SqueezeNet. Each point is color-coded by its feasibility.

MPC, and other ROS components). Without pruning and with the full pipeline running, the baseline ResNet-18 models deliver 0.9 FPS, which is significantly less than the minimal FPS requirement described in Section 6.5.1 (2 FPS @ 0.5 m/s). Section 6.5.2 demonstrates pruning can provide significant performance gains with some loss in accuracy. Two questions are left unclear: (a) are these performance gains enough to achieve the minimal FPS for crash-free navigation, and (b) whether the additional accuracy loss affects the navigation quality. We use ResNet-18 models in these experiments.

For speed 1.2 m/s, Figure 6.5 shows how varying the prune levels affects the FPS and the navigation quality (number of crashes). The purple line shows the minimal required FPS from Figure 6.2, which is 5 FPS for speed 1.2 m/s. The red line shows the FPS achieved at the different prune levels. As higher prune levels are evaluated (left to right on x-axis), the FPS increases (right-side y-axis), while the blue line showing the number of crashes (left-side y-axis) steadily decreases. At prune level 9, the FPS surpasses the minimal FPS, and the number of crashes reduce to 0. This result shows that pruning not only provides the minimal FPS, but also the prediction accuracy of the models is still reasonable enough to correctly guide the control decisions. We evaluate the prediction accuracy of this setting further in Section 6.5.6.

Figure 6.6 shows the minimal prune level required to achieve the minimal FPS (of ResNet-18) on the Pi4 across different speeds. Higher speeds require higher FPS, as shown in Figure 6.2, and hence higher prune levels.

6.5.4 Comparing ResNet and SqueezeNet Pruned Models

We discuss how different prune levels of ResNet-18 and SqueezeNet compare in terms of navigation quality. Figure 6.5 shows the evaluated prune levels (x-axis) for ResNet-18 at speed 1.2 m/s. For ResNet-18, the navigation quality increases as the FPS surpasses the minimum, and does not decrease at higher prune levels (13 and 20), even though the prediction error is expected to increase, as shown in Figure 6.4.

For a comparison, Figure 6.7 shows the prune levels, the FPS, and the number of crashes of SqueezeNet for each evaluated setting. The baseline SqueezeNet provides an FPS of 2.1, which is insufficient for stable navigation and results in 11 crashes. Then, SqueezeNet provides 6.1 FPS at prune level 4 and 10.8 FPS at prune level 6, both surpassing the minimum and resulting in no crashes. However, unlike ResNet-18, prune level 9 that delivers sufficient FPS (19.5 FPS) still results in entirely unstable navigation with 9 crashes. The higher prune level results in much higher errors (see Figure 6.4), causing the increased crashes. *This result shows the importance of tuning the prune levels to find a suitable tradeoff between performance and prediction accuracy.* In Section 6.5.6, we further discuss how high prediction errors in the vision models can lead to poor quality control decisions.

6.5.5 Navigating with Task – Real-time Stand Counting

We examine the fourth research question – how model pruning improves stand counting results – using the two stand counting algorithms in Section 6.2.3. We run 2 instances of stand counting in parallel (for left and right video streams), concurrently with the navigation pipeline, all running on the Pi4. For accuracy evaluation, we calculate the *mean relative error* of stand counts over a video data set (Section 6.4):

$$MRE(pr, gt) := \frac{1}{N} \sum_{i=0}^{N-1} \frac{|pr[i] - gt[i]|}{gt[i]} \tag{6.2}$$

where N is the size of the dataset, pr is the predicted stand counts, and gt is the groundtruth stand counts. Throughout this experiment, we use an acceptable accuracy threshold of 15% mean relative error, and since the evaluation dataset is recorded at 0.5 m/s, we set navigation speed to 0.5 m/s.

Hard FPS Constraints. Based on the navigation speed of 0.5 m/s, we calculate that the stride at which `standc-stride` skips entire frames is 15, i.e., two frames that are 15 frames apart would contain non-overlapping but adjacent views. Therefore, 1 image is sent to detector per 15 images. Together, this yields a minimum requirement of 2 FPS which is

a hard constraint. Similarly, we find 16 FPS to be a hard constraint for `standc-track`, which uses the SORT object tracker, below which the stand counting quality is irrecoverable. Intuitively, this is because SORT does not consider image features (only bounding boxes). It is significantly more difficult to track stands below some FPS where consecutive frames move more than the distance between bounding boxes.

Algorithm Comparison. Figures 6.8 and 6.9 shows the stand count errors and FPS achieved by the detector respectively. In Figure 6.8, errors are measured at the FPS requirement of each algorithm. With `standc-track`, we find even the most pruned model cannot provide 16 FPS with 2 instance running. Therefore, in Figure 6.9, FPS of `standc-track` is measured with 1 instance, while `stand-stride` is with 2 instances.

For each algorithm, a prune level is feasible only if its error is below threshold in Figure 6.8 and its FPS is above threshold in 6.9. Therefore for `standc-track`, prune level 18 gives the lowest error of 11.7%, while for `standc-stride`, the lowest error of 11.8% is achieved at step 10. While both algorithms achieve similar lowest error within the given hardware constraint, `standc-stride` outperforms `standc-track` as it can execute with 2 instances. When counting the stands in a field, this will reduce the travel distance of the robot by half as the robot only need to cover alternate rows. We will use `standc-stride` for the stand counting algorithm in the following evaluation.

Tradeoff Space of Stand Counting and Navigation. Figure 6.10 illustrates the overall tradeoff space, presented by prune level options in both navigation and stand counting. For the vision models in the navigation pipeline, there are two "families" of models: Rx represents ResNet-18 at prune level x, and Sx represents SqueezeNet at prune level x. They are sorted on the x-axis by the FPS they provide. Each point represents two choices, made for the two prune levels, and leads to one of three outcomes: 1. *Low FPS*, when either navigation or stand counting doesn't meet its FPS requirement; 2. *Low Accuracy*, when either component doesn't meet its accuracy requirement; 3. *Feasible*, when there is no violation in accuracy or FPS.

The FPS numbers are measured with `standc-stride` and navigation pipeline running together. The accuracy of stand counting is shown above in this subsection, while for the navigation accuracy we use results from 6.5.3 and 6.5.4.

Given the heavily compute-constrained hardware, the feasible region in the figure is far away from the origin (baseline). The first feasible navigation model, SqueezeNet at prune level 6, is 6.5× faster than SqueezeNet baseline and 21.5× faster than ResNet-18 baseline. This shows aggressive approximations are necessary in both components for the system to meet its requirements.

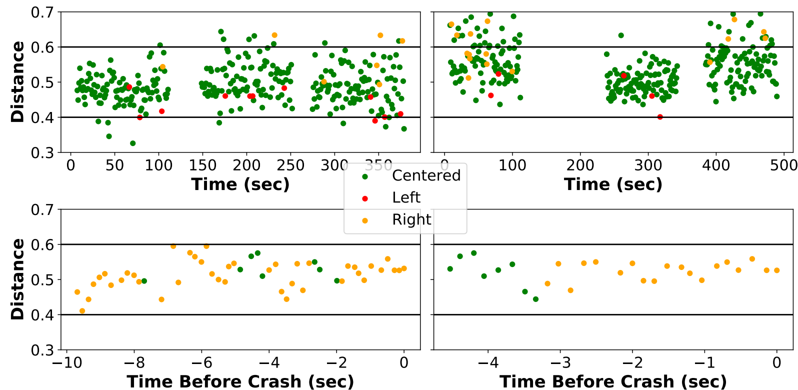


Figure 6.11: Distance prediction (point location) and groundtruth (point color) of ResNet prune level 9 (top-left) and 20 (top-right), and two crashes from SqueezeNet prune level 9 (bottom).

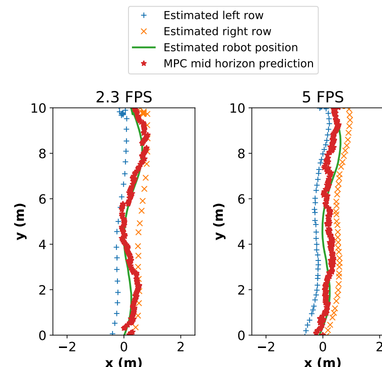


Figure 6.12: MPC predictions - ResNet prune level 4 (2.3 FPS) vs ResNet prune level 4 (5 FPS).

6.5.6 Analyzing Navigation Quality Across Pruning Settings

The evaluations above has shown that both the prediction accuracy and FPS of the models affect navigation quality, by comparing the navigation quality of different prune levels. Below we further demonstrate these effects with some additional analyses, and show the different failing patterns of navigation when prediction error is too high or FPS is too low.

Effect of model prediction error. Here we compare the distance prediction and groundtruth in some runs to show the distance prediction error.

Since the groundtruth distance values in a run is unknown, we manually annotate some image frames retrieved from rosbag recordings. For prune level 9 and 20 of ResNet-18 that are crash-free, we label the whole 360-meter run. In addition, we select 2 crashes from prune level 9 of SqueezeNet and label 10 seconds before each crash. We annotate images with categorical labels: images visibly on the left or right boundary are labelled as “left” and “right”, otherwise “center.” Two human annotators label these images and cross-check the labels to reduce annotation ambiguity.

Top of Figure 6.11 shows the distance predictions and groundtruth over time in the run for ResNet-18 prune level 9 (left plot) and ResNet-18 prune level 20 (right plot). Each dot represents one time-step (1 second granularity); its y-value is the distance prediction at the moment, and its color indicates the groundtruth label of the corresponding image frame. The distance predictions used here are IMU-fused EKF distance (Section 6.2.2) instead of the distance model’s output, which better display how mispredictions mislead MPC even with IMU corrections. These predictions are between 0 (left-most) and 1 (right-most). The range 0.4-0.6 is considered the center of the corn row (demarcated on the figure).

This figure shows two types of mispredictions. A red (left) or yellow (right) dots in the center region is a *false negative*, which tells the controller the robot is centered when it's not, and may lead to a crash. A green dot outside the center region is a less dangerous *false positive*, because when the robot is centered, there is more time to recover from an incorrect turn.

Prune level 20 of ResNet-18 has more false positives, and is less well-centered (seen from more yellow dots) than prune level 9. However, in both runs most of the predictions are overall correct, and our field experiments (Section 6.5.3) confirmed that both settings lead to crash-free navigation. In contrast, prune level 9 of SqueezeNet results in multiple crashes and an overall unstable run. Figure 6.11 shows the 2 crashes selected from this run; other crashes have similar behavior. For both crashes, the central region of the figures show that the model consistently predicts the robot to be in the center while it is on the right boundary. This makes the robot susceptible to crashing into the right corn boundary. Due to these mispredictions, 9 crashes in total occurred in this run.

Effect of model FPS. The output of MPC at a given moment shows the path that the robot decides to take in the near future, which we find to be effective in highlighting the effect of vision model FPS.

We first define the *mid-horizon prediction* output from MPC. At each moment, the MPC solves an optimization problem to obtain N "amount of turn" values (presented as angular velocity values) that the robot executes in the subsequent time steps. More specifically, with the Dubins' car model [200], the path that the robot will follow in the near future (over the *MPC horizon*) is a function of these N angular velocity values. MPC optimizes these values to minimize a cost function that measures the centeredness of the MPC horizon. The MPC horizon is 4 meters long with 19 angular velocities commands spaced at 0.2 meters. The *mid-horizon prediction* is defined as the mid-point (2 meters from current position) of the full MPC horizon. We pick the mid-horizon prediction for illustration since subsequent MPC output updates the current one, and the robot is unlikely to follow the full horizon from any output.

Figure 6.12 shows the estimated robot position (from encoders and IMU), left/right row (from EKF using IMU and vision models), and MPC mid-horizon prediction (from MPC), when running navigation at speed 1.2 m/s with the ResNet-18 models at prune level 4. As shown in Figure 6.3, ResNet-18 prune level 4 results in crashes since it only delivers 2.3 FPS while 5 FPS is the minimal FPS requirement at speed 1.2 m/s. Left of Figure 6.12 shows 10 seconds before one such crash using the rosbag recording for this run without modification.

On the right of Figure 6.12, we *reenact* the same crash but with the vision model running at a higher FPS. We apply the same vision models to the front camera video from rosbag

recording at 5 FPS, and supply these to MPC to generate new MPC mid-horizon predictions; this is performed offline.

Figure 6.12 illustrates how increasing FPS of CNN predictions can potentially improve the control decisions and avoid a crash. In the left plot, the MPC mid-horizon predictions are closer to the row boundaries (blue and orange dotted lines) and overlaps in some cases, indicating that the robot is likely to border on or collide into the boundary when following this given path. The right plot shows that MPC predictions are relatively more centered and in no instance overlaps with the row boundaries. This is because higher FPS predictions allow the MPC to quickly calibrate the angular velocity commands, and accordingly the path predictions. Overall, the experiment shows how high FPS of CNN predictions can improve navigation control.

6.6 DISCUSSION AND CONCLUSIONS

Alternate Choices for Compute Hardware. For vision-based navigation, the $59 Jetson Nano (2GB) the $75 Myriad were also feasible alternatives to the NUC, providing 25 FPS (on GPU) and 10 FPS respectively, for vision models. However, Intel Myriad needs a host device to which it connects via USB, and hence the cost of compute would need to account for a host board. In contrast, the Jetson Nano is a standalone device with on board CPU and GPU. Since Nano provides sufficient FPS on the baseline models (using the GPU), pruning the models is not strictly necessary for navigation alone. However pruning models on the Nano is still beneficial since it can potentially frees up the GPU (more compute-capable than CPU) for heavy-weight workloads while only using the CPU (which has similar specs to Raspberry Pi4) for the full navigation pipeline. The Raspberry Pi4 was our hardware of choice considering it has significantly lower cost ($35) and power consumption (7W) compared to Nano (10W) [201].

Future Research. A promising avenue for future research is other agriculture tasks such as berry-picking [180] that also use compute-intensive models. Efficiently running these tasks on commodity edge hardware is an open research problem.

In conclusion, our study shows that exploiting accuracy-performance tradeoffs can offer significant opportunities for optimization in autonomous robot navigation systems used in production. We believe that these results will open up further avenues for relaxing accuracy constraints in other related application domains, such as autonomous drones, autonomous vehicles, and other similar mobile robots.

CHAPTER 7: IMPLICATIONS FOR CURRENT PRACTICE

With the slowdown of Moore's law and end of Dennard scaling, there is increasing expectation that production compilers and runtimes deliver software optimizations that help bridge the gap between compute demands and limited compute resources. The promising results in this thesis show that there is an opportunity for production compilers and runtimes to re-envision application accuracy as a tunable quality knob as opposed to a strict correctness issue, and exploit approximation techniques to achieve performance and energy improvements. While many proposed approximations in literature have shown value in improving performance for a wide range of domains, approximate computing remains a largely untapped opportunity for optimization in production systems. I believe ApproxHPVM is the first step towards practical adoption of approximations in real-world applications and deployments.

ApproxHPVM and ApproxTuner are majorly focused on optimizing convolutional neural networks (CNNs). The performance optimizations shown for these CNN models indicate that there is potential for optimizing other machine learning models such as recurrent neural networks (RNNs), Transformers, and generative adversarial networks (GANs). Most of these models consist of similar tensor operations, for instance, convolutions and matrix multiplication, and hence approximations in this work are also applicable in the context of other machine learning applications, and more generally other tensor-based domains (e.g., image processing applications). ApproxHPVM is currently limited to supporting approximations and approximation-tuning for tensor operations, hence, in its current form it cannot be used for non-tensor domains. ApproxHPVM is being continuously extended to include support for a wider range of application domains and operation types.

The predictive tuning models used in ApproxTuner have only been evaluated in the context of CNNs and an image processing benchmark. For non-tensor domains, I believe there is potential for exploring the same idea of predicting end-to-end accuracy from the accuracy impact on individual operations. The predictive model Π_1 is less generalizable to other domains since it uses and sums up the tensor differences before evaluating the predicted QoS. Π_2 was shown to be less precise in predicting the accuracy for CNNs but is more generalizable to other domains since it only considers the differences in the end-to-end QoS.

ApproxHPVM draws a distinction between *hardware-independent* and *hardware-specific approximations*. This has implications for application developers that make their programs available for download on package managers and want to benefit from ahead-of-time optimizations before shipping the application to a user's device (mobile, laptop, tablet etc.).

Hardware-independent approximations produce the same result output across hardware devices and can hence be performed in the ahead-of-time development-tuning phase; this approach has the benefit of reducing install-times, improving the end-use experience. On the other hand, hardware-specific approximations (e.g., accelerators with approximation knobs) are an important optimization opportunity that an application should exploit to achieve maximum performance. Application developers are encouraged to identify the components that may benefit from install-time tuning and use ApproxHPVM to autotune the program with these additional hardware-specific knobs.

ApproxHPVM and ApproxTuner take a slightly different approach to approximation tuning. ApproxHPVM uses accuracy metrics to assign error budgets in an approximation-agnostic manner, while ApproxTuner directly tunes the program with approximations applied in autotuning loop. We found the approach of using L1 and L2 norm accuracy metrics to work well with errors that are independent of the inputs used (random Gaussian errors) and not well with approximations that introduce noise that is highly dependent on the values in the input tensors (e.g., perforation, sampling). If developers are to use approximations that introduce i.i.d. errors (independent and identically distributed), approximation metrics have the major benefit of enabling a very fast install-time phase where approximation selection is merely a lookup table. In contrast, ApproxTuner uses the relatively more expensive approach of retuning the program with additional hardware-specific choices, but it applies to a much wider range of approximations since it makes no assumptions about the types of error introduced by approximation.

ApproxHPVM is developed with the philosophy of making it easier for developers to use approximations by exposing only high-level accuracy specifications to use for tuning. The choice of high-level accuracy specifications to use and selection of application component to approximate are left to the developer/user. In scenarios where the amount of tolerable accuracy loss is unclear, users are encouraged to experiment with a variety of different thresholds and measure the impact of the overall application quality, in the specific context of use. For instance, the amount of PSNR loss acceptable on a image processing filter used in a photo editing application is likely dependent on how much noise (introduced by approximations) is imperceptible to the human eye.

Approximations are particularly relevant in application domains and usage scenarios where loss in accuracy has minimal effect on the overall quality of the larger application deployment. For instance, in this thesis, we show that relaxing accuracy requirements from convolutional neural networks used in a robot navigation system had no noticeable impact on the quality of the autonomous navigation. However, not all applications or deployment settings may have such error tolerance. In safety critical applications (self driving vehicles, autonomous

planes and drones, medical equipment etc.) where correctness is paramount, even minor degradation in accuracy may lead to unintended consequences. Hence the opportunity for relaxing accuracy constraints is highly dependent on the context of use and the willingness of the user to accept the worst-case result on an infrequent basis. For instance, for the agriculture monitoring robot we experimented with approximations, the worst case result is a robot crash - which may cause some minimal localized damage to crops and require manual human intervention. This is contrast to settings where human life is dependent on the guaranteed correctness of the application program.

CHAPTER 8: FUTURE WORK

I believe that the research directions proposed in this dissertation open up further opportunities for systems research in approximate computing. Below, I describe some possible future directions for extending the research presented in this dissertation.

8.1 SUPPORT FOR MORE APPLICATION DOMAINS

In our work, we extended HPVM IR with tensor operations used in the deep learning and image processing domains. The design choice of supporting high-level operations at the IR level enables mapping to efficient hardware level primitives, facilitates translation to high-level library operations, and allows for leveraging domain-specific analyses and optimizations.

Our implementation includes a small but representative set of operators. I believe there is much room for incorporating a wider range of tensor operations to support a wider range of machine learning, and image/video processing applications. Moreover, there is an opportunity to extend HPVM for other application domains including but not limited to graph processing, probabilistic computing, augmented and virtual reality (AR/VR). Incorporating support for these domains requires: a) identifying common high-level operators and algorithms used in these domains, b) developing approximate algorithms and transformations that provide useful accuracy and performance trade-offs, c) tailoring the approximation-tuning analyses to work with new kinds of operators, and d) identifying and developing metrics to use for tuning.

8.2 TRANSLATION TO APPROXIMATE HARDWARE ACCELERATORS

ApproxHPVM and ApproxTuner can map high-level IR operators to hardware knobs that control accuracy and performance trade-offs. To demonstrate this, we used the PROMISE analog compute accelerator that provides efficient tensor computations with some reductions in accuracy. As part of future work, we would like to support more such (production and research) hardware accelerators that include probabilistic and/or non-deterministic components.

In the presence of stochastic operators, we empirically evaluate a candidate configuration (mapping of approximation knobs to operators) to provide statistical guarantees (e.g., percentage of success over multiple runs) on the end-to-end computational accuracy being higher than the user-given threshold. I believe there is significant room for developing static

analyses that provide stronger guarantees on the accuracy bounds and is extensible to different kinds of hardware compute units and approximation choices. These analyses should consider: a) the random variation in stochastic hardware/software operators, b) the approximation knobs selected for each computation, and c) the error propagation properties of the target program, to compute bounds on the end-to-end program accuracy/quality.

8.3 AUTOMATIC GENERATION OF APPROXIMATE KERNELS

Currently, we use hand-written approximate kernels for convolution and reduction operations. These kernels are hand-optimized for the specific GPU (NVIDIA Jetson) and CPU architectures (ARM A57). Due to this, these kernels are not expected to exhibit performance speedups (and energy reductions) across different kinds of architectures. The choice of the target architecture impacts many optimization choices including tile sizes, SIMD processors for parallelization and vectorization, cache and scratchpad sizes, available registers among other such low-level hardware characteristics.

Systems such as TVM [67] include support for automatically generating low-level architecture specific code from high-level operators (e.g., matrix multiplication) and automatically tune the optimization parameters. I believe there is an opportunity for combining these automatic code generation capabilities in other similar systems with the approximation tuning capabilities in ApproxHPVM and ApproxTuner. To facilitate the use of approximations across multiple hardware architectures, it is important to relieve developers from the burden of hand-optimizing code for each architecture by automatically generating low-level optimized code for approximate kernels.

8.4 DOMAIN-SPECIFIC APPROXIMATION TECHNIQUES

In ApproxTuner and ApproxTuner we support software and hardware approximations for tensor operations, mainly convolution operations. To maintain the generality of our framework, we focused on supporting approximations that apply generally to tensor-based programs and avoid using domain-specific techniques. Some domain-specific techniques that have shown to provide significant performance benefits include: neural network pruning [45, 202, 97, 120, 203], neural network quantization [124, 67] for low-bit operators, low-rank factorization [44] for neural networks, sample skipping for image processing, frame skipping for video processing among others. We believe that combining generic tensor-based approximations such as input sampling and perforation (both supported in ApproxTuner)

with domain-specific optimizations can yield to higher performance improvements than either of these alone. Our preliminary experiments on applying perforation and sampling (using ApproxTuner) to pruned CNN models shows that there is potential for further improvements (Section 6.3).

8.5 EXTENDING APPROXIMATION-TUNING WITH MODEL RETRAINING SUPPORT.

Deep learning model optimizations such as DNN pruning use model retraining (fine-tuning the weights) to recover the accuracy loss due to pruning network connections [43, 45]. Similarly, for the deep learning benchmarks supported in our framework model retraining can be used to help recover the accuracy loss due to approximations. Retraining can enable higher approximation levels (i.e., more operations approximated) since fine-tuning model weights can create a program state that is more resilient to errors due to approximations.

Retraining is usually a time-consuming task (for any reasonably sized model architecture) and hence it is entirely infeasible to perform a retraining step for each candidate configuration navigated in the autotuning search space. There is room for developing techniques that selectively apply retraining to a few promising configurations, achieving a balance between the quality of the configurations discovered and the autotuning times.

REFERENCES

[1] R. St. 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 *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2678373.2665746 pp. 505–516. *[Cited on pages 1, 2, and 25.]*

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. *[Cited on pages 1 and 98.]*

[3] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017. *[Cited on page 1.]*

[4] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016. *[Cited on page 1.]*

[5] X. Zhou, R. Canady, S. Bao, and A. Gokhale, "Cost-effective hardware accelerator recommendation for edge computing," in *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020. *[Cited on page 1.]*

[6] H. Li, K. Ota, and M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, Jan 2018. *[Cited on pages 1 and 64.]*

[7] D. Azariadi, V. Tsoutsouras, S. Xydis, and D. Soudris, "ECG signal analysis and arrhythmia detection on IoT wearable medical devices," in *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, May 2016, pp. 1–4. *[Cited on pages 1 and 64.]*

[8] M. Mehrabani, S. Bangalore, and B. Stern, "Personalized speech recognition for internet of things," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec 2015, pp. 369–374. *[Cited on pages 1 and 64.]*

[9] L. Ran, Y. Zhang, Q. Zhang, and T. Yang, "Convolutional neural network-based robot navigation using uncalibrated spherical images," *Sensors*, vol. 17, no. 6, p. 1341, 2017. *[Cited on pages 1, 15, and 93.]*

[10] W. Chen, T. Qu, Y. Zhou, K. Weng, G. Wang, and G. Fu, "Door recognition and deep learning algorithm for visual based robot navigation," in *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*. IEEE, 2014, pp. 1793–1798. *[Cited on pages 1, 15, and 93.]*

[11] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556 *[Cited on pages 1, 51, and 79.]*

[12] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778. *[Cited on pages 1, 51, and 79.]*

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017. *[Cited on page 1.]*

[14] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. D. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, “Exploiting errors for efficiency: A survey from circuits to algorithms,” *CoRR*, vol. abs/1809.05859, 2018. [Online]. Available: http://arxiv.org/abs/1809.05859 *[Cited on pages 1 and 64.]*

[15] J. San Miguel, M. Badr, and N. E. Jerger, “Load value approximation,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 127–139. *[Cited on page 2.]*

[16] M. Sutherland, J. San Miguel, and N. E. Jerger, “Texture cache approximation on gpus,” in *Workshop on Approximate Computing Across the Stack*, 2015. *[Cited on page 2.]*

[17] Y. Fang, H. Li, and X. Li, “SoftPCM: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write,” in *2012 IEEE 21st Asian Test Symposium*. IEEE, 2012, pp. 131–136. *[Cited on page 2.]*

[18] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, “Scalable effort hardware design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 2004–2016, 2014. *[Cited on page 2.]*

[19] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, “Precimonious: Tuning assistant for floating-point precision,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12. *[Cited on pages 2 and 26.]*

[20] P. Düben, S. Yenugula, J. Augustine, K. Palem, J. Schlachter, C. Enz, T. Palmer et al., “Opportunities for energy efficient computing: A study of inexact general purpose processors for high-performance and big-data applications,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 764–769. *[Cited on page 2.]*

[21] M. Jung, D. M. Mathew, C. Weis, and N. Wehn, “Approximate computing with partially unreliable dynamic random access memory-approximate DRAM,” in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–4. *[Cited on page 2.]*

[22] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, “Flikker: saving DRAM refresh-power through critical data partitioning,” in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 213–224. *[Cited on page 2.]*

[23] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "PROMISE: An End-to-end Design of a Programmable Mixed-signal Accelerator for Machine-learning Algorithms," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00015 pp. 43–56. *[Cited on pages 2, 5, 12, 25, 27, 29, 36, 46, 47, 64, 65, 68, 71, and 80.]*

[24] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, "RRAM-based analog approximate computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, pp. 1905–1917, 2015. *[Cited on page 2.]*

[25] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 820–825. *[Cited on page 2.]*

[26] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *2011 24th Internatioal Conference on VLSI Design*. IEEE, 2011, pp. 346–351. *[Cited on page 2.]*

[27] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.48 pp. 449–460. *[Cited on pages 2 and 25.]*

[28] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Communications of the ACM*, vol. 58, no. 1, pp. 105–115, 2014. *[Cited on page 2.]*

[29] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing approximations to mapreduce frameworks," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 383–397. *[Cited on page 2.]*

[30] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 665–670. *[Cited on page 2.]*

[31] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," 2009. *[Cited on pages 2, 3, 4, 7, 8, and 21.]*

[32] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133 pp. 124–134. *[Cited on pages 2, 3, 4, 7, 19, 20, and 26.]*

[33] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based Approximation for Data Parallel Applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541948 pp. 35–50. *[Cited on pages 2, 3, 4, 7, 8, 15, 20, and 26.]*

[34] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, ser. RACES '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2414729.2414738 pp. 51–60. *[Cited on pages 2, 4, 7, and 26.]*

[35] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103710 pp. 441–454. *[Cited on pages 2, 3, 4, 7, 26, 71, and 78.]*

[36] 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 *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542481 pp. 38–49. *[Cited on pages 2, 3, 4, 6, 7, 8, 15, 19, 20, and 31.]*

[37] G. Keramidas, C. Kokkala, and I. Stamoulis, "Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders," in *Workshop on Approximate Computing (WAPCO'15)*, 2015. *[Cited on page 2.]*

[38] A. Rahimi, L. Benini, and R. K. Gupta, "Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 12, pp. 847–851, 2013. *[Cited on page 2.]*

[39] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions Embedded Computing Systems (TECS)*, vol. 12, pp. 88:1–88:26, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465790 *[Cited on pages 2, 26, and 44.]*

[40] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," ser. ASPLOS, 2011. *[Cited on pages 2, 3, 7, 8, 19, 21, and 26.]*

[41] H. Hoffmann, "JouleGuard: Energy Guarantees for Approximate Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 198–214. *[Cited on pages 2, 3, 7, 8, and 21.]*

[42] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, S. Misailovic, and S. Bagchi, "VideoChef: Efficient Approximation for Streaming Video Processing Pipelines," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/xu-ran pp. 43–56. *[Cited on pages 2, 15, and 52.]*

[43] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," 2015. *[Cited on pages 2, 3, 16, 19, 26, 94, 99, and 115.]*

[44] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2016, pp. 123–136. *[Cited on pages 2, 3, 7, 8, 15, 22, and 114.]*

[45] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage," in *Advances in neural information processing systems*, 1990, pp. 598–605. *[Cited on pages 2, 3, 26, 99, 114, and 115.]*

[46] M. Figurnov, A. Ibraimova, D. P. Vetrov, and P. Kohli, "PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions," in *Advances in Neural Information Processing Systems*, 2016, pp. 947–955. *[Cited on pages 2, 4, and 7.]*

[47] C. Sakr and N. Shanbhag, "An analytical method to determine minimum per-layer precision of deep neural networks," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 1090–1094. *[Cited on pages 2 and 27.]*

[48] A. Bulat and G. Tzimiropoulos, "XNOR-Net++: Improved binary neural networks," *arXiv preprint arXiv:1909.13863*, 2019. *[Cited on page 2.]*

[49] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540711 pp. 13–24. *[Cited on pages 2, 3, 4, 7, 8, 15, 19, and 21.]*

[50] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. 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, D. Killebrew, 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017. [Online]. Available: https://doi.org/10.1145/3079856.3080246 pp. 1–12. *[Cited on pages 2, 3, 5, 25, 29, 64, and 65.]*

[51] "NVDLA Deep Learning Inference Compiler is Now Open Source," https://developer.nvidia.com/blog/nvdla/, 2019. *[Cited on pages 2, 3, 5, 25, 29, and 65.]*

[52] "Intel Neural Compute Stick 2," https://www.intel.com/content/dam/support/us/en/documents/boardsandkits/neural-compute-sticks/NCS2_Product-Brief-English.pdf, 2019. *[Cited on pages 2, 3, 5, 25, 29, and 65.]*

[53] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in *2017 IEEE International symposium on circuits and systems (ISCAS)*. IEEE, 2017, pp. 1–4. *[Cited on page 2.]*

[54] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, "MASR: A Modular Accelerator for Sparse RNNs," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 1–14. *[Cited on page 2.]*

[55] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 629–634. *[Cited on page 2.]*

[56] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806808 pp. 25–34. *[Cited on pages 3, 4, 7, 20, and 26.]*

[57] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. ACM, 2012, pp. 51–60. *[Cited on page 3.]*

[58] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *2013 IEEE international conference on acoustics, speech and signal processing.* IEEE, 2013, pp. 6655–6659. *[Cited on page 3.]*

[59] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660231 pp. 309–328. *[Cited on pages 3, 4, 7, 8, 15, 18, and 19.]*

[60] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509546 pp. 33–52. *[Cited on pages 3, 4, 7, 8, 15, 18, and 19.]*

[61] 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 *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518 pp. 164–174. *[Cited on pages 3, 4, 7, 8, 15, and 18.]*

[62] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," in *U. Washington, Tech. Rep. UW-CSE- 15-01-01*, 2015. [Online]. Available: https://dada.cs.washington.edu/research/tr/2015/01/UW-CSE-15-01-01.pdf *[Cited on pages 3, 4, 7, 8, and 19.]*

[63] S. Li, S. Park, and S. A. Mahlke, "Sculptor: Flexible approximation with selective dynamic loop perforation," in *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018.* ACM, 2018. [Online]. Available: https://doi.org/10.1145/3205289.3205317 pp. 341–351. *[Cited on pages 3, 4, 7, 8, and 22.]*

[64] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, G. Maghanath, and S. Bagchi, "ApproxNet: Content and Contention Aware Video Analytics System for the Edge," *arXiv preprint arXiv:1909.02068*, 2019. *[Cited on pages 3, 4, 7, 8, 15, and 22.]*

[65] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *OOPSLA.* ACM, 2015, pp. 470–487. *[Cited on pages 4, 7, 8, 15, and 19.]*

[66] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes, “Image Perforation: Automatically accelerating image pipelines by intelligently skipping samples,” *ACM Transactions on Graphics (TOG)*, vol. 35, no. 5, pp. 1–14, 2016. *[Cited on pages 4 and 7.]*

[67] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “TVM: An Automated End-to-end Optimizing Compiler for Deep Learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18. Berkeley, CA, USA: USENIX Association, 2018. [Online]. Available: https://dl.acm.org/doi/10.5555/3291168.3291211 pp. 579–594. *[Cited on pages 4, 7, 8, 15, 19, 23, 39, and 114.]*

[68] C. Álvarez, J. Corbal, and M. Valero, “Fuzzy memoization for floating-point multimedia applications,” *IEEE Trans. Computers*, vol. 54, no. 7, pp. 922–927, 2005. [Online]. Available: https://doi.org/10.1109/TC.2005.119 *[Cited on pages 4 and 7.]*

[69] 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 *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2628071.2628110 p. 493–494. *[Cited on pages 4 and 7.]*

[70] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia et al., “Machine Learning at Facebook: Understanding Inference at the Edge,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344. *[Cited on pages 5, 29, and 65.]*

[71] NVIDIA, “PTX: Parallel thread execution ISA version 2.3,” *NVIDIA COMPUTE Programmer’s Manual*, vol. 3, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf *[Cited on pages 5 and 30.]*

[72] B. Sander, “HSAIL: Portable compiler IR for HSA,” in *Hot Chips Symposium 2013*, 2013, pp. 1–32. *[Cited on pages 5 and 30.]*

[73] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, “Language and Compiler Support for Auto-tuning Variable-accuracy Algorithms,” in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/2190025.2190056 pp. 85–96. *[Cited on pages 5, 20, 26, and 30.]*

[74] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A Survey on Compiler Autotuning using Machine Learning,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018. *[Cited on pages 6 and 31.]*

[75] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628092 pp. 303–316. *[Cited on pages 6, 12, 20, 31, 43, 60, 72, 81, and 85.]*

[76] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Proceedings of the 35th annual Design Automation conference*, 1998, pp. 732–737. *[Cited on pages 6 and 32.]*

[77] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely, "Green queue: Customized large-scale clock frequency scaling," in *2012 Second International Conference on Cloud and Green Computing*. IEEE, 2012, pp. 260–267. *[Cited on pages 6 and 32.]*

[78] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. VLSI Syst.*, vol. 8, no. 3, pp. 273–286, 2000. [Online]. Available: https://doi.org/10.1109/92.845894 *[Cited on page 7.]*

[79] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: Heterogeneous Parallel Virtual Machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178493 pp. 68–80. *[Cited on pages 9, 24, 35, 37, 38, and 67.]*

[80] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," San Jose, CA, USA, Mar 2004, pp. 75–88. *[Cited on pages 9, 35, and 37.]*

[81] D. Franklin, "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge," NVIDIA Developer Blog, 2018. [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge *[Cited on pages 11, 31, and 48.]*

[82] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes, "Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples," *ACM Trans. Graph.*, vol. 35, no. 5, pp. 153:1–153:14, 2016. [Online]. Available: https://doi.org/10.1145/2904903 *[Cited on page 15.]*

[83] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020. *[Cited on pages 15 and 93.]*

[84] D. Ball, P. Ross, A. English, P. Milani, D. Richards, A. Bate, B. Upcroft, G. Wyeth, and P. Corke, "Farm workers of the future: Vision-based robotics for broad-acre agriculture," *IEEE Robotics & Automation Magazine*, vol. 24, no. 3, pp. 97–107, 2017. *[Cited on pages 15 and 93.]*

[85] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford et al., "The limits and potentials of deep learning for robotics," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 405–420, 2018. *[Cited on pages 15 and 93.]*

[86] Y. Gu, Z. Li, Z. Zhang, J. Li, and L. Chen, "Path tracking control of field information-collecting robot based on improved convolutional neural network algorithm," *Sensors*, vol. 20, no. 3, p. 797, 2020. *[Cited on pages 15 and 93.]*

[87] S. G. Vougioukas, "Agricultural robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 2, pp. 365–392, 2019. *[Cited on pages 15 and 93.]*

[88] V. A. Higuti, A. E. Velasquez, D. V. Magalhaes, M. Becker, and G. Chowdhary, "Under canopy light detection and ranging-based autonomous navigation," *Journal of Field Robotics*, vol. 36, no. 3, pp. 547–567, 2019. *[Cited on pages 15 and 93.]*

[89] "A Growing Presence on the Farm: Robots," https://www.nytimes.com/2020/02/13/science/farm-agriculture-robots.html, 2020. *[Cited on pages 16 and 94.]*

[90] "Learned visual navigation for under-canopy agricultural robots," in *Under submission to Proc. of Robotics: Science and Systems (RSS)*, 2021. *[Cited on pages 16, 94, 95, 96, 97, and 100.]*

[91] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *arXiv preprint arXiv:1608.08710*, 2016. *[Cited on pages 16, 71, 94, and 99.]*

[92] A. Renda, J. Frankle, and M. Carbin, "Comparing Rewinding and Fine-tuning in Neural Network Pruning," in *International Conference on Learning Representations*, 2019. *[Cited on pages 16, 91, 94, and 99.]*

[93] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G.-Y. Wei, "Structured compression by weight encryption for unstructured pruning and quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1909–1918. *[Cited on pages 16 and 94.]*

[94] E. Malach, G. Yehudai, S. Shalev-Schwartz, and O. Shamir, "Proving the Lottery Ticket Hypothesis: Pruning is all you Need," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6682–6691. *[Cited on pages 16 and 94.]*

[95] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks," *arXiv preprint arXiv:1803.03635*, 2018. *[Cited on pages 16 and 94.]*

[96] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast Sparse ConvNets," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 14 629–14 638. *[Cited on pages 16 and 94.]*

[97] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143. *[Cited on pages 16, 90, 94, and 114.]*

[98] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018. *[Cited on pages 16 and 94.]*

[99] R. P. Foundation, "Buy a Raspberry Pi 4 Model B," https://www.raspberrypi.org/products/raspberry-pi-4-model-b/, 2019. *[Cited on pages 17 and 94.]*

[100] "Intel NUC Mini PC," https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html, 2020. *[Cited on pages 17 and 94.]*

[101] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. USA: IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.48 p. 449–460. *[Cited on pages 19 and 64.]*

[102] H. Sharif, P. Srivastava, M. Huzaifa, M. Kotsifakou, K. Joshi, Y. Sarita, N. Zhao, V. S. Adve, S. Misailovic, and S. V. Adve, "ApproxHPVM: A Portable Compiler IR for Accuracy-aware Optimizations," *PACMPL*, vol. 3, no. OOPSLA, pp. 186:1–186:30, 2019. [Online]. Available: https://doi.org/10.1145/3360612 *[Cited on pages 19 and 67.]*

[103] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737969 pp. 379–390. *[Cited on page 20.]*

[104] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620 pp. 198–209. *[Cited on pages 21 and 26.]*

[105] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "SiblingRivalry: Online Autotuning Through Local Competitions," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2380403.2380425 pp. 91–100. *[Cited on page 22.]*

[106] "Domain-specific compiler for linear algebra to optimize tensorflow computations," https://www.tensorflow.org/xla/, 2018. *[Cited on pages 23 and 39.]*

[107] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," *arXiv preprint arXiv:2002.11054*, 2020. *[Cited on pages 23 and 24.]*

[108] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. [Online]. Available: http://arxiv.org/abs/1805.00907 *[Cited on page 23.]*

[109] A. J. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Understanding portability of a high-level programming model on contemporary heterogeneous architectures," *IEEE Micro*, vol. 35, no. 4, 2015. *[Cited on page 23.]*

[110] 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 Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, pp. 1–25, 2014. *[Cited on page 24.]*

[111] "Coral," https://coral.ai/, 2020. *[Cited on page 25.]*

[112] S. Eldridge, F. Raudies, D. Zou, and A. Joshi, "Neural network-based accelerators for transcendental function approximation," in *Proceedings of the 24th edition of the great lakes symposium on VLSI.* ACM, 2014, pp. 169–174. *[Cited on page 25.]*

[113] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," ser. ICS, 2006. *[Cited on page 26.]*

[114] S. Chakradhar, A. Raghunathan, and J. Meng, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications," ser. IPDPS, 2009. *[Cited on page 26.]*

[115] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," ser. IPDPS, 2010. *[Cited on page 26.]*

[116] S. Misailovic and D. Roy and M. Rinard, "Probabilistically Accurate Program Transformations," ser. SAS, 2011. *[Cited on page 26.]*

[117] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS.* ACM, 2015, pp. 383–397. *[Cited on page 26.]*

[118] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," ser. PLDI, 2014. *[Cited on page 26.]*

[119] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix-up: Relaxing program semantics to unleash parallelization," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* IEEE Computer Society, 2015, pp. 235–245. *[Cited on page 26.]*

[120] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082. *[Cited on pages 26, 27, 90, and 114.]*

[121] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017. *[Cited on pages 26, 90, and 99.]*

[122] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 806–814. *[Cited on pages 26 and 90.]*

[123] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9. *[Cited on pages 26 and 90.]*

[124] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858. *[Cited on pages 27 and 114.]*

[125] "NVIDIA Tesla V100 GPU Architecture," http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. *[Cited on page 29.]*

[126] "NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. *[Cited on page 29.]*

[127] "The NVIDIA GeForce GTX 1080 GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation," https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/5, 2016. *[Cited on page 29.]*

[128] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf pp. 265–283. *[Cited on page 29.]*

[129] F. Chollet et al., "Keras," https://github.com/fchollet/keras, 2015. *[Cited on page 29.]*

[130] "PyTorch," https://pytorch.org/, 2016. *[Cited on page 29.]*

[131] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274 *[Cited on page 29.]*

[132] "OpenMP," https://www.openmp.org/. *[Cited on pages 29 and 30.]*

[133] "CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit. *[Cited on page 29.]*

[134] "OpenACC," https://www.openacc.org/. *[Cited on page 29.]*

[135] "Intel oneAPI Threading Building Blocks ," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html. *[Cited on page 29.]*

[136] "Data Parallel C++," https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/data-parallel-c-dpc.html. *[Cited on page 29.]*

[137] NVIDIA, "NVIDIA Jetson TX2 Developer Kit," 2018. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2 *[Cited on pages 36, 47, 48, and 80.]*

[138] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing, 2017. *[Cited on page 36.]*

[139] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in nvidia gpus," *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2017. *[Cited on page 46.]*

[140] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ *[Cited on page 46.]*

[141] NVIDIA, "Jetson TX2 Power Monitor with I2C," https://devtalk.nvidia.com/default/topic/1000830/jetson-tx2/jetson-tx2-ina226-power-monitor-with-i2c-interface, 2018. *[Cited on page 49.]*

[142] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750374 pp. 707–719. *[Cited on page 49.]*

[143] D. A. Jamshidi, M. Samadi, and S. Mahlke, "D2MA: Accelerating Coarse-grained Data Transfer for GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628072 pp. 431–442. *[Cited on page 49.]*

[144] S. K. Gonugondla, M. Kang, and N. R. Shanbhag, "A variation-tolerant in-memory machine learning classifier via on-chip training," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 11, pp. 3163–3173, Nov 2018. *[Cited on page 49.]*

[145] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist, 1998. *[Cited on pages 51 and 79.]*

[146] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012. *[Cited on pages 51 and 79.]*

[147] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Handwritten Digit Recognition with a Back-propagation Network," in *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, ser. NIPS '89. Cambridge, MA, USA: MIT Press, 1989. [Online]. Available: https://dl.acm.org/doi/10.5555/2969830.2969879 pp. 396–404. *[Cited on page 51.]*

[148] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS '12. USA: Curran Associates Inc., 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257 pp. 1097–1105. *[Cited on pages 51 and 79.]*

[149] W. Yang, "Classification on CIFAR-10/100 and ImageNet with PyTorch," https://github.com/bearpaw/pytorch-classification/blob/master/models/cifar/alexnet.py, 2019. *[Cited on page 51.]*

[150] Y. Geifman, "VGG16 models for CIFAR-10 and CIFAR-100 using Keras," https://github.com/geifmany/cifar-vgg, 2019. *[Cited on page 51.]*

[151] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861 *[Cited on pages 51 and 79.]*

[152] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," in *2004 Conference on Computer Vision and Pattern Recognition Workshop*, June 2004, pp. 178–178. *[Cited on page 51.]*

[153] N. Thomos, N. V. Boulgouris, and M. G. Strintzis, "Optimized Transmission of JPEG2000 Streams Over Wireless Channels," *IEEE Transactions on Image Processing*, vol. 15, no. 1, January 2006. *[Cited on page 52.]*

[154] X. Li and J. Cai, "Robust transmission of JPEG2000 encoded images over packet loss channels," in *Proceedings of the 2007 IEEE International Conference on Multimedia and Expo, ICME 2007, July 2-5, 2007, Beijing, China*, 2007, pp. 947–950. *[Cited on page 52.]*

[155] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759 *[Cited on page 62.]*

[156] J. Wang, J. Zhang, W. Bao, X. Zhu, B. Cao, and P. S. Yu, "Not just privacy: Improving performance of private deep learning in mobile cloud," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2407–2416. *[Cited on page 64.]*

[157] P. Daniels and K. Iwago, "The suitability of cloud-based speech recognition engines for language learning." *JALT CALL Journal*, vol. 13, no. 3, pp. 229–239, 2017. *[Cited on page 64.]*

[158] G. S. Brar, "Malleable contextual partitioning and computational dreaming," Ph.D. dissertation, Virginia Tech, 2015. *[Cited on page 64.]*

[159] J. H. Ahnn, "A practical approach to scalable big data computing for the personalization of services at samsung," in *2014 IEEE/ACM International Symposium on Big Data Computing*. IEEE, 2014, pp. 64–73. *[Cited on page 64.]*

[160] Mark Harris, NVIDIA, "Mixed-Precision Programming with CUDA 8," https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/, 2016. *[Cited on page 64.]*

[161] P. Konsor, "Performance benefits of half precision floats," https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats, 2011, accessed: 2019-11-21. *[Cited on page 64.]*

[162] ARM, "Half-precision floating-point number format," https://developer.arm.com/docs/dui0774/e/other-compiler-specific-features/half-precision-floating-point-number-format, 2019. *[Cited on page 64.]*

[163] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, 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," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014. *[Cited on page 64.]*

[164] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 62:1–62:26, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2836168 *[Cited on page 64.]*

[165] J. S. Miguel, M. Badr, and N. E. Jerger, “Load Value Approximation,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 127–139. *[Cited on page 64.]*

[166] R. Smith, G. Smith, and A. Wardani, “Software reuse in robotics: Enabling portability in the face of diversity,” in *IEEE Conference on Robotics, Automation and Mechatronics, 2004.*, vol. 2. IEEE, 2004, pp. 933–938. *[Cited on page 65.]*

[167] J. D. Knowles, L. Thiele, and E. Zitzler, “A tutorial on the performance assessment of stochastic multiobjective optimizers,” *TIK-Report*, vol. 214, 2006. *[Cited on page 69.]*

[168] M. Figurnov, D. P. Vetrov, and P. Kohli, “PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions,” *CoRR*, vol. abs/1504.08362, 2015. [Online]. Available: http://arxiv.org/abs/1504.08362 *[Cited on page 71.]*

[169] W. Yuan, S. Adve, D. Jones, and R. Kravets, “Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems,” 01 2003. *[Cited on page 78.]*

[170] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. *[Cited on page 79.]*

[171] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. *[Cited on page 79.]*

[172] M. Haddad, S. Cheng, L. Thao, and J. Santos, “Autonomous Navigation Powered by Jetson TX2 and Robot Operating System.” *[Cited on page 80.]*

[173] A. Milioto, P. Lottes, and C. Stachniss, “Real-time semantic segmentation of crop and weed for precision agriculture robots leveraging background knowledge in cnns,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 2229–2235. *[Cited on page 80.]*

[174] C. Wiltz, “Magic Leap One Teardown: A Leap Forward for AR/VR?” 2018. [Online]. Available: https://www.designnews.com/design-hardware-software/magic-leap-one-teardown-leap-forward-arvr/204060129459400 *[Cited on page 80.]*

[175] NVIDIA Developer Forums , “Power Monitoring on Jetson TX2. (2018)),” https://forums.developer.nvidia.com/t/jetson-tx2-ina226-power-monitor-with-i2c-interface/48754, 2018. *[Cited on page 80.]*

[176] R. Krishnamoorthi, “Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018. *[Cited on page 90.]*

[177] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-NET: ImageNet Classification using Binary Convolutional Neural Networks,” in *European conference on computer vision*. Springer, 2016, pp. 525–542. *[Cited on page 90.]*

[178] W. McAllister, D. Osipychev, A. Davis, and G. Chowdhary, "Agbots: Weeding a field with a team of autonomous robots," *Computers and Electronics in Agriculture*, vol. 163, p. 104827, 2019. *[Cited on page 95.]*

[179] R. R Shamshiri, C. Weltzien, I. A. Hameed, I. J Yule, T. E Grift, S. K. Balasundram, L. Pitonakova, D. Ahmad, and G. Chowdhary, "Research and development in agricultural robotics: A perspective of digital farming," 2018. *[Cited on page 95.]*

[180] N. K. Uppalapati, B. Walt, A. Havens, A. Mahdian, G. Chowdhary, and G. Krishnan, "A berry picking robot with a hybrid soft-rigid arm: Design and task space control," *Proceedings of Robotics: Science and Systems, Corvalis, Oregon, USA*, 2020. *[Cited on pages 95 and 109.]*

[181] N. Mott, "Intel's Frost Canyon NUC 10 Makes Its Retail Debut," 2020. [Online]. Available: https://www.tomshardware.com/news/intels-frost-canyon-nuc-10-makes-its-retail-debut *[Cited on page 97.]*

[182] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 0.5MB model size," 2016. *[Cited on page 97.]*

[183] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia, "An overview of Internet of Things (IoT) and data analytics in agriculture: Benefits and challenges," *IEEE Internet of Things Journal*, vol. 5, no. 5, pp. 3758–3773, 2018. *[Cited on page 98.]*

[184] C. Samplawski, J. Huang, D. Ganesan, and B. M. Marlin, "Towards Objection Detection Under IoT Resource Constraints: Combining Partitioning, Slicing and Compression," in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, 2020, pp. 14–20. *[Cited on page 98.]*

[185] Z. Zhang, E. Kayacan, B. Thompson, and G. Chowdhary, "High precision control and deep learning-based corn stand counting algorithms for agricultural robot," *Autonomous Robots*, vol. 44, no. 7, pp. 1289–1302, 2020. *[Cited on page 98.]*

[186] W. Luo, X. Zhao, and T. Kim, "Multiple object tracking: A review," *CoRR*, vol. abs/1409.7618, 2014. [Online]. Available: http://arxiv.org/abs/1409.7618 *[Cited on page 98.]*

[187] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173191 p. 751–766. *[Cited on page 98.]*

[188] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2 *[Cited on page 98.]*

[189] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2019. *[Cited on page 98.]*

[190] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," *2016 IEEE International Conference on Image Processing (ICIP)*, Sep 2016. [Online]. Available: http://dx.doi.org/10.1109/ICIP.2016.7533003 *[Cited on page 98.]*

[191] P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler, and L. Leal-Taixé, "MOT20: A Benchmark for Multi Object Tracking in Crowded Scenes," *arXiv:2003.09003[cs]*, Mar. 2020, arXiv: 2003.09003. [Online]. Available: http://arxiv.org/abs/1906.04567 *[Cited on page 98.]*

[192] M. Figurnov, A. Ibraimova, D. Vetrov, and P. Kohli, "PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions," *arXiv preprint arXiv:1504.08362*, 2015. *[Cited on page 99.]*

[193] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. S. Doermann, "Towards optimal structured CNN pruning via generative adversarial learning," *CoRR*, vol. abs/1903.09291, 2019. [Online]. Available: http://arxiv.org/abs/1903.09291 *[Cited on page 99.]*

[194] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312 *[Cited on page 100.]*

[195] NVIDIA, "NVIDAI Jetson Nano Developer Kit," https://developer.nvidia.com/embedded/jetson-nano-developer-kit, 2020. *[Cited on page 101.]*

[196] O. Robotics, "rosbag – ROS Wiki," 2020. [Online]. Available: http://wiki.ros.org/rosbag *[Cited on page 101.]*

[197] J. Bai, F. Lu, K. Zhang et al., "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019. *[Cited on page 101.]*

[198] J. Bai, F. Lu, K. Zhang et al., "ONNX: Open Neural Network Exchange," https://www.onnxruntime.ai/about.html, 2020. *[Cited on page 101.]*

[199] NVIDIA, "TensorRT," https://developer.nvidia.com/tensorrt, 2020. *[Cited on page 101.]*

[200] X.-N. Bui, B. Jean-Daniel, P. Soueres, and J.-P. Laumond, "Shortest path synthesis for dubins non-holonomic robot," in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 1994, pp. 2–7 vol.1. *[Cited on page 108.]*

[201] A. A. Suzen, B. Duman, and B. Sen, “Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN,” in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–5. *[Cited on page 109.]*

[202] S. Anwar, K. Hwang, and W. Sung, “Structured pruning of deep convolutional neural networks,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2017. *[Cited on page 114.]*

[203] H. Zhou, J. M. Alvarez, and F. Porikli, “Less is More: Towards Compact CNNs,” in *European Conference on Computer Vision*. Springer, 2016, pp. 662–677. *[Cited on page 114.]*