COMPILERS FOR PORTABLE PROGRAMMING OF HETEROGENEOUS PARALLEL & APPROXIMATE COMPUTING SYSTEMS

BY

PRAKALP SRIVASTAVA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair
Professor Sarita Adve
Dr. Michael Garland, NVIDIA Research
Assistant Professor Sasa Misailovic
Professor Naresh Shanbhag

ABSTRACT

Programming heterogeneous systems such as the System-on-chip (SoC) processors in modern mobile devices can be extremely complex because a single system may include multiple different parallelism models, instruction sets, memory hierarchies, and systems use *different combinations* of these features. This is further complicated by software and hardware approximate computing optimizations. Different compute units on an SoC use different approximate computing methods and an application would usually be composed of multiple compute kernels, each one specialized to run on a different hardware. Determining how best to map such an application to a modern heterogeneous system is an open research problem.

First, we propose a parallel abstraction of heterogeneous hardware that is a carefully chosen combination of well-known parallel models and is able to capture the parallelism in a wide range of popular parallel hardware. This abstraction uses *a hierarchical dataflow graph with side effects and vector SIMD instructions*. We use this abstraction to define a parallel program representation called HPVM that aims to address both functional portability and performance portability across heterogeneous systems.

Second, we further extend HPVM representation to enable accuracy-aware performance and energy tuning on heterogeneous systems with multiple compute units and approximation methods. We call it ApproxHPVM, and it automatically translates end-to-end application-level accuracy constraints into accuracy requirements for individual operations. ApproxHPVM uses a hardware-agnostic accuracy-tuning phase to do this translation, which greatly speeds up the analysis, enables greater portability, and enables future capabilities like accuracy-aware dynamic scheduling and design space exploration.

We have implemented a prototype HPVM system, defining the HPVM IR as an extension of the LLVM compiler IR, compiler optimizations that operate directly on HPVM graphs, and code generators that translate the virtual ISA to NVIDIA GPUs, Intel's AVX vector units, and to multicore X86-64 processors. Experimental results show that HPVM optimizations achieve significant performance improvements, HPVM translators achieve performance competitive with manually developed OpenCL code for both GPUs and vector hardware, and that runtime scheduling policies can make use of both program and runtime information to exploit the flexible compilation capabilities. Furthermore, our evaluation of ApproxHPVM shows that our framework can offload chunks of approximable computations to special-purpose accelerators that provide significant gains in performance and energy, while staying within a user-specified application-level accuracy constraint with high probability.

To my family, for their love and support.

ACKNOWLEDGMENTS

There are several people whom I would like to thank for supporting me and believing in me throughout my Ph.D. journey. First, I want to extend my sincere gratitude to my advisor, Vikram Adve, for being my mentor through this journey. He always inspired me to do my best and match up to his uncompromising high standards. I thank him for his time and effort to train me as an independent researcher. I would also like to thank Sarita Adve and Sasa Misailovic for their guidance in the ApproxHPVM project. I thank Naresh Shanbhag and Nam Sung Kim for their collaboration and guidance through the PROMISE project. I also sincerely thank my Ph.D. committee for their insightful comments and suggestions for improvements on my thesis.

I am thankful to Maria Kotsifakou for her collaboration on the HPVM and ApproxHPVM project. I would also thank Hashim Sharif and Huzaifa Muhammad for their collaborations on the ApproxHPVM project. I have learned a lot by working closely with these three. In addition, I am also thankful for my other collaborators on these projects, Mingu Kang, Sujan Kumar Gonugondla, Matt Sinclair, Rakesh Komuravelli, John Alsop, Sungmin Lim and Keyur Joshi. I thank my other lab-mates, Joshua, Nathan, Sandeep, Sean, Theo, and Will who played a very important role in not only providing a great and fun research environment in the lab but also provided several thoughtful philosophical discussions. I thank the Computer Science department at Illinois for providing a wonderful Ph.D. curriculum and flexibility for conducting research. Specifically, I would like to thank the staff members, Kathy, Jennifer, Maggie, and Mary Beth who on several occasions helped me with administrative chores.

My work has been supported by the National Science Foundation under grants CCF 13-02641 and CCF 16-19245, by the Center for Future Architectures Research (C-FAR) and SONIC - two of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by Intel Corporation. Part of this work was also supported by DARPA through the Domain Specific System on Chip (DSSoC) program and by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

I have made many valuable friends during my stay here at UIUC with whom I have experienced some of the best moments of my graduate life. My immense thanks to Shikha, Shashank, Vivek, Urwashi, Ankur, Manila, Advitya, Komal, Mayank, Pranav, Eros, Ankita, Kini, Shishir, Shikhar, Renato, Jia, and Fardin. This is by no means an exhaustive list.

Finally, it goes without saying how much I am indebted to my family for their understanding and encouragement for all these years. This journey of my Ph.D. would have been much harder if it were not for you and I proudly dedicate this dissertation to you.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 HETEROGENEOUS SYSTEMS: CURRENT TRENDS AND FUTURE

Earlier Moore's Law and Dernard's scaling allowed computer architects to design single core processors with smaller transistors that were faster than their predecessors because smaller transistors can switch at higher speeds with power density remaining constant [1]. However, the end of Dernard's scaling around 2006 meant that as transistors became smaller, the power density increases. In other words, architects can no longer design smaller and faster single core processors with same power density. As a result, the computer architects turned to multicore processors and parallelism [2]. This allowed manufacturers to double the number of transistors every 2 years according to Moore's Law but this approach is limited by power constraints [3]. Thus, the design of energy efficient processors became a major goal for the computer architecture community. Heterogeneous Computing has emerged as a promising approach to pursue this goal as specialized compute units can be orders of magnitude more energy efficient than general purpose processing.

Heterogeneous computing systems are becoming increasingly popular in systems ranging from portable mobile devices to high-end supercomputers to data centers [3]. Such systems are attractive because they use specialized computing elements, including GPUs, vector hardware, FPGAs, and domain-specific accelerators, that can greatly improve energy efficiency, performance, or both, compared with traditional homogeneous systems [4]. Two studies [5, 6] of a media encoder and TCP offload engine illustrate the large energy-efficiency improvement that is possible. They achieve these goals through specialized computing which includes exploiting parallelism, data access patterns, and other forms of domain specific knowledge [7, 8]. Also, domains such as machine learning are inherently tolerant to imprecision which is effectively exploited by specialized accelerators [9, 10].

The term "heterogeneous systems" is very broad in its scope and depending on the context and current trends could mean anything from single ISA heterogeneous systems such as ARM's big.LITTLE [11] chip to a system composed of compute units that differ not only in microarchitecture but have different instruction-set architectures (ISAs), many of them could be highly specialized domain-specific accelerators. At the time of writing this thesis, the term "heterogeneous systems" typically refers to systems with general purpose CPUs and GPUs, usually on the same integrated circuit. CPUs can run the operating system and traditional serial tasks, while GPUs and CPU vector instructions can be used to exploit data parallelism in applications. It is also fairly common to have some domain-specific accelerators on the

same integrated chip which may or may not support limited programmability. For example, Qualcomm's Snapdragon [12] system-on-a-chip (SoC) has accelerators such as audio/video encoders/decoders, programmable Hexagon [13] DSP integrated on the chip, apart from Kryo CPU [12] and Adreno GPU [14]. In May'2018 Intel released the Xeon Gold 6138P product which includes a FPGA along with the Intel Xeon processor. The CPU and the FPGA talk with one another over the Ultra Path Interconnect (UPI) bus, providing a cache-coherent coupling between the two.

We expect this trend of increasing diversity in heterogeneous systems to continue [15]. The computation requirements of applications is increasing, specially for domains which rely on processing large amounts of data [16]. Application specific hardware have been shown to be orders of magnitude more energy efficient than general purpose processors [5, 6]. To meet the power and energy constraints, it is expected that the level of heterogeneity in modern computing systems would keep increasing gradually, as further scaling of fabrication technologies allows for formerly discrete components to become integrated parts of a SoC [3].

This heterogeneity is not limited to different ISAs, memory hierarchies and parallelism models. Approximate computing has been shown to be viable in multiple domains [17] and provides throughput and energy benefits in these domains. Researchers and engineers have successfully exploited applications' tolerance to imprecision using a diverse range of approximate computing techniques, both in software and hardware [18, 19, 20, 21, 22, 23].

1.2 PROGRAMMABILITY CHALLENGES

Heterogeneous parallel computing systems, including both mobile System-on-Chip (SoC) designs such as Qualcomm's Snapdragon and nVidia's Tesla, or high-end supercomputers like Cray's BlueWaters (which has many GPU coprocessors) or Convey's FPGA-based HC-1, raise numerous difficult programming challenges. We believe these challenges arise from *four fundamental root causes* and we first discuss these root causes and then outline the challenges they engender.

1.2.1 Root Causes of Programmability Challenges

(1) Diverse models of parallelism: Different hardware components in heterogeneous systems support different models of parallelism. We tentatively identify five broad classes of *programmable* hardware that have qualitatively different models of parallelism:

1.	General purpose cores	Flexible multithreading
2.	Vector hardware	Vector parallelism
3.	GPUs	Restrictive data parallelism
4.	FPGAs	Customized dataflow
5.	Custom accelerators	Various forms

In addition, applications running on multiple such components may exhibit asynchronous or synchronous parallelism relative to each other. OpenCL [24] and OpenMP [25] are efforts to achieve portability at source language level. They follow a similar programming model where a single-threaded kernel function is replicated across a large number of cores. This maps well to GPUs and vector parallelism. Also, they have been used to compile to FPGAs. However, they make it difficult to express important kinds of parallelism, like pipelined parallelism useful for custom accelerators.

(2) Diverse memory architectures: With the different parallel models come deep differences in the memory system. Common choices in the various components above include cache-coherent memory hierarchies, vector register files, private or "scratchpad" memory, stream buffers, and custom memory designs used in custom accelerators. These differences in memory architectures strongly influence both algorithm design and application programming. Moreover, the performance tradeoffs are becoming even more complex as new architectures provide more options, e.g., nVidia's Fermi architecture allows a 64 KB block of SRAM to be partitioned flexibly into part L1 cache and part private scratchpad memory.

(3) Diverse hardware-level instruction sets and execution semantics: Finally, the various hardware components have very different instruction sets, register architectures, performance characteristics, and execution semantics. These differences have an especially profound effect on object-code portability. They also have other negative effects, described below.

(4) Diverse approximate computing methods: Software and hardware techniques that exploit application's tolerance to approximation further exacerbate the problem. Different compute units on an SoC use different software and hardware methods to trade-off energy and performance for accuracy using approximate computing. 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 require 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 voltage swings).

1.2.2 Major Programmability Challenges

These fundamental forms of diversity create deep programmability challenges for heterogeneous systems. First, it is extremely difficult *to design a single algorithm* for a given problem that works well across a range of such different models of parallelism, with such different memory systems, as previous work has shown [26, 27]. We envisage two options to address this problem: design algorithms that achieve good, but not optimal, performance across the targeted range of hardware, or use multiple algorithms for a given problem and select among them when the actual hardware configuration is known (e.g., at install time, load time, or run time) [28, 27]. In practice, both approaches will likely be necessary.

Second, it is much more difficult *to design effective source-level programming languages* for heterogeneous systems. A single programming language or library typically supports only one or two models of parallelism. For example, CUDA, OpenCL and AMP naturally support fine-grain data parallelism, with a macro function replicated across a large number of threads, but other parallelism models (like more flexible dataflow) are not specifically addressed. Similarly, BlueSpec [29] and Lime [30], which have proved successful for FPGAs, are both dataflow models but it is not clear whether these can be mapped effectively to GPUs. The consequence today is that, a programmer must program each hardware component differently, which creates a huge barrier to entry for widespread use of heterogeneous hardware.

A third challenge is *source code portability*. Heterogeneous systems can provide *different combinations of hardware*, both within a single manufacturer's family of devices and across different manufacturers' devices. This makes source-code level portability difficult, in two ways. First, each component must solve the algorithm portability problem (above). Second, compilers must map source code embodying one or more algorithms for each component down to the various hardware components on which those algorithms must run. This task is greatly complicated by all three forms of diversity.

Fourth, *performance tuning* for heterogeneous systems will also be significantly more complex. The disparate parallelism models, memory architectures, and lower-level performance details require significantly different performance models and tuning strategies. Because of these disparities, the programmer training, software tools, and application libraries all become prohibitively expensive as the number of different hardware components grows.

Finally, *object-code portability* across the same and different manufacturers' devices is essential as well. An application vendor must be able to ship a single software version for a broad range of devices – it is impractical to create, test, market and support different versions of an application package for all the different devices running a single platform, e.g.,

all the smartphones running Android. Today, Android solves this problem by using Java bytecode, *but only for CPU application code: few application components take advantage of the on-phone GPU or DSPs*, and those are usually native libraries. Moreover, the debuggers, profilers and performance tools for a family of heterogeneous systems must support the full range of available hardware, both within a single system and also across different system configurations. Because both tuning and debugging often need to go down to the level of object code, these tools become expensive to develop, learn and use for each family of hardware.

1.2.3 Scope of Heterogeneous Systems for this Thesis

Our aim, in this thesis, is to address these programmability and portability challenges. Thus, it is also important to define the scope of heterogeneous systems for this thesis. For this work, we envision a SoC similar to Qualcomm's Snapdragon SoC. It has multicore CPUs (with SIMD instruction support), GPUs, DSPs, programmable accelerators and possibly FPGAs. An example of the programmable accelerator would be PROMISE [18], a mixed signal accelerator for machine learning designed as part of this thesis.

1.3 SUMMARY OF RELATED WORK

Virtual Instruction Sets (virtual ISAs). An approach to solving these programmability challenges is by eliminating all the root causes. This can be achieved by abstracting away the differences in heterogeneous hardware, and presenting a more uniform hardware abstraction across devices to software. More specifically, a virtual instruction set (virtual ISA) can encapsulate all the relevant programmable hardware components on target systems. In this instruction set, source-level applications are compiled, optimized, and shipped as virtual object code and then translated down to a specific hardware configuration, usually at install time, using system- specific compiler back ends (translators). Some widely used heterogeneous devices, such as GPUs, define a *virtual instruction set* (ISA) spanning one or more families of devices, e.g., PTX for NVIDIA GPUs, HSAIL for GPUs from several vendors and SPIR for devices running OpenCL. Except for SPIR, which is essentially a lower-level representation of the OpenCL language, these virtual ISAs are primarily focused on GPUs and do not specifically address other hardware classes, like vector hardware or FPGAs. Moreover, *none of these virtual ISAs* aim to address the other challenges, such as algorithm design, language design, support for approximation and compiler optimizations, across diverse heterogeneous devices.

Compiler IRs. Previous parallel compiler IRs we know of (for example, [31, 32, 33, 34, 35]) use hierarchical task dependence graph as parallel program representation. None of them can be used as a parallel program representation for runtime scheduling (because they are not retained after static translation to native code, which is a non-trivial design challenge).

Framework for Approximate Computing. To deal with heterogeneity of approximation methods, existing systems for accuracy-aware optimizations do not provide a fully automated framework that is able to target multiple heterogeneous devices without requiring programmer-guided annotations. The ACCEPT [36] framework uses a programmer-guided approach with source-level approximation annotations. It targets accelerators using a combination of static analysis and autotuning to choose between multiple approximation options. However, ACCEPT cannot support systems with diverse compute units which is a key goal of our work. Also, it composes errors linearly which is quite arbitrary because how errors compose would depend on the functions involved. Valid configurations where after linearly composing errors, the end-to-end constraints are not met would be discarded. Our approach detailed in Chapter 5 does not assume that errors compose linearly and executes the program to see if the end-to-end accuracy threshold is met and thus, does not unnecessarily discard valid configurations. Moreover, ACCEPT does not decouple hardware-independent accuracy constraints from hardware-specific knobs, thereby reducing the extensibility of the technique to multiple kinds of hardware devices. EnerJ [37] presents a type system that separates approximate and precise data. Chisel [38] and Rely [39] programming languages introduced the idea of quantifiable reliability and accuracy at the program level. Introducing approximation metrics as part of a new programming language hurts program portability since applications need to be ported at the source-level.

1.4 CONTRIBUTIONS

Our contributions in this work can be broken down into three logical components, which also follows the chronological order in which we broke down and devised solution for the programmability and portability challenges of heterogeneous systems.

First, we aim to address the programmability challenges of heterogeneous systems where underlying compute units have different ISAs, different memory hierarchy and exploit different parallelism models. Section 1.4.1 summarizes our approach and contributions in addressing the diversity in parallelism in heterogeneous systems. This work was done jointly (and equally) with Maria Kotsifakou (kotsifa2@illinois.edu) and would appear in both of our theses.

Second, we did a joint project with Professor Naresh Shanbhag's research group [40] (especially his student Mingu Kang) to design a programmable mixed signal accelerator for machine learning algorithms. This project described in Section 1.4.2 helped in getting useful insight into the diversity of new accelerators that can be part of a heterogeneous SoC in future. There were three logical components to this projects: (1) modifications to the existing hardware which was led by Mingu Kang, (2) designing the analog ISA for the accelerator which was done jointly by Mingu and I, and (3) the compiler for this accelerator which I designed and implemented.

Lastly, we extend our work to tackle the heterogeneity of approximation techniques in heterogeneous systems. We summarize our approach in Section 1.4.3. Hashim Sharif (hsharif3@illinois.edu) contributed equally to this project and this would appear in his thesis as well.

1.4.1 Heterogeneous Parallel Systems

We believe that the programmability and portability challenges can be best addressed by developing *a single parallel program representation flexible enough to support at least three different purposes*: (1) A *compiler intermediate representation*, for compiler optimizations and code generation for diverse heterogeneous hardware. Such a compiler IR must be able to implement a wide range of different parallel languages, including general-purpose ones like OpenMP, CUDA and OpenCL, and domain-specific ones like Halide and TensorFlow. (2) A *virtual ISA*, to allow virtual object code to be shipped and then translated down to native code for different heterogeneous system configurations. This requirement is essential to enable application teams to develop and ship application code for multiple devices within a family. (3) A *representation for runtime scheduling*, to enable flexible mapping and load-balancing policies, in order to accommodate static variations among different compute kernels and dynamic variations due to external effects like energy fluctuations or job arrivals and departures. We believe that a representation that can support all these three capabilities could (in future) also simplify parallel algorithm development and influence parallel language design, although we do not explore those in this work.

In this work, we propose such a parallel program representation, *Heterogeneous Parallel Virtual Machine* (HPVM), and evaluate it for three classes of parallel hardware: GPUs, SIMD vector instructions, and multicore CPUs. Our evaluation shows that HPVM can serve all three purposes listed above: a compiler IR, a virtual ISA, and a scheduling representation, as described below.

The parallel program representation we propose is a *hierarchical dataflow graph with shared*

memory. The graph nodes can represent either coarse-grain or fine-grain computational tasks, although we focus on moderately coarse-grain tasks (such as an entire inner-loop iteration) in this work. The dataflow graph edges capture explicit data transfers between nodes, while ordinary load and store instructions express implicit communication via shared memory. The graph is hierarchical because a node may contain another dataflow graph. The leaf nodes can contain both scalar and vector computations. A graph node represents a static computation, and any such node can be "instantiated" in a rectangular grid of dynamic node instances, representing *independent* parallel instances of the computation (in which case, the incident edges are instantiated as well, as described later).

The hierarchical dataflow graphs naturally capture all the important kinds of coarse- and fine-grain data and task parallelism in heterogeneous systems. In particular, the graph structure captures coarse-grain task parallelism (including pipelined parallelism in streaming computations); the graph hierarchy captures multiple levels and granularities of nested parallelism; the node instantiation mechanism captures either coarse- or fine-grain SPMD-style data parallelism; and explicit vector instructions within leaf nodes capture fine-grain vector parallelism (this can also be generated by automatic vectorization across independent node instances).

We describe a prototype system (also called HPVM) that supports all three capabilities listed earlier. The system defines a compiler IR as an extension of the LLVM IR [41] by adding HPVM abstractions as a higher-level layer describing the parallel structure of a program.

As examples of the use of HPVM as a compiler IR, we have implemented two illustrative compiler optimizations, *graph node fusion* and *tiling*, both of which operate directly on the HPVM dataflow graphs. Node fusion achieves "kernel fusion", and the graph structure makes it explicit when it is safe to fuse two or more nodes. Similarly (and somewhat surprisingly), we find that the graph hierarchy is also an effective and *portable* method to capture tiling of computations, which can be mapped either to a cache hierarchy or to explicit local memories such as the scratchpads in a GPU.

To show the use of HPVM as a virtual ISA, we implemented *translators* for NVIDIA GPUs (using PTX), Intel's AVX vector instructions, and multicore X86-64 host processors using Posix threads. The system can translate *each HPVM graph node* to one or more of these distinct target architectures (e.g., a 6-node pipeline can generate $3^6 = 729$ distinct code configurations from a single HPVM version). Experimental comparisons against hand-coded OpenCL programs compiled with native (commercial) OpenCL compilers show that the code generated by HPVM is within 22% of hand-tuned OpenCL on a GPU (in fact, nearly identical in all but one case), and within 7% of the hand-tuned OpenCL in all but

one case on AVX. We expect the results to improve considerably by further implementation effort and tuning.

Finally, to show the use of HPVM as a basis for runtime scheduling, we developed a graph-based scheduling framework that can apply a wide range of static and dynamic scheduling policies that take full advantage of the ability to generate different versions of code for each node. Although developing effective scheduling policies is outside the scope of this work, our experiments show that HPVM enables flexible scheduling policies that can take advantage of a wide range of static and dynamic information, and these policies are easy to implement directly on the HPVM representation.

1.4.2 Programmable Machine Learning Accelerator: PROMISE

The goal of this project in the context of this thesis was to gain necessary insights into the diversity of programmable accelerators that can be part of a heterogeneous SoC in future. The project itself was motivated by the computation and large data processing needs of machine learning (ML) algorithms as they begin to perform better than humans [42] in cognitive and decision-making tasks [43]. We closely collaborated with the team behind Compute Memory [10] to work on a programmable mixed signal accelerator for ML algorithms based on deep in-memory computing.

We proposed PROMISE, the first end-to-end design of a programmable mixed-signal accelerator for diverse ML algorithms PROMISE can accomplish a high level of programmability without noticeably losing the efficiency of mixed-signal accelerators for specific ML algorithms. PROMISE exposes instruction set mechanisms that allow software control over energy-vs-accuracy tradeoffs, and supports compilation of high-level languages down to the hardware. Specifically, we make following key contributions.

PROMISE Architecture and ISA: First, we identify prevalent operations in widely-used ML algorithms and key constraints in supporting these operations for a programmable mixed-signal accelerator. These include (**C1**) intrinsic sequentiality imposed on operations and (**C2**) high variations in delay across different types of operations. **C1** limits the number of possible programmable operations and **C2** significantly affects performance and energy efficiency of mixed-signal accelerators. Second, we explore PROMISE Instruction Set Architecture (ISA), which can expose these operations and constraints to a compiler, with a programmable mixed-signal accelerator architecture built with silicon-proven components for mixed-signal operations [10]. The hardware design and ISA include mechanisms to control the accuracy-vs-energy tradeoff by varying swing voltages, which can be controlled by compiler-generated code.

Compiler for PROMISE: First, we discuss design goals for a compiler when PROMISE aims to maximize energy efficiency while delivering a desired accuracy for a given ML algorithm. Particularly, the following two aspects of PROMISE explode the solution space to explore when generating code. (**A1**) PROMISE demands software to determine the accuracy of each mixed-signal instruction in given code, while a complex interplay among accuracy of instructions strongly affects energy efficiency and overall final accuracy of the code. (**A2**) PROMISE provides many possible compositions of mixed-signal operations even under the **(C1)** and **(C2)** constraints. Considering the large solution space, we reason that it is inherently impractical for users to manually generate code for a complex ML algorithm. Second, we develop a code generator that translates machine learning kernels down to sequences of mixed-signal operations. Third, for neural network algorithms in particular, we show how to automatically map programmer-specified end-to-end accuracy error tolerances down to hardware-level swing voltage parameter values for individual PROMISE instruction (called PROMISE `Task`). A direct mapping is difficult, but we show how to break down the problem into two steps: (i) mapping error tolerance to computational bit precision (using an existing analysis [44]), and (ii) mapping bit precision to voltage swings (using simulation-based profiling). Putting these together, we develop a compiler which can take a broad range of ML algorithms described in a high-level programming language (Julia) and compile it to native PROMISE code. For neural network algorithms, we can translate a specified end-to-end error tolerance and compile it to PROMISE code that satisfies the error tolerance while approximately minimizing voltage swings (within available quantization levels).

Efficacy of PROMISE End-to-End Design: To demonstrate the efficacy of PROMISE, we first build energy and throughput models of PROMISE based on silicon-validated energy, delay and behavioral models of mixed-signal blocks. Second, we take nine popular ML algorithms, describe them in Julia, and generate the code with the PROMISE compiler. Our evaluation shows that PROMISE can offer $3.4 - 5.5\times$ lower energy and $1.4 - 3.4\times$ higher throughput than algorithm-specific digital accelerators at comparable inference accuracy. Lastly, the swing voltage optimized code by the PROMISE compiler further provides $4\% - 20\%$ (geometric mean: 15%) lower energy than the unoptimized code for complex ML kernels.

1.4.3 Portable Approximate Computing

We propose ApproxHPVM (an extension of HPVM) a unified compiler IR and framework that enables a program with application-level end-to-end error tolerance constraints to be optimized and scheduled on a heterogeneous system containing multiple approximation techniques. The ApproxHPVM system takes as input a program compiled to ApproxH-

PVM, and end-to-end constraints that quantify the acceptable difference between outputs with no approximation and approximate results. 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 all these capabilities*, especially full automation and support for multiple heterogeneous compute units. Specifically, ApproxHPVM makes the following contributions:

Decompose end-to-end error specification: For applications with multiple approximable computations, it translates end-to-end (application-level) error specifications to individual error specifications and bounds per approximable computation. For instance, a user may specify an acceptable classification accuracy degradation of 2%, allowing the tuner to reduce the accuracy requirements for an individual matrix add operation by up to 20%. In this way, the tuner facilitates the hardware scheduler in mapping error-tolerant operations to approximate hardware that provides improved performance while satisfying the end-to-end accuracy requirement.

Mapping to heterogeneous hardware with multiple approximate computing methods: 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.

Decouple autotuning into hardware-agnostic and hardware dependent stage: It greatly reduces the cost of mapping to a given heterogeneous system providing multiple approximation techniques by decoupling the mapping problem into a somewhat expensive development-time hardware-independent autotuning stage and a fast hardware-dependent stage. Such a strategy provides multiple benefits including a) shipping portable code that can be mapped to different heterogeneous hardware configurations, b) ability to dynamically schedule operations based on hardware-independent accuracy constraints, c) enabling hardware design space exploration to incorporate approximation information via fast decision algorithms.

Evaluation on Target Platform: To show the efficacy of ApproxHPVM, we envision a mobile SoC with CPUs, a GPU, and an analog accelerator called PROMISE [18] using a hybrid hardware simulator-real hardware approach. PROMISE is a mixed-signal machine learning accelerator which employs imprecise analog computation to get orders of magnitude energy and throughput benefits for large-vector reduction operations. PROMISE is particularly well-suited for large matrix multiplications, which are commonly-used kernels

in deep learning applications. Our system is built as an extension of a mobile SoC platform, NVIDIA Jetson TX2 [45], which has 8 GB shared memory between ARM cores and an NVIDIA Pascal GPU. We extend the platform by adding a PROMISE timing simulator connected to the same physical shared memory. The platform provides 9 hardware settings to trade-off energy and accuracy. The GPU provides two settings (32-bit and 16-bit floating point computation) and PROMISE provides seven more settings. We evaluate four different deep neural networks (DNNs) on our platform. Our results show that by smart scheduling of DNN operations, ApproxHPVM provides performance speedups ranging from 2.8x to 20x and energy reduction ranging from 1.9x to 7.1x for an accuracy loss of 1%.

1.5 THESIS ORGANIZATION

The remaining chapters are organized as follows. Chapter 2 gives more details on the state of the art for each of the projects part of this thesis and compares our approach to them. Chapter 3 gives details of the HPVM IR and how it address the source code portability, object code portability and performance portability for current and future heterogeneous systems. It presents the design of HPVM IR and evaluates it to show how it can serve as a virtual ISA, compiler IR and a runtime representation that facilitates flexible scheduling. Chapter 4 presents the design and evaluation of PROMISE accelerator, the design of PROMISE ISA and compiler for PROMISE. Chapter 5 proposes extensions to HPVM that help address the challenges due to heterogeneity in approximation methods. Finally, Chapter 6 summarizes the contributions and discusses some directions for future work.

CHAPTER 2: RELATED WORK

This chapter gives context to this thesis by giving a literature review of state of the art work in programming languages, compiler infrastructures, and runtimes aimed at addressing the programmability challenges of heterogeneous computing. Apart from these, I have also added the state of the art work in programmable mixed signal accelerators. This would give context to our machine learning accelerator PROMISE.

2.1 HPVM

There is a long history of work on dataflow execution models, programming languages, and compiler systems for homogeneous parallel systems [46, 47, 48, 49, 35, 50, 51, 52]. HPVM aims to adapt the dataflow model to modern heterogeneous parallel hardware. We focus below on programming technologies for heterogeneous systems.

Virtual ISAs: NVIDIA's PTX virtual ISA provides portability across NVIDIA GPUs of different sizes and generations. HSAIL [53] and SPIR [54] both provide a portable object code distribution format for a wider class of heterogeneous systems, including GPUs, vectors and CPUs. All these systems implement a model that can be described as a "grid of kernel functions," which captures individual parallel loop nests well, but more complex parallel structures (such as the 6-node pipeline DAG used in our Edge Detection example) are only expressed via explicit, low-level data movement and kernel coordination. This makes the underlying model unsuitable for use as a retargetable compiler IR, or for flexible runtime scheduling. Finally, it is difficult, at best, to express some important kinds of parallelism, such as pipelined parallelism (important for streaming applications) : they must be written explicitly using programmer-defined buffer management and enqueuing an explicit "event" for synchronizing every data transfer between every pair of pipeline stages. An event represents a single dependency between two operations, thus a different event would need to be used per stage per data item. and expressing the dependency between pipeline stages. This detailed low-level operations for expressing pipelined parallelism in PTX or HSAIL or SPIR are highly prescriptive and remove flexibility for optimizations from the compiler and run-time, whereas HPVM allows the programmer to simply express the goal (pipelining) and leave the details of implementation to the system. In contrast, HPVM allows the programmer to simply express the goal (pipelining) and leave the details of implementation to the system.

Compiler IRs with Explicit Parallel Representations: We focus on parallel compil-

ers for heterogeneous systems. The closest relevant compiler work is OSCAR [31, 55, 32], which uses a hierarchical task dependence graph as a parallel program representation for their compiler IR. They do *not* use this representation as a virtual ISA, which means they cannot provide object code portability. Their graph edges represent data and control dependences, *not dataflow* (despite the name), which is well suited to shared memory systems but not as informative for non-shared memory. In particular, for explicit data transfers, the compiler must infer automatically what data must be moved (e.g., host memory to accelerator memory). They use hierarchical graphs only for the (homogeneous) host processors, not for accelerators, because they do not aim to perform parallel compiler transformations for code running within an accelerator nor runtime scheduling choices for such code. KIMBLE [56, 33] adds a hierarchical parallel program representation to GCC, while SPIRE [34] defines a methodology for sequential to parallel IR extension. Neither KIMBLE nor SPIRE make any claim to, or give evidence of, performance portability or parallel object code portability.

Runtime Libraries for Heterogeneous Systems: Parallel Virtual Machine (PVM) [57] enables a network of diverse machines to be used cooperatively for parallel computation. An application in the PVM model is a collection of tasks (akin to a Unix process) that use an explicit, highly portable message passing library for communication. Despite the similarity in the names, the systems have different goals. What is virtualized in PVM are the application programming interfaces for task management and communication across diverse operating systems, to achieve portability and performance across homogeneous parallel systems. HPVM virtualizes the parallel execution behavior and the parallel hardware ISAs, to enable portability and performance across heterogeneous parallel hardware, including GPUs, vector hardware, FPGAs [1] and potentially domain-specific accelerators.

Several other runtime systems [58, 59, 60, 61] support scheduling and executing parallel programs on heterogeneous parallel systems.

Programming Languages: Source-level languages such as CUDA, [62] OpenCL, [63] OpenACC, [64] and OpenMP [25], all support a similar programming model that maps well to GPUs and vector parallelism. None of them, however, address object code portability and none can serve as a parallel compiler IR. They also make it difficult to express important kinds of parallelism, like pipelined parallelism as explained earlier for PTX, SPIR and HSAIL.

PENCIL [65] is a programming language defined as a restricted subset of C99, intended as an implementation language for libraries and a compilation target for DSLs. Its compiler

[1] My colleague Adel Ejjeh (aejjeh@illinois.edu) is working on efficient code generation on FPGAs using HPVM

uses the polyhedral model to optimize code and is combined with an auto-tuning framework. It shares the goals of source code portability and performance portability with HPVM. However, it is designed as a readable language with high-level optimization directives rather than as a compiler IR, per se, and it also does not address object code portability.

StreamIt [66] and CnC [67] are programming languages with a somewhat more general representation for streaming pipelines. StreamIt streams are more general, for example, allowing cycles via its *FeedbackLoop* mechanism, whereas HPVM does not. In future, it can be explored how to map StreamIt to HPVM. Most other key StreamIt features such as filters, *SplitJoin* and *Pipeline* can be naturally expressed in HPVM. They, however, focus on stream parallelism, whereas HPVM supports both streaming and non-streaming parallelism. This is crucial when defining a compiler IR or a virtual ISA for parallel systems (of any kind), because most parallel languages (e.g., OpenMP, OpenCL, CUDA, Chapel, etc.) are used for non-streaming parallel programs.

Frameworks: Habanero-Java [68] and Habanero-C [69], provide an abstraction of heterogeneous systems called *Hierarchical Place Trees (HPT)*, that supports co-location of data and computation at multiple levels of a memory hierarchy to support different communication mechanisms across memory hierarchies. HPT is aimed at handling the challenge of diversity of memory hierarchies in heterogeneous systems. However, the challenges of diversity in instruction sets of compute units are not addressed and thus do not provide object code portability.

Legion [70, 71] is a programming model and runtime system for writing high performance applications for distributed heterogeneous architectures. It focuses on the placement and partitioning of program data in the complex memory hierarchies of such systems where cost of data movement dominates the cost of computation. It provides abstractions for describing the structure of program data in a machine independent way including organization, partitioning, privileges, and coherence. In the Legion programming model, every Legion program executes as a tree of tasks with a top-level task spawning sub-tasks which can recursively spawn further sub-tasks. All tasks in Legion must specify the logical regions they will access as well as the privileges and coherence for each logical region. However, broadly it focuses on addressing the challenge of data layout for heterogeneous architectures with deep memory hierarchies and not on different parallelism models or object code portability. It does not address the portability of leaf tasks and relies on the programmer to provide multiple version of leaf tasks for each processor kind. Also, it does not support partitioning or fusion of tasks to adapt to different type of compute units. Lastly, in its current form, it focuses on systems built using CPUs and GPUs, and not DSPs, or semi-custom programmable accelerators.

Similar to Legion, Sequoia [72] provides rich memory abstractions to enable explicit control over movement and placement of data at all levels of a heterogeneous memory hierarchy.

HPVM lacks these memory related features, but does express tiling effectively and portably using the hierarchical graphs. In future, we aim to add richer memory abstractions to HPVM.

Petabricks [27] explores the search space of different algorithm choices and how they map to CPU and GPU processors. In Tangram [73], a program is written in interchangeable, composable building blocks, enabling architecture-specific algorithm and implementation selection. Exploring algorithm choices is orthogonal to, and can be combined with, our approach.

Delite [74] is a framework for developing compiled, embedded DSLs inside the programming language Scala. The Delite execution graph encodes the dependencies between computations. One important characteristic of Delite is that IR nodes are simultaneously viewed in multiple ways: a generic layer, a parallel layer, called Delite ops, and a domain-specific layer, which is the actual operation, for example, vector reduction. The IR can be optimized by viewing nodes at any layer, enabling both generic and domain-specific optimizations to be expressed on the same IR. To provide flexibility to run new ops on different hardware devices, Delite relies on the DSL developers to provide Scala, CUDA, OpenCL implementations of these computations as necessary for efficiency. HPVM on the other hand relies on the hardware vendors to provide platform specific implementation of new ops/parallel patterns in HPVM IR. The broader Delite approach can be combined with HPVM approach to ease burden on the DSL developers. The feature of simultaneous multi-level view of Delite IR helps it perform both generic and domain specific optimizations on the same IR. We think something similar can benefit HPVM as well, in future.

2.2 APPROXHPVM

Software Approximations: Many studies have introduced novel software techniques for approximation that reduce execution time and/or energy. The transformations include task skipping [75, 76, 77], loop perforation [78, 21, 79], approximate function substitution [80, 81, 23, 82], dynamic knobs [83] (dynamically changing function version), reduction sampling [84, 82], tuning floating-point operations [85, 86], and approximate parallelization [87, 88, 82, 89]. 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 as many of these optimizations as possible (we leave the goal of covering all software or hardware approximation methods as future work), 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 [37] presents a type system that separates approximate and precise data. Chisel [38] and Rely [39] introduced the idea of quantifiable reliability and accuracy at the program level. Specifically, Rely enables an application developer to manually identify unreliable instructions and variables that can be stored in unreliable memories. In other words, the developer can specify what instructions are approximable and which variables are approximable. The developer also provides a reliability specification, which identifies the minimum required probability with which the kernel must produce a correct result. Given this specification and the exact kernel computation, the Chisel analysis then automatically navigates the trade-off space between energy savings and reliability and provides an approximate computation that minimizes the energy savings while satisfying the reliability and accuracy 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 and the approximation becomes a first-class citizen in a compiler workflow, which is able to interact with various front-end languages *and* hardware-specific features. Also, the accuracy specification we use is user defined end-to-end application accuracy. It is flexible and can be different for different applications. For the machine learning classification benchmarks we use in this thesis, we use the degradation in the classification accuracy of the benchmark on a dataset as the metric. In ApproxHPVM the frontend or developer would specify which IR nodes (macro computations) are approximable. Our hardware agnostic analysis only adds errors to the outputs of such nodes. Other details such as approximable instructions, variables and memories, are specific to the underlying approximation method and hardware and have been abstracted away in ApproxHPVM IR for portability. Lastly, ApproxHPVM decouples autotuning into hardware-agnostic and hardware dependent stages unlike any other work that we know of in this field.

Approximate Hardware Accelerators: Recently there have been many proposals for machine learning accelerators [90, 91, 92, 93, 94, 19], some of which explicitly incorporate approximations. Although we have chosen PROMISE and GPUs as the accelerators in our evaluations, our approach of decoupling autotuning into hardware-agnostic and hardware dependent approach makes ApproxHPVM easily extensible to adapt to these accelerators as well. The hardware-independent metrics would remain the same. Similar to how we created a look-up table to map L-norm metrics to PROMISE and GPU configurations, we would

need to create a mapping for these accelerators as well.

Systems for Automated Accuracy-tuning: ACCEPT [36] used source-level approximation annotations for automatically offloading computations to approximate hardware. Like our work, they target accelerators, they use a combination of static and dynamic analysis, and they use autotuning to choose between multiple approximation options. We differ from ACCEPT in several ways. They require programmer involvement in an interactive optimization loop for greater programmer control, whereas ApproxHPVM is fully automatic. They cannot support systems with multiple compute units, which is a key goal of our work (in addition to multiple approximation techniques). ACCEPT composes errors linearly which is quite arbitrary because how errors compose depends on the functions involved. This would lead to valid configurations being discarded where after linearly composing errors, the end-to-end constraints are not met. Our approach does not assume that errors compose linearly and executes the program to see if the end-to-end accuracy threshold is met and thus, does not unnecessarily discard valid configurations. Lastly, they do not decouple hardware-independent from hardware-dependent tuning, which enables us to achieve *much* faster target-specific tuning times and portable object code, and enables future uses like dynamic scheduling and design space exploration.

The Petabricks programming language automatically autotunes the choice of an algorithm among multiple user-provided choices with varying accuracy and performance characteristics [95, 81]. Autotuning algorithmic choice could be incorporated in ApproxHPVM as simply another approximation mechanism. For example, different algorithms for specific IR-level operations could be mapped to different accuracy levels in the back end.

Approximation techniques for deep neural networks, e.g., fixed point quantization of pre-trained neural networks [96] or layer-grained analytical models for bit-widths of weights and activations [97], are done using highly 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.

Existing general tools, e.g. [85] tune the floating point precision of numerical computations by using a domain-neutral search derivations. While these systems are only concerned with tuning computational precision, ApproxHPVM includes more generic domain-neutral search with approximation metrics tune floating point precision together with other transformations provided by software or hardware.

2.3 HARDWARE ACCELERATORS FOR MACHINE LEARNING

Here we discuss related work in relation to the programmable analog accelerator project, PROMISE. As mentioned in Section 1.4.2 and discussed in detail in Chapter 4 the accelerator brings the benefits of deep in-memory computing and analog computing to a wide range of machine learning algorithms through PROMISE and PROMISE ISA. Here we compare it against other recent Machine Learning accelerators with/without instruction set (ISA).

2.3.1 Accelerators without ISA

The bandwidth of processor-memory interfaces is a longstanding problem in high-performance computing. Machine learning applications have greatly aggravated the problem, where memory access has begun to dominate the overall energy consumption. There have been proposal for digital ML accelerators such as Eyeriss [8] to reduce memory access energy and latency by exploiting the dataflow of convolution neural networks. Such accelerators give significant benefits over more general purpose processors like CPUs and GPUs, but are algorithm specific (Eyeriss is specific to convolution neural networks).

Many analog accelerators have also been proposed. RedEye [98] performs the initial convolution layer computations in analog domain before going to digital domain. RedEye [98] performs the initial convolution layer computations in analog domain before going to digital domain. [99, 100] employed Adaptive Boosting algorithm for image recognition processing raw analog signals from sensors. [101] suggested mixed-signal matrix multiplier via switched-capacitor. Such papers exploit the efficient analog calculation for large data processing at slightly compromised accuracy. However, they are *not* programmable and do not cover diverse ML algorithms in general.

There have also been efforts to integrate computation near memory with emerging technologies such as resistive RAM (RRAM) [102], and a Memristor crossbar based CNN accelerator [103]. These accelerators too are not programmable and limited to neural network algorithm.

2.3.2 Accelerators with ISA

Cambricon [104] and PuDianNao [105] aim to cater to a wider range of ML algorithms and have a instruction set (ISA) for the digital ML accelerator they propose. The Dian-Nao family of deep learning processors demonstrate approximately 100× improvement in both energy and speed-up compared to a GPU. PROMISE [18] employs an alternate approach of

using Compute Memory (CM) [106] technology to get even more energy benefits while at the same time keeping it programmable and applicable to wider set of ML algorithms.

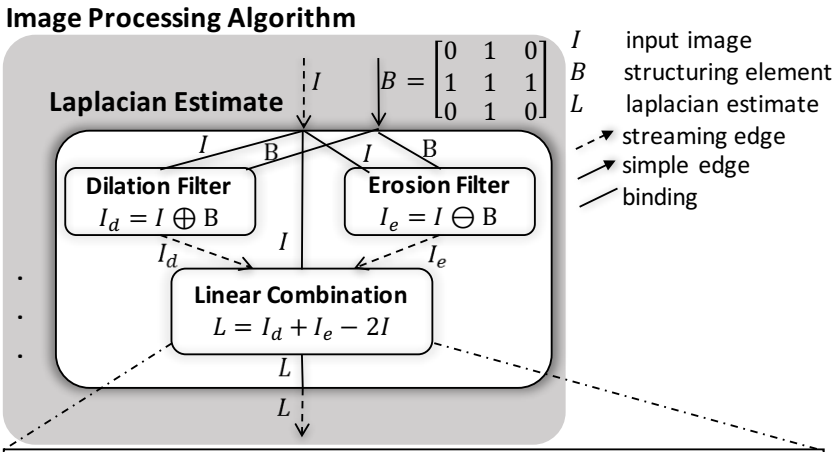
CHAPTER 3: HPVM: HETEROGENEOUS PARALLEL VIRTUAL MACHINE

We[1] studied the programmability and portability challenges of heterogeneous systems and proposed a parallel program representation designed to enable performance portability across a wide range of popular hardware, including GPUs, vector instruction sets, multicore CPUs and potentially FPGAs. The representation, which we call HPVM, is a hierarchical dataflow graph with shared memory and vector instructions. HPVM supports three important capabilities for programming heterogeneous systems: a compiler intermediate representation (IR), a virtual instruction set (ISA), and a basis for runtime scheduling; previous systems focus on only one of these capabilities. As a compiler IR, HPVM aims to enable effective code generation and optimization for heterogeneous systems. As a virtual ISA, it can be used to ship executable programs, in order to achieve both functional portability and performance portability across such systems. At runtime, HPVM enables flexible scheduling policies, both through the graph structure and the ability to compile individual nodes in a program to any of the target devices on a system. We have implemented a prototype HPVM system, defining the HPVM IR as an extension of the LLVM compiler IR, compiler optimizations that operate directly on HPVM graphs, and code generators that translate the virtual ISA to NVIDIA GPUs, Intel's AVX vector units, and to multicore X86-64 processors. Experimental results show that HPVM optimizations achieve significant performance improvements, HPVM translators achieve performance competitive with manually developed OpenCL code for both GPUs and vector hardware, and that runtime scheduling policies can make use of both program and runtime information to exploit the flexible compilation capabilities. Overall, we conclude that the HPVM representation is a promising basis for achieving performance portability and for implementing parallelizing compilers for heterogeneous parallel systems.

3.1 HPVM PARALLELISM ABSTRACTIONS

This section describes the Heterogeneous Parallel Virtual Machine parallel program representation. The next section describes a specific realization of Heterogeneous Parallel Virtual Machine on top of the LLVM compiler IR.

[1]**This work was done jointly (and equally) with Maria Kotsifakou (kotsifa2@illinois.edu) and would appear in both of our theses.**




```
define %struct.l @lincomb(double* I, double* Ie, double* Id) {
  %N = call i8* @llvm.hpvm.getNode()
  %nidx = call i32 @llvm.hpvm.getNodeInstanceID.x(i8* %N)
  %nidy = call i32 @llvm.hpvm.getNodeInstanceID.y(i8* %N)
  %vl = call i32 @llvm.hpvm.getVectorLength(i32 8)
  ; Index and base address calculation using %nidx, %nidy
  ; for I, Id, Ie, L (not shown)
  %pixel_I = load <%vl x double>* I_base
  %pixel_Id = load <%vl x double>* Id_base
  %pixel_Ie = load <%vl x double>* Ie_base
  %tmp = insertelement <%vl x double> undef,
                       double 2.0, i32 0
  %vec2 = shufflevector <%vl x double> %tmp,
                        <%vl x double> undef,
                        <%vl x i32> zeroinitializer
  %mul = mul <%vl x double> %vec2, %pixel_I
  %add = add <%vl x double> %pixel_Id, %pixel_Ie
  %res = sub <%vl x double> %add, %mul
  store <%vl x double> %res, <%vl x double>* L_base
  %out = insertvalue %struct.l undef, double* L, 0
  ret %struct.l %out
}
```

Figure 3.1: Non-linear Laplacian computation in HPVM

3.1.1 Dataflow Graph

In HPVM, a program is represented as a host program plus a set of one or more distinct dataflow graphs. Each dataflow graph (DFG) is a hierarchical graph with side effects. Nodes represent units of execution, and edges between nodes describe the explicit data transfer requirements. A node can begin execution once it receives a *data item* on every one of its input edges. Thus, repeated transfer of data items between nodes (if overlapped) yields a pipelined execution of different nodes in the graph. The execution of the pipeline is initiated and terminated by host code that launches the graph. For example, this mechanism can be used for streaming computations on data streams, e.g., processing successive frames in a video stream.

Nodes may access globally shared memory through load and store instructions ("side-effects"), since hardware shared memory is increasingly common across heterogeneous systems. These operations may result in *implicit* data transfers, depending on the mapping of nodes to hardware compute units and on the underlying memory system. Because of these side effects, HPVM is not a "pure dataflow" model. Figure 3.1 shows the HPVM version of a Laplacian estimate computation of a greyscale image, used as part of image processing filters. This will be used as a running example. The figure shows the components of the Laplacian as separate dataflow nodes – `Dilation Filter (DF)`, `Erosion Filter (EF)` and `Linear Combination (LC)` – connected by edges. The figure also shows the code for node `LC`, which is standard LLVM IR except for the new intrinsic functions named Load/store instructions access shared memory, using pointers that must be received explicitly from preceding nodes.

Dynamic Instances of Nodes and Edges: The dataflow graphs in HPVM can describe varying (data-dependent) *degrees of parallelism* at each node. In particular, a single static dataflow node or edge represents multiple dynamic instances of the node or edge, each executing the same code with different index values. The dynamic instances of a node are required to be independent of each other, so that each time a (static) node is executed on receiving a new set of data items, all dynamic instances of the node may execute in parallel, similar to invoking a parallel loop. The dynamic instances form an n-dimensional grid, with integer indexes in functions. Our implementation allows up to three dimensions. For example, the `LC` node in the example is replicated to have $dim_X \times dim_Y$ instances, where dim_X and dim_Y are computed at runtime. Similarly, a static edge between two static nodes may represent multiple dynamic edges between dynamic instances of the two nodes, as explained further in Section 3.1.1.

Dataflow Node Hierarchy: Each dataflow node in a DFG can either be a *leaf node* or an *internal node*. An internal node contains a complete dataflow graph, called a *child graph*, and the child graph itself can have internal nodes and leaf nodes. In Figure 3.1, the node `Laplacian Estimate` is an internal node, and its child graph comprises the leaf nodes `DF`, `EF`, and `LC`.

A leaf node contains scalar and/or vector code, expressing actual computations. Dynamic instances of a leaf node capture independent computations, and a single instance of a leaf node can contain fine-grain vector parallelism. Leaf nodes may contain instructions to query the structure of the underlying dataflow graph, as described in Section 3.2.

Internal nodes describe the structure of the child graph. The internal nodes are traversed by the translators to construct a static graph and generate code for the leaf nodes and edges (Section 3.3). One restriction of this model is that the dataflow graph cannot be modified

at runtime, e.g., by data-dependent code, dynamically spawning new nodes; this enables fully-static optimization and code generation at the cost of some expressivity.

The grouping and hierarchy of parallelism has several advantages. It helps express several different kinds of parallelism in a compact and intuitive manner: coarse-grain task (i.e., pipelined) parallelism via top-level nodes connected using dataflow edges; independent coarse- or fine-grained data parallelism via dynamic instances of a single static node; and fine-grained data parallelism via vector instructions within single instances of leaf nodes. It provides a flexible and powerful mechanism to express tiling of computations for memory hierarchy in a portable manner (Section 3.5.3). It enables efficient scheduling of the execution of the dataflow graph by grouping together appropriate sets of dataflow nodes. For example, a runtime scheduler could choose to map a single top-level (internal) node to a GPU or to one core of a multicore CPU, instead of having to manage potentially large numbers of finer-grain nodes. Finally, it supports a high-degree of modularity by allowing separate compilation of parallel components, represented as individual dataflow graphs that can be composed into larger programs.

Hardware feature	**Typical HPVM representation**
Heterogeneous multiprocessor system	
Major hardware compute units, e.g., CPU cores, GPUs	Top-level nodes in the DFG and edges between them
GPUs	
GPU Threads	DFG leaf nodes
GPU Thread Blocks	Parent nodes of DFG leaf nodes
Grid of Thread Blocks (SMs)	Either same as GPU Thread Blocks or parent node of DFGs representing thread blocks
GPU registers, private memory	Virtual registers in LLVM code for leaf nodes
GPU Scratch-pad Memory	Memory allocated in DFG internal nodes representing thread blocks
GPU Global Memory and GPU Constant Memory	Other memory accessed via loads and stores in DFG leaf nodes
Short-vector SIMD instructions	
Vector instructions with independent operations	Dynamic instances of first-level internal nodes, and/or Vector code in leaf nodes
Vector instructions with cross-lane dependences	Vector code in leaf nodes
Vector registers	Virtual registers in LLVM code for leaf nodes
Homogeneous host multiprocessor	
CPU threads in a shared-memory multiprocessor	One or more nodes in one or more DFGs
Shared memory	Memory accessed via loads and stores in DFG leaf nodes. HPVM intrinsics for synchronization.

Table 3.1: How HPVM represents major parallel hardware features

Dataflow Edges and Bindings: Explicit data movement between nodes is expressed with dataflow edges. A dataflow edge has the semantics of copying specified data from the source to the sink dataflow node, after the source node has completed execution. Depending on where the source and sink nodes are mapped, the dataflow edge may be translated down to an explicit copy between compute units, or communication through shared memory, or simply a local pointer-passing operation.

As with dataflow nodes, static dataflow edges also represent multiple dynamic instances of dataflow edges between the dynamic instances of the source and the sink dataflow nodes. An edge can be instantiated at runtime using one of two simple replication mechanisms: "all-to-all", where all dynamic instances of the source node are connected with all dynamic instances of the sink node, thus expressing a synchronization barrier between the two groups of nodes, or "one-to-one" where each dynamic instance of the source dataflow node is connected with a single corresponding instance of the sink node. One-to-one replication requires that the grid structure (number of dimensions and the extents in each dimension) of the source and sink nodes be identical.

Figure 3.1 shows the dataflow edges describing the data movement of input image I, dilated image I_d, eroded image I_e, and matrix B between dataflow nodes. Some edges (e.g., input B to node `Laplacian Estimate`) are "*fixed*" edges: their semantics is as if they repeatedly transfer the same data for each node execution. In practice, they are treated as a constant across node executions, which avoids unnecessary data transfers (after the first execution on a device).

In an internal node, the incoming edges may provide the inputs to one or more nodes of the child graph, and conversely with the outgoing edges, such as the inputs I and B and output L of node `Laplacian Estimate`. Semantically, these represent *bindings* of input and output values and not data movement. We show these as undirected edges.

3.1.2 Vector Instructions

The leaf nodes of a dataflow graph can contain explicit vector instructions, in addition to scalar code. We allow parametric vector lengths to enable better performance portability, i.e., more efficient execution of the same HPVM code on various vector hardware. The vector length for a relevant vector type need not be a fixed constant in the HPVM code, but it must be a *translation-time constant* for a given vector hardware target. This means that the parametric vector types simply get lowered to regular, fixed-size vector types during native code generation. Figure 3.1 shows an example of parametric vector length (`%vl`) computation and use.

Evaluating the effect of parametric vector lengths on performance is out of the scope of this work because we only support one vector target (Intel AVX) for now. Moreover, in all the benchmarks we evaluate, we find that vectorizing across dynamic instances of a leaf node is more effective than using explicit vector code, as explained in Sections 3.3.2 and 3.6.2. More complex vector benchmarks, however, may benefit from the explicit vector instructions.

3.1.3 Integration with Host Code

Each HPVM dataflow graph is "launched" by a host program, which can use `launch` and `wait` operations to initiate execution and block for completion of a dataflow graph. The graph execution is asynchronous, allowing the host program to run concurrently and also allowing multiple independent dataflow graphs to be launched concurrently. The host code initiates graph execution by passing initial data during the launch operation. It can then sustain a streaming graph computation by sending data to input graph nodes and receiving data from output nodes. The details are straightforward and are omitted here.

3.1.4 Discussion

An important consideration in the design of HPVM is to enable efficient mapping of code to key features of various target hardware. We focus on three kinds of parallel hardware in this work: GPUs, vectors, and multithreaded CPUs. Table 3.1 describes which HPVM code constructs are mapped to the key features of these three hardware families. This mapping is the role of the translators described in Section 3.3. The table is a fairly comprehensive list of the major hardware features used by parallel computations, showing that HPVM is effective at capturing different kinds of hardware.

3.2 HPVM VIRTUAL ISA AND COMPILER IR

We have developed a prototype system, also called HPVM, including a Compiler IR, a Virtual ISA, an optimizing compiler, and a runtime scheduler, all based on the HPVM representation The compiler IR is an extension of the LLVM IR, defined via LLVM intrinsic functions, and supports both code generation (Section 3.3) and optimization (Section 3.5) for heterogeneous parallel systems. The virtual ISA is essentially just an external, fully executable, assembly language representation of the compiler IR.

We define new instructions for describing and querying the structure of the dataflow graph, for memory management and synchronization, as well as for initiating and terminating

Intrinsics for Describing Graphs	
`i8* llvm.hpvm.createNode1D`(Function* F, i32 n)	Create node with n dynamic instances executing node function F (similarly llvm.hpvm.createNode2D/3D)
`void llvm.hpvm.createEdge`(i8* Src, i8* Dst, i32 sp, i32 dp, i1 ReplType, i1 Stream)	Create edge from output *sp* of node Src to input *dp* of node Dst
`void llvm.hpvm.bind.input`(i8* N, i32 ip, i32 ic, i1 Stream)	Bind input *ip* of current node to input *ic* of child node N
`void llvm.hpvm.bind.output`(i8* N, i32 op, i32 oc, i1 Stream)	Bind output *oc* of child node `N` to output *op* of the current node
Intrinsics for Querying Graphs	
`i8* llvm.hpvm.getNode()`	Return a handle to the current dataflow node
`i8* llvm.hpvm.getParentNode(i8* N)`	Return a handle to the parent of node N
`i32 llvm.hpvm.getNodeInstanceID.[xyz](i8* N)`	Get index of current dynamic node instance of node N in dimension x, y or z
`i32 llvm.hpvm.getNumNodeInstances.[xyz](i8* N)`	Get number of dynamic instances of node N in dimension x, y or z
`i32 llvm.hpvm.getVectorLength(i32 typeSz)`	Get vector length in target compute unit for type size `typeSz`
Intrinsics for Memory Allocation and Synchronization	
`i8* llvm.hpvm.malloc(i32 nBytes)`	Allocate a block of memory of size `nBytes` and return pointer to it
`i32 llvm.hpvm.xchg(i32, i32), i32 llvm.hpvm.atomic.add`(i32*, i32), ...	Atomic-swap, atomic-fetch-and-add, etc., on shared memory locations
`void llvm.hpvm.barrier()`	*Local* synchronization barrier across dynamic instances of current leaf node

Table 3.2: Intrinsic functions used to implement the HPVM internal representation. `i`N is the N-bit integer type in LLVM.

execution of a graph. We express the new instructions as function calls to newly defined LLVM "*intrinsic functions*." These appear to existing LLVM passes simply as calls to unknown external functions, so no changes to existing passes are needed.

The intrinsic functions used to define the HPVM compiler IR and virtual ISA are shown in Table 3.2 (except host intrinsics for initiating and terminating graph execution). The code for each dataflow node is given as a separate LLVM function called the "*node function*," specified as function pointer `F` for intrinsics `llvm.hpvm.createNode[1D,2D,3D]`. The node function may call additional, "auxiliary" functions. The incoming dataflow edges and their data types are denoted by the parameters to the node function. The outgoing dataflow edges are represented by the return type of the node function, which must be an LLVM struct type with zero or more fields (one per outgoing edge). In order to manipulate or query information about graph nodes and edges, we represent nodes with opaque handles (pointers of LLVM type i8*) and represent input and output edges of a node as integer indices into the list of function arguments and into the return struct type.

The intrinsics for describing graphs can only be "executed" by internal nodes; *all these*

intrinsics are interpreted by the compiler at code generation time and erased, effectively fixing the graph structure. (Only the number of dynamic instances of a node can be varied at runtime.) All other intrinsics are executable at run-time, and can only be used by leaf nodes or by host code.

Most of the intrinsic functions are fairly self-explanatory and their details are omitted here for lack of space. A few less obvious features are briefly explained here. The `llvm.hpvm.createEdge` intrinsic takes a one bit argument, `ReplType`, to choose a 1-1 or all-to-all edge, and another, `Stream` to choose an ordinary or an invariant edge. The `Stream` argument to each of the `bind` intrinsics is similar. The intrinsics for querying graphs can be used by a leaf node to get information about the structure of the graph hierarchy and the current node's position within it, including its indices within the grid of dynamic instances. `llvm.hpvm.malloc` allocates an object in global memory, shared by all nodes, although the pointer returned must somehow be communicated explicitly for use by other nodes. `llvm.hpvm.barrier` only synchronizes the dynamic instances of the node that executes it, and not all other concurrent nodes. In particular, there is no global barrier operation in HPVM, but the same effect can be achieved by merging dataflow edges from all concurrent nodes.

Finally, using LLVM functions for node code makes HPVM an "outlined" representation, and the function calls interfere with existing intraprocedural optimizations at node boundaries. We are working on adding HPVM information within LLVM IR without outlining, using a new LLVM extension mechanism.

3.3 COMPILATION STRATEGY

We describe the key aspects of the compilation strategy here.

3.3.1 Background

We begin with some background on how code generation works for a virtual instruction set, shown for HPVM in Figure 3.2. At the developer site, front-ends for one or more source languages lower source code into the HPVM IR. One or more optimizations may be optionally applied on this IR, to improve program performance, while retaining the IR structure and semantics. The possibly optimized code is written out in an object code or assembly language format, using the IR as a virtual ISA, and shipped to the user site (or associated server). A key property of HPVM (like LLVM [107]) is that the compiler IR and the virtual ISA are essentially identical. Once the target hardware becomes known (e.g.,

at the user site or server), the compiler backend is invoked. The backend traverses the Virtual ISA and uses one or more target-ISA-specific code generators to lower the program to executable native code.

Hardware vendors provide high-quality back ends for individual target ISAs, which we can often leverage for our system, instead of building a complete native back-end from scratch for each target. We do this for the PTX ISA on NVIDIA GPUs, AVX vector ISA for Intel processors, and X86-64 ISA for individual threads on Intel host processors.

In this work, we focus on using HPVM for efficient code generation (this section) and optimizations (section 3.5). We leave front ends for source languages for future work. Note that we do rely on a good dataflow graph (representing parallelism, not too fine-grained nodes, good memory organization) for good code generation. This need can be met with a combination of parallelism information from suitable parallel programming languages (such as OpenMP or OpenCL), combined with the graph optimizations at the HPVM level, described in Section 3.5. We do *not* rely on precise static data dependence analysis or precise knowledge of data transfers or memory accesses, which is important because it means that we can support irregular or data-dependent parallelism and access patterns effectively.

3.3.2 HPVM Compilation Flow

The HPVM compilation flow follows the structure shown in Figure 3.2. The compiler invokes separate back-ends, one for each target ISA. Each back end performs a depth-first traversal of the dataflow graph, maintaining the invariant that code generation is complete for all children in the graph hierarchy of a node, N, before performing code generation for N. Each back end performs native code generation for selected nodes, and associates each translated node with a host function that implements the node's functionality on the target device.

We have implemented back ends for three target ISAs: PTX (GPU), AVX (Vector), and X86-64 (CPU). Each backend emits a device-specific native code file that includes a device specific function per translated node. For now, we use simple annotations on the node functions to specify the target compute unit manually, where the annotation may specify one *or more* of `GPU,Vector,CPU`, defaulting to `CPU`. The following subsections describe each back end briefly.

Code Generation for PTX: The PTX [108] backend builds on the existing NVPTX back-end in LLVM. This back end translates an extended version of the LLVM IR called NVVM (containing PTX-specific intrinsic functions) [109] into PTX assembly.

A node annotated for GPU will usually contain a two-level or three-level DFG, depending

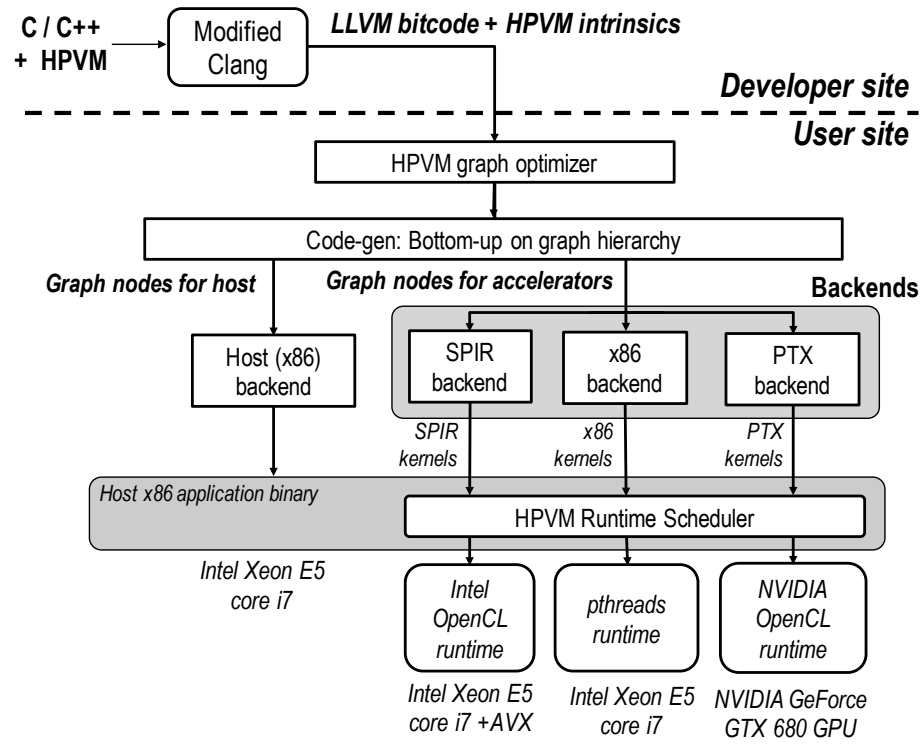


Figure 3.2: Overview of compilation flow for HPVM.

on whether or not the computation must be tiled, as shown in Table 3.1 and explained in Section 3.5.3. Our translator for PTX takes as input the internal node containing this DFG. It generates an NVVM kernel function for each leaf node, which will execute the dynamic instances of the leaf node. If the DFG is a three-level graph, and the second (thread block) level node contains an *allocation node* (defined as a leaf node that allocates memory using the `llvm.hpvm.malloc` intrinsic), the allocated memory is assigned to scratchpad memory, as explained in Section 3.5.3. All other memory is allocated by the translator to GPU global memory or GPU constant memory. The generated NVVM kernel is translated to PTX by the NVPTX back-end. Our translator also generates code to use the NVIDIA OpenCL runtime to load and run the PTX assembly of the leaf node on the GPU. This code is the host function associated with the input dataflow node on the GPU.

Code Generation for AVX: Dynamic instances of leaf nodes are independent, making it possible to vectorize *across* node instances. We leverage Intel's translator from SPIR [54] to AVX, which is part of Intel's OpenCL runtime system, for two reasons: it recognizes and utilizes the independence of SPIR work items to produce vector parallelism, and it is well tuned to produce efficient code for the AVX instruction set. Instead of writing our own

AVX code-generator directly from HPVM with these sophisticated capabilities, we instead wrote a translator that converts HPVM code to SPIR, in which the dynamic instances of leaf nodes become SPIR work items. The SPIR code is then vectorized for AVX by Intel's translator. Our translator also creates the necessary host function to initiate the execution of the SPIR kernel.

Host Code Generation: The x86 backend is invoked last, and generates the following:

- Native code for all nodes annotated as CPU nodes. We build upon the LLVM X86 backend for regular LLVM IR, adding support for HPVM query intrinsics. We translate `createNode` operations to loops enumerating the dynamic instances, and dataflow edges to appropriate data transfers (section 3.3.2).
- For nodes with multiple native versions, i.e. annotated with more than one target, a wrapper function that invokes the HPVM runtime scheduler (section 3.4) to choose which target function to execute on every invocation.
- Host-side coordination code, enforcing the order of execution dictated by the dataflow graph.
- Code to initiate and terminate execution of each dataflow graph.

Data Movement: Code generation for dataflow edges is performed as part of translating the internal dataflow node containing the edge. When the source and sink node execute on the same compute unit, or if they execute on two different compute units that share memory, passing a pointer between the nodes is enough. Such pointer passing is safe even with copy semantics: a dataflow edge implies that the source node must have *completed* execution before the sink node can begin, so the data will not be overwritten once the sink begins execution. (Pointer passing may in fact not be the optimal strategy, e.g., on NVIDIA's Unified Virtual Memory. We are developing a more effective optimization strategy for such systems.)

Some accelerators including many GPUs and FPGAs, only have private address spaces and data needs to be explicitly transferred to or from the accelerator memory. In such cases, we generate explicit data copy instructions using the accelerator API, e.g., OpenCL for GPUs.

It is important to avoid unnecessary data copies between devices for good performance. To that end, we allow explicit attributes `in`, `out`, and `inout` on node arguments, and only generate the specified data movement. Achieving the same effect without annotations would

require an interprocedural May-Mod analysis [110] for pointer arguments, which we aim to avoid as a requirement for such a key optimization.

3.4 HPVM RUNTIME AND SCHEDULING FRAMEWORK

Some features of our translators require runtime support. First, when global memory must be shared across nodes mapped to devices with separate address spaces, the translator inserts calls to the appropriate accelerator runtime API (e.g., the OpenCL runtime) to perform the copies. Such copies are sometimes redundant, e.g., if the data has already been copied to the device by a previous node execution. The HPVM runtime includes a conceptually simple "memory tracker" to record the locations of the latest copy of data arrays, and thus avoid unnecessary copies.

Second, streaming edges are implemented using buffering and different threads are used to perform the computation of each pipeline stage. The required buffers, threads, and data copying are managed by the runtime.

Finally, the runtime is invoked when a runtime decision is required about where to schedule the execution of a dataflow node with multiple translations. We use a run-time policy to choose a target device, based on the dataflow node identifier, the data item number for streaming computations, and any performance information available to the runtime. (Data item numbers are counted on the host: 0 or higher in a streaming graph, -1 in a non-streaming graph.) This basic framework allows a wide range of scheduling policies. We have implemented a few simple static and dynamic policies:

1. *Static Node Assignment*: Always schedule a dataflow node on a fixed, manually specified target, so the target depends only on the node identifier.

2. *Static Data Item Assignment*: Schedule all nodes of a graph for a particular input data item on a single target, so the target depends only on the data item number.

3. *Dynamic*: A dynamic policy that uses the node identifier as in policy #1 above, plus instantaneous availability of each device: when a specified device is unavailable, it uses the CPU instead.

We leave it to future work to experiment with more sophisticated scheduling policies within the framework. In this work, we simply aim to show that we offer the flexibility to support flexible runtime scheduling decisions. For example, the second and third policies above could use a wide range of algorithms to select the target device per data item among all available

devices. The key to the flexibility is that HPVM allows individual dataflow graph nodes to be compiled to any of the targets.

3.5 COMPILER OPTIMIZATION

An important capability of a compiler IR is to support effective compiler optimizations. The hierarchical dataflow graph abstraction enables optimizations of explicitly parallel programs at a higher (more informative) level of abstraction than a traditional IR (such as LLVM and many others), that lacks explicitly parallel abstractions; i.e., the basic intrinsics, `createNode*`, `createEdge*`, `bind.input`, `bind.output`, `getNodeInstanceID.*`, etc., are directly useful for many graph analyses and transformations. In this section, we describe a few optimizations enabled by the HPVM representation. Our long term goal is to develop a full-fledged parallel compiler infrastructure that leverages the parallelism abstractions in HPVM.

3.5.1 Node Fusion

One optimization we have implemented as a graph transformation is *Node Fusion*. It can lead to more effective redundancy elimination and improved temporal locality across functions, reduced kernel launch overhead on GPUs, and sometimes reduced barrier synchronization overhead. Fusing nodes, however, can hurt performance on some devices because of resource constraints or functional limitations. For example, each streaming multiprocessor (SM) in a GPU has limited scratchpad memory and registers, and fusing two nodes into one could force the use of fewer thread blocks, reducing parallelism and increasing pressure on resources. We use a simple policy to decide when to fuse two nodes; for our experiments, we provide the node identifiers of nodes to be fused as inputs to the node fusion pass. We leave it to future work to develop a more sophisticated node fusion policy, perhaps guided by profile information or autotuning.

Two nodes $N1$ and $N2$ are valid node fusion candidates if: (1) they both are (a) leafs, or (b) internal nodes containing an optional allocation node (see Section 3.3.2) and a single other leaf node (which we call the *compute node*); (2) they have the same parent, target, dimensions and size in each dimension, and, if they are internal nodes, so do their compute nodes and their optional allocation nodes; and (3) they are either concurrent (no path of edges connects them), or they are connected directly by 1-1 edges and there is no data transfer between $N1$'s compute and $N2$'s allocation node, if any.

The result is a fused node with the same internal graph structure, and with all incoming (similarly, outgoing) edges of $N1$ and $N2$, except that edges connecting $N1$ and $N2$ are replaced by variable assignments.

Note that fusing nodes may reduce parallelism, or may worsen performance due to greater peak resource usage. Nodes that have been fused may need to be split again due to changes in program behavior or resource availability, but fusing nodes loses information about the two original dataflow nodes. More generally, node splitting is best performed as a first-class graph transformation, that determines what splitting choices are legal and profitable. We leave this transformation to future work.

3.5.2 Mapping Data to GPU Constant Memory

GPU global memory is highly optimized (in NVIDIA GPUs) for coalescing of consecutive accesses by threads in a thread block: irregular accesses can have orders-of-magnitude lower performance. In contrast, constant memory is optimized for read-only data that is invariant across threads and is much more efficient for thread-independent data.

The HPVM translator for GPUs automatically identifies data that should be mapped to constant memory. The analysis is trivial for scalars, but also simple for array accesses because of the HPVM intrinsics: for array index calculations, we identify whether they depend on (1) the `getNodeInstanceId.*` intrinsics, which is the sole mechanism to express thread-dependent accesses, or (2) memory accesses. Those without such dependencies are uniform and are mapped to constant memory, and the rest to GPU global memory. The HPVM translator identified such candidates in 3 (spmv, tpacf, cutcp) out of 7 benchmarks , resulting in 34% performance improvement in tpacf and no effect on performance of the other two benchmarks.

3.5.3 Memory Tiling

The programmer, an optimization pass or a language front-end can "tile" the computation by introducing an additional level in the dataflow graph hierarchy. The (1D, 2D or 3D) instances of a leaf node would become a single (1D, 2D or 3D) tile of the computation. The (1D, 2D or 3D) instances of the parent node of the leaf node would become the (1D, 2D or 3D) blocks of tiles.

Memory can be allocated for each tile using the `llvm.hpvm.malloc` intrinsic in a single allocation node (see Section 3.3.2), which passes the resulting pointer to all instances of the leaf node representing the tile. This memory would be assigned to scratchpad memory on a

GPU or left in global memory and get transparently cached on the CPU.

In this manner, *a single mechanism, an extra level in the hierarchical dataflow graph, represents both tiling for scratchpad memory on the GPU and tiling for cache on the CPU*, while still allowing device-specific code generators or autotuners to optimize tile sizes separately. On a GPU, the leaf node becomes a thread block and we create as many thread blocks as the dimensions of the parent node. On a CPU or AVX target, the code results in a loop nest with as many blocks as the dimensions of the parent node, of tiles as large as the dimensions of the leaf node.

We have used this mechanism to create tiled versions of four of the seven Parboil benchmarks evaluated in Section 3.6. The tile sizes are determined by the programmer in our experiments. For the three benchmarks (`sgemm, tpacf, bfs`) for which non-tiled versions were available, the tiled versions achieved a mean speedup of 19x on GPU and 10x on AVX, with `sgemm` getting as high as 31x speedup on AVX.

3.6 EVALUATION

We evaluate the HPVM virtual ISA and compiler IR by examining several questions: (1) Is HPVM performance-portable: can we use the *same virtual object code* to get "good" speedups on different compute units, and how close is the performance achieved by HPVM compared with hand-written OpenCL programs? (2) Does HPVM enable flexible scheduling of the execution of target programs? (3) Does HPVM enable effective optimizations of target programs?

3.6.1 Experimental Setup and Benchmarks

We define a set of C functions corresponding to the HPVM intrinsics and use them to write parallel HPVM applications. We modified the Clang compiler to generate the virtual ISA from this representation. We translated the *same* HPVM code to two different target units: the AVX instruction set in an Intel Xeon E5 core i7 and a discrete NVIDIA GeForce GTX 680 GPU card with 2GB of memory. The Intel Xeon also served as the host processor, running at 3.6 GHz, with 8 cores and 16 GB RAM.

For the performance portability and hand-coded comparisons, we used 7 OpenCL applications from the Parboil benchmark suite [111]: Sparse Matrix Vector Multiplication (spmv), Single-precision Matrix Multiplication (sgemm), Stencil PDE solver (stencil), Lattice-Boltzmann (lbm), Breadth-first search (bfs), Two Point Angular Correlation Function (tpacf), and Distance-cutoff Coulombic Potential (cutcp).

In the GPU experiments, our baseline for comparison is the best available OpenCL implementation. For spvm, sgemm, stencil, lbm, bfs and cutcp, that is the Parboil version labeled `opencl_nvidia`, which has been hand-tuned for the Tesla NVIDIA GPUs [112]. For tpacf, the best is the generic Parboil version labeled `opencl_base`. We further optimized the codes by removing unnecessary data copies (bfs) and global barriers (tpacf, cutcp). All the applications are compiled using NVIDIA's proprietary OpenCL compiler.

In the vector experiments, with the exception of stencil and bfs, our baseline is the same OpenCL implementations we chose as GPU baselines, but compiled using the Intel OpenCL compiler, because these achieved the best vector performance as well. For stencil, we used `opencl_base` instead, as it outperformed `opencl_nvidia`. For bfs, we also used `opencl_base`, as `opencl_nvidia` failed the correctness test. The HPVM versions were generated to match the algorithms used in the OpenCL versions, *and that was used for both vector and GPU experiments*.

We use the largest available input for each benchmark, and each data point we report is an average of ten runs.

3.6.2 Portability and Comparison with Hand Tuning

Figures 3.3 and 3.4 show the execution time of these applications on GPU and vector hardware respectively, normalized to the baselines mentioned above. Each bar shows segments for the time spent in the compute kernel (Kernel), copying data (Copy) and remaining time on the host. The total execution time for the baseline is shown above the bar.

Compared to the GPU baseline, HPVM achieves near hand-tuned OpenCL performance for all benchmark except bfs, where HPVM takes 22% longer. The overhead is because our translator is not mature enough to generate global barriers on GPU, and thus HPVM version is based on a less optimized algorithm that issues more kernels than the `opencl_nvidia` version, incurring significant overhead. In the vector case, HPVM achieves performance close to the hand-tuned baseline in all benchmarks except lbm. In this case, the vector code generated from the Intel OpenCL compiler after our SPIR backend is significantly worse that the one generated directly from OpenCL - we are looking into the cause of this.

Note that although HPVM is a low-level representation, it requires less information to achieve performance on par with OpenCL, e.g., details of data movement need not be specified, nor distinct command queues for independent kernels. The omitted details can be decided by the compiler, scheduler, and runtime instead of the programmer.

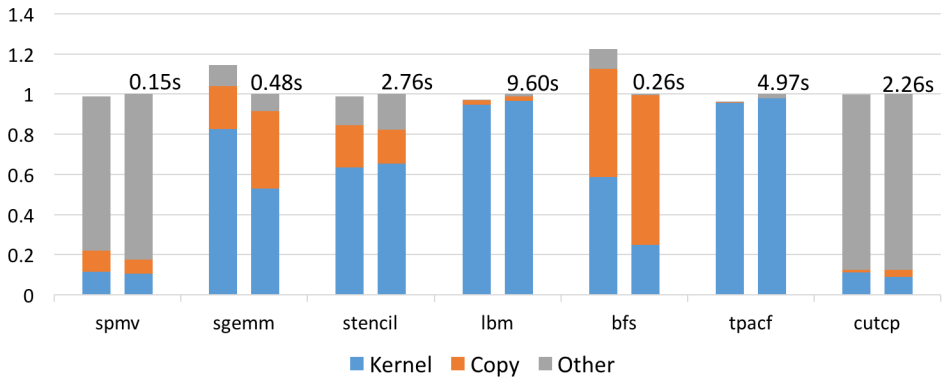


Figure 3.3: GPU Experiments - Normalized Execution Time. For each benchmark, left bar is HPVM and right bar is OpenCL.

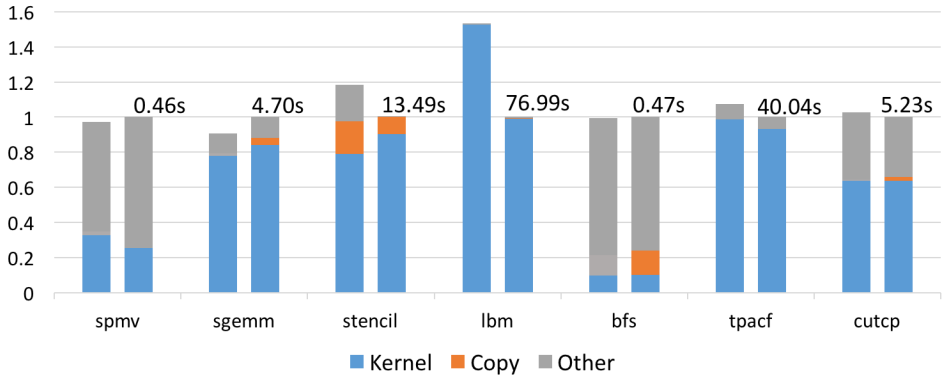


Figure 3.4: Vector Experiments - Normalized Execution Time. For each benchmark, left bar is HPVM and right bar is OpenCL.

3.6.3 Evaluation of Flexible Scheduling

We used a six-stage image processing pipeline, Edge Detection in grey scale images, to evaluate the flexibility that HPVM provides in scheduling the execution of programs consisting of many dataflow nodes. The application accepts a stream of grey scale images, I, and a fixed mask B and computes a stream of binary images, E, that represent the edges of I. We feed 1280x1280 pixel frames from a video as the input and measure the frame rate at the output. This pipeline is natural to express in HPVM. The streaming edges and pipeline stages simply map to key features of HPVM, and the representation is similar to the code presented in Figure 3.1. In contrast, expressing pipelined streaming parallelism in OpenCL, PTX, SPIR or HSAIL, although possible, is extremely awkward.

Expressing this example in HPVM allows for flexibly mapping each stage to one of three targets (GPU, vector or a CPU thread), for a total of $3^6 = 729$ configurations, all generated from a single HPVM code. Figure 3.5 shows the frame rate of 7 such configurations. The

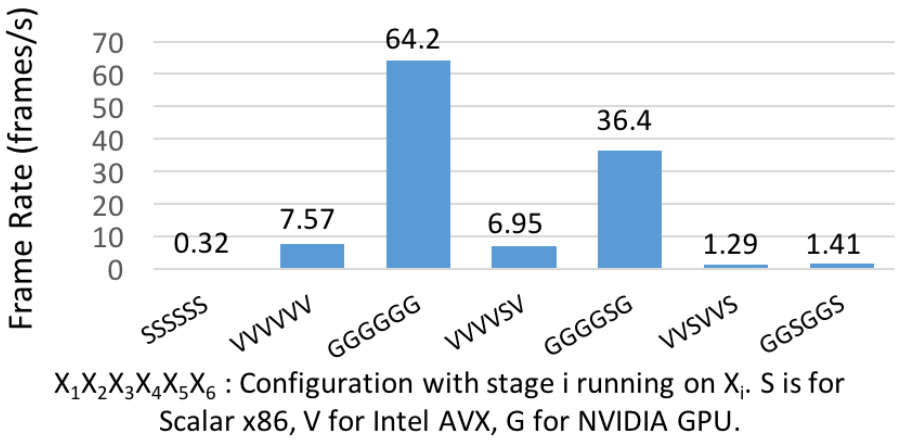


Figure 3.5: Frame rates of different configurations of Edge Detection six stage pipeline through single HPVM object code.

figure shows that HPVM can capture pipelined, streaming computations effectively with good speedups. More importantly, however, the experiment shows that HPVM is flexible enough to allow a wide range of *static* mapping configurations with very different performance characteristics from a single code.

To show the flexibility for *dynamic* scheduling, we emulate a situation where the GPU becomes temporarily unavailable, by using a thread to toggle a boolean variable indicating availability. This can arise, e.g., for energy conservation in mobile devices, or if a rendering task arrives with higher priority. When the GPU becomes unavailable, kernels that have already been issued will run to completion but no new jobs can be submitted to it. We choose to have the GPU available for intervals of 2 seconds out of every 8 seconds, because the GPU in our system is far faster than the CPU.

In this environment, we execute the Edge Detection pipeline using the three different scheduling policies described in Section 3.4. Figure 3.6 shows the *instantaneous frame rate* for each policy. Green and red sections show when the GPU is available or not respectively. We truncate the Y-axis because the interesting behavior is at lower frame rates; the suppressed peak rates are about 64 frame/s.

Static node assignment policy makes no progress during the intervals when the GPU is not available. The other two policies are able to adapt and make progress even when the GPU is unavailable, though neither is perfect. Static data item assignment policy may or may not continue executing when the GPU is unavailable, depending on when the data items that will be issued to the GPU are processed. Also, it may have low frame rate when the GPU is available, if data items that should be processed by the CPU execute while the

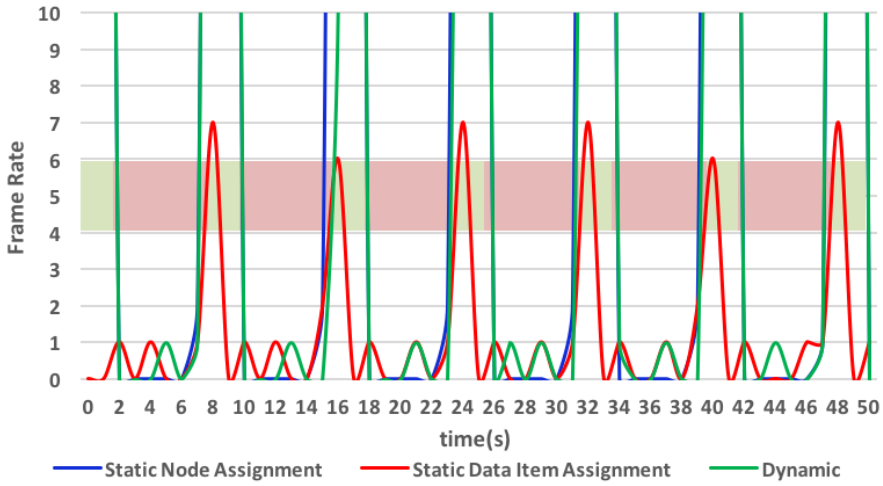


Figure 3.6: Edge Detection Frame rate with different scheduling policies. The green and red band in the graph indicates when the GPU is available or not respectively.

GPU is available. Dynamic policy will not start using the GPU to execute a dataflow node for a data item until the node is done for the previous data item. That is why the frame rate does not immediately increase to the maximum when the GPU becomes available. The experiment shows HPVM enables flexible scheduling policies that can take advantage of static and dynamic information, and these policies are easy to implement directly on the HPVM graph representation.

We also used the Edge Detection code to evaluate the overhead of the scheduling mechanism. We compared the static node assignment policy using the runtime mechanism with the same node assignment using only compiler hints. The overheads were negligible.

Overall, these experiments show that HPVM enables flexible scheduling policies directly using the dataflow graphs.

CHAPTER 4: PROMISE: AN END-TO-END DESIGN OF PROGRAMMABLE MIXED-SIGNAL ACCELERATOR FOR MACHINE LEARNING ALGORITHMS

In this chapter we[1] describe our experience in designing a novel programmable accelerator for machine learning. The underlying technology is called Compute Memory [10] and it performs computation in the SRAM memory bit-cell array in analog. This project provided us necessary lessons into how diverse the programmable accelerators can be from conventional general purpose digital processors.

4.1 MOTIVATION

Current and emerging applications are increasingly relying on the ability to extract patterns from large data sets to support inference and decision making with Machine Learning (ML) algorithms. Such ML inference algorithms (or simply ML algorithms in this thesis) have begun to offer higher performance than humans [42] in cognitive and decision-making tasks [43], but they have demanded more computing capability as they have gotten computationally more complex and required to process larger amounts of data. This coincides with the decline of Moore's Law, and thus it becomes far more challenging to meet such demands than ever. To close the gap between the demand and the offering from traditional general-purpose processors, researchers have begun to explore specialized processors (or accelerators) for ML algorithms [113, 8, 104, 114, 115]. These ML accelerators can offer orders of magnitude higher energy efficiency than general-purpose processors, but most of them are based on digital circuits and/or implement a specific ML algorithm.

In order to further improve energy efficiency of digital ML accelerators, researchers have proposed analog or mixed-signal accelerators [106, 10, 116, 117, 118, 119, 103, 98, 120, 121]. These analog and mixed-signal accelerators rely on small-signal computation which is less precise but more energy efficient than traditional large-signal computation in the digital domain. Therefore, they are suitable for acceleration of ML inference algorithms where the application domain itself is tolerant to such imprecision. However, these accelerators lack a programmable architecture, instruction sets, or compiler support necessary for supporting application software. These capabilities are essential to support high-level programming languages like Python [122] and Julia [123], which implement popular ML libraries such as

[1] **This work was done jointly and equally with Mingu Kang and appears in both of our theses. Mingu led the modifications to Compute Memory chip to make it programmable, and I led the compiler design and implementation for PROMISE. We both worked in equal measure to design the PROMISE ISA**

TensorFlow [124], and MXNet [125].

Moreover, the algorithmic error tolerance that allows hardware-level small-signal computations creates energy vs. accuracy tradeoffs that must be controlled from the application level in order to ensure that application-domain accuracy or precision goals are met. Translating these application-level metrics down to suitable hardware-level control knobs for energy and accuracy without requiring application programmers to understand hardware design concepts requires careful hardware, ISA and compiler design.

We propose PROMISE, the first end-to-end design of a programmable mixed-signal accelerator for diverse ML algorithms, which tackles all these challenges. PROMISE can accomplish a high level of programmability without noticeably losing the efficiency of mixed-signal accelerators for specific ML algorithms. PROMISE exposes instruction set mechanisms that allow software control over energy-vs-accuracy tradeoffs, and supports compilation of high-level languages down to the hardware.

4.2 BACKGROUND

In this section, we first analyze a variety of ML inference algorithms in order to identify commonalities in their data flow, and then we describe a circuit of a programmable mixed-signal accelerator which is well suited for ML algorithms and we build an ISA (Section 4.2.2) and a compiler (Section 4.3) on.

4.2.1 ML Algorithms

The ML algorithms involve repeated Vector Distance (VD) computations denoted by $D(W_j, X)$ between N-dimensional input vector X and weight vector W as depicted in [105]. Commonly used VD computations include the dot product, L1 distance (Manhattan distance), L2 distance (Euclidean distance), and Hamming distance for the ML algorithms including the Support Vector Machine (SVM), template matching, Deep Neural Network (DNN), k-Nearest Neighbor (k-NN), and matched filter as listed in Table 4.1.

These ML algorithms have the following three data-flow properties in common. **(P1)** A single VD is obtained by first computing N element-wise Scalar Distances (SDs) $(d(w[j][i], x[i]))$ followed by an aggregation step such as a sum or average generating the final scalar VD $D(W, X) = \sum_{i=1}^{N} d(w[j][i], x[i])$. **(P2)** The VD between a single query vector X and multiple (say N_o) weight vectors W_js ($j = 1, 2, ...N_o$) needs to be computed. **(P3)** The VD goes through a simple decision function $f()$ such as sigmoid or ReLu to generate

$f(D(W,X))$	Inner loop kernel $D(W,X) = \sum_{i=1}^{N} d(w[i],x[i])$	$f(\ )$
SVM	$\sum_{i=1}^{N} w[i]x[i]$	$sign$
Temp. Match. (L1)	$\sum_{i=1}^{N} \|w[i]-x[i]\|$	min
Temp. Match. (L2)	$\sum_{i=1}^{N} (w[i]-x[i])^2$	min
DNN	$\sum_{i=1}^{N} w[i]x[i]$	$sigmoid$
Feature extraction (PCA)	$\sum_{i=1}^{N} w[i]x[i]$	$--$
k-NN (L1)	$\sum_{i=1}^{N} \|w[i]-x[i]\|$	$majority\,vote$
k-NN (L2)	$\sum_{i=1}^{N} (w[i]-x[i])^2$	$majority\,vote$
Matched filter	$\sum_{i=1}^{N} w[i]x[i]$	min
Linear Regression	$\sum_{i=1}^{N} w[i]$	$accumulate$
	$\sum_{i=1}^{N} w[i]^2$	$accumulate$
	$\sum_{i=1}^{N} w[i]x[i]$	$accumulate$

Table 4.1: Machine learning (ML) algorithms.

the decision y_j. Especially, the VD computation tends to dominate the execution time and energy consumption of ML algorithms for practical problems.

4.2.2 Mixed-Signal ML Accelerator

Recently proposed compute memory [106, 126, 10, 127, 117] deeply embeds energy-efficient analog computations into the periphery circuit of a memory array consisting of conventional bit-cells. As an innovative feature, compute memory can offer seamless interface between digital- and anlog-domain computations. More specifically, compute memory stores a B_w-bit word in a column-major format (i.e., a word is stored in B_w bit-cells connected to the same Bit-Line (BL) across B_w rows), as shown in the yellow box of Figure 4.1(b)) whereas conventional memory does in a row-major format (Figure 4.1(a)). After BLs are pre-charged for a read cycle, compute memory simultaneously asserts B_w Word-Lines (WLs). The durations of these asserted WLs are proportional to the binary weight values of the corresponding bit positions in a given B_w-bit word with a binary Pulse-Width Modulated (PWM) Word-Line (WL) signaling scheme (Figure 4.1(b)). Subsequently, each BL develops a voltage drop (ΔV_{BL}) proportional to a binary weighted sum of B_w bits in the corresponding column [106], which constitutes the first processing stage of compute memory: **(S1)** analog Read (`aREAD`). `aREAD` can not only seamlessly convert digital values stored in memory into analog values for subsequent analog computation stages, but also fetches highly condensed B_w-bit information per BL, significantly improving energy efficiency and throughput.

For ML algorithms, compute memory can store pre-trained W_j in its bit-cells, then it can

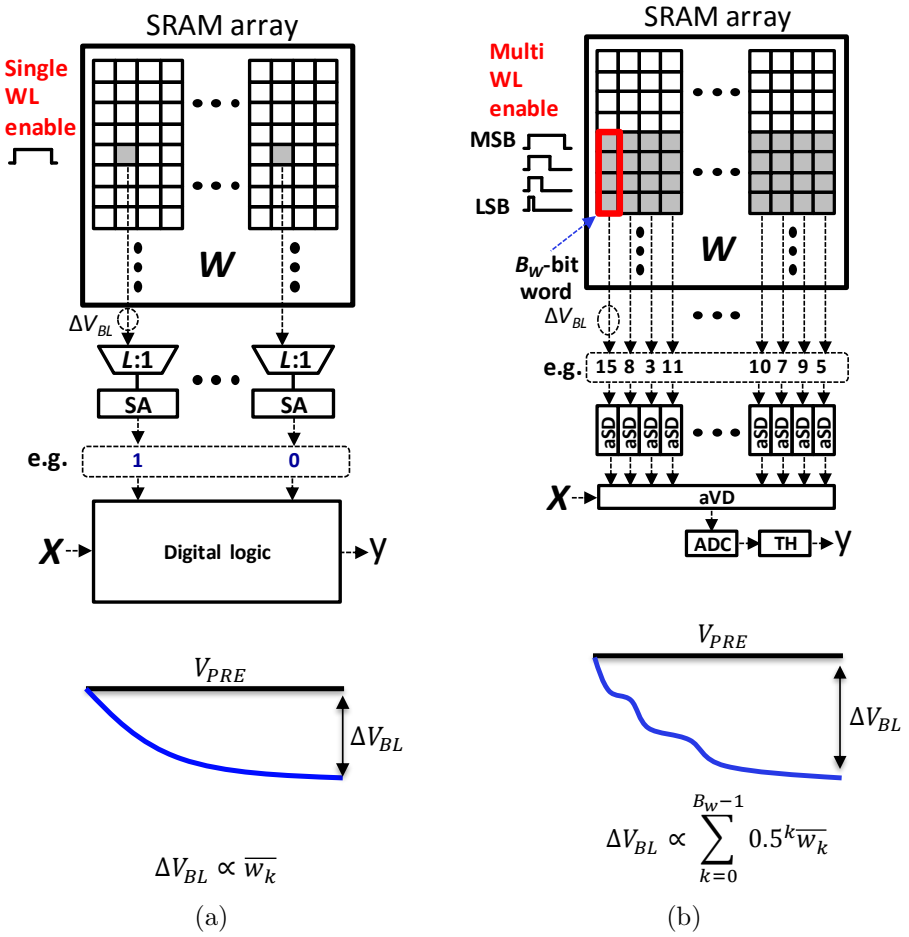


Figure 4.1: Block diagrams and read operations with bitline swing (ΔV_{BL}): (a) conventional system, (b) Compute Memory, with bit precision of scalar weight w, $B_w = 4$ and column mux ratio $L = 4$, analog domain in red.

serve as a very energy-efficient mixed-signal ML accelerator with the following three subsequent analog processing stages: **(S2)** analog Scalar Distance (aSD) implementing scalar distance computations right next to the bit-cell array; **(S3)** analog Vector Distance (aVD) performing the aggregation ($\sum_{i=1}^{N}$ in Table 4.1) by simply charge-sharing all the analog outputs from aSD blocks in one shot; and **(S4)** Analog-to-Digital Conversion (ADC) and ThresHold (TH) converting the analog output of aVD into a digital word and subsequently generating a final decision from the digital word based on a given decision function $f()$ in Table 4.1. Note that the aSD stage can support scalar comparison, multiplication, subtraction, addition, and absolute computation in bitcell pitch-matched analog circuitry [106, 126], while the ADC and TH stages consume negligible portion of total energy as they operate infrequently (once after > 128 aSD operations).

It has been shown that the compute memory can offer significantly lower energy consumption and higher throughput than digital ML accelerators, but it supports only limited reconfigurability [10], as described in this section. Furthermore, the absence of an instruction set for the compute memory limits the use for a short sequence of operations with single computation kernel, single memory bank, and fixed parameters such as a vector length.

In this section, we present PROMISE architecture as a substrate to explore ISA, discuss various challenges in developing ISA, and then propose ISA for a programmable mixed-signal accelerator.

4.2.3 PROMISE Architecture

Single Bank Architecture: PROMISE is built on compute memory as shown in Figure 4.2(a), where the standard SRAM read and write functionalities are preserved (at the bottom) for additional flexibility. Along with **(S1)** `aREAD`, **(S2)** `aSD`, **(S3)** `aVD`, and **(S4)** `ADC` and `TH` described in Section 4.2, PROMISE comprises `X-REG` and `CTRL` to transform compute memory into a programmable mixed-signal accelerator. The more detailed architectural specification of PROMISE is as follows.

A PROMISE bank consists of 256 ($= N_{COL}$) columns. An 8-bit ($= B_w$) word is distributed across four consecutive rows constituting a *word row* and two neighboring columns which store 4-bit MSB and 4-bit LSB to enhances linearity through a sub-ranged read technique [10]. That is, `aREAD` can read out a 128-element vector of digital values and seamlessly converts it to that of analog values. Furthermore, `aREAD` can simultaneously perform element-wise addition or subtraction with X, a 128-element vector representing the input operand for inference in Table 4.1. `aSD` and `aVD` are architected to perform operations on 128 analog values. `ADC` consists of eight 8-bit ADCs which operate in parallel and convert 32×2^6 analog values to digital values per second. Note that the `aVD` output of each bank is digitized by these ADCs to prevent the noise from analog operations accumulating over the iterations. This digitization also enables a multiple-bank architecture, where reliable data transfers between banks are required. `TH` implements non-linear operations such as sigmoid via piece-wise linear approximation [128] not only to compute the decision functions $f()$ in Table 4.1 but also to aggregate intermediate computed values when the vector length N is larger than 128.

Lastly, `X-REG` is a digital block similar to a vector register file in a SIMD processor, holding eight 128-element vectors representing eight X values. `CTRL` is a controller to generate enable signals for the aforementioned components based on a given *instruction* and make compute memory function as a programmable mixed-signal accelerator.

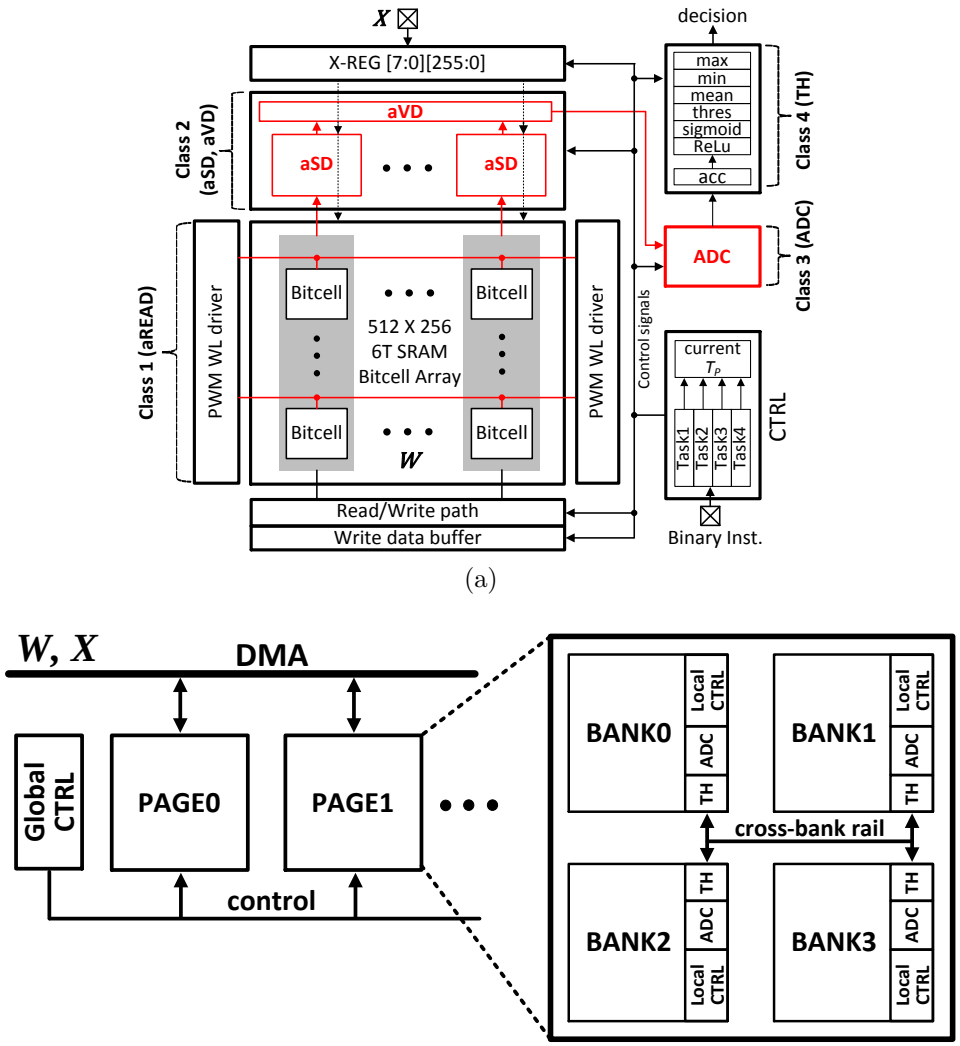


Figure 4.2: The PROMISE architecture: (a) a single-bank architecture with $N_{ROW} = 512$, $N_{COL} = 256$, analog processing marked in red, and (b) a multi-bank architecture.

Multiple bank architecture: PROMISE can be extended to a multi-bank structure (Figure 4.2(b)), which has multiple (up to eight in this work) `PAGEs`, each of which includes four banks. Thus, long (¿ 128) vectors can be distributed across multiple banks for parallel processing. PROMISE does not have the scalability limitation by the analog operations as the partial results from each bank are always converted to digital values by the ADCs. Then, the partial sums can be aggregated across different banks via cross-bank rail, similar to H-Tree in [118]. The data transfer of 8-bit ADC output from each bank to the other through the cross-bank rail takes only 0.5 pJ with activity factor of 0.5 from post-layout simulations. This is negligible (¡ 1%) as a `aREAD` consumes only 61 pJ/bank as listed in Figure 4.5(b). The control of multi-bank by the instruction is discussed in Section 4.2.5.

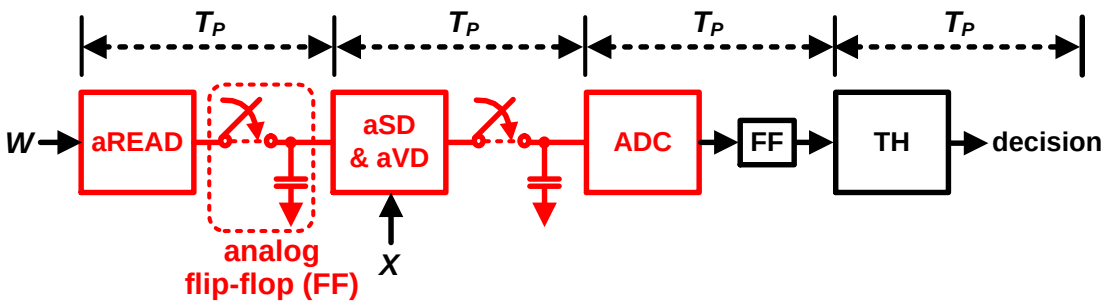


Figure 4.3: Analog pipeline in PROMISE with operating frequency $= 1/T_P$, and red marked area is analog domain.

Analog pipeline architecture: To achieve higher throughput than the original compute memory [10], PROMISE adopts analog pipelining as shown in Figure 4.3. This requires analog Flip-Flops (FFs) to hold the output of each stage. The first pipelined stage (`aREAD`) needs N_{COL} analog FFs whereas the second pipelined stage consisting of `aSD` and **aVD** needs only one analog FF to store the aggregated scalar output. Thus, we propose to reuse a pre-existing capacitor, which is a part of the analog multiplier [10] in the `aSD` block, as an analog FF for the `aREAD` stage to minimize the hardware overhead.

4.2.4 Challenges in Using Programmable Mixed-Signal Accelerator

The combination of algorithmic diversity and mixed-signal operations in PROMISE creates many challenges that we should consider when exploring ISA for a programmable mixed-signal accelerator. To support a broad range of ML algorithms, each stage needs to support diverse programmable operations. However, we identify some challenges in supporting (too) many diverse programmable operations which can significantly degrade both throughput and accuracy.

Higher Impact of Delay Variation across Operations: Analog operations often exhibit a wide range of delays depending on operation types [10], while each stage must accommodate the worst case delay out of all the operations for the stage. Furthermore, as all the stage of a pipelined mixed-signal accelerator need to operate at the same clock period T_P, T_P needs to accommodate the worst case delay across all the stages, i.e., $T_P = \max(T_{S1\_max}, T_{S2\_max}, T_{S3\_max}, T_{S4\_max})$, as illustrated in Figure 4.4). That is, supporting more programmable operations increases longer idle time for some stages. As a result, we observe up to 2× throughput degradation when designing for the 9 ML benchmarks evaluated in this work. Furthermore, such long idle time negatively affects accuracy at the algorithm level because analog values are typically stored on (area-constrained) capacitors

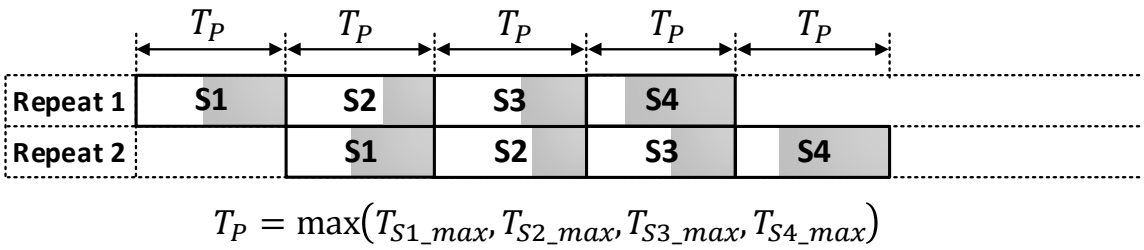


Figure 4.4: 4-stage processing in PROMISE with operational diversity per stage.

and thus degrade over time due to various leakage mechanisms. Especially, this problem is worst for the S1 stage (`aREAD`) as each BL is subject to the leakage contributions from all the bit-cells in the column (upto 0.04%/ns).

More Limit in Sequence of Operations: We also recognize some other challenges specific to analog processing, which further limits the number of possible programmable operations. Specifically, a chain of analog processing stages imposes intrinsic sequentiality of operations. Furthermore, two consecutive processing stages need to be physically closely placed to shun substantial degradation in analog voltage from one stage to the next. Lastly, large capacitor ratio between consecutive stages (input-output 20:1 to maintain the voltage drop ¡ 5%) is required to transfer the signal via charge-transfer mechanism without additional analog buffer.

4.2.5 PROMISE Instruction Set

We explore an ISA suitable for a programmable mixed-signal accelerator, considering the constraints described above.

Instruction Format: We propose a wide-word macro instruction format, which is referred to as `Task`. Akin to a Very-Large Instruction Word (VLIW), a single `Task` consists of multiple operations, *except that the operations are sequential and not parallel like VLIW architectures.* As depicted in Figure 4.5(a), the four `Class` fields specify four operations for 4 pipelined stages of PROMISE, while the three other fields, `OP_PARAM`, `RPT_NUM` and `MULTI_BANK` configure all or specific `Class` operations. More specific descriptions of these seven fields are as follows.

Operating Parameter Field: `OP_PARAM` comprises 33 bits and configures operating parameters of `Class` operations in a given `Task`, facilitating *flexible programmability.* As shown in Figure 4.5(b), `W_ADDR` specifies a compute memory address for a `Class-1` operation. `X_ADDR1`

Task

OP_PARAM (28 bits)	RPT_NUM (7 bits)	MULTIBANK (2 bits)	Class-1 (3 bits)	Class-2 (4 bits)	Class-3 (1 bits)	Class-4 (3 bits)

(a)

Contents	Bits	Description
SWING	[27:25]	ΔV_{BL} swing code – 000:min(5mV/LSB), 111:max(30mV/LSB)
ACC_NUM	[24:23]	# of operands to be accumulated for `accumulate` opcode in `Class-4`
W_ADDR	[22:14]	Bitcell array address of W in `Class-1`
X_ADDR1	[13:11]	Bitcell array address of X in `Class-1`
X_ADDR2	[10:8]	`X-REG` array address of X in `Class-2`
X_PRD	[7:6]	`X_ADDR1` & 2 circulate from 0 to "`X_PRD` - 1"
DES	[5:4]	`Class-4` output destination - 00: `ACC` input, 01: output buffer, 10: `X-REG`, 11:Write data buffer
THRES_VAL	[3:0]	Thresholding reference value for threshold opcode in `Class-4`

(b)

Class	Operation [operand]	Bit length	OP CODE	Option
1	*none*	3 bits (OPCODE)	000	*X_PRD*: *X_ADDR1* circulates from 0 to "*X_PRD*-1"
	write[W_ADDR]		001	
	read[W_ADDR]		010	
	aREAD[W_ADDR]		011	
	aSUBT[W_ADDR, X_ADDR1]		100	
	aADD[W_ADDR, X_ADDR1]		101	
2	*none*	4 bits (OPCODE + aVD)	000	aVD bit: 0: no aggregation 1: aggregation *X_PRD*: *X_ADDR2* circulates from 0 to "*X_PRD*-1"
	compare		001	
	absolute		010	
	square		011	
	sign_mult[X_ADDR2]		100	
	unsign_mult[X_ADDR2]		101	
3	*none*	1 bit (OPCODE)	0	
	ADC		1	
4	*accumulation*	3 bits (OPCODE)	000	*DES, ACC_NUM*
	mean		001	*DES*
	threshold		010	*DES, THRES_VAL*
	max		011	*DES*
	min		100	*DES*
	sigmoid		101	*DES*
	ReLu		111	*DES*

(c)

Figure 4.5: Instruction set of PROMISE: (a) instruction format, (b) operation parameters (`OP_PARAM`) (c) operations in each Class.

and `X_ADDR2` designate `X-REG` addresses for `Class-1` and `Class-2`, respectively. `SWING` controls BL swing ΔV_{BL}, e.g., `111` allows 30 mV/LSB whereas `001` allows 5 mV/LSB. This parameter is a key knob to control the trade-off between energy and accuracy under software control; Section 5.4 evaluates this accuracy-energy trade-off for further energy savings. Refer to Figure 4.5(c) for descriptions of the remaining parameters and their assignments of bit fields.

Class Fields: `Class-1` is composed of 3-bit opcode and defines six possible memory operations. `READ`, `WRITE`, or `aREAD` makes compute memory perform a digital read, digital write, or analog read operation to a compute-memory address specified by `OP_PARAM` (`W_ADDR`).

aADD or aSUB fuses an analog read and an element-wise analog addition or subtraction into a single operation where two vector operands come from compute-memory addresses specified by OP_PARAM (W_ADDR and X_ADDR1), respectively.

Class-2 consists of 4-bit opcode and specifies a composition of one of six possible aSD operations with one of two possible aVD operations. Specifically, aSD operating on a computed value from Class-1 supports three unary operations: compare, absolute, and square and two binary operations: sign_mult and unsign_mult where the other operand comes from an X-REG address specified by OP_PARAM (X_ADDR2). aVD specifies whether an aggregation should be performed or not after an aSD operation.

Class-3 and Class-4 comprise 1- and 3-bit opcode and control whether an ADC should be performed or not and specify one of seven possible TH operations, respectively. The seven possible TH operations are as follows: accumulation, mean, threshold, max, min, sigmoid, and ReLu.

In summary, Class-1, Class-2, and Class-3 define a distance computation, $D(W_j, X)$ in Table 4.1 in the analog domain, while Class-4 specifies $f(D(W_j, X))$ in the digital domain. Operations such as element-wise write back ([129]), or shuffle and compare operation ([130]) were omitted to keep T_p small. These operations are required to efficiently implement algorithms such as k-means and random forest.

Loop Control Field: RPT_NUM comprises 7 bits and specifies how many times the Task should be executed to process multiple W_js. The compute memory and X-REG addresses (W_ADDR, X_ADDR1 and X_ADDR2) are incremented sequentially every iteration. Although unconventional for modern RISC architectures, this is a natural choice for typical ML inference algorithms, which iterate sequentially through data W_js in memory for the computation $D(W_j, X)$.

Multiple Bank Control Field: MULTI_BANK comprises 2 bits and specifies the number of banks used to distribute long (> 128) vectors for parallel processing. The intermediate results from banks 1, 2,..., and $2^{MULTI\_BANK} - 1$ are transferred to bank 0 through the cross-bank rail in Figure 4.2(b) to be aggregated by digital TH accumulation operation. The long vector needs to be distributed to the same row of each bank to support the parallel processing across multiple banks. The instruction is shared by 2^{MULTI_BANK} banks as those banks process the same operation. The output of a Class-4 operation can be transferred to the X-REG of a specific bank in any PAGE by defining *DES_ADD* in OP_PARAM.

Extension to Large Scale Applications: PROMISE is well suited to process 128 dimensional vector processing. Longer vectors (> 128) can be processed by repeating the 128 dimensional vector processing sequentially by setting RPT_NUM = (W.size() / 128) and

other parameters. A word row of compute memory stores 128 words, and thus two word rows are used to store 256 dimension vector. Two consecutive iterations complete a vector processing. The address `W_ADDR` and `X_ADDR1` (or `X_ADDR2`) are incremented as the `Task` iterates to process the next 128 words. However, the `X` is re-used to compute the distances to many `W`s as explained in Section 4.2. Thus, the `X_ADDR1` and `X_ADDR2` circulates from 0 to `X_PRD-1`. For example, `X_PRD = 2` to process 256 element vector `X`.

4.2.6 Algorithm Mapping and Compiler Need

A given ML inference algorithm sometimes requires several `Tasks` for different distance metrics. That is, it can be described by one or more `Tasks`. We present an example of template matching with L1 distance kernel to find the closest 512-pixel image out of 127 candidate images (W_j) to input query image (X) by processing across four banks in parallel. The template matching is mathematically defined as:

$$j_{opt} = \arg\min_{j} \sum_{i=1}^{512} |x[i] - w[j,i]| \tag{4.1}$$

The corresponding `Task` instruction consists of: `RPT_NUM` = 127 specifying the number of candidate images; `MULTI_BANK` = 4 to distribute 512 pixels into four banks (128 pixels per bank) for parallel processing; `Class-1 aSUBT` to perform element-wise subtraction of X with W_j; `Class-2 absolute` with aggregation; and `Class-3 ADC` followed by a digital-domain `Class-4 min` to compute $f() = \arg\min_j$.

Although each `Class` offers a limited number of operations (6, 12, 2 and 7, respectively), PROMISE can perform more than 1000 compositions of operations for a given X value. Furthermore, we observe that the order of `Task` instructions in a complex ML inference algorithm and the accuracy setting of each `Task` through the `SWING` parameter significantly affect the accuracy at the algorithm level, exploding the solution space for code generation. Hence, it is not efficient and inherently sub-optimal to manually generate code for a complex ML inference algorithm. This in turn gives a compiler an opportunity to generate and optimize code implementing a given ML inference algorithm, which will be explored in the subsequent section.

4.3 COMPILER

In this section, we discuss the goals of a compiler for PROMISE, describe how PROMISE compiler design meets these goals while translating a given ML algorithm described in a high-level language into the PROMISE ISA, and how the compiler addresses the programmability challenges mentioned in Section 4.2.4.

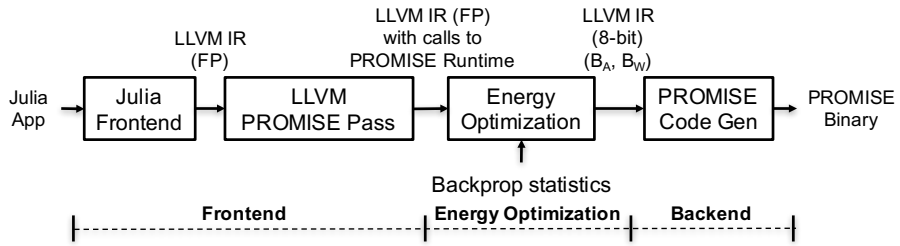


Figure 4.6: Compiler Pipeline. LLVM(FP) implies that data arrays are in floating point. 8-bit implies fixed point data arrays

4.3.1 Goals

Hardware Abstraction: The compiler intermediate representation (IR) should abstract away low-level details of the hardware, so that front ends don't have to be concerned about hardware details like `Class-1` vs. `Class-2` vector operations, or the specifics of the `OP_PARAM` parameters. At the same time, the IR should enable compilers to perform optimizations, and should capture essential information for generating efficient code on PROMISE. This is similar to how mid-level compiler IRs abstract away details like finite register files and different register classes from front ends, while enabling sophisticated register allocation algorithms to manage these details [131].

Accuracy-Energy Tradeoff: As explained in Section 4.2.2 the PROMISE ISA allows the compiler to use the `SWING` parameter as a knob to tune ΔV_{BL} to exploit the tradeoff between energy and accuracy. However, it is very hard to go from a high level description of "accuracy" that programmer understands in the context of her algorithm, to hardware specific parameter such as the voltage swing. This is especially true for applications which would have several smaller computations to offload (for example, DNNs can offload each layer computation) to a hardware accelerator such as PROMISE. It is unclear how the error in a single computation would affect the overall accuracy of the application. Towards this end, the PROMISE compiler must provide a compiler optimization to find the optimal `SWING` parameter for each `Task`.

Hardware Specific Optimizations: Furthermore, for applications where data does not fit into PROMISE memory, the compiler needs to find efficient dataflow pattern based on the size of data arrays.

Easily Extensible to ML Domain Specific Languages: Domain specific languages (DSLs) and libraries for ML are evolving fast: there are already a wide range of popular DSLs such as Torch, Theano, Tensorflow, MXNet, Keras, and others [132, 133, 124, 125, 134], implemented on top of dynamic programming languages such as Python, Julia, R, Scala, Perl, etc. Thus, a desirable goal of the PROMISE compiler is to be easily extendable to new DSLs. To achieve this, we must provide a language-neutral IR as an interface for the compiler/programmer to reason about operations on PROMISE.

4.3.2 AbstractTask and PROMISE Compiler IR

In this section we first define an `AbstractTask` which is an abstraction of a PROMISE `Task` described in Section 4.2.5. `AbstractTask` is based on our observation that a vector operation can be either a `Class-1` (addition/subtraction) or a `Class-2` (signed/unsigned multiplication) operation, but that distinction is only relevant for late stage code generation, not front ends and other compiler optimizations. `AbstractTask` is also oblivious to hardware-specific parameters such as the number of elements in a vector (i.e., the length of the bitcell array in PROMISE), size of the bitcell array, etc.

An `AbstractTask` has the following seven fields: **(F1)** `W`: address of a 2D data array; **(F2)** `X`: address of a 1D data array; **(F3)** `output`: address of the output 1D data array; **(F4)** `vecOp`: element-wise vector operation between a row of `W` and `X`; **(F5)** `redOp`: reduction operation on the output of `vecOp`; **(F6)** `digitalOp`: unary operation on the output of `redOp`; **(F7)** `vectorLen`: number of elements in `X`; **(F8)** `loopIterations`: number of iterations of the loop; **(F9)** `threshold`: threshold value for `Class-4 threshold` operation of PROMISE `Task`; and **(F10)** `swing`: swing parameter corresponding to a voltage swing at which this should run on PROMISE. The `swing` field is initialized to the value `0b111` (maximum accuracy) by the frontend, and is fine-tuned later by energy optimization pass as described in Section 4.3.4.

The PROMISE compiler IR is a directed acyclic graph (DAG) of such `AbstractTask`s, where each node represents an `AbstractTask`, and a directed edge between two nodes `P` and `C` represents the dataflow from the output of P to the input (`W` or `X`) of C. For example, for a DNN inference algorithm of fully connected layers, the compiler IR would be a sequential pipeline of `AbstractTask`s. Each `AbstractTask` would represent the computation of the corresponding fully connected layer of DNN.

The IR is acyclic even though a task is always an iterative computation because the loop count is simply represented as the `RPT_NUM` field of a task (Figure 4.5(a)). Loops surrounding a sequence of one or more tasks are always executed on the host processor and not on the PROMISE accelerator.

4.3.3 Code Generation

Figure 4.6 shows the PROMISE compiler pipeline for Julia. There are three parts: (1) a front end to map Julia applications to the IR, (2) energy optimizations on the IR, and (3) a back end to translate the IR to the PROMISE ISA. The IR, energy optimizations and back end are all designed to be independent of the source-level language, to make it easily extendable to other languages or DSLs.

Frontend (Julia program to PROMISE compiler IR): We chose Julia as the source language because such a high-level language enables us to easily identify patterns of computations that can be offloaded to PROMISE. The ML kernels are built using high-level library calls to matrix/vector operations, and so we do not need to use sophisticated compiler analysis to identify these operations. Julia also supports several ML libraries such as MXNet [135], Flux [136], and others and is already used to develop ML applications. (Julia also has a working open-source LLVM front-end. Other choices like TensorFlow didn't have one available at the time we started.)

The Julia frontend translates applications to LLVM IR. The PROMISE pass runs over each LLVM function and uses pattern matching to identify computations that can be offloaded to PROMISE, translates it into an `AbstractTask`, and replaces the computations with a call to the PROMISE runtime to pass the parameters of the `AbstractTask`.

The LLVM IR [107] representation is a collection of Single Static Assignment (SSA) [137] graphs, one graph per function in a program. Each node in an SSA graph corresponds to an LLVM IR instruction. In each function, the PROMISE pass looks for matrix-matrix operations with a reduction component (e.g., matrix-matrix multiplication, which uses the dot product). When it finds one, it also looks for a single basic block "natural loop" [131] enclosing it. If such a loop is found, the compiler pass identifies the induction variable of the loop and checks if the SSA graph of the basic block matches with the SSA graph pattern shown in Figure 4.7, and explained below. If matched, the single basic block loop can be offloaded to PROMISE. Since the pattern matching can be fragile to minor variations in generated LLVM IR, e.g., the loop index variable being incremented instead of decremented in each loop iteration, the compiler converts all single basic block loops into canonical loops before pattern matching.

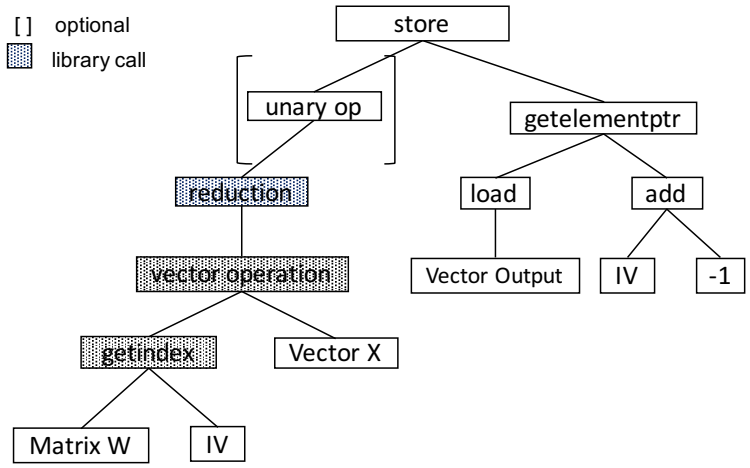


Figure 4.7: SSA Pattern for single basic block loops. The shaded nodes are calls to Julia library. The part enclosed in square brackets is optional for pattern matching.

Starting from the bottom left of the SSA graph, library call `getindex` is used to get the IV_{th} vector of matrix W. Vector X which is a loop invariant (i.e., its definition must be outside the loop), is used to perform an element-wise vector operation with vector W_{IV}. X being loop invariant is essential for the computation to be efficient on PROMISE: X gets stored in the buffer `X_REG` which is designed to hold a vector with temporal locality and is constant throughout the execution of a `Task`. The output of the vector operation undergoes `reduction` operation using a Julia library function call. The right child of the `store` SSA node uses the `getelementptr` instruction in LLVM to compute the address where computed value needs to be stored in the vector *Output*.

This pattern captures many widely used ML inference kernels, like template matching, support vector machines, k-nearest neighbor, matched filtering, matrix-vector multiplication, etc., and is used as a canonical form which can be mapped to an `AbstractTask`.

Translation of the SSA graph to a `AbstractTask` is straightforward. The SSA nodes `Matrix W`, `Vector X` and `Vector Output` map to the `W`, `X`, and `Output` fields of a `AbstractTask`, respectively. `loopIterations` can be computed at run-time from the induction variable `IV` and corresponding conditional branch at the end of the basic block. `VectorLen` can be obtained from `X`. The `swing` field is initially set to maximum value of `0b111` corresponding to the maximum ΔV_{BL} for `aREAD` operation, and optimized later as described in Section 4.3.4. After mapping to an `AbstractTask`, we replace the loop from the LLVM IR representation with a call to PROMISE runtime, passing it the fields of *AbstractTask*.

Backend (compiler IR to PROMISE ISA): The backend of the compiler maps the compiler IR to ISA by mapping each `AbstractTask` to `Task`. This involves two parts: (1)

compile time code generation for `Class 1-4`, and (2) computing the `OP_PARAM, RPT_NUM, MULTI_BANK` fields at runtime and passing them to the PROMISE run-time, which launches the task.

Code generation for `Class 1-4` is relatively straightforward: most fields have a one-to-one mapping to the corresponding field of the ISA. For example, the digital operation directly maps to the `Class-4` field, and the `Class-3` operation is always `ADC`. Primarily, the backend identifies where the `vecOp` of `AbstractTask` would execute - `Class-1` or `Class-2`. If the `vecOp` is element-wise addition or subtraction, `Class-1` operation is set to `aADD` or `aSUB` respectively, and `Class-2` performs just the reduction of the resulting values according to `redOp` field of `AbstractTask`. Otherwise, for signed/unsigned multiplication, `Class-1` performs the `aREAD` operation and `Class-2` performs the element-wise vector multiplication with `X` operand from `X_REG` and performs the aggregation as well.

The `OP_PARAM, RPT_NUM` and `MULTI_BANK` fields require runtime information and are computed on the host before task launch. For vector lengths > 128, `X_PRD` is set to the number of rows required to fit one vector, i.e., `vectorLen/128`. `RPT_NUM` is set to the product of the number of loop iterations and the number of rows per vector, i.e., `X_PRD * loopIterations`. Setting other fields such as `W_ADDR` in `OP_PARAM` is straightforward and we skip those for lack of space.

4.3.4 Energy Optimization

In the ML domain, classification accuracy of a model is an important metric. For instance, neural networks for a handwritten digit recognition can achieve classification accuracy of $p_{model} = 98\%$ on the MNIST [138] dataset running on floating point architectures such as GPUs. Often, the broader application context that uses the results of the classifier can tolerate lower accuracy, but that is application-specific. In our work, we allow the programmer to express the additional error they can tolerate when running their model on PROMISE, which we call the mismatch probability (p_m). Formally, mismatch probability (p_m) is the upper bound on difference between the classification accuracy of an algorithm running on PROMISE ($p_{PROMISE}$) and the classification accuracy of the ML model (p_{model}), i.e., $p_{model} - p_{PROMISE} \leq p_m$

The energy optimization in the compiler pipeline in Figure 4.6 takes the mismatch probability p_m from a programmer and determines the `swing` field of each `AbstractTask` in the application that would ensure that error tolerance is met. Mapping a high-level parameter like p_m directly to a suitable `swing` voltage is difficult, and is even more challenging for algorithms such as neural networks that have multiple tasks. We solve this problem by breaking

it down into two parts, taking advantage of prior work [44]: (a) determining a minimum bit precision required to achieve the given mismatch probability, using the results of [44]; and (b) modeling the accuracy of `AbstractTask` in terms of equivalent bit precision, as a function of the swing voltage. These are explained briefly below, second one first.

To achieve B-bit precision in the final output, the error introduced must be less than $1/2^{B+1}$. The major source of error in PROMISE due to lowering the swing voltage is from `aREAD` operations (see Section 4.2.2). The output of `aREAD` follows normal distribution $\hat{W} \sim \mathcal{N}(W, \sigma_W^2)$, where $\sigma_W = |W| \cdot f(SWING)$ and $f(SWING)$ is a function of the `SWING` parameter and ranges from 0.08 ~ 0.75. The `SWING` parameter and $f(SWING)$ are inversely proportional, and hence, the σ_W is minimized with higher `SWING` parameter. After the aggregation of N such vector elements through charge-sharing, the standard deviation of the aggregated value (σ_{agg}) of output is $\sigma_W/\sqrt{N}$. Since W is in range [-1, 1], we assume $|W| = 1$ for all values to maximize σ_W and σ_{agg}. In this work, we choose a confidence level of 99%, which corresponds to $2.6 \times \sigma_{agg}$. To guarantee B-bit precision at the output of aggregation, this yields:

$$2.6\sigma_{agg} = 2.6\frac{f(SWING)}{\sqrt{N}} < \frac{1}{2^{B+1}} \tag{4.2}$$

If we can estimate the required bit precision for a given p_m (part (a), above), we can use (2) to compute the minimum swing voltage. To achieve (a), we leverage prior work [44]. In that work, Sakr et al. analyze the quantization (floating-point to fixed-point conversion) tolerance of neural network and give a relationship between the accuracy degradation and the bit precisions used to store the activation and weights of a neural network model. Mathematically, it says that given the bit precision of weights and activations (B_W and B_A), they can provide a bound on the mismatch probability $p_{fl} - p_{fp}$, where p_{fl} is the accuracy of the floating-point model, and p_{fp} is the accuracy of the same model quantized to fixed-precision representation for weights and activations. The analysis bounds the mismatch probability by

$$p_m \leq \Delta_A^2 E_A + \Delta_W^2 E_W,$$

where E_A and E_W are statistics of the model obtained while training the model, and $\Delta_A = 2^{-(B_A-1)}, \Delta_W = 2^{-(B_W-1)}$.

Since, neural network inference is simply a series of repeated *vector distance* computations described in Section 4.2, computation of activations in each layer can be described as a PROMISE `AbstractTask` with W equal to a weight matrix, and vector X equal to the activations of the previous layer. The vector operation is element-wise multiplication and

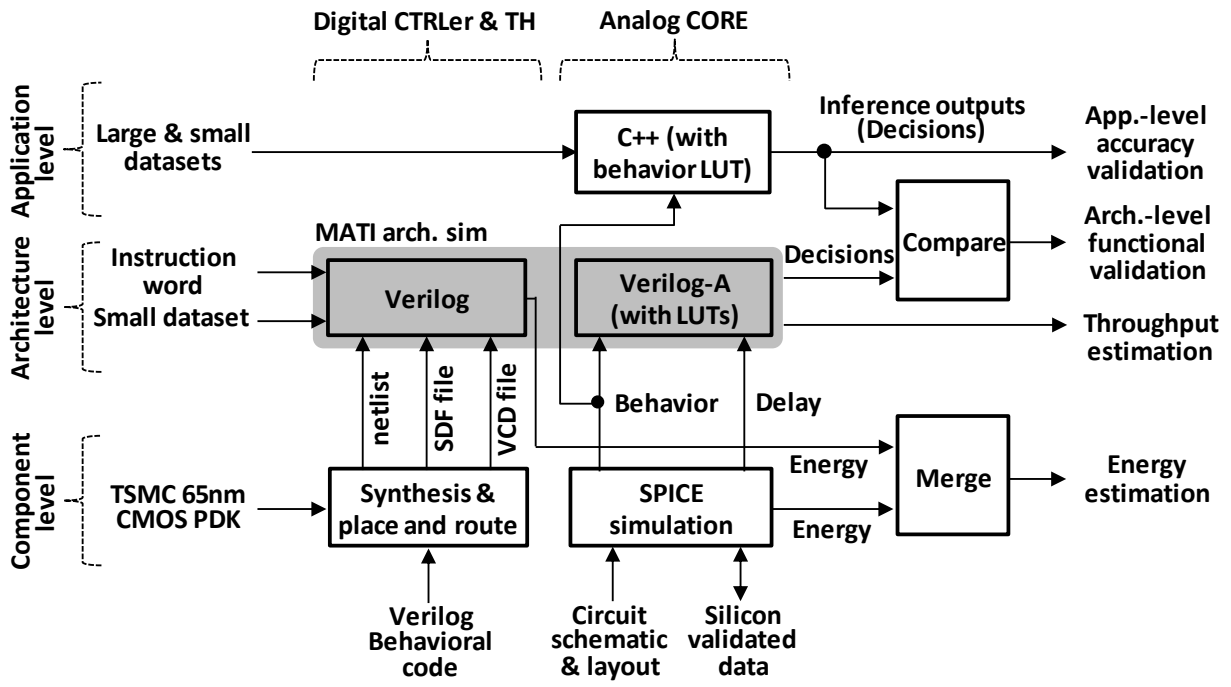


Figure 4.8: PROMISE validation methodology.

the reduction operation is the sum of all the values in the result. The activation function such as `sigmoid/ReLu/tanh` are supported as `Class-4` operations in PROMISE. Thus, a neural network inference is a sequence of `AbstractTask`s.

In the context of PROMISE, we can use Sakr's model to calculate the bit precision B_X for X in each `AbstractTask` given the mismatch probability p_m for the model (sequence of `AbstractTask`s), weight precision $B_W = 7$ since the PROMISE bitcell array uses 8-bits to store a value, including one sign bit.

Putting it all together, we use the back propagation statistics, E_A and E_W of the trained application model, along with the desired p_m, as input to analysis of [44] to estimate B_A and B_W. We then use B_X and `vectorLength` as input to (2) to estimate the minimum swing voltage, which we pass as the `Class-0 SWING` parameter to the PROMISE run-time when launching the task.

The analysis above – like that of [44] – has focused on neural networks. For other combinations of `Class-1` and `Class-2` operations, we would have to extend Sakr's analysis [44]. Doing so is straightforward, but is outside the scope of this work. We can still optimize kernels with only a single task by using a brute-force sweep through all 8 swing voltage levels to look for the optimal value.

4.4 VALIDATION METHODOLOGY

This section describes our methodology for validating PROMISE's energy, delay, and accuracy benefits as well as the benefits of the compiler generated code. Figure 4.8 summarizes our validation methodology. Specifically, we (1) develop energy, delay, and behavioral models of PROMISE components in TSMC 65 nm GP process including analog non-idealities; (2) incorporate these component-level analog models (in Verilog-A) with PROMISE Verilog model the digital CTRL to ensure correct functionality and estimate accuracy over small data sets; and (4) develop a PROMISE C++ model with component level behavioral models for verifying accuracy over large data sets of compiler generated code.

Component-level Models: The entire mixed-signal chain was post-layout simulated in SPICE in TSMC 65 nm GP process to obtain the energy and delay numbers listed in Table 4.3. The total energy and delay for the mixed-signal blocks were compared with the measured results reported in [10] and the differences were found to be within 10% and 9%, respectively. We conducted Monte-Carlo SPICE simulations of all the analog components to capture their behavior in the presence of analog non-idealities,. Behavioral models incorporating these non-ideal analog effects were extracted from SPICE data in the form of look-up tables (LUTs). These behavioral LUTs and delays were then incorporated into component-level Verilog-A models for analog blocks. Verilog models of all the digital components including `CTRL` and `TH` block were developed and synthesized with TSMC 65 nm GP library via Synopsys Design Compiler.

Application-level Validation: Verilog and Verilog-A models described in Section 4.4 were integrated to obtain a cycle and functionally accurate PROMISE Verilog model. The digital blocks was verified by generating the correct control signals at the right time in the presence of post-layout parasitics when presented with the appropriate PROMISE instruction word. In addition, a functional PROMISE C++ model incorporating the LUT-based analog behavioral models described in earlier in Section 4.4 was also developed. This C++ model was run on large data sets to obtain PROMISE's application-level accuracy. The compiler generated code was verified with the Verilog models of the digital components along with the Verilog-A models to evaluate energy and accuracy of PROMISE.

Benchmarks: The commonly employed ML algorithms listed in Table 4.2 were mapped to PROMISE. As PROMISE is programmable, it employs 8-bit data to cover diverse applications and algorithms as shown in many other implementations [115, 139, 140, 42]. For application level energy optimization analysis we also choose 3 variants of DNN of different complexity to demonstrate the architecture. These benchmarks give us diversity in complexity and allows us to explore the energy accuracy trade-offs.

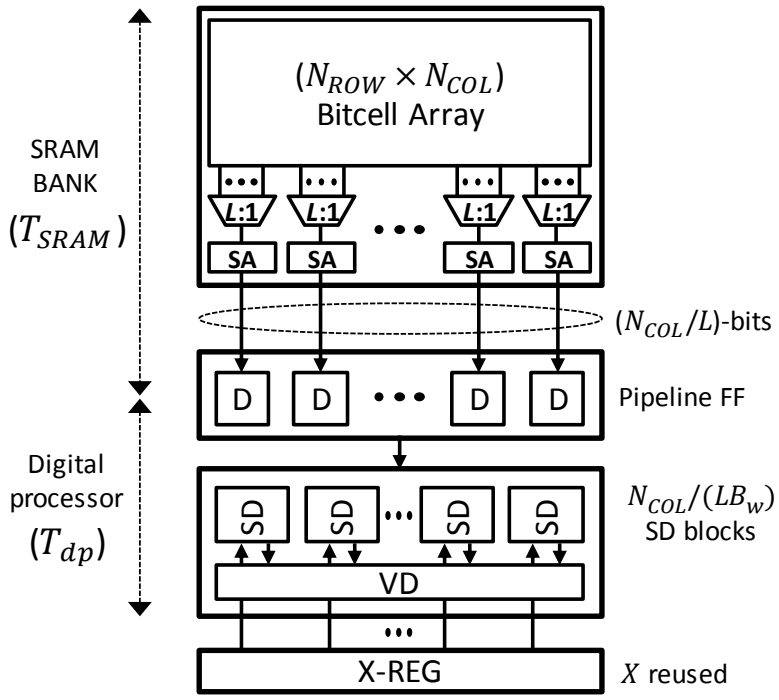


Figure 4.9: Single bank of CONV-8b with $L = 4$, $N_{ROW} = 512$, $N_{COL} = 256$, where an SRAM communicates with an algorithm-specific synthesized digital processor over a pipelined interface.

Baseline Architectures: It is well known that ASICs are order-of-magnitude more energy and delay-efficient compared to general-purpose processors (CPU/GPU) [143]. Therefore, we choose the following four ASICs as our comparison baselines for the conservative evaluation of PROMISE: (a) **CM** : The programmability overhead is estimated via the comparison with CM. (b) **State-of-arts**: We also compare PROMISE with the recent prior arts with silicon IC prototypes [115, 114], which implement similar algorithms as PROMISE. The DNN is compared with [114] and k-NN and template matching with L1 and L2 distances are compared with [115]. (c) **CONV-8b**: We build the baseline digital architecture (Fig. 4.9), which consists of the computational logic synthesized for the specific algorithm + conventional SRAM. (d) **CONV-OPT**: This is the same as CONV-8b but with minimum bit precision required per benchmark.

Even though prior arts exist for some of benchmarks many of them do not have a relevant previous ASIC. Furthermore, configurations such as process technology and on-chip memory capacity are not perfectly identical to PROMISE. In order to perform a conservative comparison, the CONV-8b/CONV-OPT (CONV) is chosen to operate at maximum speed while only restricting the number of SRAM banks are employed to be same as PROMISE. The

Algorithm	Application	Database	Data size (N)	Number of Categories	Problem size	# of AbstractTasks	AbstractTask	W	X	Comments	Opt. swing (p_m=1%)
DNN (Multilayer Perceptron)	Hand-written character recognition	MNIST	(8-bit) 22×23	10	10 categories, 60000 training samples, 10000 test samples	4	vecOp: multiplication redOp: sum	Weights	Test samples	5-layer DNN with nodes as follows: 784-512-256-128-10	3,2,3,3
Matched filtering	Event (gun-shot) Detection	Gun-shot Mono sound	(8-bit) 256, 512, 1024	2	100 test vectors	1	vecOp: multiplication redOp: sum	Filter weights	Test samples		1
Template matching (w/ L1 & L2)	Face Recognition	MIT-CBCL	(8-bit) 16×16 22×23 32×33	64	256 Candidates	1	vecOp: *subtraction* redOp: L1: *absolute* L2: *square*	Candidate faces	Test samples	Nearest candidate based on either L1 or L2 distance	2
Linear SVM	Face Detection	MIT-CBCL	(8-bit) 16×16	2	2 categories, 2000 training samples, 858 test samples	1	vecOp: multiplication redOp: sum	Weights	Test samples	Face data converted into a vector, linear SVM applied on it	6
k-NN (w/ L1 & L2)	Hand-written character recognition	MNIST	(8-bit) 16×16 22×23 32×33	10	10 categories, 54210 training samples, 200 test samples	1	vecOp: *subtraction* redOp: L1: *absolute* L2: *square*	Training samples	Test samples	Sorting is done in external processor after processing on MATI	1
Feature extraction (PCA)	Face Detection	MIT-CBCL	(8-bit) 16×16	—	2000 samples	1	vecOp: multiplication redOp: sum	Weights	Samples	Four features used for face detection based on PCA	—
Linear regression	Modeling linear predictor	Synthetic data	(8-bit) 2 dim.	—	8192 samples	4	vecOp: *None* redOp: AT1,AT2: *mean* AT3: *square,* AT4: *sign_mult*	AT1: U AT2: V AT3: U AT4: U	T4: V	2-D linear regression : $slope = \frac{\Sigma uv - \Sigma \bar{u}\bar{v}}{\Sigma u^2 - \Sigma \bar{u}^2}$ $y\text{-}intercept = \bar{v} - slope \cdot \bar{u}$	—

Table 4.2: Benchmarks for PROMISE Simulations [141, 142, 138].

SRAM fetches $N_{COL}/(LB_w)$ words per single read access of bank.Therefore, CONV operates with maximum achievable throughput of :

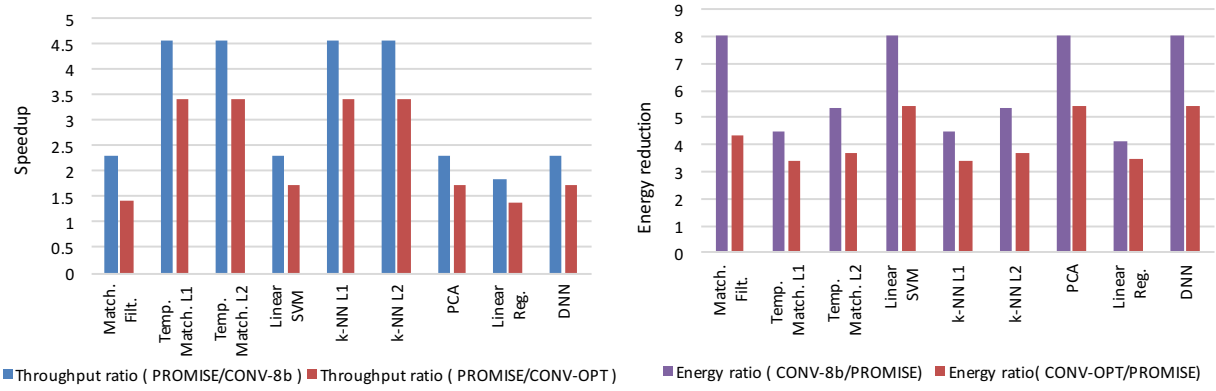
$$f_{CONV} = \left(\frac{N_{COL}/L}{B_w}\right)\left(\frac{1}{T_{SRAM}}\right) \tag{4.3}$$

The CONV (Fig. 4.9) consists of computation logic synthesized for each specific benchmark therefore it only incurs the energy costs of that specific benchmark and additional routing, dataflow and control energy are neglected. Additionally, CONV-OPT has the minimum precision for each benchmark thereby making it the most conservative baseline to compare with PROMISE in terms of accuracy, energy and throughput.

4.5 EVALUATION

In this section we present evaluation of PROMISE. We estimate the energy and delay for each operation using the methodology presented in Section 4.4. We present the gains of the compiler-based energy optimization, compared with the maximum (unoptimized) swing voltages, and then compare PROMISE (with the maximum swing voltages) against the baselines described in Section 4.4.

PROMISE executes 128-element vector operation per bank within T_P, i.e., its throughput in terms of "number of OPs per bank per unit time" can be expressed by $f_{PROMISE} = 128/T_P$



(a) (b)

Figure 4.10: PROMISE (with $SWING = 111$) compared to CONV in terms of (a) speed-up and (b) energy savings.

per bank, where OP includes the SRAM access and MAC (multiply + accumulation) for single word. The energy consumption of PROMISE can be divided as:

$$E_{PROMISE} = \sum_{i=1}^{4} E_{Class,i} + E_{LEAK} + E_{CTRL}, \tag{4.4}$$

where $E_{Class,i}$ is the energy consumed by Class,i instruction, E_{CTRL} and E_{LEAK} are the CTRL block and leakage energies, respectively. Table 4.3 shows the energy consumed for each operation at $SWING = 111$.

4.5.1 Effectiveness of Compiler

Code Generation: The benchmarks for evaluation are listed in Table 4.2. The benchmarks use a wide variety of combination of operations in different Classes. Encoding these diverse algorithms by hand or using a library is not feasible. Coding these algorithms in Julia, and using the compiler to generate PROMISE ISA was both more efficient and error free. Moreover, it is also suboptimal to find the `SWING` parameter for benchmarks which use more than one `Task`s. For example, for DNN benchmark with 3 hidden layers, the number of `SWING` combinations is $8^4 = 4096$. Inter `Task` compiler analysis is required to find the optimal `SWING` parameter for such algorithms as shown in later sections. Lastly, the compiler also handles different vector sizes for these benchmarks.

Energy Optimization: We evaluate the energy benefits on PROMISE by choosing the

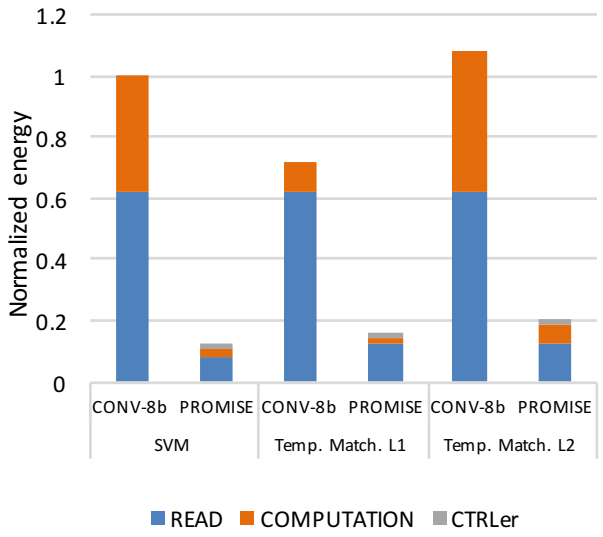


Figure 4.11: Energy breakdown (normalized to SVM with CONV-8b)

optimal `swing` values obtained via compiler directed energy optimization. Figure 4.12 shows the energy benefits for energy optimized code generated for PROMISE. The figure contains the energy estimates of PROMISE for the benchmarks under test for two cases: (1) Full Precision - all `Task`s use maximum `SWING`, and (2) Optimized - `Task`s use the optimized `SWING` set by the energy optimization pass. We limit the degradation in accuracy to 1% ($p_m = 1\%$). Feature Extraction and Linear Regression are omitted from this evaluation as they are not classification kernels, and mismatch probability is defined for classification algorithms only.

The first six benchmarks in Figure 4.12 compile down to a single `AbstractTask` in PROMISE compiler IR, and the optimal `swing` is obtained by doing a sweep over all the 8 values of `swing`. The last three benchmarks, DNN-N1, DNN-N2, and DNN-N3 are variants of the multilayer perceptron algorithm shown in Table 4.2. The three DNNs have 3, 4, and 5 layers respectively and translate to 2, 3, and 4 `AbstractTask`s. The search space for optimal `swing` for the three DNNs increases exponentially with increase of each layer. We use the energy optimization analysis to obtain the `swing` values for these DNNs. The optimal swing for the `AbstractTask`s in DNN-1 is (3, 6), for DNN-2 is (5, 7, 7), and for DNN-3 is (3, 3, 4, 6). The maximum energy savings come from the lower layers of each DNN, which are wider and also more tolerant to imprecision. Overall the benefits of the optimization range from 4% for Linear SVM to about 25% for the two k-NN versions (geometric mean: 17%).

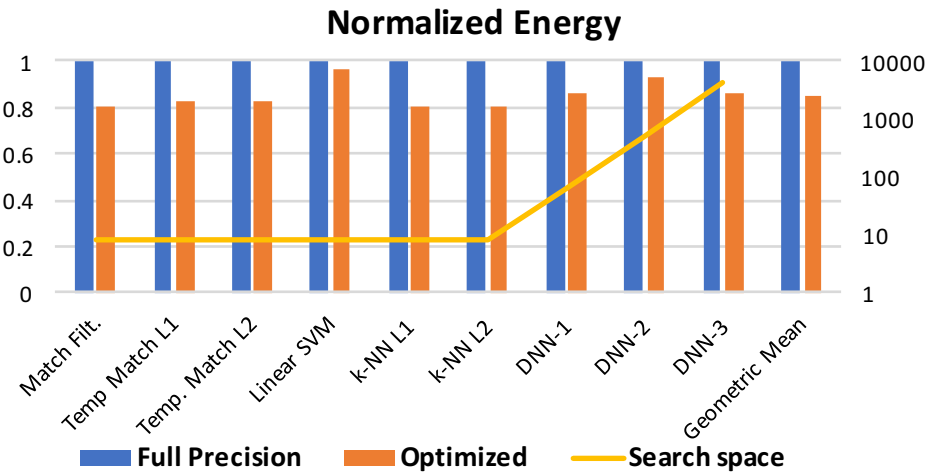


Figure 4.12: Energy gains by compiler directed energy optimization. DNN-(1, 2, 3) are trained on MNIST dataset. Their structures are: DNN-1(784-128-10), DNN-2(784-256-128-10), and DNN-3(784-512-256-128-10).

4.5.2 Performance and Energy

Comparison with State-of-Arts: The k-NN accelerator [115] with L1 and L2-distances is implemented in a 14 nm FinFET process, where 8-bit 128-dimension X is processed with 128 W_js. The k-NN accelerator demonstrates 3.37 (3.84) nJ/decision (processing single input X) with 21.5M (20.3M) decisions/s with L1 (L2) distance. PROMISE achieves 18 (22.9) nJ/decision with 1.12M (0.98M) decisions/s with L1 (L2) distance for the same benchmark with single bank. Though PROMISE achieves lower energy efficiency and throughput, PROMISE is implemented in a 65 nm process (vs. 14 nm FinFET in [115]). If the energy and delay numbers in [115] are scaled to a 65 nm process based on ITRS roadmap [144], PROMISE achieves 4.1× (3.7×) smaller energy and 3.1× (3.4×) lower throughput, achieving 1.3× (1.1×) energy-delay product (EDP) reduction with L1 (L2) distance.

The DNN accelerator [114] was implemented in a 28 nm process for 8-bit 5-layer DNN with a network size of 784-256-256-256-10. The accelerator employs total 1 MB SRAM (to test up to 16-bit case), zero-skipping, and RAZOR technique [145], demonstrating 0.57 uJ/decision and 28K decisions/s. On the other hand, PROMISE enables 8-bit 5-layer DNN with a size of 784-512-256-128-10 in 36 banks (= 576 KB). The network size is not identical, but comparable (PROMISE's network is slightly larger, requiring 69% higher number of coefficients W_js and MAC operations). PROMISE achieves 0.49 uJ/decision and 558K decisions/s, achieving 1.15× energy saving and 19.9× throughput improvement with 22×

Class	Operation	Delay	Energy/ Bank (pJ)	Operation	Delay	Energy/ Bank (pJ)
1	`write`	2	73	`aSUBT`	7	103
	`read`	2	33	`aADD`	7	103
	`aREAD`	5	61	–		
2	`compare`	6	5	`sign_mult`	14	16
	`absolute`	6	12	`unsign_mult`	14	16
	`square`	8	38	–		
3	`ADC`	138	6	–		
4	`mean`	3	≈ 0	`accumulation`	4	≈ 0
	`min`	4	≈ 0	`threshold`	2	≈ 0
	`max`	4	≈ 0	`sigmoid`	3	≈ 0
	`ReLu`	3	≈ 0	–		
Leakage energy per cycle			0.6	CTRLer energy per cycle		5.4

Table 4.3: Energy and Delay (# of cycles). 1 cycle = 1ns.

EDP reduction though PROMISE was implemented in a 65 nm process (vs. 28 nm in [114]).

Comparison with CONV: Figure 4.10(a) shows that PROMISE provides a speed-up of $1.4 - 3.4\times$ compared to CONV-OPT across the benchmarks. PROMISE's speed-up is the least for linear regression because it needs to re-access the same SRAM data every Task because analog data cannot be stored due to leakage whereas CONV stores the data in a local register (pipeline FF in Fig. 4.9) and reuse it. Figure 4.10(b) shows that PROMISE achieves a $3.4 - 5.5\times$ energy savings compared to CONV-OPT leading to an EDP improvements of $4.7 - 12.6\times$ compared to CONV-OPT. The key reason for PROMISE's energy efficiency is due to its aREAD (`Class-1`) and aSD/aVD (`Class-2`) being executed with low-voltage swing mixed signal computation block (See Figure 4.11).

The programability overhead of PROMISE is minimal, rather our estimates show the concepts introduced in this work actually can improve over CM. Our results show that PROMISE achieves up to $1.9\times$ speed-up over CM due to the analog pipeline in spite of its operational diversity. In spite of the increased complexity of CTRL to support the programmability, PROMISE was found to achieve 5.5% energy savings over CM due to reduced leakage as PROMISE can go to sleep mode quicker after completing the given `Task`s due to the throughput gain.

CHAPTER 5: APPROXHPVM

With the end in Moore's Law and Denard scaling, the gap between hardware performance and the ever-increasing requirements of modern applications continues to widen [19]. 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. Approximate computing is particularly relevant for application domains that can tolerate small errors with acceptable loss in the final output. Examples of such domains include signal processing, speech recognition, sensor networks, information retrieval, data mining, video decoding, game engines, and machine learning [146].

Approximate computing techniques span many architectural components: floating-point units, caches, DRAM, and analog and digital accelerators [18, 19, 20]. Software techniques are similarly diverse, including loop perforation [21], barrier elision [22], reduction sampling, and function substitution [23]. 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 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 require 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 voltage swings).

Existing systems for accuracy-aware optimizations do not provide a fully automated framework that is able to target multiple heterogeneous devices without requiring programmer-guided annotations. The ACCEPT [36] framework uses a programmer-guided approach with source-level approximation annotations. It targets accelerators using a combination of static analysis and autotuning to choose between multiple approximation options. However, ACCEPT cannot support systems with multiple compute units and multiple approximation techniques, which is a key goal of our work. Moreover, it does not decouple hardware-independent accuracy constraints from hardware-specific knobs, thereby reducing the extensibility of the technique to multiple kinds of hardware devices. EnerJ [37] presents a type system that separates approximate and precise data. Chisel [38] and Rely [39] programming languages introduced the idea of quantifiable reliability and accuracy at the program level. Introducing approximation metrics as part of a new programming language hurts program

portability since applications need to be ported at the source-level.

We[1] 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. ApproxHPVM automatically translates end-to-end application-level accuracy constraints into accuracy requirements for individual operations. ApproxHPVM uses a hardware-agnostic accuracy-tuning phase to do this translation, which greatly speeds up the analysis, enables greater portability, 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 four benchmarks from the deep learning domain. 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 a user-specified application-level accuracy constraint with high probability. Across four benchmarks, we observe performance speedups ranging from 2.8x to 20x and energy reduction ranging from 1.9x to 7.1x for an accuracy loss of 1%.

5.1 THE HPVM INTERNAL REPRESENTATION

ApproxHPVM is inspired by and builds on HPVM [147], a dataflow graph compiler IR for heterogeneous parallel hardware. We extend the HPVM IR to support execution of basic linear algebra tensor computations and approximate computing metrics. In this section we first briefly discuss the HPVM IR and then describe our extensions to it.

5.1.1 Background: HPVM dataflow graph

HPVM [147] 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. It can also capture nested

[1]**This project was done jointly and equally with Hashim Sharif (hsharif3@illinois.edu) and it would appear in both of our theses.**

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

Table 5.1: Tensor intrinsics in the HPVM representation.

parallelism, and SPMD style data parallelism exploited in GPUs. The authors 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. Thus, we leverage the existing infrastructure of HPVM and extend it to compile to our heterogeneous approximate computing platform.

The following HPVM details are relevant for this work. In HPVM, a program is represented as a host program plus 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. Nodes in the HPVM dataflow graph (DFG) represent units of execution, and edges between nodes describe explicit data transfer requirements. 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 execution of the DFG is initiated and terminated by host code.

Lastly, the HPVM DFG is hierarchical which allows a node in HPVM DFG to be a leaf node describing the computation or an internal node describing a HPVM DFG itself. 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 in 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.

5.1.2 Tensor operations in HPVM

Domain-specific languages such as Tensorflow and Pytorch allow for improved programmer productivity and are thus gaining wide-spread adoption. Accordingly, compilers 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. Recently, Google's Tensorflow infrastructure introduced XLA [148], a compiler framework that lowers Tensorflow programs to linear algebra operations and efficiently calls existing GPU and CPU backends such as Eigen and cuDNN.

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 used by a target backend or runtime. 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, and machine learning. For instance, convolutional neural networks may be expressed using generic linear-algebra operations. This design choice provides two essential benefits: a) It enables efficient mapping of tensor operations to special purpose hardware and highly optimized target-specific runtime libraries, such as cuDNN for GPUs. 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.

The tensor operations in HPVM are represented as calls to intrinsic functions. The intrinsic calls appear to existing LLVM passes as calls to unknown external functions, so no changes to existing passes are needed. 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. The list of tensor intrinsics introduced in HPVM are listed in Table 5.1.

Figure 5.1 shows a single-layer fully connected neural network represented in HPVM using tensor intrinsics. 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 *tensorMulNode*, *tensorAddNode* and *tensorTanhNode* (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.

5.1.3 Approximation Metrics in the IR

The core feature we add to HPVM design is the introduction 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:

- L_1 error:

$$L_1^e = \frac{L_1(A - G)}{L_1(G)} \tag{5.1}$$

  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.

- L_2 error:

$$L_2^e = \frac{L_2(A - G)}{L_2(G)} \tag{5.2}$$

  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.

Figure 5.2 shows how the approximation metrics are represented in the compiler IR. The approximation parameters for each tensor operation are attached as additional arguments to the respective intrinsic functions. While our current system only uses the two metrics described, our implementation and analyses can be easily extended to include additional approximation metrics.

```
define i8* @tensorMulNode(i8* %t1, i8* %t2) {
  %result = call i8* @hpvm.tensor.mul(i8* %t1, i8* %t2)
  return i8* %result
}
define i8* @tensorAddNode(i8* %t1, i8* %t2) {
  %result = call i8* @hpvm.tensor.add(i8* %t1, i8* %t2)
  return i8* %result
}
define i8* @tensorTanhNode(i8* %t) {
  %result = call i8* @hpvm.tensor.tanh(i8* %t)
  return i8* %result
}
; Root node of the Dataflow Graph
define void @DFG_root(i8* %W, i8* %X) {
  ; Creating DFG nodes
  %nodeMul = call i8* @hpvm.createNode(i8* @tensorMulNode)
  %nodeAdd = call i8* @hpvm.createNode(i8* @tensorAddNode)
  %nodeTanh = call i8* @hpvm.createNode(i8* @tensorTanhNode)
  ; Creating data-flow edges between different DFG nodes
  call void @hpvm.createEdge(i8* %nodeMul, i8* %nodeAdd, 1, 0, 0, 0)
  call void @hpvm.createEdge(i8* %nodeAdd, i8* %nodeTanh, 1, 0, 0, 0)
  ; Binding the parent input to inputs of the leaf nodes
  call void @hpvm.bind.input(i8* %nodeMul, 0, 0, 0)
  call void @hpvm.bind.input(i8* %nodeMul, 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* %nodeTanh, 0, 0, 0)
}
```

Figure 5.1: Single layer fully connected DNN in ApproxHPVM

```
define i8* @tensorMulNode(i8* %t1, i8* %t2) {
  %result = call i8* @hpvm.tensor.mul(i8* %t1, i8* %t2, float %relative_l1, float %relative_l2)
  return i8* %result
}
define i8* @tensorAddNode(i8* %t1, i8* %t2) {
  %result = call i8* @hpvm.tensor.add(i8* %t1, i8* %t2, float %relative_l1, float %relative_l2)
  return i8* %result
}
define i8* @tensorTanhNode(i8* %t) {
  %result = call i8* @hpvm.tensor.tanh(i8* %t, float %relative_l1, float %relative_l2)
  return i8* %result
}
```

Figure 5.2: Tensor intrinsics annotated with accuracy metrics. The accuracy metrics are stored in LLVM metadata associated with the tensor intrinsic call

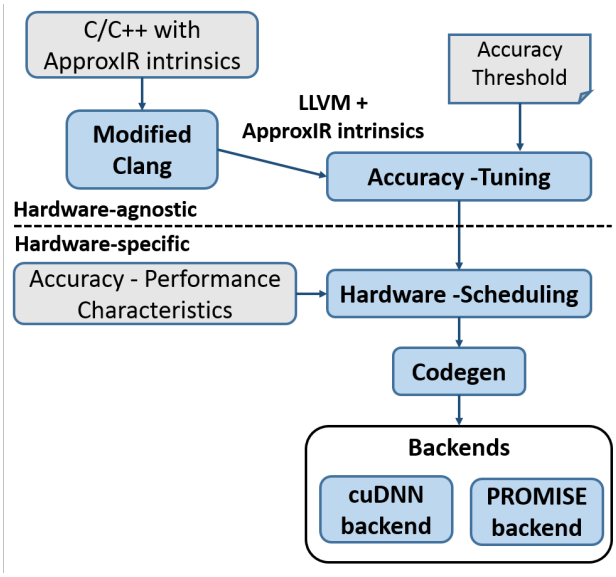


Figure 5.3: ApproxHPVM System Workflow

5.2 SYSTEM WORKFLOW

Figure 5.3 shows the overall ApproxHPVM workflow, which extends the HPVM infrastructure developed by Kotsifakou et. al. [147]. The primary input is a program written in C/C++ with HPVM intrinsics for representing data-flow and ApproxHPVM intrinsics for the tensor operations. We modified the Clang compiler to generate ApproxHPVM, which feeds into the later passes. A second input is the desired end-to-end accuracy requirement, a domain-dependent parameter. For the neural network domain, we use the final classification accuracy as the parameter.

The 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 accuracy. We decompose this mapping problem into a hardware-agnostic first stage and a hardware-specific second stage, for several reasons: (1) The end-to-end accuracy must be decomposed into separate accuracy constraints for each tensor operation, but this is generally a very expensive step, as described next: in fact, it proved impractical to do directly for our hardware. The decomposition allows us to perform this step without considering the specific mapping choices and approximation decisions, which greatly speeds up the analysis and also allows it to be done once ahead of time. (2) The decomposition generates hardware-agnostic ApproxHPVM code with accuracy constraints, which enables portability of the ApproxHPVM virtual object code. (3) The second stage is extremely fast, enabling techniques like dynamic accuracy-aware scheduling and rapid accuracy-aware hardware design space exploration, which would be impractical if

the expensive first stage were needed (both of these are key goals of our ongoing research, although out of scope for this paper).

We therefore use a hardware-agnostic accuracy-tuning phase that takes an end-to-end accuracy requirement and computes the accuracy requirements for individual ApproxHPVM operations, adding these requirements in the IR. 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 [107]. For each target, a (static) accuracy-aware hardware scheduling phase chooses which compute units should execute each tensor operation, and optimizes any approximation parameters available on each compute unit to minimize energy and/or maximize performance, while satisfying the individual accuracy constraints on each operation. Finally, the code generation phase leverages the hardware-specific backends to generate code for each compute unit. In our work, we build a) GPU backend that targets the cuDNN and cuBLAS libraries that are optimized for high-level tensor operations, and b) a PROMISE backend that targets the library that performs optimized tensor computations on the PROMISE hardware simulator. The GPU can use FP32 (golden) or FP16 values for the network weights and bias values. PROMISE can only use 8-bit integers, and offers a choice of seven voltage values to further trade-off accuracy for energy (see Section 5.2.2).

5.2.1 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 accuracy constraint 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 5.4 describes the overall workflow of the accuracy-tuning phase. The heart of the accuracy-tuner is an autotuning search that uses statistical error injection to model potential run-time errors and directly executes the program on a standard CPU to measure the end-

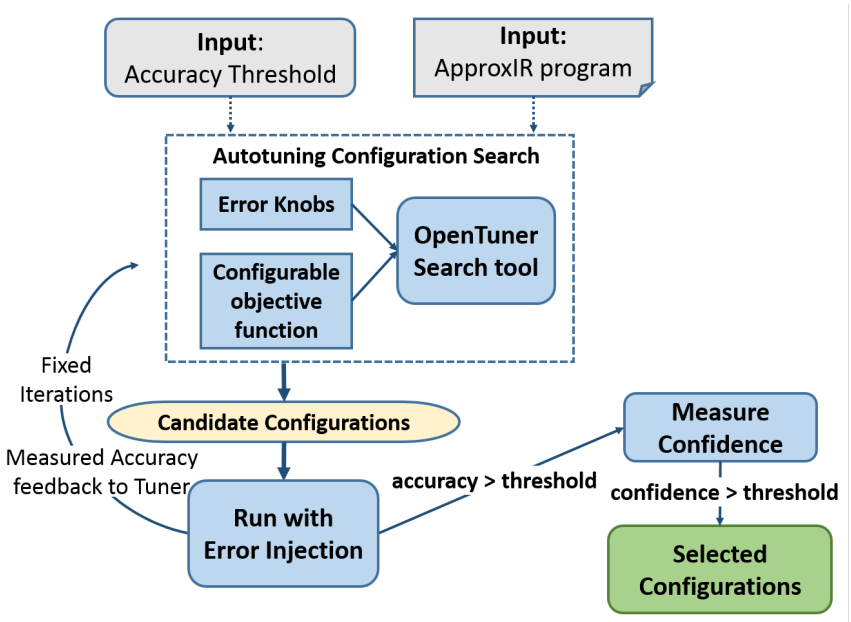


Figure 5.4: Hardware-agnostic accuracy-tuning workflow.

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 [149], an extensible framework for building domain-specific autotuners. OpenTuner allows users to configure a domain-specific search space and specify a custom objective function. Prior work has shown that OpenTuner provides promising results with enormous search spaces, exceeding 10^{3600} possible configurations. Leveraging Opentuner, we build our custom accuracy tuner that tunes the error knob for each tensor operation while minimizing an objective function. The objective functions we use are described below. In our experiments, we are able to extract high-quality configurations while searching through only a small subset of the full search space. For our experiments, we run OpenTuner for a total of 5000 iterations, where each iteration generates a unique configuration; the longest runs took about 2 hours.

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 5.1 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 pre-defined 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 [87], 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:

$$C_{Total}(config) = \sum_{i=0}^{N} C(op(i), E(i)) \tag{5.3}$$

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:

$$C_1(op, E) = \frac{N_c(op)}{\log E} \tag{5.4}$$

$$C_2(op, E) = \frac{N_c(op)}{E} \tag{5.5}$$

$$C_3(op, E) = \frac{N_c(op)}{E^2} \tag{5.6}$$

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.

5.2.2 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 [18]. The computations are offloaded to an NVIDIA Pascal GPU using the cuDNN library which supports both 32-bit (FP32) and 16-bit floating point (FP16) operations. Prior work has shown that FP16 computation reduces execution time and energy by 1.5-4x compared to FP32 [150, 151], at the cost of reduced accuracy.

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 [18], PROMISE consumes 3.4-5.5x lower 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.

Thus, in total we have 9 different choices (FP16, FP32 on GPU and P1, P2, ..., P7 on PROMISE) for mapping each tensor operation. Figure 5.5 shows the speedup, energy reduction, and accuracy of 3 hardware settings – P1, P7, and FP16. These are measured for a matrix multiplication of matrix M_1 of size $5000 \times K$ and matrix M_2 of size $K \times 256$, where $K \in \{2^8, 2^9, \ldots, 2^{15}\}$. The matrices are initialized with random values from uniform distribution $\mathcal{U}(0, 1)$. For readability, we do not show curves for P2-P6, which follow the same trends as P1 and P7. The left Y-axis shows speedup and energy reduction over FP32. The right Y-axis depicts error in the computation by showing L_1^e of the matrix multiplication for each hardware setting.

The graph shows that the L_1^e of different hardware settings remains constant for different

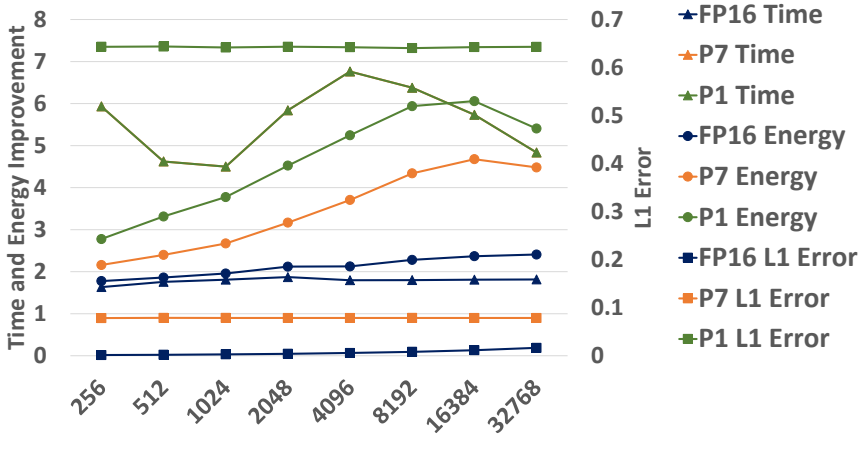


Figure 5.5: Time and energy improvement, and L_1^e for the following hardware knobs: FP16, P7, and P1. It can be seen that PROMISE is faster and less accurate than FP16, which is faster and less accurate than FP32.

values of K. FP16 is most accurate followed by P7 and P1 in that order. FP16 is also faster than P7 and P1 for all K and consumes more energy than P7 and P1, except for an anomaly for $K = 256, 512$. P7 and P1 curves for execution time overlap i.e., they execution time is same for different hardware settings for PROMISE. P1 however, is more imprecise and consumes less energy than P7. The hardware settings P2-P6 follow the same pattern.

Mapping L_1^e and L_2^e metrics to Hardware Setting: We generated similar graphs (similar to Figure 5.5) for all ApproxHPVM tensor operations for each hardware setting FP16, P7, P6, ...P1. These operations include tensor multiplication, addition, convolution, activations (tanh sigmoid,relu), and max-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. Thus, 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. This makes our hardware specific mapper very lightweight, which in future can be used for dynamic scheduling or for SoC design space exploration.

5.2.3 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, the backend support 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 (called a "task"). 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.). PROMISE can perform the entire layer computation as one PROMISE instruction [18]. 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.

5.3 METHODOLOGY

5.3.1 Platform

For our backend, we assume a modern System-on-Chip (SoC) architecture with different system agents (CPUs, GPUs, accelerators) that communicate via main memory. While a cycle-accurate integrated CPU-GPU-PROMISE simulator would be ideal to model such a system, we instead opted for a split approach to model the SoC. We ran the GPU tensor operations on a real mobile GPU and the PROMISE tensor operations on a 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 real SoC architecture.

For our GPU experiments, we used an NVIDIA Jetson TX2 developer kit [45]. This board contains the NVIDIA Tegra TX2 SoC [152], which has the same system architecture as our target SoC. For our PROMISE experiments, we obtained the simulator from its authors [18, 153] and extended it with a memory timing and energy model. Table 5.2 lists the relevant characteristics of both Tegra TX2 and the PROMISE simulator. Finally, due to

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

Table 5.2: System parameters for TX2 and PROMISE.

our split approach, the functional and timing aspects of our experiments were split as well.

5.3.2 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. We ran each configuration 200 times to obtain the mean and standard deviation of the classification accuracy, and $R_{success}$ of the configuration.

5.3.3 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 a DNN is running, the tool continuously reads GPU and DRAM power from Jetson's voltage rails via an I2C interface [154] at 1 KHz (1 ms period). Furthermore, it associates each GPU tensor operation with a begin and end timestamp pair. Once the DNN 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

Network	Layers	Network	Layers
LeNet	Conv, Conv, FC, FC	DNN2	FC, FC, FC
DNN1	FC, FC	DNN3	FC, FC, FC, FC

Table 5.3: Four DNNs and their layers. Conv is a convolution layer. FC is a fully connected layer.

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 how the input matrices will be mapped to PROMISE's banks, computes the time and energy of loading the data from main memory, computes the time and energy of performing the computation, and finally calculates the time and energy of writing data back to main memory. All memory transfers occur via a programmable DMA engine (pDMA) [155, 156]. PROMISE operates on INT8 data and requires a data layout transformation, both of which are handled by pDMA.

For the compute model, we used the pipeline parameters obtained from the authors of PROMISE [153]. 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. PROMISE performed quantization internally while a CUDA kernel performed quantization for the GPU.

5.3.4 Benchmarks

Our evaluations use four DNNs with structures shown in Table 5.3. Convolution layers map to Convolution, Bias add, Pooling, and Tanh tensor operations. Fully connected layers map to GEMM, Bias add, and Tanh tensor operations. The GPU performs one tensor operation at a time, while PROMISE computes the entire layer together (Section 5.2.3).

For each network, we studied an accuracy loss of 1% ($Loss_{1\%}$) and 2% ($Loss_{2\%}$). For each loss level, we evaluated the top three configurations – C1, C2, and C3 – provided by the accuracy-aware hardware scheduler. The baseline for all experiments was everything on FP32 with no approximation.

We used the MNIST [138] data set for all four DNNs. MNIST contains 10,000 28x28 pixel images of handwritten digits 0 through 9. We trained our networks on 5000 images from the data set and used the other 5000 for inference. Therefore, the input was a $5000 \times 28 \times 28$ tensor and the output was a 1×10 vector (one entry per digit).

5.4 EVALUATION

This section presents an evaluation of ApproxHPVM. We first present the aggregated results for speedup, energy reduction and accuracy on all 4 networks. We then go into the details by presenting the layer wise breakdown of LeNet in Section 5.4.2.

5.4.1 Performance and Energy Evaluation

Figures 5.6, 5.7, and 5.8 show the aggregate results for all four networks. They show the speedup and energy reduction over baseline, of all four networks for $Loss_{1\%}$ and $Loss_{2\%}$ experiments (higher is better). For each network, three configurations are shown. 5.6: Speedup. 5.7: Energy reduction. 5.8: Mean accuracy $\pm$ standard deviation. 5.9: Hardware knob mappings of configurations C1, C2, and C3 for each network. Hardware Settings – FP32: 32-bit floating point on GPU; FP16: 16-bit floating point on GPU; Px: PROMISE with swing x. Performance speedup and energy reduction are in comparison to the baseline (no approximation, all operations run on the GPU with 32-bit FP) and higher is better. For each network, we report mean classification accuracy with standard deviation over 200 runs of full batch of 5000 images in MNIST dataset. For each network, we show 3 bars – C1, C2, and C3 – which correspond to the top 3 configurations given by the ApproxHPVM framework. The specific mapping of C1, C2, and C3 to hardware settings for each network is shown in Figure 5.9.

The results shows that all configurations given by the ApproxHPVM framework achieve performance speedup and energy benefits. The performance speedup ranges from 2.7x (LeNet C3) to 20x (DNN1 C1) for $Loss_{1\%}$ and $Loss_{2\%}$ experiments. Similarly, we observe energy reduction of 1.8x (DNN3 C3) to 6.6x (DNN1 C1) for $Loss_{1\%}$ experiments and 2.5x (DNN3 C3) to 8.3x (DNN1 C1) for $Loss_{2\%}$ experiments. Figure 5.8 shows the mean and standard deviation of the accuracy of different configurations. The final accuracy of each configuration is within the accuracy loss threshold of 1% and 2%. We also measured the success rate $R_{Success}$ of our configurations by measuring the number of runs where the measured accuracy violated the accuracy loss threshold of $Loss_{1\%}$ and $Loss_{2\%}$. All our configurations achieved R of $> 95\%$, with an average $R_{Success}$ of 99.39% over all configurations. This shows that our hypothesis of decoupling accuracy, performance and energy tuning into hardware agnostic accuracy tuning, and hardware specific mapping of individual tensor operations yields configurations which benefit from approximation and yet remain within the constraints specified by the programmer. If we compare execution times for $Loss_{1\%}$ and $Loss_{2\%}$, we observe that ApproxHPVM finds configurations that achieve higher speedup and energy reduction for 2% accuracy loss.

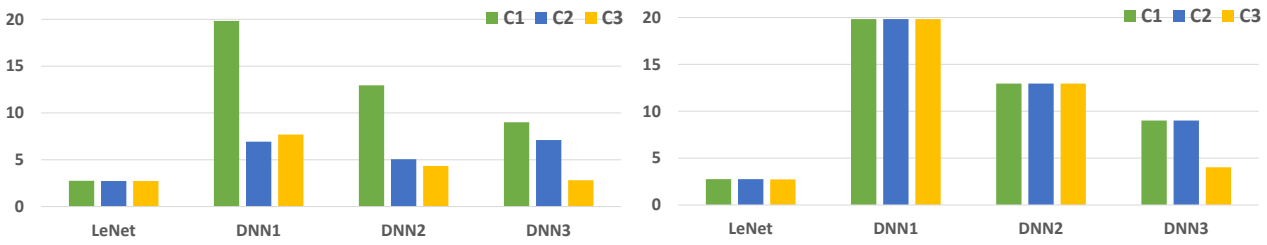


Figure 5.6: Speedup (a) $Loss_{1\%}$, (b) $Loss_{2\%}$

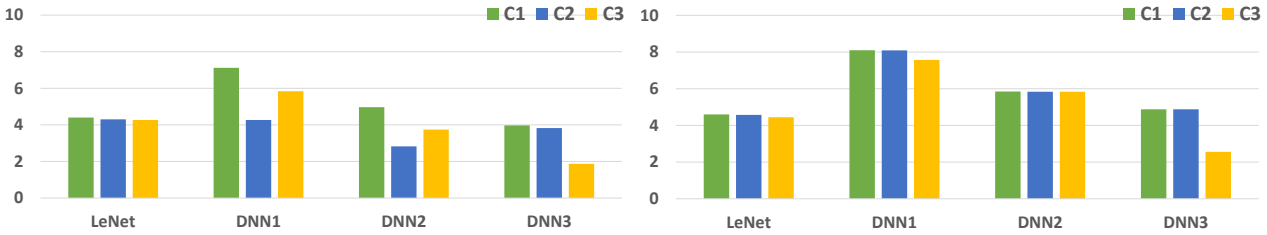


Figure 5.7: Energy Reduction (a) $Loss_{1\%}$, (b) $Loss_{2\%}$

	LeNet	DNN1	DNN2	DNN3
FP32	98.9	90.23	94.24	94.93
C1	98.83 ± 0.07	89.99 ± 0.22	94.06 ± 0.19	94.76 ± 0.22
C2	$98.87 \pm .04$	90.19 ± 0.13	94.13 ± 0.17	94.90 ± 0.15
C3	98.87 ± 0.06	90.18 ± 0.14	94.22 ± 0.14	94.96 ± 0.10

(a)

	LeNet	DNN1	DNN2	DNN3
FP32	98.9	90.23	94.24	94.93
C1	98.80 ± 0.09	89.49 ± 0.46	93.14 ± 0.48	94.10 ± 0.44
C2	98.81 ± 0.07	89.73 ± 0.27	93.27 ± 0.42	94.26 ± 0.36
C3	98.87 ± 0.06	89.64 ± 0.34	93.42 ± 0.40	94.09 ± 0.58

(b)

Figure 5.8: Mean classification accuracy (a) $Loss_{1\%}$, (b) $Loss_{2\%}$

	LeNet	DNN1	DNN2	DNN3
C1	FP16-P6-P6-P6	P7-P7	P7-P7-P7	P7-P7-P6-P7
C2	FP16-P6-P6-FP16	P7-FP16	P7-FP16-P7	P6-P7-P7-FP16
C3	FP16-P6-P7-FP16	P7-FP32	P7-FP32-FP32	P6-FP16-FP16-FP16

(a)

	LeNet	DNN1	DNN2	DNN3
C1	FP16-P4-P4-P6	P5-P6	P5-P4-P4	P4-P4-P4-P5
C2	FP16-P5-P4-P6	P5-P7	P5-P5-P4	P4-P4-P4-P6
C3	FP16-P4-P6-FP16	P6-P6	P5-P5-P5	P4-FP16-P4-P4

(b)

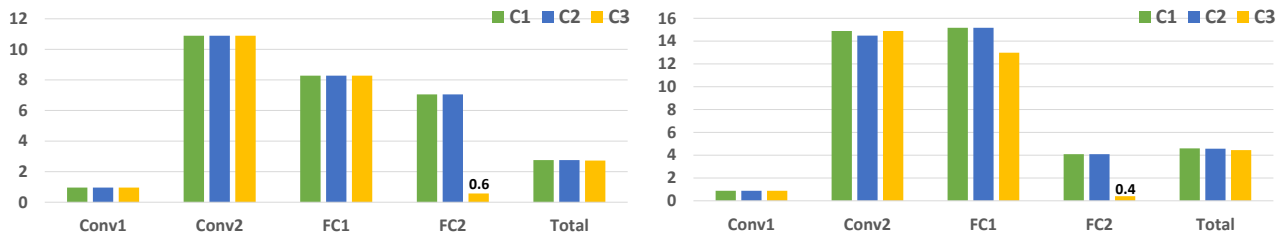
Figure 5.9: Configurations (a) $Loss_{1\%}$, (b) $Loss_{2\%}$

5.4.2 LeNet: Layer-wise Breakdown

Figure 5.10 shows the layer-wise breakdown of (a) performance and (b) energy benefits on LeNet for $Loss_{2\%}$ experiments, respectively. The layers Conv1, Conv2, FC1, and FC2 consume 28%, 58.3%, 13.4%, and 0.3% of the total execution time for baseline respectively. In the same order, the baseline energy breakdown of these layers is 14%, 65.7%, 20.1%, and 0.2%. The three configurations of LeNet (in the three bars) are listed in Figure 5.9 (b).

All three configurations map Conv1 to FP16 as it cannot be run on PROMISE. This is because PROMISE is highly imprecise for short vector (< 64 elements) dot products. Mapping the Conv1 layer to FP16, however, incurs overhead of casting from `float` (FP32) to `half` (FP16). This is the reason for the slight slowdown and energy increase in Conv1.

Conv2 is mapped to P4 for C1, C3 and P5 for C2. It gives 10.9x speedup, and 14.9x (C1, C3) and 14.4x (C2) energy reduction. FC1 is also mapped to PROMISE (P4 for C1, C2 and P6 for C3) and gives 8.3x speedup for all three configurations. It gets 12.9x energy



(a) (b)

Figure 5.10: Speedup (a) and energy reduction (b) over baseline of all four layers of LeNet for 3 different configurations ($Loss_{2\%}$). Note that C3's overall result is not affected by FC2, as FC2 constitutes only a small fraction of the network's computation.

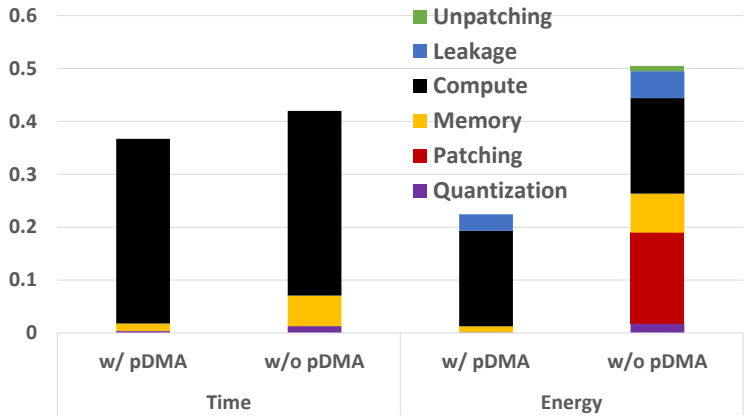


Figure 5.11: Normalized execution time and energy for LeNet with and without pDMA. Without pDMA, PROMISE relies on the GPU to perform quantization, patching, and unpatching, the overheads of which significantly reduce the performance and energy improvement over FP32.

reduction in C3 and 15x energy reduction in C1 and C2. This shows that mapping FC1 to P4 gets 14% further energy reduction over mapping it to P6.

The last layer FC2 is mapped to P6 for C1, C2 and to FP16 for C3. The computation time of this layer is small and the overheads of type conversion to FP16 make it 1.7x (2.5x) slower (higher energy consumption). Ideally FC2 should be mapped to FP32 avoiding the unnecessary type conversion overheads. However, our model assumes that FP16 is always faster and more energy efficient than FP32. We found this assumption largely holds except for some very small layers like FC2. This is not a major concern as this layer is less than 0.3% of the whole network in terms of execution time and has minimal impact on total execution time and energy. Overall, ApproxHPVM achieves a mean speedup of 2.75x and energy reduction of 4.5x on LeNet across all three configurations.

5.4.3 Hardware Sensitivity

To validate our choice of target hardware, we study the impact of pDMA and number of PROMISE banks on performance and energy.

pDMA: In deep learning, a convolution layer can be implemented using 4 different algorithms - Direct, GEMM, FFT, and Winograd. The GEMM-based approach 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. Both NVIDIA's cuDNN library [157] and PROMISE perform GEMM-based convolution (Section 5.2.3). The overhead of GEMM-based convolution consists of two data layout transformations: "patching" to generate matrices P_X and P_W, and "unpatching" to

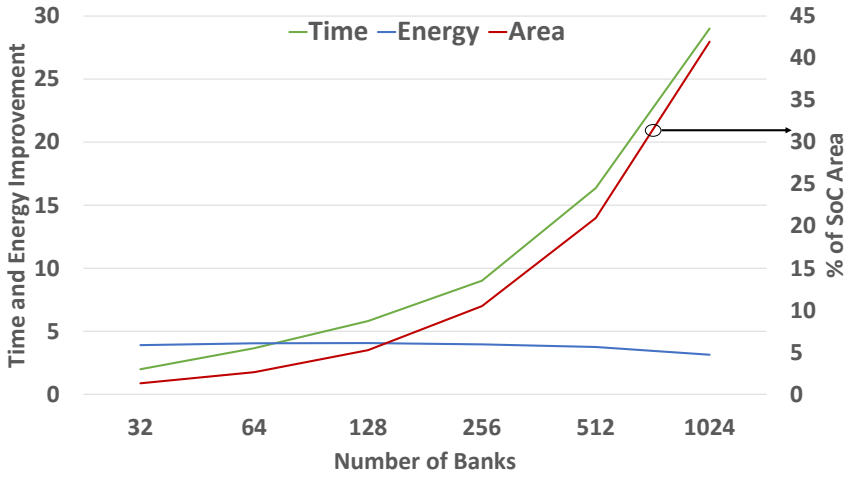


Figure 5.12: Speedup and energy reduction over FP32, and area for DNN3 vs the number of PROMISE banks. The SoC contains 4B transistors.

convert the GEMM's output to the application's desired format.

In cuDNN, patching and unpatching are done in on-chip memory to minimize this overhead [157]. In PROMISE, we can either use the pDMA scheme described in Section 5.3.3 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 5.11 shows execution time and energy, normalized to FP32, for LeNet configuration FP16-P4-P6-FP16 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 25x larger than the input matrix) causes both time and energy to increase further. Nonetheless, PROMISE without pDMA still achieves a 2.38x speedup and 1.98x energy reduction compared to FP32. While the benefits are higher with pDMA , these results show that our approach is effective regardless of which method is used.

#Banks: We performed a scaling study of the number of PROMISE banks to establish the suitability of a 256 bank configuration. Figure 5.12 shows the execution time and energy of DNN3 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.

CHAPTER 6: CONCLUSIONS AND FUTURE DIRECTIONS

6.1 CONCLUSIONS

For many years, Moore's law and Dennard's scaling helped architects design faster and more energy efficient processors. With Moore's law slowing down and end of Dennard's scaling the microprocessor industry and research community is betting on heterogeneous systems with specialized compute units to meet the performance and energy efficiency requirements of modern workloads. However, the diversity of underlying compute units makes them hard to program.

In this thesis, we list what the programmability challenges are and identify four root causes behind them: (1) diversity of parallelism models, (2) diversity of memory hierarchy, (3) diversity of instructions sets, and (4) different software and hardware approximate computing methods. We propose prototype systems to address these root problems. In order to better understand the heterogeneity of future accelerators we worked on a programmable analog machine learning accelerator, PROMISE, based on deep in-memory computing [106]. In summary we made the following contributions.

- **Efficient compilation to heterogeneous parallel hardware:** We proposed HPVM, a parallel program representation that can map down to diverse parallel hardware. HPVM is a hierarchical dataflow graph with side effects and vector instructions. We present a prototype system based on the HPVM parallelism model to define a compiler IR, a virtual instruction set, and a flexible scheduling framework. We implement two optimizations as transforms on the HPVM IR — node fusion and tiling —, and translators for NVIDIA's GPUs, Intel's AVX vector units, and multicore X86-64 processors. Our experiments show that HPVM achieves performance portability across these classes of hardware and significant performance gains from the optimizations, and is able to support highly flexible scheduling policies.

- **Energy efficient computation of ML inference algorithms:** We presented PROMISE, the first end-to-end design of a programmable mixed-signal accelerator for diverse ML algorithms. PROMISE accomplishes a high level of programmability without losing the efficiency of mixed-signal accelerators for specific ML algorithms. We designed the PROMISE ISA to allow software control over energy-vs-accuracy tradeoffs, and supports compilation of high-level languages like Julia down to the hardware. Our evaluation shows better energy efficiency than digital ASICs, despite much greater

programmability, and significant energy savings from small programmer-specified error tolerances.

- **Portable Approximate Computing:** We propose ApproxHPVM, an extension of HPVM, a promising basis for achieving performance portability and for implementing parallelizing compilers and schedulers for heterogeneous parallel systems. 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 a accuracy-tuning analysis that lowers the accuracy requirements of IR operations given an end-to-end accuracy constraint, while the hardware scheduling phase uses the extracted constraints to map to target hardware. Our results show that ApproxHPVM provides promising results on a heterogeneous target platform with multiple hardware compute units. Overall, across four benchmarks, we observe performance speedups ranging from 2.8x to 20x and energy reduction ranging from 1.9x to 7.1x for an accuracy loss of 1%. 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 be mapped to software techniques for approximation.

Overall, this thesis makes fundamental contributions in addressing the programmability of heterogeneous systems. Not only does it address the diversity of parallelism models, memory hierarchies and instruction sets, but also takes a step in addressing the diversity in approximate computing software and hardware techniques (an emerging challenge).

6.2 FUTURE DIRECTIONS

6.2.1 HPVM Compilation to FPGAs

HPVM was designed to be compiled to and generate efficient code for FPGAs as well. However, currently the HPVM infrastructure does not have a FPGA beckend and hence that claim has not been quantitatively evaluated in this thesis.

HPVM IR is a dataflow graph and it has been shown that dataflow representations map well to FPGAs [158, 159]. This is how its features map well to FPGAs: (1) DAG representation maps well to a hardware pipeline. (2) Streaming edges between HPVM IR nodes representing explicit data transfers between kernels map well to hardware FIFOs. (3) Edges representing data transfers between host and kernels map well to hardware load/store units.

(4) Dynamic instances of nodes maps well to spatial replication of kernels in FPGA. (5) Multiple independent nodes map well to different sections of the FPGA, and can therefore run in parallel.

Since HPVM also serves as a compiler IR, future research can explore FPGA specific optimizations given the hardware details of the target FPGA. In our research group, Adel Ejjeh (aejjeh@illinois.edu) is working on using HPVM infrastructure to compile to FPGAs.

6.2.2 HPVM Extensions for Complex Memory Hierarhcy

HPVM abstractions are well suited to handle the diversity of parallelism models, instruction sets. It also expresses tiling effectilvely and portably using graph hierarchy. However, there is scope for further research in the memory abstractions. Legion [70, 71] decouples logical regions from their physical layout and mapping. It allows programmers to explicitly declare properties of program data including organization, partitioning, privileges, and coherence. In HPVM we have allocate memory regions using the `llvm.hpvm.malloc` intrinsic. Adding information related to how this memory is accessed, partitioned can allow further optimizations and is particularly useful for memory bound applications running on distributed memory heterogeneous systems.

6.2.3 HPVM Extensions for Dynamically Varying Parallel Structure

HPVM dataflow graphs are currently static and cannot change at runtime. This is sufficient for many programs, even highly irregular ones that require dynamic load balancing, because the assignment of node instances to compute elements or threads does not change the graph structure itself. However, programs that change parallelism structure or communication patterns while they execute would require run-time modifications to the graph, e.g., when new types of tasks are spawned or patterns of communication edges are created during an execution in a data-dependent manner, such as in Delaunay Mesh Refinement, where the neighbors of a vertex evolve as the computation proceeds, requiring HPVM edges to be added or deleted at run-time. IR, code generation and run-time changes need to be explored to support dynamic changes to HPVM graphs. Most importantly, the assignment of graph node instances to compute elements and/or the assignment of shared memory to compute nodes may need to be performed at runtime, which can require more sophisticated runtime scheduling as well as more flexible communication mechanisms

6.2.4 Efficient Scheduling Policy of HPVM Dataflow Graph

In the evaluation section in Chapter 3 we showed how HPVM IR as a flexible runtime representation can be used to do ensure progress in the event a preferred compute unit becomes unavailable due to load from other applications. The HPVM runtime representation provides fine grained control to the runtime scheduler. Although we do present a scenario showing the benefits of such flexibility, implementation of a sophisticated scheduling policy especially in case of dynamically varying dataflow graph is an open problem to solve.

6.2.5 Aggressive Hardware Dependent Tuning in ApproxHPVM

A key feature of ApproxHPVM is that it decouples autotuning into a hardware-agnostic stage and a hardware dependent stage. The output of the hardware agnostic stage is multiple configurations where each configuration annotates the nodes in the ApproxHPVM dataflow graph with the maximum error it can tolerate (in terms of the error metric). In the hardware dependent tuning stage, we map a subgraph of ApproxHPVM IR to a compute unit. The mapping involves choosing the right configuration on the hardware unit corresponding to the error metric with the goal of optimizing for performance and energy efficiency. In it's current form, this mapping is very conservative, choosing the configuration which satisfies the error metric of all the nodes in the subgraph. However, as the errors would compose, we can be more aggressive in choosing the configuration at the same time satisfying the end-to-end error of the subgraph. If one can predict how errors compose in a subgraph when mapped to a particular hardware unit, we should be able to further improve the ApproxHPVM benefits.

REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures," in *Proceedings of 27th International Symposium on Computer Architecture*, June 2000, pp. 248–259.

[2] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, July 2008.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012.

[4] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 225–236.

[5] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *In Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA 2010)*, 2010. [Online]. Available: http://www.duke.edu/~BCL15/documents/hameed2010-isca-h264.pdf

[6] Y. Hoskote, B. A. Bloechel, G. E. Dermer, V. Erraguntla, D. Finan, J. Howard, D. Klowden, S. G. Narendra, G. Ruhl, J. W. Tschanz, S. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar, "A tcp offload accelerator for 10 gb/s ethernet in 90-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1866–1875, Nov 2003.

[7] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," *Commun. ACM*, vol. 51, no. 7, pp. 91–97, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364802

[8] Y.-H. Chen, J. Emer, and V. Sze, "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, jun 2016, pp. 367–379.

[9] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[10] M. Kang, S. K. Gonugondla, A. Patil, and N. R. Shanbhag, "A Multi-Functional In-Memory Inference Processor Using a Standard 6T SRAM Array," *JSSC*, 2018.

[11] P. Greenhalgh, "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, vol. 17, 2011.

[12] Qualcomm, "Hexagon snapdragon." [Online]. Available: https://www.qualcomm.com/products/snapdragon-855-mobile-platform

[13] Qualcomm, "Hexagon digital signal processor." [Online]. Available: https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor

[14] Qualcomm, "Adreno graphics processing units." [Online]. Available: https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu

[15] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1941487.1941507

[16] C. Lynch, "Big data: How do your data grow?" *Nature*, vol. 455, no. 7209, p. 28, 2008.

[17] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 62, 2016.

[18] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[19] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 505–516, 2014.

[20] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti et al., "Exploiting errors for efficiency: A survey from circuits to algorithms," *arXiv preprint arXiv:1809.05859*, 2018.

[21] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

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

[23] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 441–454.

[24] Khronos Group, "OpenCL," http://www.khronos.org/opencl/.

[25] E. Stotzer, "Tutorial: OpenMP accelerator model," Tech. Rep., 2014.

[26] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, "Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment," in *Proceedings of the 7th international conference on High performance computing for computational science*, ser. VECPAR'06. Berlin, Heidelberg: Springer-Verlag, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1761728.1761755 pp. 305–318.

[27] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," ser. PLDI, 2009.

[28] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A programming model for heterogeneous multi-core systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346318 pp. 287–296.

[29] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, June 2004, pp. 69–70.

[30] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469 pp. 89–108.

[31] M. Okamoto, K. Yamashita, H. Kasahara, and S. Narita, "Hierarchical macro-dataflow computation scheme," in *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings.* IEEE, 1995. [Online]. Available: https://doi.org/10.1109/pacrim.1995.519406

[32] Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara, *A Parallelizing Compiler Cooperative Heterogeneous Multicore Processor Architecture.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[33] N. Benoit and S. Louise, "Using an intermediate representation to map workloads on heterogeneous parallel systems," in *24th Euromicro Conference*, 2016.

[34] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt, "SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension," ser. CPC, 2012.

[35] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into llvm's intermediate representation," ser. PPoPP, 2017.

[36] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," in *U. Washington, Tech. Rep. UW-CSE- 15-01-01*, 2015.

[37] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," ser. PLDI, 2011.

[38] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 309–328.

[39] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509546 pp. 33–52.

[40] N. R. Shanbhag. [Online]. Available: http://shanbhag.ece.illinois.edu

[41] L. Project, "LLVM Language Reference Manual," 2003. [Online]. Available: http://llvm.org/docs/LangRef.html

[42] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[44] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017. [Online]. Available: http://proceedings.mlr.press/v70/sakr17a.html pp. 3007–3016.

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

[46] R. S. Nikhil, "The parallel programming language Id and its compilation for parallel machines," ser. IJHSC, 1993.

[47] D. Culler, S. Goldstein, K. Schauser, and T. Voneicken, "TAM - a compiler controlled threaded abstract machine," 1993.

[48] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Commun. ACM*, 1977.

[49] Google, "Google Cloud Dataflow," 2013. [Online]. Available: https://cloud.google.com/dataflow/

[50] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Supporting stateful tasks in a dataflow graph," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 435–436.

[51] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Integrating dataflow abstractions into the shared memory model," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, Oct 2012, pp. 243–251.

[52] J. Zhou and B. Demsky, "Bamboo: A data-centric, object-oriented approach to many-core software," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 388–399.

[53] H. Foundation, "HSAIL," 2015. [Online]. Available: http://www.hsafoundation.com/standards/

[54] Khronos Group, "SPIR 1.2 specification," https://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf, 2012.

[55] T. Miyamoto, S. Asaka, H. Mikami, M. Mase, Y. Wada, H. Nakano, K. Kimura, and H. Kasahara, "Parallelization with automatic parallelizing compiler generating consumer electronics multicore API," in *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, dec 2008.

[56] N. Benoit and S. Louise, "Extending GCC with a Multi-grain Parallelism Adaptation Framework for MPSoCs," in *2nd Int'l Workshop on GCC Research Opportunities*, 2010.

[57] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT press, 1994.

[58] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, 2011.

[59] Qualcomm Technologies, Inc., "MARE: Enabling applications for heterogeneous mobile devices," White Paper, Tech. Rep., 2014.

[60] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, W. R. van der, and V. Sarkar, "OCR: The open community runtime interface," Tech. Rep., September 2015.

[61] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 235–248.

[62] NVIDIA, "CUDA Toolkit Documentation v7.5," http://docs.nvidia.com/cuda/.

[63] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science and engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[64] "OpenACC-Standard." [Online]. Available: http://www.openacc-standard.org/

[65] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 138–149.

[66] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," ser. International Conference on Compiler Construction, 2002.

[67] Z. Budimlic, M. Burke, V. Cav, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent Collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[68] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical place trees: A portable abstraction for task parallelism and data movement," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 172–187.

[69] D. Majeti and V. Sarkar, "Heterogeneous habanero-c (h2c): A portable programming model for heterogeneous processors," ser. IPDPS Workshop, 2015.

[70] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," ser. SC, 2012.

[71] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628084 pp. 263–276.

[72] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," ser. SC, 2006.

[73] L. Chang, A. Dakkak, C. I. Rodrigues, and W. mei Hwu, "Tangram: a high-level language for performance portable code synthesis," ser. MULTIPROG 2015, 2015.

[74] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," ser. ACM TECS, 2014.

[75] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," ser. ICS, 2006.

[76] S. Chakradhar, A. Raghunathan, and J. Meng, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications," ser. IPDPS, 2009.

[77] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," ser. IPDPS, 2010.

[78] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," ser. ICSE, 2010.

[79] S. Misailovic, D. Roy, and M. Rinard, "Probabilistically accurate program transformations," ser. SAS, 2011.

[80] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," ser. PLDI, 2010.

[81] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 85–96.

[82] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," ser. ASPLOS, 2014.

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

[84] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," ser. POPL, 2012.

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

[86] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," ser. PLDI, 2014.

[87] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, p. 88, 2013.

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

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

[90] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[91] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 393–405.

[92] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[93] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun et al., "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[94] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[95] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice.* ACM, 2009, vol. 44, no. 6.

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

[97] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *International Conference on Machine Learning*, 2017, pp. 3007–3016.

[98] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *Proceedings of the 43rd International Symposium on Computer Architecture.* IEEE Press, 2016, pp. 255–266.

[99] W. Rieutort-Louis, T. Moy, Z. Wang, S. Wagner, J. C. Sturm, and N. Verma, "16.2 a large-area image sensing and detection system based on embedded thin-film classifiers," in *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International.* IEEE, 2015, pp. 1–3.

[100] S. Joshi, C. Kim, S. Ha, Y. M. Chi, and G. Cauwenberghs, "21.7 2pj/mac 14b 8× 8 linear transform mixed-signal spatial filter in 65nm cmos with 84db interference suppression," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International.* IEEE, 2017, pp. 364–365.

[101] E. H. Lee and S. S. Wong, "24.2 a 2.5 ghz 7.7 tops/w switched-capacitor matrix multiplier with co-designed local memory in 40nm," in *Solid-State Circuits Conference (ISSCC), 2016 IEEE International.* IEEE, 2016, pp. 418–419.

[102] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 44. IEEE, jun 2016, pp. 27–39.

[103] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 44. IEEE, jun 2016, pp. 14–26.

[104] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An Instruction Set Architecture for Neural Networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 44. IEEE, jun 2016, pp. 393–405.

[105] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, Z. Xuehai, and Y. Chen, "PuDianNao : A Polyvalent Machine Learning Accelerator," *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 43, pp. 369–381, mar 2015.

[106] M. Kang, M.-S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 8326–8330.

[107] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.

[108] NVIDIA, "PTX: Parallel Thread Execution ISA," 2009, http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[109] NVIDIA, "NVVM IR," http://docs.nvidia.com/cuda/nvvm-ir-spec, 2013.

[110] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2002.

[111] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep., 2012.

[112] Li-wen Chang, "Personal communication," August 2015.

[113] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao Family: Energy-efficient accelerators for machine-learning," *Communications of the ACM*, vol. 59, no. 11, pp. 105–112, November 2016.

[114] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, "A 28nm SoC with a 1.2 GHz 568nJ/prediction sparse deep-neural-network engine with¿ 0.1 timing error rate tolerance for IoT applications," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 242–243.

[115] H. Kaul, M. A. Anders, S. K. Mathew, G. Chen, S. K. Satpathy, S. K. Hsu, A. Agarwal, and R. K. Krishnamurthy, "14.4 a 21.5 m-query-vectors/s 3.37 nj/vector reconfigurable k-nearest-neighbor accelerator with adaptive precision in 14nm tri-gate cmos," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 260–261.

[116] M. Kang, S. K. Gonugondla, and N. R. Shanbhag, "A 19.4 nj/decision 364k decisions/s in-memory random forest classifier in 6t sram array," in *ESSCIRC 2017 - 43rd IEEE European Solid State Circuits Conference*, Sep. 2017, pp. 263–266.

[117] J. Zhang, Z. Wang, and N. Verma, "In-memory computation of a machine-learning classifier in a standard 6t sram array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 915–924, April 2017.

[118] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 481–492.

[119] Y. Huang, N. Guo, M. Seok, Y. Tsividis, K. Mandli, and S. Sethumadhavan, "Hybrid analog-digital solution of nonlinear partial differential equations," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124550 pp. 665–678.

[120] F. N. Buhler, P. Brown, J. Li, T. Chen, Z. Zhang, and M. P. Flynn, "A 3.43 tops/w 48.9 pj/pixel 50.1 nj/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm cmos," in *VLSI Circuits, 2017 Symposium on*. IEEE, 2017, pp. C30–C31.

[121] R. Genov and G. Cauwenberghs, "Kerneltron: support vector "machine" in silicon," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 1426–1434, Sept 2003.

[122] Python Core Team, *Python: A dynamic, open source programming language*, Python Software Foundation, 2015. [Online]. Available: https://www.python.org

[123] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

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

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

[126] M. Kang, S. K. Gonugondla, M.-S. Keel, and N. R. Shanbhag, "An energy-efficient memory-based high-throughput VLSI architecture for Convolutional Networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2015.

[127] J. Zhang, Z. Wang, and N. Verma, "A machine-learning classifier implemented in a standard 6t sram array," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, June 2016, pp. 1–2.

[128] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *International Conference on Field Programmable Logic and Applications*, 2009, pp. 32–37.

[129] J. Lee, D. Shin, Y. Kim, and H.-J. Yoo, "A 17.5-fj/bit energy-efficient analog sram for mixed-signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[130] M. Kang, S. K. Gonugondla, and N. R. Shanbhag, "A 19.4 nJ/decision 364K decisions/s In-memory Random Forest Classifier in 6T SRAM Array," in *IEEE European Solid-State Circuits Conf. (ESSCIRC)*, Sept 2017.

[131] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2^{nd} Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[132] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[133] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas et al., "Theano: A python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[134] F. Chollet et al., "Keras," https://github.com/fchollet/keras, 2015.

[135] The Apache Software Foundation (ASF), "MXNet.jl," http://dmlc.ml/MXNet.jl/latest, 2015.

[136] M. J. Innes, "Flux: The Julia Machine Learning Library," https://github.com/FluxML/Flux.jl, 2015.

[137] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[138] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist, 1998.

[139] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR, abs/1502.02551*, vol. 392, 2015.

[140] B. Murmann, D. Bankman, E. Chai, D. Miyashita, and L. Yang, "Mixed-signal circuits for embedded machine-learning applications," in *2015 49th Asilomar Conference on Signals, Systems and Computers*. IEEE, 2015, pp. 1341–1345.

[141] Production Crate, "Gun Shot Sounds," `http://soundscrate.com/gun-related/`.

[142] Center for Biological and Computational Learning, MIT, "CBCL Face Database No. 1," 2001, `http://cbcl.mit.edu/software-datasets/index.html`.

[143] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38. ACM, 2010, pp. 37–47.

[144] ITRS, "ITRS Roadmap," http://www.itrs2.net/, 2015.

[145] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner et al., "RAZOR: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003, pp. 7–18.

[146] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 113.

[147] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178493 pp. 68–80.

[148] "Domain-specific compiler for linear algebra to optimize tensorflow computations," https://www.tensorflow.org/xla/overview, 2018.

[149] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

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

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

[152] NVIDIA, "NVIDIA Tegra TX2 SoC," https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge, 2018.

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

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

[155] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, P. Srivastava, M. Kotsifakou, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache it Too," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 707–719.

[156] D. A. Jamshidi, M. Samadi, and S. Mahlke, "D2MA: Accelerating Coarse-grained Data Transfer for GPUs," in *PACT*, 2014.

[157] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[158] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192379 pp. 296–311.

[159] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872415 pp. 651–665.