ApproxHPVM: A Portable Compiler IR for Accuracy-Aware Optimizations

HASHIM SHARIF, University of Illinois at Urbana-Champaign, USA
PRAKALP SRIVASTAVA, University of Illinois at Urbana-Champaign, USA
MUHAMMAD HUZAIFA, University of Illinois at Urbana-Champaign, USA
MARIA KOTSIFAKOU, University of Illinois at Urbana-Champaign, USA
KEYUR JOSHI, University of Illinois at Urbana-Champaign, USA
YASMIN SARITA, Cornell University, USA
NATHAN ZHAO, University of Illinois at Urbana-Champaign, USA
VIKRAM S. ADVE, University of Illinois at Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois at Urbana-Champaign, USA
SARITA ADVE, University of Illinois at Urbana-Champaign, USA

We propose ApproxHPVM, a compiler IR and system designed to enable accuracy-aware performance and energy tuning on heterogeneous systems with multiple compute units and approximation methods. ApproxH-PVM 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 and enables future capabilities like accuracy-aware dynamic scheduling and design space exploration.

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 our framework on nine benchmarks from the deep learning domain and five 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.

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

Additional Key Words and Phrases: Compiler, Virtual ISA, Approximate Computing, Heterogeneous Systems, Deep Neural Networks

1 INTRODUCTION

With the slowdown of Moore's Law and the end of Denard scaling, the gap between hardware performance and the ever-increasing requirements of modern applications continues to widen [St. Amant et al. 2014]. Recent paradigms such as approximate computing attempt to bridge the gap by introducing novel hardware architectures and software optimizations that trade-off accuracy for gains in performance and energy [Stanley-Marbell et al. 2018]. Approximate computing is particularly relevant for application domains that can tolerate small errors with acceptable loss in the final output, such as signal processing, speech recognition, sensor networks, information retrieval, data mining, video decoding, game engines, and machine learning.

Approximate computing techniques can be realized in many architectural components: floating-point units, caches, DRAM, and analog and digital accelerators [Esmaeilzadeh et al. 2012; Srivastava et al. 2018; St. Amant et al. 2014]. Software techniques are similarly diverse, such as loop perforation [Sidiroglou-Douskos et al. 2011], barrier elision [Misailovic et al. 2012], reduction sampling, and function substitution [Zhu et al. 2012]. A given computational algorithm or kernel may benefit from multiple different approximation techniques, and moreover, a realistic application will contain several (or many) distinct kernels. Determining how best to map such an application to a modern heterogeneous system while achieving the best overall tradeoff between end-to-end application-level accuracy and performance or energy is an open research problem. 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).

In addition, software portability is a critical requirement for modern applications, not just at the source-code level but also the ability to *ship* software that can execute efficiently on a wide range of systems. Modern 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) and even multiple hardware generations (e.g., across x86 processors). GPUs, for example, provide virtual instructions sets, e.g., PTX [NVIDIA 2010] or HSAIL [Sander 2013], to enable software to be shipped as "virtual object code" that is translated to particular hardware instances only on the end-user's system. This is a major challenge for approximate computing approaches because hardware-specific accuracy-performance-energy choices can make orders-of-magnitude difference in the performance and energy benefits achieved in exchange for relaxing accuracy. 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 [Ansel et al. 2011] (e.g., on the end-user's device or on servers in an app store).

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 without requiring programmer-guided low-level annotations. We propose **ApproxHPVM**, a unified compiler IR and framework that solves both the problems above – 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. This enables approximate computing techniques to be supported in situations such as dynamic scheduling (where accuracy-aware mapping decisions must be performed at run-time) and hardware design space exploration (for designing hardware variations with different approximation options or parameter settings). We are exploring both opportunities in our current research.

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 [Kotsifakou et al. 2018]. HPVM itself is built on LLVM [Lattner and Adve 2004], 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 [NVIDIA 2018], 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 [Srivastava et al. 2018]. 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.

2 APPROXHPVM INTERNAL REPRESENTATION AND SYSTEM WORKFLOW

Figure 1 shows the overall ApproxHPVM workflow. The primary input is a program written using high-level abstractions of the Keras library [Gulli and Pal 2017], 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

Table 1. Tensor intrinsics in the ApproxHPVM representation.

Tensor Intrinsic	Description
i8* **@tensor.mul**(i8* lhs, i8* rhs)	Performs a matrix multiply operation on the input tensors.
i8* **@tensor.conv**(i8* input, i8* filter, i32 stride, i32 padding)	Applies a convolution filter on input tensor with given stride and padding.
i8* **@tensor.add**(i8* lhs, i8* rhs)	Element-wise addition on input tensors.
i8* **@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* **@tensor.relu**(i8* input)	Element-wise relu activation function.
i8* **@tensor.clipped.relu**(i8* input)	Element-wise clipped relu activation function.
i8* **@tensor.tanh**(i8* input)	Element-wise tanh activation function.

problem into a hardware-agnostic first stage and a hardware-specific second stage, for the reasons described in Section 1.

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 [Lattner and Adve 2004]. 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

Fig. 1. ApproxHPVM System Workflow

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 3.2).

ApproxHPVM is inspired by and builds on HPVM [Kotsifakou et al. 2018], 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 first briefly discuss the HPVM IR in the next subsection, and then describe our extensions to it.

2.1 Background: HPVM Dataflow Graph

HPVM [Kotsifakou et al. 2018] 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
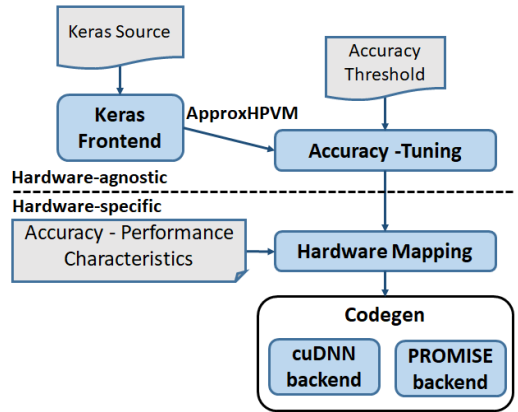
representation that uses hierarchical dataflow graphs to capture a diverse range of coarse- and fine-grain data and task parallelism including pipeline parallelism, nested parallelism, and SPMD-style (single program, multiple data) data parallelism. We showed that these abstractions allow HPVM to compile 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.

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 [The XLA Team 2019] and TVM for MxNet [Chen et al. 2018] 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

```
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, %B) { ; Root node of the Dataflow Graph
  ; 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)
}
```

Fig. 2. Convolution Layer sub-operations represented as ApproxHPVM tensor intrinsics in HPVM dataflow nodes. The data-flow nodes are connected through explicit dataflow edges using HPVM intrinsics.

optimized target-specific runtime libraries, such as cuDNN for GPUs, and b) it allows approximation analyses to leverage domain-specific information, because the approximation properties, parameters, and analysis techniques usually are determined by properties of the domain.

Table 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 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.

```
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
}
```

Fig. 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.

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 $L_1(X) = \|X\|_1 = \sum_i |x_i|$
  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 $L_2(X) = \|X\|_2 = \sqrt{\sum_i x_i^2}$
  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 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.

2.4   Keras Frontend

Keras [Gulli and Pal 2017] is a popular neural-network library that can run on top of Tensorflow and other frameworks. Keras provides a simple, programmable interface for providing high-level descriptions of neural networks. We choose Keras since it allows us to identify the higher level tensor computations that can be mapped to the ApproxHPVM tensor intrinsics. Moreover, since Keras internally maintains the data-flow relations across operations, this allows the front end to extract the data-flow information and translate it to data-flow edges in ApproxHPVM relatively easily.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5),
                 activation='relu', padding = 'same',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu', padding = 'same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dense(num_classes, activation='relu'))
model.add(Activation('softmax'))
```

Fig. 4. LeNet-5 defined in Keras

Figure 4 presents the popular LeNet-5 neural network [LeCun et al. 1998] in Keras. The LeNet architecture consists of 2 convolution layers and 2 fully-connected layers followed by a softmax layer. Moreover, the convolution layer is followed by pooling layers that downsample the input size. The frontend translates these high-level operations to ApproxHVPM tensor intrinsics. For instance, the Conv2D operator in the example is translated to 3 tensor operations in the IR - @tensor.conv, @tensor.add, and @tensor.relu. Similarly, the Dense operator (fully-connected layer) is mapped to @tensor.mul, @tensor.add, and @tensor.relu. The MaxPooling operator is mapped one-to-one to the @tensor.reduce_window IR operation with appropriate parameters. The parameters passed to each Keras operator (kernel sizes, pool sizes etc.) are also appropriately passed as parameters to the intrinsic calls in the IR. Figure 2 shows how a single Conv2D operator in Keras maps to ApproxHPVM code with high-level tensor intrinsics.

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 3.1) to determine per-operation accuracy requirements and an efficient accuracy-aware scheduler (Section 3.2) that maps the approximable components to hardware compute units and hardware-level system parameters.

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 5 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).

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 [Ansel et al. 2014], 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

Fig. 5. Hardware-agnostic accuracy-tuning workflow.

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 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 2 may look like:

```
Configuration: {
    hpvm.tensor.mul: 5,
    hpvm.tensor.add: 6,
    hpvm.tensor.tanh: 4
}
```
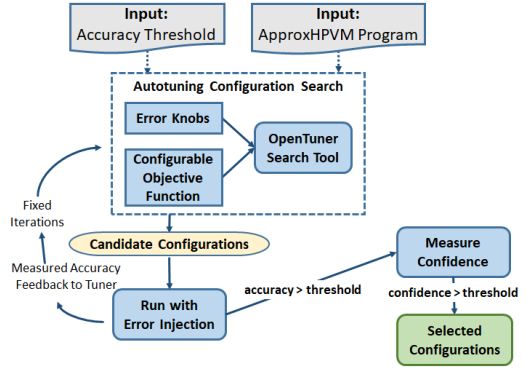
For every configuration generated by the accuracy tuner, the final accuracy is empirically evaluated by running the program with the tuned level of error injection. If the measured end-to-end accuracy is below the threshold, the configuration is rejected. Otherwise, the configuration is saved as a candidate 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 [Misailovic et al. 2013], 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:

$$C_{Total}(config) = \sum_{i=0}^{N} C(op(i), E(i)) \tag{1}$$

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:

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

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.

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 [Srivastava et al. 2018]. 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 [Ho and Wong 2017; Micikevicius et al. 2018].

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 [Srivastava et al. 2018], 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 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 $K \in \left\{2^8, 2^9, \ldots, 2^{15}\right\}$. The matrices are initialized with random values from uniform distribution $\mathcal{U}(0, 1)$. 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.

Fig. 6. Time and energy improvement, and L_1^e for the following hardware knobs: FP16, P7, and P1. It can be seen that PROMISE is faster and less accurate than FP16, which is faster and less accurate than FP32. Note that P1 and P7 Time curves overlap since execution time is constant across different swing values.

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 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
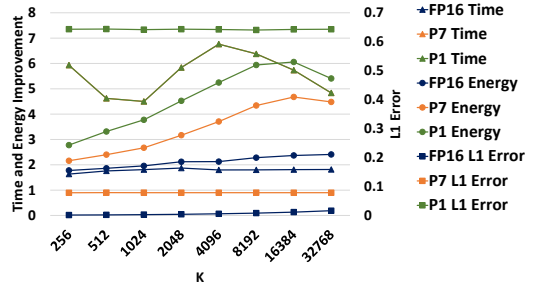
a tensor operation to the least accurate hardware setting that meets the L_1^e and L_2^e constraints of the operation. Since the mapping from individual operation L1 error and L2 error to hardware knobs is merely a table lookup operation, hardware scheduling is an inexpensive step. This makes our hardware 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.

3.3 Code Generation

In its final phase, the ApproxHPVM compiler generates code for each operation corresponding to the selected compute unit. We added new backends for PROMISE and for an optimized cuDNN and cuBLAS based library runtime for GPU. Since the support for backends is flexible, it can be extended to other approximate computing hardware platforms. The back-end code generators translate dataflow graph nodes (containing tensor intrinsics such as @hpvm.tensor.mul) to functions that invoke the corresponding DNN operations for GPU or PROMISE.

Code generation for PROMISE requires an extra pattern-driven fusion operation because the hardware can perform an entire layer operation as a single PROMISE instruction [Srivastava et al. 2018]. A layer operation in a DNN usually maps to the following common patterns for fully-connected and convolution layers, respectively:

$$Y_{FC} = f(X \cdot W + B) \qquad Y_{Conv} = f(X \circledast W + B)$$

where W, X and B are the weight tensor, input tensor, and bias tensor, and $f(\cdot)$ is the activation function (sigmoid, relu, tanh, etc.). We implement a pattern-driven Node Fusion transformation that identifies sequences of nodes performing these operations and fuses the nodes into a single ApproxHPVM dataflow node if they are all mapped to PROMISE.

4 METHODOLOGY

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 [NVIDIA 2018], augmented with the PROMISE programmable machine learning accelerator [Srivastava et al. 2018]. 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.

Table 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

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 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 [NVIDIA 2018]. This board contains the NVIDIA Tegra TX2 SoC [Franklin 2018], 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 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.

4.2 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.

4.3 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 [NVIDIA Developer Forums 2018] 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

Table 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

(pDMA) [Jamshidi et al. 2014; Komuravelli et al. 2015]. 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 [Gonugondla et al. 2018]. For the main memory model, we empirically measured peak sustained 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 Benchmarks

Our evaluation includes 9 DNN benchmarks and 5 image processing pipelines, detailed in Table 3a and Table 3b, respectively.

DNN Benchmarks. We include a range of different convolutional neural networks for 3 different datasets: MNIST [LeCun et al. 1998], CIFAR-10, and CIFAR-100 [Krizhevsky and Hinton 2009]. 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[LeCun et al. 1989], AlexNet [Krizhevsky et al. 2012] (reference implementation [Yang 2019]), ResNet-18 [He et al. 2016], VGG-16 [Simonyan and Zisserman 2014] (reference implementation [Geifman 2019]), MobileNet [Howard et al. 2017], and Shallow MobileNet [Howard et al. 2017]. 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 [Geifman 2019]. 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 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 [Fei-Fei et al. 2004] 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 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.

4.5 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 [Li and Cai 2007; Thomos et al. 2006].) To illustrate the visual impact, Figure 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,
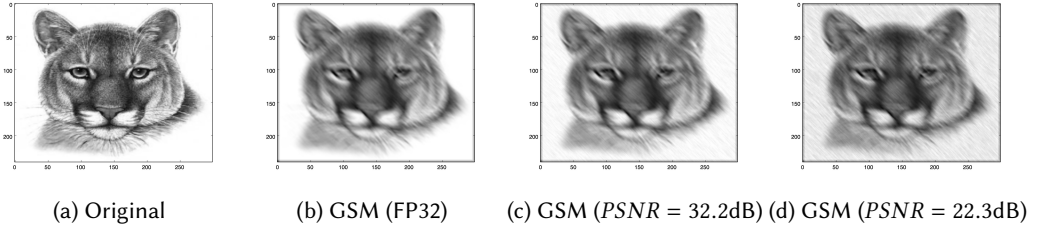
(a) Original   (b) GSM (FP32)   (c) GSM ($PSNR = 32.2$dB) (d) GSM ($PSNR = 22.3$dB)

Fig. 7. Sample output from GSM (Gaussian-Sharpen-MotionBlur) benchmark. 7a: Original image; 7b: GSM baseline output (FP32 without approximation); 7c, 7d: GSM output approximated at $PSNR_{30}$ and $PSNR_{20}$ respectively.

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 [Xu et al. 2018], we use 5% as an acceptable threshold for the violation rate. Similar to the DNN benchmarks, the baseline uses FP32 for all filter computations.

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?

5.1 Performance and Energy Evaluation

DNN Benchmarks. Figure 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 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 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 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
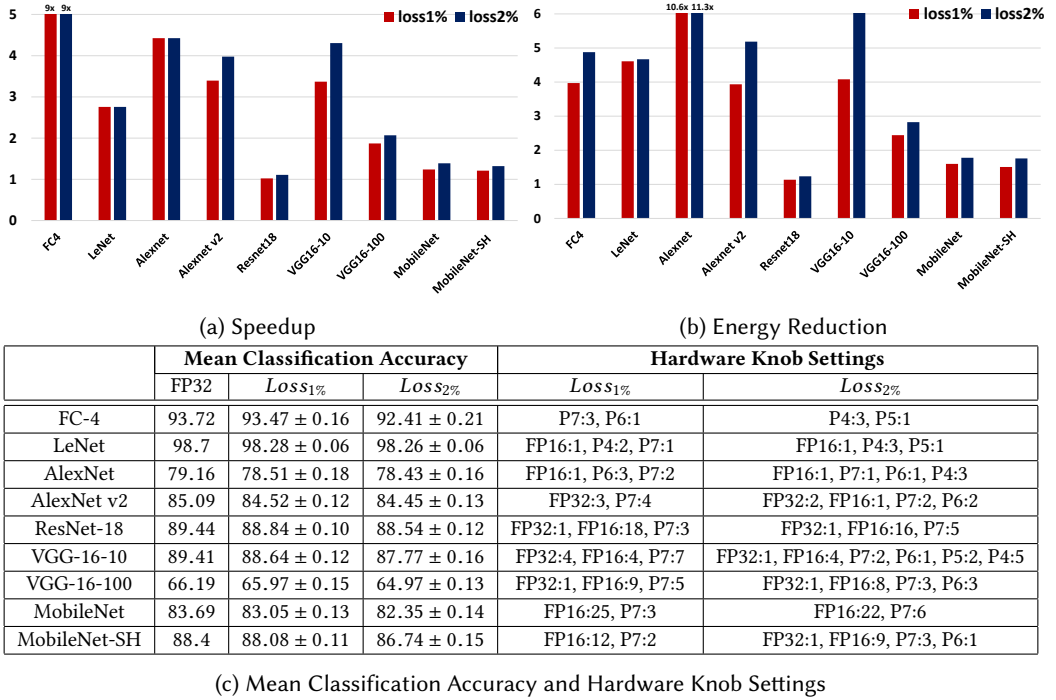
(a) Speedup

(b) Energy Reduction

	Mean Classification Accuracy			Hardware Knob Settings	
	FP32	$Loss_{1\%}$	$Loss_{2\%}$	$Loss_{1\%}$	$Loss_{2\%}$
FC-4	93.72	93.47 ± 0.16	92.41 ± 0.21	P7:3, P6:1	P4:3, P5:1
LeNet	98.7	98.28 ± 0.06	98.26 ± 0.06	FP16:1, P4:2, P7:1	FP16:1, P4:3, P5:1
AlexNet	79.16	78.51 ± 0.18	78.43 ± 0.16	FP16:1, P6:3, P7:2	FP16:1, P7:1, P6:1, P4:3
AlexNet v2	85.09	84.52 ± 0.12	84.45 ± 0.13	FP32:3, P7:4	FP32:2, FP16:1, P7:2, P6:2
ResNet-18	89.44	88.84 ± 0.10	88.54 ± 0.12	FP32:1, FP16:18, P7:3	FP32:1, FP16:16, P7:5
VGG-16-10	89.41	88.64 ± 0.12	87.77 ± 0.16	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 ± 0.15	64.97 ± 0.13	FP32:1, FP16:9, P7:5	FP32:1, FP16:8, P7:3, P6:3
MobileNet	83.69	83.05 ± 0.13	82.35 ± 0.14	FP16:25, P7:3	FP16:22, P7:6
MobileNet-SH	88.4	88.08 ± 0.11	86.74 ± 0.15	FP16:12, P7:2	FP32:1, FP16:9, P7:3, P6:1

(c) Mean Classification Accuracy and Hardware Knob Settings

Fig. 8. Speedup and energy reduction (over baseline) of all nine DNNs for $Loss_{1\%}$ and $Loss_{2\%}$ experiments (higher is better). 8a: Speedup. 8b: Energy reduction. 8c: Mean accuracy ± 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.

(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 performance (energy). Figure 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 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 ranging 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
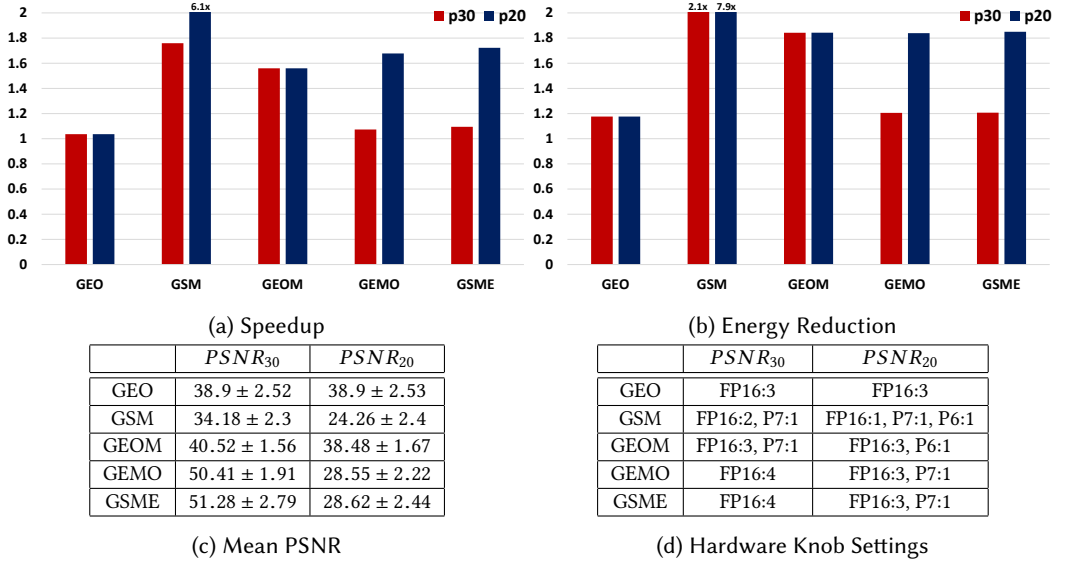
(a) Speedup

	$PSNR_{30}$	$PSNR_{20}$
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

(b) Energy Reduction

	$PSNR_{30}$	$PSNR_{20}$
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

Fig. 9. Speedup and energy reduction (over baseline) of all 5 image processing benchmarks for $PSNR_{30}$ (p30) and $PSNR_{20}$ (p20) thresholds. 9a: Speedup. 9b: Energy reduction. 9c: Mean PSNR ± standard deviation. 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.

$Loss_{2\%}$. In Section 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 9 shows the aggregate results for all 5 image processing benchmarks. Figures 9a, 9b, 9c show the performance improvement, energy reduction, and mean PSNR of $PSNR_{30}$ and $PSNR_{20}$ experiments.

Figure 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 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 8c and 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 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
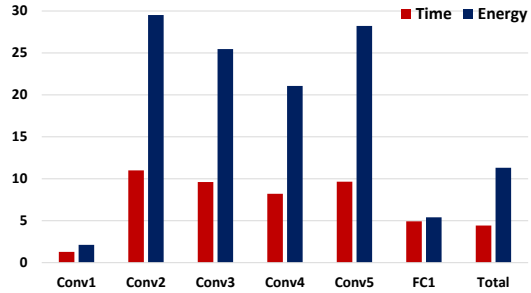

Fig. 10. Speedup and energy reduction over baseline for all six layers of AlexNet ($Loss_{2\%}$). Conv1 cannot be mapped to PROMISE and is executed using FP16. It thus observes the smallest benefit and also significantly reduces the overall time and energy improvement.

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 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.

5.2 Hardware-Agnostic vs Hardware-Specific Tuning

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

(a) Normalized DNN Speedup

(b) Normalized DNN Energy Reduction

(c) Normalized Image Benchmark Speedup

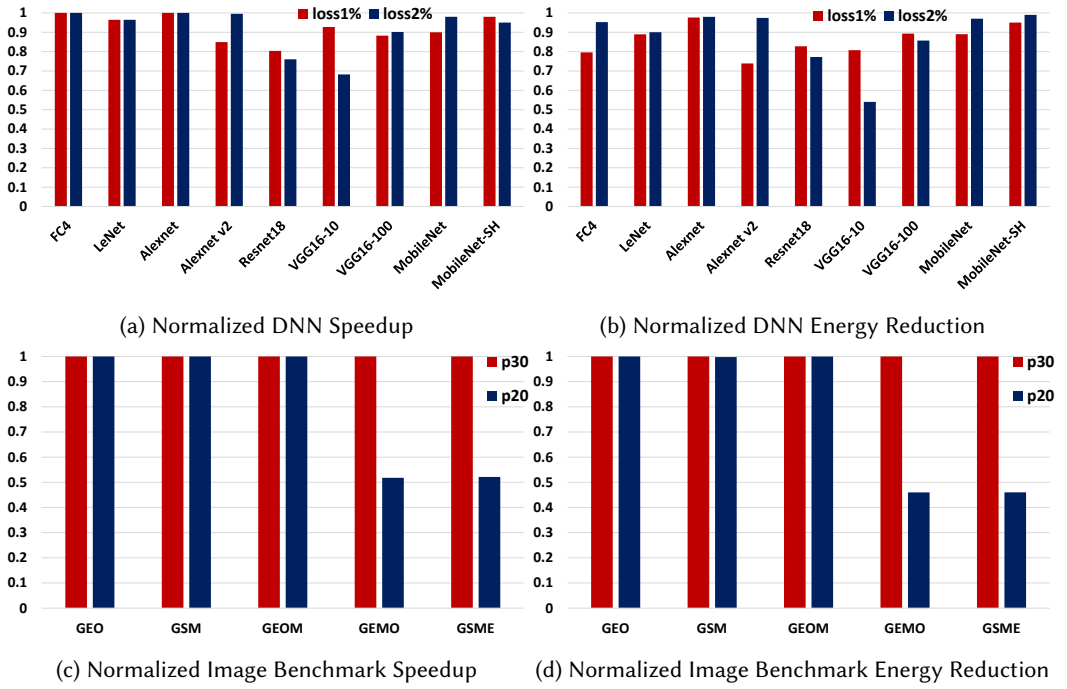(d) Normalized Image Benchmark Energy Reduction

Fig. 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.

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 11a, 11b, 11c, and 11d.

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 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 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.

5.3 Autotuning Times

We built our autotuner by leveraging the interface provided by the OpenTuner framework [Ansel et al. 2014]. 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 in hours for hardware-agnostic and hardware-specific experiments for each benchmark are included in Table 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

Table 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

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.

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 NVIDIA's cuDNN library [Chetlur et al. 2014] and PROMISE perform GEMM-based convolution (Section 3.3). 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.
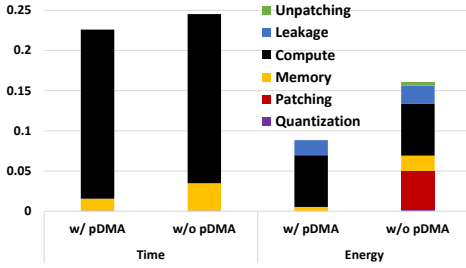
Fig. 12. Normalized execution time and energy for AlexNet with and without pDMA. Without pDMA, PROMISE relies on the GPU to perform quantization, patching, and unpatching; these overheads reduce the performance and energy improvement over FP32.

In cuDNN, patching and unpatching are done in on-chip memory to minimize this overhead [Chetlur et al. 2014]. In PROMISE, we can either use the pDMA scheme described in Section 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 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.
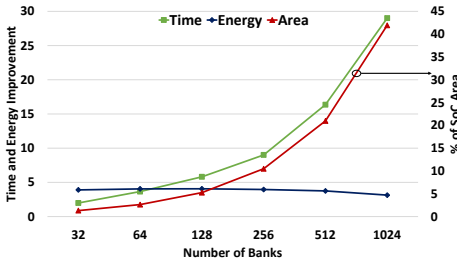

Fig. 13. Speedup and energy reduction over FP32, and area for FC4 vs the number of PROMISE banks. The SoC contains 4B transistors.

#Banks: We performed a scaling study of the number of PROMISE banks to establish the suitability of a 256 bank configuration. Figure 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 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.

6 RELATED WORK

Software Approximations. Many studies have introduced novel software techniques for approximation that reduce execution time and/or energy. The transformations include task skipping [Meng et al. 2009, 2010; Rinard 2006], loop perforation [Misailovic et al. 2011, 2010; Sidiroglou-Douskos et al. 2011], approximate function substitution [Ansel et al. 2011; Baek and Chilimbi 2010; Samadi et al. 2014; Zhu et al. 2012], dynamic knobs [Hoffmann et al. 2011] (dynamically changing function version), reduction sampling [Goiri et al. 2015; Samadi et al. 2014; Zhu et al. 2012], tuning floating-point operations [Rubio-González et al. 2013; Schkufza et al. 2014], and approximate parallelization [Campanoni et al. 2015; Misailovic et al. 2013, 2012; Samadi et al. 2014]. These techniques have been shown to work well across a variety of application domains resilient to small errors. We envision that ApproxHPVM will provide a general framework for expressing and developing these optimizations, leveraging the ability of the underlying IR to explicitly specify parallelism and data

dependency. In conjunction, our IR allows a developer to explicitly specify approximation metrics and the error tolerance allowing the compiler to apply more flexible approximations.

Approximation-Aware Languages. EnerJ [Sampson et al. 2011] presents a type system that separates approximate and precise data. The developer needs to annotate each variable in the source code as precise or approximate before the type system can ensure that the approximate data is never assigned to the precise variable. More recently, Decaf [Boston et al. 2015] performs type inference to reduce the developer annotation effort.

Rely [Carbin et al. 2013] and Chisel [Misailovic et al. 2014] introduced the idea of quantifiable reliability and absolute error at the program level. They define function-level specifications that express the maximum probability with which the function can produce an inaccurate result. These specifications separate the optimization within the function from the uses of the function: the code that calls the function can rely on the specification, while the body of the function can be modified separately and Rely and Chisel can statically verify that those implementations satisfy the specification.

ApproxHPVM introduces the concept of quantifiable reliability at the IR level. Incorporating approximation metrics at the IR level provides a more portable alternative, since the metrics are preserved even after compiling the program. Effectively, the approximation becomes a first-class citizen in the compiler workflow, which is able to leverage the interaction of the accuracy-aware IR with various front-end languages *and* hardware-specific features, especially in heterogeneous systems. We also adopted and generalized Chisel's sensitivity profiling based on error-injection to infer the acceptable thresholds for a wider range of specifications for tensor operations.

Systems for Automated Accuracy Tuning. The Petabricks programming language automatically tunes the program to select among multiple user-provided versions of the algorithm with varying accuracy and performance characteristics [Ansel et al. 2009, 2011; Ding et al. 2015]. Its auto-tuner uses heuristic algorithm based on genetic programming to search among alternative program implementations. OpenTuner [Ansel et al. 2014] extends these ideas to provide a general autotuning framework for programs written in conventional languages.

Green [Baek and Chilimbi 2010] presents a combined offline and run-time accuracy tuning mechanism. Its offline heuristic algorithm selects the parameters of the approximation opportunities exposed by the developer. Its run-time system infrequently re-executes the exact sub-computations, compares it to the approximate one and adjusts the approximation accordingly. Loop perforation [Misailovic et al. 2010; Sidiroglou-Douskos et al. 2011] presents an off-line auto-tuner, which automatically searches for the loops to approximate. Its search consists of two phases: 1) sensitivity testing, which checks whether the perforated loop will crash the program, slow it down, cause memory leaks, or produce illegal outputs, and 2) accuracy tuning, which finds the loops with maximum speedup for every end-to-end accuracy loss bound. More recently, Accept [Sampson et al. 2015] builds on this strategy, by including the developer annotations and type system from EnerJ to constrain the search space and automatically apply several approximate transformations, including offloading to approximate hardware (using the approach from Esmaeilzadeh et al. [2012]).

Unlike ApproxHPVM, these previous approaches consider only end-to-end accuracy of the computation and do not decouple hardware-independent from hardware-dependent tuning. Also these approaches (except Accept) have been used only for optimizing CPU code. In contrast, decoupling the tuning and IR-level accuracy specifications enable us to achieve portable approximate object code, and provide natural abstractions for future uses like dynamic scheduling and hardware-level design space exploration.

Approximation techniques for deep neural networks, e.g., fixed point quantization of pretrained neural networks [Lin et al. 2016] or layer-grained analytical models for bit-widths of weights

and activations [Sakr et al. 2017], require domain-specific algorithmic knowledge. In contrast, our accuracy-tuning does not require any domain-specific knowledge, instead using an efficient search-based approach for determining the accuracy requirements for different operations.

Approximate Hardware Accelerators. Recently there have been many proposals for machine learning accelerators [Chen et al. 2014, 2016; Du et al. 2015; Esmaeilzadeh et al. 2012; Liu et al. 2016; St. Amant et al. 2014], some of which explicitly incorporate approximations. Although we have chosen PROMISE and GPUs as the accelerators in our evaluations, our framework is more broadly applicable to the wide range of emerging approximate accelerator platforms.

Compiler-based Systems for Machine Learning. TVM [Chen et al. 2018] proposes a compiler framework that supports the compilation and optimization of machine learning workloads on multiple hardware targets. Similarly, Glow [Rotem et al. 2018] and XLA [The XLA Team 2019] are also ML-based compiler frameworks that leverage DNN-specific operations in the IR design, thereby facilitating domain-specific optimizations. As ApproxHPVM also leverages domain-specific information with the inclusion of high-level tensor intrinsics, it also facilitates such domain-specific optimizations. None of these systems include approximation metrics in the IR design and hence do not provide the portability and flexibility offered by ApproxHPVM. While these existing systems provide support for precision-tuning to FP16 and INT8, ApproxHPVM provides more extensive support for approximation since it also allows for mapping computations to accelerators that provide performance-energy-accuracy trade-offs. Moreover, ApproxHPVM enables approximation mechanisms that are not limited to the machine learning domain.

7   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.

REFERENCES

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. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1542476.1542481

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 (PACT '14). ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

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 *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 85–96. http://dl.acm.org/citation.cfm?id=2190025.2190056

Woongki Baek and Trishul M. Chilimbi. 2010. 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 (PLDI '10)*. ACM, New York, NY, USA, 198–209. https://doi.org/10.1145/1806596.1806620

Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *OOPSLA*. ACM, 470–487.

Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 235–245. http://dl.acm.org/citation.cfm?id=2738600.2738630

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. 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 & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 33–52. https://doi.org/10.1145/2509136.2509546

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622. https://doi.org/10.1109/MICRO.2014.58

Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Vol. 44. IEEE, 367–379. https://doi.org/10.1109/ISCA.2016.40

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

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

Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 92–104. https://doi.org/10.1145/2749469.2750389

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 449–460. https://doi.org/10.1109/MICRO.2012.48

Li Fei-Fei, Rob Fergus, and Pietro Perona. 2004. 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*. 178–178. https://doi.org/10.1109/CVPR.2004.383

Dustin Franklin. 2018. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. NVIDIA Developer Blog. (2018). https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge

Yonatan Geifman. 2019. VGG16 models for CIFAR-10 and CIFAR-100 using Keras. https://github.com/geifmany/cifar-vgg. (2019).

Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. 2015. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*. ACM, 383–397.

S. K. Gonugondla, M. Kang, and N. R. Shanbhag. 2018. A Variation-Tolerant In-Memory Machine Learning Classifier via On-Chip Training. *IEEE Journal of Solid-State Circuits* 53, 11 (Nov 2018), 3163–3173. https://doi.org/10.1109/JSSC.2018.2867275

Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091072

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 199–212. https://doi.org/10.1145/1950365.1950390

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. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

D. Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. 2014. D2MA: Accelerating Coarse-grained Data Transfer for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 431–442. https://doi.org/10.1145/2628071.2628072

Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 707–719. https://doi.org/10.1145/2749469.2750374

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

Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS '12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=977395.977673

Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. 1989. Handwritten Digit Recognition with a Back-propagation Network. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems (NIPS '89)*. MIT Press, Cambridge, MA, USA, 396–404. http://dl.acm.org/citation.cfm?id=2969830.2969879

Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. 1998. The MNIST database of handwritten digits. (1998). http://yann.lecun.com/exdb/mnist

Xiangjun Li and Jianfei Cai. 2007. 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*. 947–950.

Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 2849–2858. http://dl.acm.org/citation.cfm?id=3045390.3045690

Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 393–405. https://doi.org/10.1109/ISCA.2016.42

Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. 2009. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/IPDPS.2009.5160991

Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Surendra Byna. 2010. Exploiting the forgiving nature of applications for scalable parallel execution. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS '10)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470469

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. https://openreview.net/forum?id=r1gs9JgRZ

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. 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 (OOPSLA '14)*. ACM, New York, NY, USA, 309–328. https://doi.org/10.1145/2660193.2660231

Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Transactions Embedded Computing Systems (TECS)* 12, Article 88 (May 2013), 26 pages. Issue 2s. https:

//doi.org/10.1145/2465787.2465790

Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. 2011. Probabilistically Accurate Program Transformations. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 316–333. http://dl.acm.org/citation.cfm?id=2041552.2041576

Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of Service Profiling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/1806799.1806808

Sasa Misailovic, Stelios Sidiroglou, and Martin C. Rinard. 2012. Dancing with Uncertainty. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2414729.2414738

NVIDIA. 2010. PTX: Parallel thread execution ISA version 2.3. *NVIDIA COMPUTE Programmer's Manual* 3 (2010). http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf

NVIDIA. 2018. NVIDIA Jetson TX2 Developer Kit. (2018). https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2

NVIDIA Developer Forums. 2018. Power Monitoring on Jetson TX2. (2018). https://devtalk.nvidia.com/default/topic/1000830/jetson-tx2/jetson-tx2-ina226-power-monitor-with-i2c-interface

Martin Rinard. 2006. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 324–334. https://doi.org/10.1145/1183401.1183447

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503296

Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. 2017. Analytical Guarantees on Numerical Precision of Deep Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML '17)*. 3007–3016. http://dl.acm.org/citation.cfm?id=3305890.3305992

Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 35–50. https://doi.org/10.1145/2541940.2541948

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*. https://dada.cs.washington.edu/research/tr/2015/01/UW-CSE-15-01-01.pdf

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. 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 (PLDI '11)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

Ben Sander. 2013. HSAIL: Portable compiler IR for HSA.. In *Hot Chips Symposium 2013*. 1–32.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. 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 (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556

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 *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 43–56. https://doi.org/10.1109/ISCA.2018.00015

Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose Code Acceleration with Limited-precision Analog Computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 505–516. http://dl.acm.org/citation.cfm?id=2665671.2665746

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

The XLA Team. 2019. XLA: Domain-specific compiler for linear algebra that optimizes TensorFlow computations. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/g3doc/overview.md. (2019).

N. Thomos, N. V. Boulgouris, and M. G. Strintzis. 2006. Optimized Transmission of JPEG2000 Streams Over Wireless Channels. *IEEE Transactions on Image Processing* 15, 1 (January 2006).

Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Sasa Misailovic, and Saurabh Bagchi. 2018. VideoChef: Efficient Approximation for Streaming Video Processing Pipelines. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 43–56. https://www.usenix.org/conference/atc18/presentation/xu-ran

Wei Yang. 2019. Classification on CIFAR-10/100 and ImageNet with PyTorch. https://github.com/bearpaw/pytorch-classification/blob/master/models/cifar/alexnet.py. (2019).

Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 2012. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 441–454. https://doi.org/10.1145/2103656.2103710