HPVM: Hardware-Agnostic Programming for Heterogeneous Parallel Systems

Adel Ejjeh
Aaron Councilman
Akash Kothari
Maria Kotsifakou*
Leon Medvinsky
Abdul Rafae Noor
Hashim Sharif
Yifan Zhao
Sarita Adve
Sasa Misailovic
Vikram Adve

{aejjeh,aaronjc4,akashk4,leonkm2,arnoor2,hsharif3,yifanz16,sadve,misailo,vadve}@illinois.edu
University of Illinois at Urbana-Champaign
Computer Science
Urbana, IL

*maria.kotsifakou@runtimeverification.com
Runtime Verification, Inc.
Champaign, IL

***Abstract*—We present Heterogeneous Parallel Virtual Machine, or HPVM, a compiler framework for hardware-agnostic programming on heterogeneous compute platforms. HPVM introduces a hardware-agnostic parallel Intermediate Representation (IR) with constructs for hierarchical task, data, and pipeline parallelism, including dataflow parallelism, and supports multiple front-end languages. Additionally, HPVM provides optimization passes that navigate performance, energy, and accuracy tradeoffs, and includes retargetable back ends for a wide range of diverse hardware targets including CPUs, GPUs, domain-specific accelerators, and FPGAs. Across diverse hardware platforms, HPVM optimizations provide significant performance and energy improvements, while preserving object code portability. With these capabilities, HPVM facilitates developers, domain experts, and hardware vendors in programming modern heterogeneous systems.**

HPVM: Hardware-Agnostic Programming for Heterogeneous Parallel Systems

WITH the slowdown of Moore's Law and the end of Dennard scaling, heterogeneous architectures are increasingly dominating the systems used for modern applications. These systems have been evolving to include a plethora of compute- and energy-efficient processing elements (PEs), ranging from GPUs and FPGAs to fixed-function and programmable domain-specific accelerators. Enabling *hardware-agnostic* programming of these heterogeneous systems is important to facilitate their use by a broad range of software developers. By *hardware-agnostic*, we mean that the entire programming process, including the software itself and the iterative manual development and tuning processes, should not be specific to a particular target system. Moreover, any system-specific performance tuning should be automated by the compilation flow as much as possible, and if that is not entirely possible, the programming process should minimize the need for changes to the application source code.

Currently, programming such heterogeneous devices presents many challenges, including the need to use diverse hardware-specific programming languages, poor source code portability, lack of object-code portability which is essential in certain domains such as mobile applications, and the need for system-specific performance tuning [1]. Moreover, in many emerging application domains (e.g., video analytics, AR/VR, mobile robotics, etc.), it is also crucial – but challenging for application developers – to increase performance and/or energy efficiency by sacrificing small amounts of accuracy or application quality.

We believe that the solution to these challenges lies in a suitable compiler and autotuning infrastructure, with the right abstractions of parallelism, that allows seamless compilation of *hardware-agnostic* code into multiple general-purpose and/or specialized hardware targets. Additionally, the compiler must automatically optimize programs using efficient hardware-level primitives and domain-specific transformations while requiring minimal source-code tuning by the developer for performance. This can be achieved by separating the hardware-agnostic functional specification from the hardware-specific tuning, which itself can be automated using autotuning.

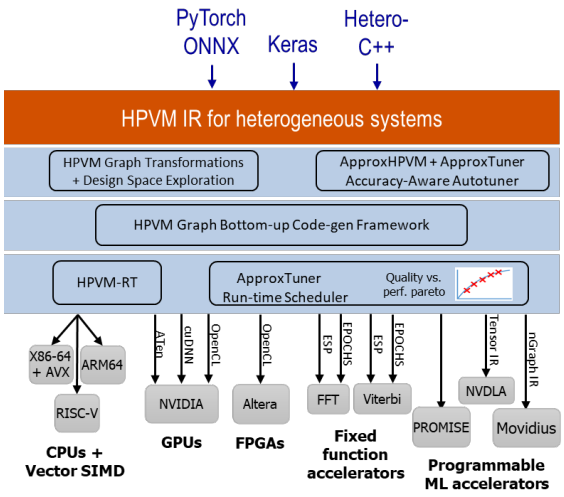

Figure 1: HPVM Compiler Infrastructure. General-purpose applications use HeteroC++ and are compiled using graph transformations and autotuning, and run using HPVM-RT. Tensor-domain applications use Keras, PyTorch or ONNX, compiled to the ApproxHPVM tensor IR extensions, and optimized by ApproxTuner using static and dynamic approximation-tuning.

We present Heterogeneous Parallel Virtual Machine (HPVM) [2], a compiler framework that achieves the aforementioned goals. HPVM addresses all the programming challenges by providing a **compiler IR** design and code-generation framework that is *retargetable* to a wide range of heterogeneous parallel architectures, **hardware-agnostic language front ends** that support ease of programming, and a **domain- and hardware-specific** optimization framework that automatically navigates performance, energy, and accuracy tradeoffs. HPVM's flexibility allows **programmers** to easily write and optimize code for heterogeneous platforms, while giving **hardware vendors** the ability to easily extend the compiler

infrastructure with new hardware back ends and corresponding optimizations.

No existing infrastructure we know of provides the combination of features needed for hardware-agnostic programming of a broad range of accelerator-based systems, including a flexible abstraction of parallelism, source- and object-code portability, and the separation of functionality from (automatable) performance tuning and accuracy-aware approximation tuning. Spatial [3] is based on a valuable parallelism abstraction, but emphasizes hardware design. TVM [4] supports diverse hardware targets, but is narrowly focused on machine learning. Perhaps the best alternative is MLIR, which is flexible enough to support a wide range of compiler goals and all missing features could be added in future. At present, it lacks a virtual object code format essential for object code portability in accelerator-based systems. It also lacks an approximation framework (including diverse approximations, support for approximation tuning and dynamic adaptation of approximations), which is crucial for many emerging applications on heterogeneous systems. We believe it would be valuable to add new "dialects" to MLIR based on the techniques described here in order to provide these missing capabilities.

HPVM Overview

HPVM is an open-source[1] retargetable compiler infrastructure that uses a common abstraction of parallelism to define the compiler intermediate representation (IR), a virtual instruction set architecture (ISA), and a runtime system. Figure 1 shows the HPVM compiler stack. HPVM's parallel abstraction is designed to capture the multiple forms of parallelism available on heteregeneous systems, in a hardware-agnostic manner.

The HPVM IR is currently built on top of LLVM, i.e., it uses LLVM IR to represent (scalar and vector) computations. HPVM is a modular framework that facilitates compiler optimizations, and benefits from the advanced optimization and code-generation capabilities offered by LLVM. Our ApproxHPVM [5] and ApproxTuner [6] extensions provide a tensor-domain programming model and enable tradeoffs between accuracy

[1]https://gitlab.engr.illinois.edu/llvm/hpvm-release/

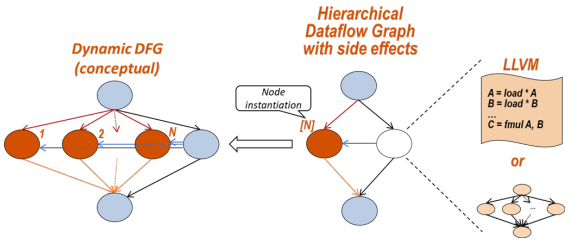

Figure 2: HPVM DFG Representation.

and performance/energy for tensor applications, which we describe later. With that, HPVM provides a framework that can seamlessly integrate domain-specific tensor code with general-purpose code.

Starting with hardware-agnostic code written in a supported language – at present, these include a parallel dialect of C++ (HeteroC++) for general-purpose parallel code, and Keras, PyTorch, and ONNX for deep learning – HPVM's front ends translate the parallelism in the application into equivalent code using HPVM's parallel IR abstractions. HPVM's optimization frameworks, which support the optimization needs of different domains, tune the code for the target system. Finally, code is generated for individual (specified or default) target devices and for the host CPU to generate executable binaries, leveraging existing LLVM code generators for each device wherever possible. Optional features enable design space exploration for FPGAs [7] and GPUs, and an approximation autotuning and dynamic adaptation framework for tensor programs.

HPVM Parallel Abstraction and IR

A parallel program is represented in HPVM as a host program plus one or more acyclic dataflow graphs (DFGs) (Figure 2). Each DFG is hierarchical: a node is either a *leaf node* representing a unit of computation, or an *internal node* containing an entire "child" dataflow graph. Edges in the DFG represent explicit, logical data transfer between nodes, with blocking semantics, i.e., the sink node of an edge will block if it attempts to access the data coming in via that edge. Leaf nodes can include references to a global shared memory. HPVM programs are *assumed* to be data race free, i.e., explicit ordering through DFG edges or fine-grain synchronization operations must be used to enforce ordering or mutual exclusion

between conflicting accesses to the same shared memory location.

Each node in a DFG has one or more dynamic instances, represented using a dynamic replication factor (e.g. for the iterations of a parallel loop). Node instances are assumed to be fully independent, i.e., safe to run in parallel. At the IR level, hints from the source program or front end are used to specify which device a node must be targeted to, although an automatic partitioning system built on the autotuner described here will allow automatic assignment of nodes to devices.

HPVM Compiler Internal Representation

The HPVM compiler IR is a direct implementation of the hierarchical graph representation, using LLVM IR for both the host program and the node computations. Every HPVM node is represented by an ordinary LLVM function. The HPVM IR defines a set of LLVM "intrinsic functions" to describe the details of the dataflow graphs, as well as host operations to launch a graph computation asynchronously, wait for completion, and to transfer inputs and outputs to and from a graph (either as one-time transfers or as streams). An LLVM-to-DFG translation pass converts the intrinsics-based description into explicit graph data structures, before graph transformations and back-end code generation occur.

HPVM leaf nodes use LLVM scalar and vector instructions for individual computational tasks, and can use LLVM atomics and other synchronization operations to ensure data race freedom. Just like with LLVM, an HPVM IR program is a fully self-contained and executable "virtual object code" program, which can be used to achieve object code portability (via install-time translation) for any targets that support portable LLVM IR.

ApproxHPVM and Tensor Extensions

We extend the LLVM IR used by HPVM with higher level tensor intrinsics such as matrix multiplication and convolution. This enables HPVM to retain domain specific semantics for tensor application domains such as deep learning and image processing. We leverage these intrinsics for three purposes: (a) domain specific optimizations, such as operator fusion; (b) code generation targeting machine learning accelerators, such as Movidius and NVDLA; and (c) accuracy-aware optimizations for tensor operations, which apply and tune software and hardware approximation techniques on the tensor operators systematically such that user-specified quality of service (QoS) constraints are respected.

```
void* Section = __hetero_section_begin();
for(int i = 0; i < left_dim ; i++){
        for(int j = 0; j < right_dim; j++){
          __hetero_parallel_loop(
          /* Num Parallel Enclosing Loops */ 2,
          /* Num Input Objects */ 6,
          Res, Res_Size, V1, V1_Size,  V2, V2_Size,
          left_dim, right_dim, common_dim,
          /* Num Output Objects */ 1,
          Res, Res_Size ,
          /* Optional Node Name */ "matmul_parallel_loop" );
          __hetero_hint(/* TARGET DEVICE  */ Target::GPU);
          Res[i*right_dim + j] = 0;
          for(int k = 0 ; k < common_dim; k ++){
            // Res[i,j] += V1[i,k] + V2[k,j]
            Res[i*right_dim + j] += V1[i*common_dim + k] * V2[k*right_dim +j];
          }
}
 __hetero_section_end(Section);
```

2 Dimensional Parallel Loops corresponding 2D HPVM Node with dynamic replication factor: `(left_dim x right_dim)`

Specifying the incoming objects (fat pointers or scalars) and the outgoing objects (fat pointers or scalars) for Parallel Loop

Source level annotation specifying where to offload computation

Figure 3: Matrix Multiply written in HeteroC++. The outer two loops over i and j are parallel, whereas the innermost loop executes sequentially over k. Note that a pointer and the size of the allocated memory it points to together constitute a single input/output object.

HPVM Front Ends

HeteroC++ Front End

HeteroC++ is an experimental parallel dialect of C/C++ that enables easy parallelization using HPVM. Computations in HeteroC++ are described as parallel tasks or parallel loops, which are lowered to nodes in the HPVM DFG using function outlining. The programmer annotates the incoming and outgoing pointer variables for each task or loop (Figure 3), which are used to infer the edges between nodes. Importantly, HeteroC++ directives are *completely hardware-agnostic*, lacking any target-specific design or tuning parameters. Through this programming interface, the same high-level program in HeteroC++ can be compiled for CPUs, GPUs, FPGAs and hardware accelerators by generating target-agnostic HPVM IR. HeteroC++ can be viewed as a small subset of OpenMP, supporting flexible nested loop and task parallelism and implicit offloading with "target hints." We are currently extending our autotuning framework to automatically determine the target device for a node given a target heterogeneous system.

Deep Learning Front Ends

HPVM supports front ends for three popular neural network frameworks: Keras, PyTorch,

and ONNX (a neural network exchange format). These front ends compile high-level code written in these frameworks to HPVM IR dataflow graphs with tensor operations (convolutions, matrix multiplications etc.) in the leaf nodes, and tensor data on the DFG edges.

HPVM Back End Code Generation

To generate code for the different target devices, a bottom-up traversal of the DFG is performed by each device back end, translating every leaf node to device-specific code for one or more devices using the following device-specific translators, and generating host code as needed. Standard LLVM CPU back ends for the x86, ARM, and RISC-V families are used to generate CPU code, both for the host program and to compile DFG nodes targeted for the CPU.

GPU Back end

For every leaf node targeted to the GPU, the GPU back end generates an OpenCL kernel in C, replacing HPVM's thread-indexing intrinsics with OpenCL API calls. Additionally, the necessary code to launch the kernel, set up its arguments, and copy arguments to device memory is generated for the host program. The dynamic replication factor of the kernel's leaf node determines the OpenCL workgroup size.

FPGA Back End

The FPGA back end also performs OpenCL code generation for the corresponding leaf nodes in a similar manner to the GPU back end. The OpenCL kernels are compiled using Intel FPGA SDK for OpenCL to generate an FPGA bitstream for Altera FPGAs. Intel recommends using Single Work Item Kernels, so we added a node sequentialization transformation to generate sequential loops from the dynamic replication factors of nodes. Finally, the back end uses separate OpenCL command queues for each kernel to support concurrent execution on the FPGA, and OpenCL events are used to synchronize accordingly. This is done by issuing the corresponding calls to the HPVM Runtime.

Compiling to Fixed Function Accelerators

The HPVM fixed function accelerator back end requires LLVM functions (e.g., generated from C source code) representing the functional semantics of each of the target accelerator's operations. It looks for matches between application leaf node functions and accelerator operations by using a semantic matching scheme for LLVM functions described in [8] based on program dependence graph (PDG) isomorphism. When a match is found, it replaces the leaf node function body with code to launch the corresponding accelerator construct. This back end has been used to generate code for FFT and Viterbi fixed function accelerators.

Back Ends for Deep Learning Accelerators

NVDLA is NVIDIA's programmable accelerator for energy-efficient tensor operations commonly-used in deep learning (convolutions, matrix multiplication, relu, max pooling, etc.). HPVM directly maps HPVM tensor IR operations to NVDLA's tensor IR constructs for operations like relu, max pooling, etc. It fuses HPVM IR operations such as convolution and tensor add before mapping to NVDLA constructs. The NVDLA IR is then compiled to object code by the vendor NVDLA compiler. We support two NVDLA modes: FP16 and INT8. For INT8, we perform floating-point to integer quantization using Distiller, a third-party tool for neural network quantization [9].

Intel Movidius VPU (Vision Processing Unit) is a deep learning accelerator used on edge devices like the Neural Compute Stick. Similar to the NVDLA workflow, the HPVM-Movidius back end translates HPVM's DFG of tensor operations to the Intel nGraph IR - a DFG-based IR with tensor operations specific to the Movidius back end. The HPVM-Movidius back end directly invokes nGraph compiler interfaces (linked with the HPVM toolchain), which apply hardware-specific optimizations and generate object code. These interfaces also insert code for offloading the compute kernels to the Movidius accelerator.

ATen Back End

HPVM tensor operations can be translated to high-performance libraries. We support compilation to the ATen back end - the tensor library used by PyTorch for compiling to GPUs (using cuDNN) and CPUs (using MKL-DNN). This back end enables HPVM to map to efficient

approximate constructs supported by ATen, including support for sparse tensor operations (used in pruned neural networks) and quantized tensor operations.

HPVM Optimization Frameworks

HPVM includes two code optimization frameworks:

- a graph optimization framework that automatically tunes an HPVM program for a specific accelerator target; and
- an accuracy-aware optimization framework, ApproxTuner, that uses approximation techniques to gain performance and energy improvements.

Optimizations and Autotuning

The HPVM graph optimization framework includes both HPVM DFG graph optimizations and regular LLVM optimizations on the node functions. These optimizations are applied to HPVM leaf nodes (i.e. kernels) singly or in pairs. They include:

- **Argument Privatization** finds pointer arguments that are marked as thread-private and creates a local (`private`) copy of them.
- **Loop Unrolling** unrolls loops using a specified unroll factor.
- **Greedy Loop Fusion** considers fusing all pairs of LLVM loops in a single leaf-node function that are legal to fuse, from the outermost nesting level to the innermost.
- **Node Fusion** fuses DFG nodes that are connected with an edge and have the same dynamic replication factor and no fusion-preventing dependencies.

HPVM's optimization framework incorporates autotuning using the HyperMapper design space exploration framework [10] to automatically tune hardware-agnostic programs for FPGA [7] and GPU (separately for now) using the above optimizations. A performance model is used for FPGA tuning to avoid long synthesis times, while we use direct execution on the GPU for GPU tuning. The autotuner selects which optimizations will be applied and, for loop unrolling, what unroll factor to use for each loop.

ApproxTuner

ApproxTuner is an end-to-end accuracy-aware optimization framework for tensor-based components of programs, such as deep neural networks and image processing pipelines. ApproxTuner takes a program compiled to HPVM IR and a desired end-to-end quality (QoS) threshold, and automatically maps tensor operations to different approximations to maximize performance and/or energy benefits while ensuring that the QoS is achieved.

ApproxTuner uses a *novel three-phase, predictive tuning* strategy to map approximations on diverse hardware and maintain object code portability. To enable efficient tuning, ApproxTuner uses a *predictive tuning approach*, which uses accuracy and performance prediction models instead of expensive empirical evaluations.

At development-time, ApproxTuner tunes the program with *hardware-independent* approximations, finding a number of configurations. A *configuration* is a mapping from each tensor operator to one or more approximations and parameter settings for those approximations. Predictive tuning is used to compute a Pareto-optimal frontier of these configurations in the performance-accuracy tradeoff space, and this Pareto curve is shipped with the HPVM IR. At install-time, ApproxTuner retunes these configurations with any hardware-specific knobs that exist on the target hardware, and refines the Pareto curve by measuring real performance on the target hardware. The final Pareto curve is shipped with the application binary and used by a dynamic tuner at run-time (described later).

ApproxTuner supports software approximations such as reduction sampling (using a subset of inputs in the reduction), perforated convolutions, and sampled convolutions. At the hardware-level, ApproxTuner supports reduced precision using FP16 or INT8, and mapping to low-voltage knobs on PROMISE, an experimental analog ML accelerator [11].

Runtime Systems and Schedulers

HPVM Runtime

The HPVM Runtime (HPVM-RT) enables programs compiled in HPVM to efficiently execute on a diverse range of hardware platforms.

HPVM-RT provides a memory tracker which maintains the location of the most recent *dirty* copy of memory objects used across HPVM DFGs, and uses that to determine when a memory copy between host and device is necessary.Also, HPVM-RT communicates with the corresponding device runtimes (OpenCL runtime for GPU and FPGA), creating the necessary OpenCL objects (Platform, Context, Kernels, Command Queues), copying memory back and forth, setting kernel arguments, and launching kernels on the corresponding devices.

ApproxTuner Runtime Approximation Tuning

ApproxTuner's run-time approximation tuner enables applications to maintain performance goals under changing system or application conditions, e.g., low-power modes. The tuner adapts per-operation approximation knobs (described earlier) to adapt the accuracy-performance tradeoff while using a system monitor to detect system slowdowns. The tuner uses the Pareto curves shipped with the HPVM program binary to choose the most accurate approximation parameter settings that satisfy desired quality metrics. These parameter settings are used as arguments for the tensor operations on each invocation, making it easy to change configurations quickly [6].

Third-party SoC Schedulers

HPVM includes support for two third-party SoC schedulers as part of a collaborative project. The first is the ESP system [12] from Columbia, a hardware-design framework that enables easy integration of new accelerators into SoC's. ESP is being used as part of the EPOCHS project (led by IBM) to design an SoC specialized for autonomous vehicles with several accelerators, such as FFT, Viterbi, and NVDLA, and a RISC-V host. This approach enables hardware-agnostic programmability as well as potential accuracy-aware optimization for SoCs designed using ESP.

The second is the novel EPOCHS scheduler to schedule different application "tasks" onto the available accelerators in an SoC, including static and dynamic mappings. Our compiler targets the scheduler library API to launch the "tasks" and specify the possible target devices for each. These HPVM back ends are easily extensible to other similar SoC design and task scheduling frameworks.

Experimental Evaluation

Because of space constraints, we focus on results from recent work. In an early HPVM publication, we reported results for 7 Parboil benchmarks, showing that the HPVM infrastructure can compile the *same* hardware-agnostic code for NVIDIA GPUs and Intel vector instructions (AVX) and achieve performance competitive with that of separately hand-tuned OpenCL code for each target [2].

Below, we show experimentally that: (a) our GPU autotuner achieves excellent speedups; (b) a *single, hardware-agnostic* program can be partitioned for GPU and FPGA, achieving much higher speedups than on GPU alone; (c) when small reductions in end-to-end DNN inference accuracy are tolerable, ApproxTuner can roughly double the performance of a wide range of DNNs [6] ; and (d) dynamic approximation tuning enables a DNN to maintain image classification throughput despite a large reduction in GPU clock frequency, with only a small loss in inference accuracy [6]. The first two experiments are new for this work.

Optimizing Applications for GPU

To evaluate the GPU autotuner, we conducted an experiment on seven benchmarks: a five-stage camera image processing pipeline (CAVA)[2], an Edge Detection program for grayscale images from [2] whose DFG is a six-node DAG, and five multi-kernel benchmarks from Rodinia [13] : breadth-first search (BFS), backpropagation (Backprop), speckle reducing anisotropic diffusion (SRAD), and both the "Euler" and "Pre-euler" implementations of the computational fluid dynamics solver (CFD). The hardware was an Intel Xeon 4216 CPU and NVIDIA RTX 2080 Ti GPU. Each data point is the average of five runs, with error bars showing the range.

Figure 4(a) shows speedups compared to baselines compiled from the programs using HPVM's single-threaded CPU back end, without applying our optimizations. Orange and gray bars show speedups achieved by HPVM without and with our optimizations, respectively, including autotuning in the latter. For Euler, Pre-Euler and BFS,

[2]https://github.com/yaoyuannnn/cava/

autotuning does not add much benefit over the GPU version. For the other four benchmarks, the widely varying configurations selected by autotuning demonstrate the importance of autotuning to achieve source code portability and hardware-agnostic programming. For example, Edge and Backprop had loops that were fully unrolled, Euler and SRAD had partially-unrolled loops, while CAVA had only one loop unrolled. Making similar design choices manually requires trial and error, and achieving sufficient coverage of the search space through manual exploration is often impractical, and most seriously, can lead to hardware-specific tuned code, which compromises source code portability.

SRAD suffers a slowdown on the GPU, even with autotuning, due to a sum-reduction kernel that is not parallelized by HPVM. Through autotuning, loop unrolling was able to reduce the slowdown from $5.5\times$ to $1.7\times$. Automating the parallelization of reductions for GPUs is planned, along with support for other important GPU optimizations like tiling for GPU registers and scratchpad memory.

Including initial random sampling to initialize HyperMapper, 212 designs were evaluated for CAVA and Backprop, while 400 were evaluated for Edge Detection, proportional to their parameter counts. The remaining benchmarks had so few parameters that all designs in the search space were evaluated before reaching their set iteration counts, producing between 2 and 48 samples. This contributed to a large range of autotuning times, taking between 1 and 445 minutes for a full run (average 146). Time per sample ranged between 23 and 388 seconds (average 122), since each sample is executed on the GPU during autotuning.

Partitioning Applications on multiple devices

Partitioning applications across multiple devices (e.g. GPU and FPGA) poses two main challenges: 1) programmability, since different devices tend to have different programming models and languages, and 2) partitioning decisions, which requires an intimate knowledge of the performance tradeoffs for each target device. HPVM allows us to overcome the first challenge by supporting a unified hardware-agnostic programming language and IR that can be targeted to multiple different devices. As a proof-of-concept, we manually partitioned the Edge Detection benchmark of the previous section using device hints on the nodes, putting the reduction kernel (which dominates the execution time) on the FPGA and leaving the rest on the GPU. Our target GPU+FPGA system uses an Intel Xeon W-2275 CPU, an NVIDIA Quadro P1000 GPU, and an Intel Arria 10 GX FPGA. Figure 4(b) shows the performance results averaged over ten runs. The speedup increases by around $3\times$ by moving the reduction kernel to the FPGA, demonstrating the strong benefits of such a partitioning. (The base GPU speedup is lower than in 4(a) because the Quadro P1000 is much slower than the RTX 2080 used there.) As in the previous experiment, the GPU performance of the reduction could be improved by parallelizing it, but the nature of pipeline parallelism in an FPGA is better suited for reductions and can handle non-associative reductions which cannot be parallelized on a GPU.

This experiment shows that HPVM is able to overcome the first challenge described above. Additionally, once our autotuner is extended to support multiple target devices at the same time, the partitioning decisions would be automated and optimized as well, thus solving the second challenge of multi-device partitioning.

Performance Improvements using ApproxTuner

Figure 5(a) shows our evaluation results [6] using ApproxTuner on ten popular CNN benchmarks measured on the Jetson Tegra Tx2 GPU. The graph shows speedups obtained compared to a baseline that does all operations in FP32 on the GPU, with no approximations. We configured ApproxTuner to choose from three different hardware-independent approximations per tensor operation: FP16, perforated convolutions, and sampled convolutions.

Across benchmarks, when allowed merely one percentage point drop in accuracy, ApproxTuner achieves a mean $2.1\times$ speedup compared with the baseline. Relaxing the accuracy threshold to 2 and 3 percentage points (red and orange bars, respectively) provides only a small increase in speedups, to $2.2\times$ and $2.3\times$, showing that most of the benefits are gained simply by allowing any approximation at all.

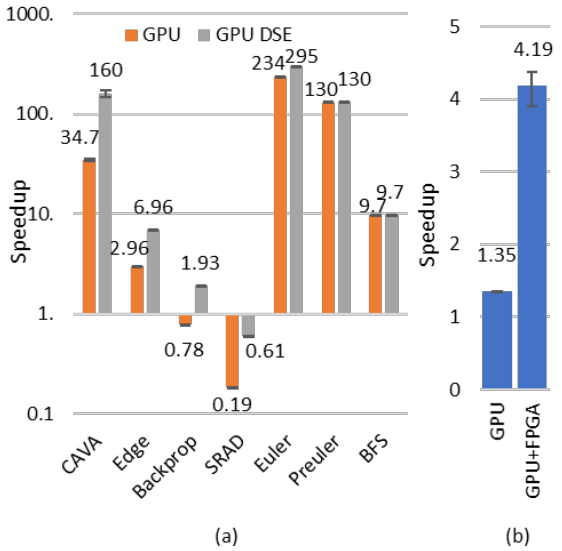

Figure 4: (a) Results for GPU autotuning. Figure shows speedup of hardware-agnostic code running unoptimized on GPU (orange bar) and optimized using autotuning (grey bar) compared to CPU. GPU is NVIDIA 2080 Ti. (b) Speedups compared with CPU for two device partitionings of Edge Detection. GPU is NVIDIA Quadro P1000 and FPGA is Arria 10 GX.

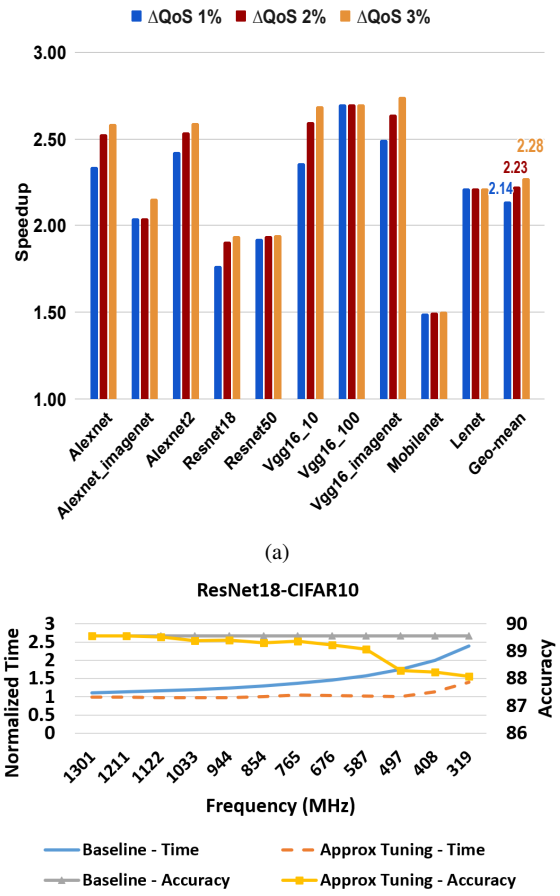

Figure 5: (a) Speedups achieved on GPU using approximations for $\Delta QoS_{1\%}$, $\Delta QoS_{2\%}$, $\Delta QoS_{3\%}$. (b) Runtime approximation tuning maintains stable responsiveness (red line) with little lost of accuracy in most of the range (yellow), when GPU frequency is reduced. Time on the y-axis is relative to that at the highest GPU frequency (1.3Ghz). Without dynamic approximations, the application slows down (blue line). (from ApproxTuner [6]).

As expected, each network is amenable to a different set of approximations; there does not exist an approximation that provides the best accuracy-performance tradeoff on all networks. For example, Alexnet is amenable to perforated convolution and more sensitive to sampled convolution, while it is the opposite for the VGG networks, which ApproxTuner discovers through tuning. In addition, ApproxTuner finds that the first few and last convolution layer in a network cause the highest errors due to approximations, and it approximates these layers conservatively.

Figure 5(b) shows that ApproxTuner can counteract system slowdowns induced by low frequency modes on the GPU. As frequency lowers from left to right (x-axis), the normalized batch processing time (y-axis) shown by the blue line increases. ApproxTuner uses the shipped Pareto curve to pick configurations that increase speedups in order to counteract these slowdowns, while sacrificing small amounts of accuracy. The red dotted line shows batch processing times stabilize when ApproxTuner dynamic tuning is enabled. The yellow line shows that as frequency decreases, the neural network accuracy gradually decreases since higher approximation levels are needed to counter greater slowdowns.

Directions for Further Work

As heterogeneous systems adopt more diverse accelerators, we will continue expanding HPVM with more device back ends, and more advanced optimization and tuning techniques. This includes extending our autotuner to automatically partition programs across PEs, while also optimizing them for each target.

We are also working on making HPVM even more retargetable for a wide range of emerging tensor architectures, including CPU ISA extensions (Intel AMX, Power MMA), GPU extensions

(NVIDIA's Tensor Cores, AMD's Matrix Cores), and custom ML accelerators (Amazon Inferentia/Trainium, Google TPU, NVDLA).

We also aim to leverage HPVM to greatly simplify DSL design and implementation for high-level applications that benefit from heterogeneous systems.

We are also interested in extending approximation tuning to emerging application domains, particularly edge computing domains such as mobile robotics, AR/VR, and video analytics.

Acknowledgements

This work was supported in part by NSF Grants CCF 13-02641 and CCF 16-19245, the Semiconductor Research Corporation and DARPA through the Center for Future Architectures Research (C-FAR) and the Applications Driving Architectures (ADA) center, the DARPA DSSoC Program, by grants from Intel Corp, and by the Amazon AWS Machine Learning Research Awards and Amazon Research Awards programs.

Sidebar: Key Takeaways

- HPVM enables hardware-agnostic programming of heterogeneous systems via a hierarchical dataflow graph abstraction of parallelism that supports retargetable compilation to diverse hardware, such as CPUs, GPUs, FPGAs, and domain-specific accelerators.
- HeteroC++, PyTorch and other hardware-agnostic front ends simplify programming heterogeneous systems, and facilitate offloading different application components to different devices while preserving source-code and (optionally) object-code portability.
- Sophisticated HPVM and LLVM optimizations, together with target-specific autotuning, deliver significant performance improvements without manual tuning, which greatly improves source-level portability and maintainability.
- The ApproxTuner automated approximation tuning framework for tensor operations supports powerful accuracy-aware optimizations and run-time adaptation, while preserving hardware-agnostic programming and object-code portability.

REFERENCES

1. V. Adve *et al.*,"Virtual Instruction Set Computing for Heterogeneous Systems," *USENIX Workshop on Hot Topics in Parallelism*, 2012.
2. M. Kotsifakou *et al.*, "HPVM: Heterogeneous Parallel Virtual Machine," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018.
3. D. Koeplinger *et al.*,"Spatial: a language and compiler for application accelerators," *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.
4. T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," *USENIX Symposium on Operating Systems Design and Implementation*, 2018.
5. H. Sharif *et al.* "ApproxHPVM: a portable compiler IR for accuracy-aware optimizations," in *Proc. ACM Programming Languages*, 2019.
6. H. Sharif *et al.*, "ApproxTuner: a compiler and runtime system for adaptive approximations," *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.
7. A. Ejjeh *et al.*, "HPVM2FPGA: Enabling True Hardware-Agnostic FPGA Programming," *Proc. IEEE International Conference on Application-specific Systems, Architectures, and Processors*, 2022.
8. S. Dasgupta *et al.*, "Scalable Validation of Binary Lifters," *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.
9. N. Zmora *et al.*. "Neural network distiller: A python package for dnn compression research," *arXiv*, 2019.
10. L. Nardi *et al.*, "Practical design space exploration," *Proc. IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2019.
11. P. Srivastava *et al.* "PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," *Proc. ACM/IEEE International Symposium on Computer Architecture*, 2018.
12. P. Mantovani *et al.*, "Agile SoC Development with Open ESP," *Proc. IEEE/ACM International Conference On Computer Aided Design*, 2020.
13. S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," *Proc. IEEE International Symposium on Workload Characterization*, 2009.

Adel Ejjeh is a PhD student at University of Illinois at Urbana-Champaign.

Aaron Councilman is a PhD student at University of Illinois at Urbana-Champaign.

Akash Kothari is a PhD student at University of Illinois at Urbana-Champaign.

Maria Kotsifakou is a software engineer at Runtime Verification, Inc. in Urbana, IL.

Leon Medvinsky is a PhD student at University of Illinois at Urbana-Champaign.

Abdul Rafae Noor is a PhD student at University of Illinois at Urbana-Champaign.

Hashim Sharif is a postdoctoral researcher at University of Illinois at Urbana-Champaign.

Yifan Zhao is a PhD student at University of Illinois at Urbana-Champaign.

Sarita Adve is a professor at University of Illinois at Urbana-Champaign.

Sasa Misailovic is an assistant professor at University of Illinois at Urbana-Champaign.

Vikram Adve is a professor at University of Illinois at Urbana-Champaign.