ApproxTuner: A Compiler and Runtime System for Adaptive Approximations

Hashim Sharif
University of Illinois at Urbana-Champaign, USA
hsharif3@illinois.edu

Yifan Zhao
University of Illinois at Urbana-Champaign, USA
yifanz16@illinois.edu

Maria Kotsifakou
Runtime Verification, Inc., USA
maria.kotsifakou@runtimeverification.com

Akash Kothari
University of Illinois at Urbana-Champaign, USA
akashk4@illinois.edu

Ben Schreiber
University of Illinois at Urbana-Champaign, USA
bjschre2@illinois.edu

Elizabeth Wang
University of Illinois at Urbana-Champaign, USA
eyw3@illinois.edu

Yasmin Sarita
Cornell University, USA
ycs4@cornell.edu

Nathan Zhao
University of Illinois at Urbana-Champaign, USA
nz11@illinois.edu

Keyur Joshi
University of Illinois at Urbana-Champaign, USA
kpjoshi2@illinois.edu

Vikram S. Adve
University of Illinois at Urbana-Champaign, USA
vadve@illinois.edu

Sasa Misailovic
University of Illinois at Urbana-Champaign, USA
misailo@illinois.edu

Sarita Adve
University of Illinois at Urbana-Champaign, USA
sadve@illinois.edu

Abstract

Manually optimizing the tradeoffs between accuracy, performance and energy for resource-intensive applications with flexible accuracy or precision requirements is extremely difficult. We present **ApproxTuner**, an automatic framework for accuracy-aware optimization of tensor-based applications while requiring only high-level end-to-end quality specifications. ApproxTuner implements and manages approximations in algorithms, system software, and hardware.

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 of 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*, which significantly speeds up autotuning by analytically predicting the accuracy impacts of approximations.

We evaluate ApproxTuner across 10 convolutional neural networks (CNNs) and a combined CNN and image processing benchmark. For the evaluated CNNs, using only hardware-independent approximation choices we achieve a mean speedup of 2.1x (max 2.7x) on a GPU, and 1.3x mean speedup (max 1.9x) on the CPU, while staying within 1 percentage point of inference accuracy loss. For two different accuracy-prediction models, ApproxTuner speeds up tuning by 12.8x and 20.4x compared to conventional empirical tuning while achieving comparable benefits.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Approximate Computing, Compilers, Heterogeneous Systems, Deep Neural Networks

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3446108

1 Introduction

With the ubiquitous deployment of highly compute-intensive machine-learning and big data processing workloads, optimizing these workloads is becoming increasingly important. A wide range of applications using these workloads are being deployed on both the cloud and the edge, including image processing, object classification, speech recognition, and face recognition [8, 43, 46]. Many of these workloads are very computationally demanding which makes it challenging to achieve the desired performance levels on resource-constrained systems.

Many modern machine learning and image-processing applications are inherently "*approximate*" in the sense that the input data are often collected from noisy sensors (e.g., image and audio streams) and output results are usually probabilistic (e.g., for object classification or facial recognition). Such applications can *trade off small amounts of result quality (or accuracy) for improved performance and efficiency* [60], where result quality is an application-specific property such as inference accuracy in a neural network or peak signal-to-noise ratio (PSNR) in an image-processing pipeline. 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 [3, 7, 14, 34, 45]. Recent specialized accelerators incorporate hardware-specific approximations that provide significant improvements in performance and energy in exchange for relaxed accuracy [1, 2, 11, 29, 30, 48, 59]. Such techniques can provide a crucial strategy to achieve the necessary performance and/or energy for emerging applications in edge computing.

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 to automatically select, configure, and tune the parameters for combinations of approximation techniques*, while meeting *end-to-end requirements on energy, performance, and accuracy.*

To address this broad challenge, we need to solve several specific challenges. First, the variety of software and hardware approximation choices, and the tuning knobs for each of them, induce a large search space for accuracy-aware optimization – up to 7^{91} configurations in our benchmarks. Second, efficiently searching such large spaces is made even more difficult because individual "candidate configurations" (sample points in the search space) can be expensive to measure on edge systems for estimating accuracy, performance and energy. For example, measurement-based (aka, "empirical") tuning for the ResNet50 neural network took 11 days on a server-class machine. Third, the diversity of hardware compute units often used in edge-computing devices [63] yields diverse hardware-specific approximation options with varying accuracy-performance tradeoffs. Moreover, optimizations in specialized hardware accelerators often create important opportunities for orders-of-magnitude improvements. These hardware-specific tuning opportunities are critical, but difficult to exploit without sacrificing software portability, which is crucial in some important domains, like mobile devices. Fourth, the best accuracy-performance-energy tradeoffs may vary significantly at run time, depending on system load, battery level, or time-varying application requirements.

1.1 ApproxTuner System

We present **ApproxTuner**, an automatic framework for accuracy-aware optimization of tensor-based applications (an important subset of edge computing applications). It finds application configurations that maximize speedup and/or energy savings of an application, subject to end-to-end quality specifications. It addresses all of the challenges above, *and is the first and only system to handle all of them*, as we discuss in Section 9.

Tensor computations are widely used in machine learning frameworks, and many important domains such as image processing, scientific computing, and others [3, 31–33]. ApproxTuner focuses on tensor-based computations for two reasons. First, limiting the system to these computations enables *algorithmic* approximations specific to tensor operations (in addition to generic software and hardware approximations). Adding support for other classes of computations can be done in a similar fashion, but is outside the scope of this work. Second, many current and emerging hardware accelerators focus on tensor computations [1, 2, 29, 30, 48], enabling novel hardware-specific approximations for these computations. ApproxTuner includes support for a novel experimental, analog-domain accelerator [59] that exemplifies such approximation opportunities and associated challenges.

ApproxTuner tackles the last two challenges above — and enables hardware-specific, yet portable, tuning and run-time adaptation – by decomposing the optimization process into three stages: development-time, install-time and run-time. At development time, ApproxTuner selects hardware-independent approximations and creates a *tradeoff curve* that contains the approximations with highest quality and performance ApproxTuner found during search. At install time, the system refines this curve using hardware-specific optimizations and performance measurements. The final tradeoff curve is included with the program binary. The program can use the refined curve for the best static choices of approximations before the run, and it can further adapt these choices using the same curves based on run-time conditions. Using the final tradeoff curve at run-time as well keeps the overheads of run-time adaptation negligible.

ApproxTuner tackles the first two challenges – and enables efficient navigation of the large tradeoff space and efficient estimation of performance and quality by introducing novel *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. ApproxTuner also facilitates distributed approximation-tuning since the error profile collection can happen at multiple client devices, while a centralized server can perform time-intensive autotuning. It makes install-time tuning (with hardware-specific approximations) feasible which would otherwise be prohibitively expensive on a single resource-constrained edge device.

1.2 Contributions

In summary, our contributions are:

- A system that combines a wide range of existing *hardware, software and algorithmic approximations*, supports diverse heterogeneous systems, and provides an easy-to-use programming interface for accuracy-aware tuning. Our results show that different kinds of approximations and approximation knobs are best 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) construction of tradeoff curves for hardware-independent approximations at *development-time*, 2) mapping to hardware-specific approximations at *install-time*, and 3) a fast approximation selection at *runtime*.
- A novel *predictive approximation-tuning* technique that uses compositional models for accuracy prediction to speed up both development-time and install-time tuning. For two different accuracy-prediction models, our predictive tuning strategy speeds up tuning by 12.8x and 20.4x compared to conventional empirical tuning while achieving comparable benefits.
- Experimental evaluation for 11 benchmarks – 10 CNNs and 1 combined CNN + image processing benchmark – shows:
  Hardware-independent Approximations. When using only hardware-independent approximations, ApproxTuner achieves geomean speedup of 2.1x and energy reduction of 2x on GPU, geomean speedup of 1.3x and energy reduction of 1.3x on CPU, with 1 percentage point accuracy loss.
  Hardware Approximations. At install time, ApproxTuner maps tensor operations to an energy-efficient analog compute accelerator, PROMISE, and provides a geomean energy reduction of 3.25x (across benchmarks) with 1 percentage point drop in inference accuracy. ApproxTuner exploits such hardware-specific approximations without sacrificing object-code portability.
  Runtime Adaptation for Approximations. To counteract performance slowdowns imposed by runtime conditions such as low-power modes, ApproxTuner can dynamically tune approximation knobs with extremely low run-time overheads, by using tradeoff curves shipped with the application binary.

2 ApproxTuner Overview

Figure 1 shows the high-level workflow for ApproxTuner. ApproxTuner builds on the HPVM and ApproxHPVM compiler

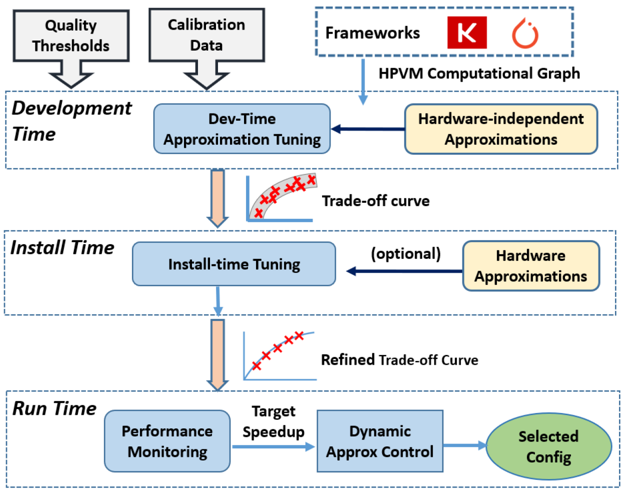

Fig. 1. ApproxTuner workflow.

systems [35, 57], 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 [35] (for other tensor-based programs), and compiles them to the ApproxHPVM internal representation (IR) [57]. ApproxHPVM manages the compilation to various compute units (Section 2.1). ApproxTuner optimizes the computations in this IR using three phases: 1) development-time, 2) install-time, and 3) run-time (Section 2.2). ApproxTuner's goal is to select combinations of software and hardware approximations (detailed in Section 2.3) that meet performance, energy, and accuracy targets.

2.1 Preliminaries and Terminology

HPVM and ApproxHPVM. HPVM IR is a dataflow graph-based parallel program representation that captures coarse- and fine-grain data and task parallelism [35]. HPVM provides a portable IR and a retargetable compiler framework that can target diverse compute units, e.g., CPUs, GPUs and FPGAs.

ApproxHPVM extends HPVM with support for tensor operations and limited support for accuracy-aware tuning (see Section 9) [57]. The tensor operations represent data-parallel computations on tensors, commonly used in CNNs and image processing applications. ApproxHPVM also adds back-ends to the HPVM system for a few custom machine learning accelerators/ libraries, including one for cuDNN on NVIDIA GPUs and one for a programmable analog in-memory compute accelerator called PROMISE [59].

This work focuses on approximations for a set of predefined tensor operations included in ApproxHPVM, such as convolutions, matrix multiplication, ReLU, map, and reduce. We refer to Sharif et al. [57, Table 1] for the definition of these operations in ApproxHPVM. 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 outputs (i.e., the semantics) of a program is fixed, regardless of the hardware used to execute the operation. Examples include filter sampling, perforated convolutions, and reduced precision, described below (Section 2.3). Some approximations, like the use of (IEEE) FP16 instead of FP32 may have hardware-independent semantics, and yet may be implemented in hardware for efficiency (in fact, they may be too inefficient to use without hardware support, possibly making them non-portable). Other approximations are called *hardware-specific*; the output quality impact of these approximations is specific to the choice of hardware used. The impact on energy and performance will usually be hardware-dependent for all kinds of approximations.

Quality of Service. A quality-of-service (QoS) metric is a (usually domain-specific) metric over the outputs of some computation, such as inference accuracy or PSNR. The QoS metric function tor a tensor-based program, $QoS : T_{\text{out}} \times T_{\text{gold}} \rightarrow \mathbb{R}$ takes the output tensor T_{out} and the exact output T_{gold}, and produces a scalar value.

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

Knobs, Configurations, and Tradeoff Curves. An *approximation knob* is a discrete-valued parameter of an approximation method (represented using integers in ApproxTuner) that can be modified to control the quality, energy, and running time. A zero value denotes no approximation.

Each tensor operation may be assigned an approximation choice or none. A *configuration* is a map $Config : op \rightarrow \text{Int}$ that assigns an approximation knob value to every tensor operation in the program. The *search space* is the set of all possible configurations.

A *tradeoff point* is a triple, $(QoS, Perf, config)$, which specifies the quality-of-service and the performance of the configuration on specified inputs. 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 [18]: a point $s_1 = (QoS_1, Perf_1, config_1)$ is dominated by a 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 have unequal QoS or $Perf$.

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*. A Pareto set of a set S is its subset that consists of non-dominated points:

$$PS\,(S) = \{\, s \mid s \in S \;\wedge\; \forall s' \in S \,.\, s \not\prec s' \,\} \qquad (1)$$

This set defines a *tradeoff curve*, which contains linear segments between the points in the Pareto set. Intuitively, these points have the best tradeoffs that the search algorithm was able to find. We describe how to select the configurations or combinations of configurations from a tradeoff curve in Section 5. We also define a relaxed version of the curve, PS_ε, which includes tradeoff points that are within ε distance from points in the Pareto set:

$$PS_\varepsilon\,(S) = \{\, s \mid s \in S \;\wedge\; \exists s^* \in PS(S) \,.\, dist(s, s^*) \leq \varepsilon \,\} \qquad (2)$$

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

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 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 optimize a hardware-agnostic performance metric (e.g., operation count) and produce QoS values within the constraint. In practice, the QoS and performance calculated at development time can be imprecise because they use hardware-agnostic estimates of performance (e.g., operation count) and optionally of accuracy (with predictive tuning). To make it more likely we will capture points with good measured QoS and performance, we create the relaxed tradeoff curve, $PS_\varepsilon(S_0)$, which is shipped with the application.

The install-time stage (Section 4) uses the development-time tradeoff curves and measures the actual performance of each configuration in those tradeoff curves on the target hardware. Next, we create a refined tradeoff 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 available.

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 and generates a new tradeoff curve.

The run-time stage (Section 5) takes the program's final tradeoff curve from the install-time phase and uses it 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.

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.

Filter sampling for convolutions. Li et al. [42] 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. We vary 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. This approximation has 9 knob settings in all: i) three possible sampling rates: 50% (skip 1 out of 2), 33% (skip 1 out of 3), and 25% (skip 1 out of 4) and ii) offset values of $0 \dots k-1$ for a skip rate of 1-out-of-k.

Perforated convolutions. Figurnov et al. [17] 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 (again, with varying initial offsets), then interpolates the missing output elements. It has 18 knob settings in all: 1) skipping rows or columns, 2) *skip rate* (1-out-of-k): 50%, 33%, and 25%, and 3) k choices of initial offset.

Reduction sampling. Zhu et al. [67] 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. [59]. Our system 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. [59] show that PROMISE consumes 3.4-5.5x less energy and has 1.4-3.4x higher throughput compared even to fully-custom non-programmable digital accelerators.

IEEE FP16. Our implementation has FP16 versions of all tensor operations, including FP16 variants for each of the knobs supported by filter sampling, perforated convolutions, and reduction sampling. FP16 can be used or not for an operation; there is no additional knob value, though it can combine with other approximation knobs. For instance, both FP16 and 50% filter sampling can be applied to a convolution operation. Although FP16 requires hardware support, its semantics (impact on QoS) are hardware-independent and hence can be accounted for in the development stage.

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, there are 63 = (9 * 2 + 18 * 2 + 2 + 7) knobs for each convolution operation, 8 = 3 * 2 + 2 knobs for each reduction, and 2 choices for other tensor operations.

3 Development-time Tuning

At development time, we tune the application for hardware-independent approximations. We propose *predictive approximation tuning*, which decomposes tuning into a profile collection phase and an autotuning phase. In profile collection phase, one operation at a time is approximated and its impact on QoS and performance is measured. The autotuning phase uses these profiles and compositional models to predict the end-to-end QoS and performance impact of multiple approximations applied simultaneously, and uses these 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.

3.1 Overview of Predictive Tuning

Algorithm 1 describes our predictive approximation tuning in the function `PredictiveTuner`. It consists of five steps:

- **Profile collection**: Lines 12-15 in Algorithm 1 collect a QoS profile for the given application *per operation*, *per knob setting* (§ 3.2).
- **QoS predictor refinement**: Lines 18-20 refine a parameter α of the *QoS prediction model* (§ 3.3) used in the following autotuning, so that the predictor fits better to the program to be tuned.
- **Autotuning**: Lines 23-30 invoke an off-the-shelf autotuner to heuristically explore the configuration space. The QoS prediction model (§ 3.3) and *performance prediction model* (§ 3.4) direct this search towards better configurations. We use the OpenTuner tool [5] which supports multiple algorithms for exploring large configuration spaces. It uses the estimates of QoS and performance from our models to calculate the fitness of the current configuration and select the next one.
- **Tradeoff curve construction**: Line 33 selects autotuning configurations that are in or close to the Pareto set (§ 3.5).
- **QoS validation**: Lines 36-40 empirically measure the QoS of configurations in the previous step, and select configurations with measured QoS greater than threshold.

Our presentation focuses on time savings. Our autotuning can be directly adapted for tuning other goals such as energy savings by providing a corresponding prediction model.

Algorithm 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 $\alpha$
7      • nIters: number of autotuning iterations
8      • $\varepsilon_1$, $\varepsilon_2$: maximum distances of a configuration to the Pareto set
9  Output: A tradeoff curve for P
10 Function PredictiveTuner(P,C,K,$QoS_{min}$,nCalibrate,nIters,$\varepsilon_1$,$\varepsilon_2$)
11     // Step 1: Collect QoS Profile
12     map qosProfiles;
13     foreach (op, knob) ∈ K do
14         ($\Delta Q$, $\Delta T$) = gatherQoSProfile(P, C, op, knob);
15         qosProfiles[(op, knob)] = ($\Delta Q$, $\Delta T$);
16
17     // Step 2: Initialize and tune predictor to find coefficient $\alpha$
18     autotuner = AutoTuner (P, K, nIters, $QoS_{min}$);
19     predictor = Predictor (qosProfiles);
20     $\alpha$ = 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, $\alpha$);
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 $\varepsilon$ distance of the tradeoff curve
33     paretoConfigs = $PS_{\varepsilon_1}$ (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 $PS_{\varepsilon_2}$ (filteredConfigs);
```

3.2 Gathering QoS Profiles

QoS profiles are gathered for each unique pair of *tensor operation* and *approximation knobs*. Algorithm 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 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.

3.3 Models for QoS Prediction

We propose and evaluate two error composition models Π_1 and Π_2, for a program P and configuration $config \in Config$:

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

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

where $knob = config(op)$, α is the coefficient to be refined, and

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

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

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 *config*, 2) adding this sum to the baseline raw tensor output (T_{base}), and then 3) computing the QoS relative to the exact output (T_{gold}). 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. Model Π_1 works well with classification accuracy (as QoS metric) in scenarios where relative probabilities of predicted class(es) remain mostly unchanged despite changes in the absolute output values.

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. The scope of these models is discussed in Section 8.

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.

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 predicted execution cost of a configuration is the sum of the cost for each operator with the knob the tuner selected. This is computed as:

$$Cost_{\text{Total}}(config) = \sum_{(op, knob) \in config} Cost(op,\ knob).$$

We assume the compute and memory 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 operator *op* with approximation knob *knob* by:

$$Cost(op, knob) = \frac{N_m(op)}{R_m(knob)} + \frac{N_c(op)}{R_c(knob)}, \qquad (3)$$

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. For example, for FP16 50% *filter sampling*, $R_m = 4$ since the operator 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 compute and memory 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 compute and memory operations is likely faster than one that reduces fewer such operations. Therefore, this performance predictor *ranks* configurations correctly by their speedup, which suffices for autotuning purposes.

3.5 Constructing Tradeoff Curve

Autotuning (Lines 23-30 of Algorithm 1) often discovers many `candidateConfigs`. On Line 33, points in the Pareto set or with distance to the Pareto set less than ε_1 are kept (by Equation 2). This step serves the purpose of reducing the overhead of QoS validation (Line 36- 40), by filtering away configurations that are not in or close to the Pareto set. ε_2 controls the configuration selection after QoS validation (Line 41). Using ε_1 to filter configurations reduces the number of configurations that are empirically evaluated to measure the real QoS. However, if many configurations are filtered (Lines 36- 40), this can potentially result in a very small set of valid configurations. Setting ε_2 to be higher (or equal) to ε_1 adds flexibility for including more points in the shipped tradeoff curve. Thus ε_1 and ε_2 give the developer the ability to control the quality and the size of tradeoff curve and the time of our three-stage tuning.

Since FP16 availability is not guaranteed on each hardware platform, we allow users to tune the program with and without FP16 support, creating two separate curves - one each for FP32 and FP16. Users can simply ship a single curve if FP16 hardware availability is known ahead of time.

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} that participate in the distributed tuning process, and the other parameters from the development-time tuning. This step refines the shipped tradeoff curves with real performance (or other properties, such as energy usage) measurements and creates a new tradeoff curve. When hardware-specific approximations exist on the target platform, distributed predictive tuning is invoked to further optimize the program by exploiting those approximations, as described below.

Software-only knobs. In this case, all steps are done on the edge-device. It runs the configurations from the input tradeoff curve PS_ε on the inputs from C, and measures *both* the real QoS and performance (development-time stage collected only QoS, but lacks access to real edge hardware to measure real performance), and filters the configurations that do not satisfy the thresholds. Finally, for the resulting filtered set S^* it constructs the final tradeoff curve $PS(S^*)$, which has only the points with best measured QoS-performance tradeoffs.

Hardware-specific knobs. We distribute predictive tuning across the server and edge-devices in three main steps, in order to reduce the profiling and validation burden per device:

- 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 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 1. Because the approximation choices cannot be decoupled (as we found in initial prototype experiments), we cannot simply reuse the curve from development-time, but instead perform a fresh autotuning step that combines software and hardware approximations and constructs a new tradeoff curve.
- 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 1).
- The server receives the filtered configurations from each of the participating edge devices and computes the final tradeoff curve, $PS(S_1^* \cup S_2^* \cup ... \cup S_n^*)$, where S_i^* are Pareto sets returned by edge device i; these are configurations with measured QoS higher than the user-specified QoS threshold. This is the final curve that the server sends back to the devices.

5 Runtime Approximation Tuning

A key capability of ApproxTuner is the ability to adapt approximation settings at run-time 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., [66] and can be incorporated in our approach.

ApproxTuner allows users to specify a desired target for performance, and then uses the tradeoff curve *PS* (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 tradeoff space.

A key property of *all* our approximation techniques is that the different approximation knob settings are simply numerical parameters to the tensor operations (e.g., perforation and sampling rates). Therefore, the runtime tuner can switch between configurations *with negligible overhead*, simply by using different parameter values each time.

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 N most recent batch executions ($k-N, \dots, k-2, k-1$). 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 tradeoff curve that provides the speedup necessary to achieve the target performance level.

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

1. **Enforce Required Speedup in each Invocation.** Picks a configuration with performance no smaller than $Perf_T$. This is a $O(\log(|PS|)$ operation, using binary search.
2. **Achieve Average Target Performance over Time.** Probabilistically selects between two configurations, with performance the closest below and above $Perf_T$ over a time interval (on average). The selection probabilities, p_1 and p_2, are such that $p_1 \cdot Perf_1 + p_2 \cdot Perf_2 = Perf_T$, as in [67]. For instance, if $Perf_T = 1.3$x and the closest points provide 1.2x and 1.5x speedup, these two configurations are randomly selected with respective probabilities $^2/_3$ and $^1/_3$.

Policy 1 is less flexible and is better suited for hard or soft real-time systems, where deadlines are important. Policy 2 is a better choice when application throughput is a goal.

6 Evaluation Methodology

Benchmarks. We use several CNNs (Table 1) and an image processing benchmark that combines a CNN (AlexNet2) with the Canny [10] edge detection pipeline.

Datasets. We use MNIST [40], CIFAR-10 [36] and the ImageNet dataset ILSVRC 2012 [53], each with 10K images. For ImageNet, we use 10K randomly sampled images (from 200 randomly selected classes) from its test set of 50K images. We divide the 10K images into calibration set (for autotuning) and test set (for evaluation), with 5K images each.

Table 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 [37]	CIFAR-10	6	79.16%	5e+8
AlexNet [37]	ImageNet	8	55.86%	5e+8
AlexNet2	CIFAR-10	7	85.09%	2e+10
ResNet-18 [24]	CIFAR-10	22	89.44%	3e+22
ResNet-50 [24]	ImageNet	54	74.16%	7e+91
VGG-16 [58]	CIFAR-10	15	89.41%	3e+22
VGG-16 [58]	ImageNet	15	72.88%	3e+22
MobileNet [28]	CIFAR-10	28	83.69%	1e+26
LeNet [39]	MNIST	4	98.70%	3e+3

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:

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

($PSNR_y$ 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.

6.2 Implementation

Our tensor library, targeted by our compiler back ends, 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 and sampling. Instead, we developed a 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.

6.3 Hardware Setup

Client Device Setup. Our client device (Table 2) is the NVIDIA Jetson Tegra TX2 board [49], commonly used in robotics and small autonomous vehicles [47, 62].

We model an SoC that adds to the TX2 a simulated PROMISE accelerator for machine learning [59]. 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 [50] 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 a validated timing and energy model [59].

Table 2. System parameters for the Edge Device. NVIDIA Tegra TX2 board including simulated 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

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.

6.4 Autotuning Setup

For autotuning search, we use the OpenTuner [5] 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 does not improve over 1K consecutive iterations. The number of 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 1); after autotuning, we select and ship configurations within ε_2 distance to the Pareto set (Line 41). These distance thresholds, ε_1 and ε_2, are computed per benchmark to limit the maximum number of configurations validated and shipped. We chose ε_1 and ε_2 so that at most 50 configurations are retained.

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. Remaining 99 distributed invocations are emulated 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 is to maintain the same level of performance (execution time) offered at the highest frequency mode (1.3Ghz). The frequency is reduced after a batch of inputs has been processed and before the next batch starts, and applied instantaneously.

For reliable measurements, we run 200 batches of 500 images each: we divide the 5K test set into 10 batches and cycle through them 20 times. We average the processing time and accuracy across batches. The experiments use Control strategy 2 from Section 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 execution time, and ApproxTuner picks a new configuration for batch $n + 2$. The configuration switching overhead is negligible (Section 5).

7 Evaluation

We experimentally evaluate the benefits of ApproxTuner. We analyze the three stages in Sections 7.1-7.3 (development-time), Section 7.4 (install-time), and Section 7.5 (runtime). We characterize tuned approximations in Section 7.2. We evaluate predictive tuning in Section 7.3. We demonstrate tuning a composite CNN + image processing benchmark with multiple QoS metrics in Section 7.6.

7.1 Performance and Energy Improvements

Improvements for GPU. Figures 2a and 2b show the performance and energy benefits achieved on the Tegra's GPU, for accuracy reductions of 1%, 2% and 3%. The X-axes list the benchmarks and the Y-axes show improvements over the FP32 baseline. These results only use hardware-independent approximations: FP16, perforation, sampling. The results are reported after trying both predictors, Π_1 and Π_2, and choosing the best result (we compare the predictors in Section 7.3).

For $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, and $\Delta QoS_{3\%}$, the geomean speedups are 2.14x, 2.23x, and 2.28x. 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 geomean speedup, on top of FP16.

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

Energy reductions (Fig. 2b) are positively correlated with speedups, as expected. For $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, and $\Delta QoS_{3\%}$, the mean energy reductions are 1.99x, 2.06x and 2.11x.

Improvements for CPU. The mean speedups for the CPU for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$ and $\Delta QoS_{3\%}$ are 1.31x, 1.38x and 1.42x (figure omitted for space). The maximum speedup is 1.89x for VGG16-CIFAR10. The energy benefits are quite similar. 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 less amenable to sampling or perforation.

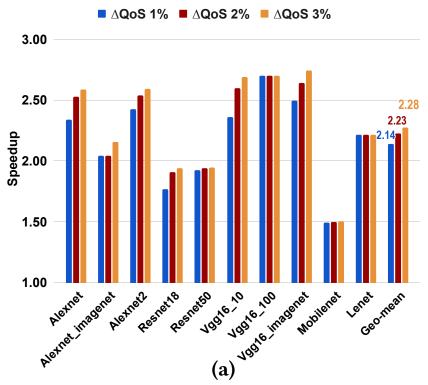
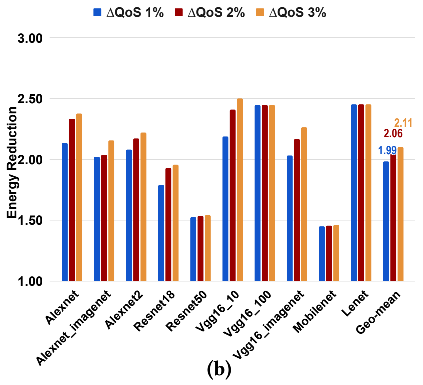

Fig. 2. (a) Speedups and (b) Energy reductions achieved on GPU using hardware-independent approximations for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, $\Delta QoS_{3\%}$.

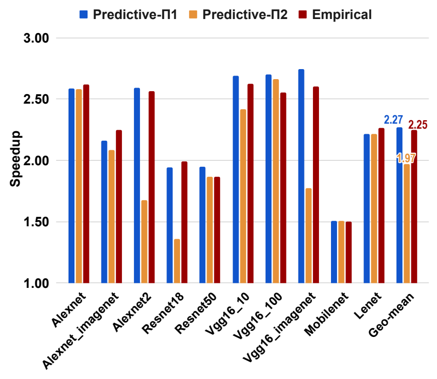

Fig. 3. Comparing predictive vs. empirical tuning for $\Delta QoS_{3\%}$.

7.2 Characteristics of Approximated Benchmarks

Table 3 shows the best performing GPU configurations (number of operations mapped to approximation knobs) for $\Delta QoS_{3\%}$.

Table 3. 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	FP16:42 perf-50%:2 perf-33%:3 perf-25%:6

General Trends. We find that the first few layers in the CNNs are relatively less amenable to approximations. For 4 benchmarks, the first layer is not mapped to perforation or sampling (only FP16). For ResNet-18 and MobileNet, the first 3 layers are not mapped to perforation or sampling.

We summarize interesting insights for several representative CNNs. The other CNNs show similar behaviors (e.g., AlexNet2-CIFAR10 behaves similarly to AlexNet-CIFAR10).
AlexNet-CIFAR10: None of the layers in AlexNet map to the perforated convolutions approximation, while all layers (in most configurations) are amenable to filter sampling.
ResNet18: Across all configurations in the Pareto set, 7 of the 21 convolution layers are not mapped to any approximation. Interestingly, 4 layers map only to 33% perforation and all use different start offsets, showing the importance of combining varying start offsets.
VGG16-100: We find that 3 layers in VGG16-100 can only be mapped to column-based perforation while row-based perforation leads to low accuracy.

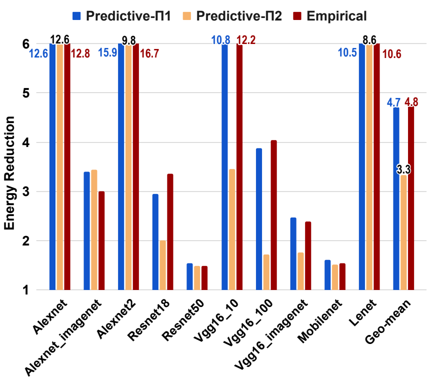

Fig. 4. Energy reductions on GPU + PROMISE with install-time distributed predictive tuning (Π_1, Π_2) and empirical tuning for $\Delta QoS_{3\%}$.

MobileNet: For the best configuration, only 8 layers out of 28 could be mapped to approximations without significant accuracy loss, which is why MobileNet has the lowest performance improvement (1.50× speedup).
ResNet50-ImageNet: At most 11 convolutions (from 53) are mapped to any approximation other than FP16, with 6 convolution operators mapped to 25% perforation. As a result, ResNet50-ImageNet achieves much of its speedup from FP16, although the overall speedup of 1.95× is quite significant for just 1% accuracy reduction.

Overall, these insights show the importance of combining different approximations and the importance of tuning the choices of combinations to balance accuracy vs. performance gains.

7.3 Predictive vs Empirical Tuning

Comparing Speedups. We compare our predictive approximation tuning with empirical tuning, both using OpenTuner [5]. Figure 3 illustrates results for the maximum bound of 3%. It shows that predictors Π_1 and Π_2 provide geomean speedups of 2.27x and 1.97x, compared with 2.25x for empirical tuning. For most benchmarks, Π_1 effectiveness is similar to empirical tuning, but Π_2 provides lower speedups because it systematically underestimates accuracy loss for

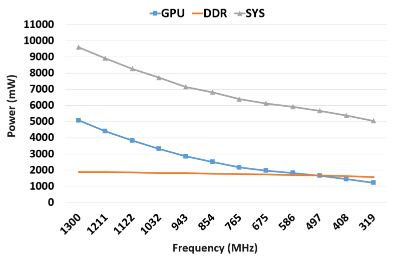

Fig. 5. GPU, DDR, and overall system power at different GPU frequency levels for ResNet18. As GPU frequency reduces, GPU and overall power consumption reduce.

Table 4. Predictive Tuning times compared to Empirical (in minutes). "Π_1-red" and "Π_2-red" are speedups compared to Empirical.

Benchmark	Empirical	Pred-Π_1	Pred-Π_2	Π_1-red	Π_2-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	30.5	13.68x	12.25x
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	11112.8	716.4	246.3	15.51x	45.12x
Geomean				**12.76x**	**20.37x**

some benchmarks, and therefore chooses configurations that are later removed during accuracy validation.

Autotuning Times. Table 4 shows the autotuning time for predictive tuning compared to empirical tuning using OpenTuner. Predictive-Π_1 and Predictive-Π_2 are on average 12.76x and 20.37x 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 more precise but requires more memory.

Size of Tradeoff Curves. For our benchmarks, autotuning (before configuration selection) on average generates 4360 configurations. As ApproxTuner keeps at most 50 configurations (§ 6.4), the size of the tradeoff curve is reduced by 87x.

7.4 Install-time tuning

We evaluate the efficacy of our install-time tuning strategy in exploiting the low voltage knobs in the PROMISE accelerator that trade accuracy for energy savings (Section 2.3).

Energy Savings. Figure 4 shows the energy reductions achieved with install-time predictive distributed tuning compared to empirical tuning for $\Delta QoS_{3\%}$. 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. With the exception of ResNet50, all benchmarks have some tensor operations mapped to PROMISE. For LeNet, AlexNet, AlexNet2, and VGG16-CIFAR10 more than 50% of convolution operations can be mapped to PROMISE (for $\Delta QoS_{3\%}$). On average, predictors Π_1 and Π_2 provide 4.7x and 3.3x energy reductions, compared to 4.8x reduction for empirical tuning. Π_2 is lower than Π_1 due to higher prediction error, which leads the search to explore less effective configurations. These results show that the predictive install-time tuning in ApproxTuner can exploit hardware-specific approximations by intelligently selecting operations to offload to accelerators, and achieves significant energy improvements.

Tuning Times. We separately measure the times on the edge device for error profile collection and the autotuning time on the server. A single distributed error profile collection phase on the edge device ranges from 1 minute (LeNet) to 6 hours (ResNet50), with a geometric mean of 25 minutes across all benchmarks. To put these results into context, with a back-of-the-envelope calculation, we estimate that empirical tuning for ResNet50 on a single Tegra Tx2 would take more than 4 months. The autotuning time on the server ranges from 24 minutes (LeNet) to 10.22 hours (ResNet-50), with a geometric mean of 1.9 hours across benchmarks.

7.5 Runtime Approximation Tuning

We vary GPU frequency to mimic lower power modes. As Figure 5 shows, both the GPU power and total system (SYS) power decrease substantially with decreasing frequency, by ~$\dot{7}$x for the GPU and ~$\dot{1}$.9x for the system when frequency decreases from 1300MHz to 318MHz. DDR power decreases only slightly because DDR frequency is kept constant. These results are averaged over 10 runs of ResNet18-CIFAR10.

We show the effectiveness of run-time adaptation for three evaluated CNNs in Figure 6 (other CNNs exhibit similar behavior). The X-axis presents the different frequencies, which we vary over time. The left Y-axis shows the *normalized execution time* (averaged over 200 batches), relative to time taken at the highest frequency level (1.3Ghz). The right Y-axis presents the model accuracy (in %).

As we reduce the frequency, the baseline configuration slows down substantially (solid blue line), while accuracy remains unaffected. ApproxTuner's dynamic tuning counteracts the slowdown and maintains the original average batch processing time through most of the frequency range (dashed orange lines), while gracefully degrading the inference accuracy (yellow lines, right Y-axis). To counteract higher slowdowns, relatively higher QoS degradation has to be imposed. For instance, for ResNet18, a potential slowdown of 1.45x in the baseline case (at 675MHz) can be counteracted with an accuracy drop of 0.33 percentage points, but as much as 1.75x (at 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). 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 are not shown for AlexNet and AlexNet2).

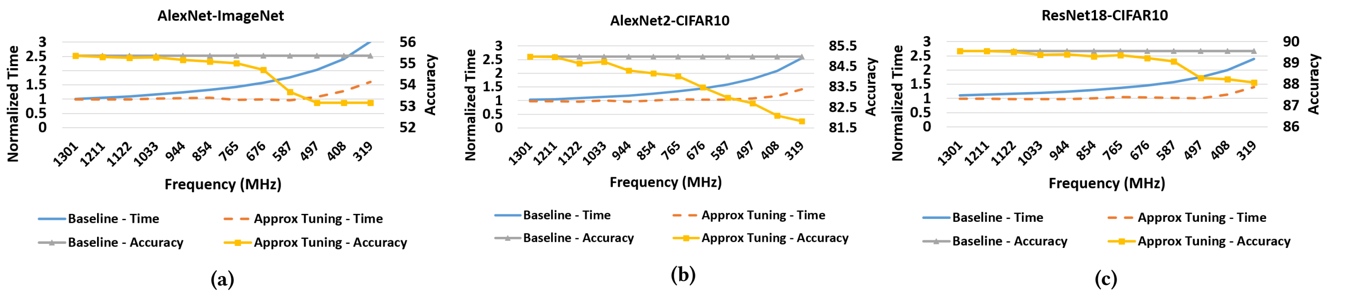

Fig. 6. Figures a), b) and c) show that runtime approximation tuning trades off accuracy to maintain the same level of responsiveness when frequency levels are reduced. The times on the y-axis are normalized with respect to performance achievable at the highest frequency (1.3Ghz). Without dynamic approximations (the blue line), applications slow down.

These results demonstrate that ApproxTuner's dynamic approximation-tuning capabilities can help maintain a target performance in the face of lower power/frequency modes.

7.6 Combining CNNs and Image Processing

In our final experiment, we demonstrate that ApproxTuner can optimize components from multiple tensor-based domains, each with its own QoS metric, using a benchmark that combines CNN and Image processing. The application consists of a CNN (AlexNet2 on CIFAR-10) that classifies images to one of 10 classes, and images from five of the classes are forwarded for Canny edge detection [10].

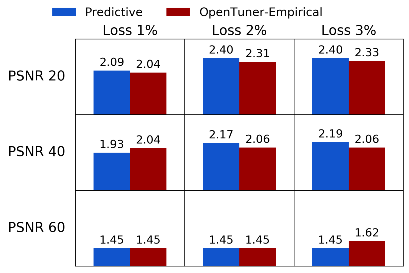

Fig. 7. Speedups achieved on GPU for a grid of accuracy (horizontal) and PSNR (vertical) thresholds.

For this benchmark, the QoS we consider is a pair *(PSNR, accuracy)*, using PSNR for image quality of edge detection and accuracy for the CNN classifier. Figure 7 presents the best configurations achieved on the GPU with each QoS pair. Nine QoS pairs are evaluated, with three different values each of PSNR and accuracy. The bar labels show speedups compared to the FP32 baseline. As either threshold (PSNR or accuracy) is relaxed, speedup increases, since the tuner finds more opportunities for approximation.

Predictive tuning again enables dramatically faster tuning (graphs omitted) while preserving optimization gains. For all *(PSNR, accuracy)* pairs, predictive and empirical tuning find configurations with comparable speedups, while predictive tuning is 20.1× faster on average. 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). For this benchmark, we only apply model Π_2 since the size of the output tensor is not fixed (CNN classification results affect output tensor size), and model Π_1 requires each approximation knob error profile to be equal-sized tensors (Section 8).

8 Discussion and Future Work

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. Our models are better at estimating QoS when these conditions are satisfied: 1) The control flow is deterministic and input-independent. 2) The operators have no side effects. 3) For predictor Π_1, the shapes of raw tensor outputs must match, since these outputs are summed up in the model. As part of future work, we hope to incorporate models for other kinds of computations, beyond tensor operations. There is also potential for evaluating and extending these models to work with input-dependent control-flow and operations with side-effects.

Interaction with Model Compression Techniques. We believe that the approximation-tuning capabilities in ApproxTuner open avenues for new research opportunities. We perform preliminary experiments to investigate the potential for incorporating compression techniques such as pruning [22, 41, 52] as part of our tuning framework. We create pruned models (using the approach in [52]) for MobileNet, VGG16, and ResNet18 on CIFAR-10. We find that, starting with the pruned models, applying perforated convolutions using empirical tuning in ApproxTuner reduces MAC (multiply-accumulate) operations by 1.3x for MobileNet and VGG-16, and by 1.2x for ResNet18, while reducing inference accuracy by less than 1% point compared with the pruned model. The key takeaway is that approximation techniques can be applied to pruned models with little loss of accuracy, and potentially yielding valuable additional speedups. In future work, we aim to incorporate model compression techniques, including pruning, into ApproxTuner and conduct a systematic evaluation of the accuracy-performance tradeoffs.

9 Related Work

Table 5 compares ApproxTuner to select related systems across four broad kinds of capabilities: a) support for approximations, b) programmability, c) tuning strategies, and d)

Table 5. Comparing capabilities of ApproxTuner versus most closely related state-of-the-art systems.

	Approximations				**Programmability**			**Tuning Strategies**			**Model Approxs**	
	Algorithmic Approxs	*Accelerator -specific Approxs*	*Multi Domain Approx*	*Precision Tuning*	*No Code Changes*	*Retarget*	*Portable Object Code*	*Joint Dev+Install Time Tuning*	*Runtime Approx Tuning*	*Predictive Tuning*	*Model Approxs*	*Support for Re-training*
ApproxTuner	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
ApproxHPVM [57]	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
TVM [12], AutoTVM [13]	✗	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓
ACCEPT [56]	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
PetaBricks [4]	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗

support for DNN model optimizations (pruning, quantization) and model retraining.

Compilers for Accuracy Tuning: ApproxHPVM [57] is a compiler we developed for accuracy-aware mapping of applications to heterogeneous systems. ApproxHPVM only supports hardware approximations (FP16 and PROMISE) and uses standard empirical autotuning at development time. It has no install-time or run-time tuning. Its reliance on L-norms is insufficient for algorithmic approximations with heavy-tailed error distributions, as we discovered for both perforated convolutions and feature sampling.

TVM [12] is a deep-learning compiler that generates optimized tensor-operator code for CPUs, GPUs, and accelerators. It supports reduced precision (FP16 and INT8) and DNN binarization [19] (reducing operator precision to 1 or 2 bits), but applies each of them for the entire network and does not tune individual operations with different choices or knobs. It does not support approximation tuning or arbitrary software/hardware approximations and does not dynamically adjust approximation knobs.

ACCEPT [56] uses programmer-specified type declarations to approximate programs on CPUs or FPGAs. Unlike ApproxTuner, it requires extensive source annotations, does not support tuning strategies for combinations of approximations, and does not support run-time adaptation.

Petabricks uses heuristic search to select among multiple user-provided versions of an algorithm with varying accuracy [4, 6, 16]. It requires developers to rewrite programs in the Petabricks programming language to provide alternate algorithm implementations, does not offer built-in, generic approximation options such as reduced precision, and does not support run-time approximation choices.

Dynamic Approximation Tuning: Many systems have supported approximation changes at run-time on CPU [9, 26, 27] and GPU [23, 54, 55]. These systems 1) do not target heterogeneous systems beyond GPUs, and 2) do not support efficient tuning for combining multiple kinds of approximations.

Model Optimizations for DNNs. *Deep Compression* [21] uses pruning, quantization, and compression to reduce model size (as much as 49x for VGG-16). However, pruning introduces sparsity in the computation which limits performance gains, and can even lead to slowdowns on GPUs [44, 61, 65]. To make sparse tensor computation efficient, researchers proposed software and architectural techniques [20, 25, 38, 44, 51, 64]. ApproxTuner does not reduce model size, but optimizes various tensor operator approximations, yielding significant speedups. Our preliminary study (Section 8) shows that there is potential for combining perforation and sampling with pruned models to have both model size and performance improvements. Our work is also complementary to systems that automatically generate implementations for low-precision quantized tensor computations [15, 19].

10 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 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 run-time 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.

Acknowledgements

This work is supported in part by DARPA through the Domain-Specific System on Chip (DSSoC) program, the National Science Foundation under Grants CCF 17-03637, CCF 18-46354, and CCF 19-56374, a Google Faculty Research award, a grant from the Amazon AWS Machine Learning Research Awards Program, and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] 2020. Coral. https://coral.ai/.

[2] 2020. Qualcomm Redefines Premium with the Flagship Snapdragon 888 5G Mobile Platform. https://www.qualcomm.com/news/releases/2020/12/02/qualcomm-redefines-premium-flagship-snapdragon-888-5g-mobile-platform.

[3] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[4] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. *PetaBricks: a language and compiler for algorithmic choice*. Vol. 44. ACM. https://doi.org/10.1145/1543135.1542481

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316. https://doi.org/10.1145/2628071.2628092

[6] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 85–96. https://doi.org/10.1109/CGO.2011.5764677

[7] ARM. 2019. Half-precision floating-point number format. https://developer.arm.com/docs/dui0774/e/other-compiler-specific-features/half-precision-floating-point-number-format. Accessed: 2019-11-21.

[8] D. Azariadi, V. Tsoutsouras, S. Xydis, and D. Soudris. 2016. ECG signal analysis and arrhythmia detection on IoT wearable medical devices. In *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 1–4. https://doi.org/10.1109/MOCAST.2016.7495143

[9] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. *SIGPLAN Not.* 45, 6 (June 2010), 198–209. https://doi.org/10.1145/1809028.1806620

[10] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.

[11] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Olivier Temam, Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Olivier Temam, Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGPLAN Notices* 49, 4 (2014), 269–284. https://doi.org/10.1145/2644865.2541967

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594. https://www.usenix.org/system/files/osdi18-chen.pdf

[13] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*. 3389–3400.

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[15] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 305–316.

[16] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 379–390. https://doi.org/10.1145/2737924.2737969

[17] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. 2016. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions. In *Advances in Neural Information Processing Systems*. 947–955.

[18] Carlos M Fonseca, Joshua D Knowles, Lothar Thiele, and Eckart Zitzler. 2005. A tutorial on the performance assessment of stochastic multiobjective optimizers. In *Third International Conference on Evolutionary Multi-Criterion Optimization (EMO 2005)*, Vol. 216. 240.

[19] Josh Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. 2020. RIPTIDE: Fast End-to-end Binarized Neural Networks. *Proceedings of Machine Learning and Systems (MLSys)* (2020). http://ubicomplab.cs.washington.edu/pdfs/riptide.pdf

[20] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. *arXiv preprint arXiv:2006.10901* (2020).

[21] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv:cs.CV/1510.00149

[22] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[23] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. 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, 123–136. https://doi.org/10.1145/2906388.2906396

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

[25] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333. https://doi.org/10.1145/3352460.3358275

[26] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. 2009. Using code perforation to improve performance, reduce energy consumption, and respond to failures. (2009).

[27] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic Knobs for Responsive Power-Aware Computing *(ASPLOS)*. https://doi.org/10.1145/1961295.1950390

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

[29] Intel. 2018. Intel Movidius Vision Processing Units (VPUs). https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/myriad-x-product-brief.pdf.

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi

Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[31] Boris N Khoromskij. 2012. Tensors-structured numerical methods in scientific computing: Survey on recent advances. *Chemometrics and Intelligent Laboratory Systems* 110, 1 (2012), 1–19.

[32] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. https://doi.org/10.1145/3133901

[33] Joseph C Kolecki. 2002. An introduction to tensors for students of physics and engineering. (2002).

[34] Patrick Konsor. 2011. Performance Benefits of Half Precision Floats. https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats. Accessed: 2019-11-21.

[35] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’18)*. ACM, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[36] Alex Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (05 2012).

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*. 1097–1105. https://doi.org/10.1145/3065386

[38] Vadim Lebedev and Victor Lempitsky. 2016. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2554–2564.

[39] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based Learning Applied to Document Recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[40] Yann LeCun, Corinna Cortes, and Christopher JC Burges. 1998. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist.

[41] Yann LeCun, John S Denker, and Sara A Solla. 1990. Optimal brain damage. In *Advances in neural information processing systems*. 598–605.

[42] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning Filters for Efficient Convnets. *arXiv preprint arXiv:1608.08710* (2016).

[43] H. Li, K. Ota, and M. Dong. 2018. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network* 32, 1 (Jan 2018), 96–101. https://doi.org/10.1109/MNET.2018.1700202

[44] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 806–814.

[45] Mark Harris, NVIDIA. 2016. Mixed-Precision Programming with CUDA 8. https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/.

[46] M. Mehrabani, S. Bangalore, and B. Stern. 2015. Personalized speech recognition for Internet of Things. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 369–374. https://doi.org/10.1109/WF-IoT.2015.7389082

[47] Andres Milioto, Philipp Lottes, and Cyrill Stachniss. 2018. 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, 2229–2235.

[48] NVIDIA. 2018. NVDLA. http://nvdla.org/.

[49] NVIDIA. 2018. NVIDIA Jetson TX2 Developer Kit. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules.

[50] NVIDIA Developer Forums . 2018. Power Monitoring on Jetson TX2. (2018)). https://forums.developer.nvidia.com/t/jetson-tx2-ina226-power-monitor-with-i2c-interface/48754.

[51] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736. https://doi.org/10.1109/HPCA.2018.00067

[52] Alex Renda, Jonathan Frankle, and Michael Carbin. 2019. Comparing Rewinding and Fine-tuning in Neural Network Pruning. In *International Conference on Learning Representations*.

[53] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[54] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications *(ASPLOS)*. https://doi.org/10.1145/2541940.2541948

[55] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 13–24. https://doi.org/10.1145/2540708.2540711

[56] Adrian Sampson, Andre Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. In *U. Washington, Tech. Rep. UW-CSE- 15-01-01*.

[57] Hashim Sharif, Prakalp Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S. Adve, Sasa Misailovic, and Sarita V. Adve. 2019. ApproxHPVM: a portable compiler IR for accuracy-aware optimizations. *PACMPL* 3, OOPSLA (2019), 186:1–186:30. https://doi.org/10.1145/3360612

[58] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[59] Prakalp Srivastava, Mingu Kang, Sujan K Gonugondla, Sungmin Lim, Jungwook Choi, Vikram Adve, Nam Sung Kim, and Naresh Shanbhag. 2018. PROMISE: An End-to-End Design of a Programmable Mixed-Signal Accelerator for Machine-Learning Algorithms. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. https://doi.org/10.1109/ISCA.2018.00015

[60] Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie D. Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. 2018. Exploiting Errors for Efficiency: A Survey from Circuits to Algorithms. *CoRR* abs/1809.05859 (2018). arXiv:1809.05859 http://arxiv.org/abs/1809.05859

[61] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*. 2074–2082.

[62] Chris Wiltz. 2018. Magic Leap One Teardown: A Leap Forward for AR/VR? (2018). https://www.designnews.com/design-hardware-software/magic-leap-one-teardown-leap-forward-arvr/204060129459400

[63] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. 2019. Machine learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344. https://doi.org/10.1109/HPCA.2019.00048

[64] Leonid Yavits, Amir Morad, and Ran Ginosar. 2014. Sparse matrix multiplication on an associative processor. *IEEE Transactions on Parallel and Distributed Systems* 26, 11 (2014), 3175–3183.

[65] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. 2017. Scalpel: Customizing DNN pruning to the Underlying Hardware Parallelism. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 548–560. https://doi.org/10.1145/3140659.3080215

[66] Wanghong Yuan, Klara Nahrstedt, Sarita Adve, Douglas L Jones, and Robin H Kravets. 2003. Design and evaluation of a cross-layer adaptation framework for mobile multimedia systems. In *Multimedia Computing and Networking 2003*, Vol. 5019. International Society for Optics and Photonics, 1–13. https://doi.org/10.1117/12.484069

[67] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. 2012. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 441–454. https://doi.org/10.1145/2103621.2103710