HPVM2FPGA: Enabling True Hardware-Agnostic FPGA Programming

Adel Ejjeh*, Leon Medvinsky*, Aaron Councilman*, Hemang Nehra*, Suraj Sharma*, Vikram Adve*,
Luigi Nardi†‡, Eriko Nurvitadhi§, and Rob A Rutenbar¶.
*University of Illinois at Urbana-Champaign, Urbana, IL, USA
†Lund University, Lund, Sweden. ‡Stanford University, Stanford, CA, USA
§Intel Corp, Hillsboro, OR, USA. ¶University of Pittsburgh, Pittsburgh, PA, USA.

***Abstract*—Current FPGA programming tools require extensive hardware-specific manual code tuning to achieve performance, which is intractable for most software application teams. We present HPVM2FPGA, a novel *end-to-end compiler and auto-tuning system* that can automatically tune hardware-agnostic programs for FPGAs. HPVM2FPGA uses a hardware-agnostic abstraction of parallelism as an intermediate representation (IR) to represent hardware-agnostic programs. HPVM2FPGA's powerful optimization framework uses sophisticated compiler optimizations and design space exploration (DSE) to automatically tune a hardware-agnostic program for a given FPGA. HPVM2FPGA is able to support software programmers by shifting the burden of performing hardware-specific optimizations to the compiler and DSE. We show that HPVM2FPGA can achieve up to $33\times$ speedup compared to unoptimized baselines and can match the performance of hand-tuned HLS code for three of four benchmarks. We have designed HPVM2FPGA to be a modular and extensible framework, and we expect it to match hand-tuned code for most programs as the system matures with more optimizations. Overall, we believe that it constitutes a solid step closer to fully hardware-agnostic FPGA programming, making it a suitable cornerstone for future FPGA compiler research.**

***Index Terms*—High-level synthesis, FPGA, hardware-agnostic FPGA programming, compilers for FPGA.**

I. Introduction

Recently, FPGAs have become widely available in heterogeneous systems and public clouds, taking them beyond the traditional audience of hardware designers and making them accessible to the much larger category of software application developers. When used for acceleration, FPGAs have been known to provide significant benefits in energy and performance for a wide range of domains, from image and signal processing [1], to communications and networking [2], and most recently machine learning [3]. However, these devices have been primarily used by hardware designers due to their complex programming models that rely heavily on low-level hardware knowledge. Even high-level synthesis (HLS) frameworks, which raised the abstraction level for FPGA programming compared to RTL, are primarily focused on increasing the productivity of hardware engineers, and require significant hardware-specific manual code modifications and tuning to achieve good performance.

Consider the Intel FPGA SDK for OpenCL, or AOC, as an example of a popular HLS tool. Getting good performance with AOC is non-trivial since it requires extensive hardware-specific tuning that requires developers to adopt optimization strategies and techniques proposed in the official supporting documents. Xilinx's HLS tools, VivadoHLS and VitisHLS, are very similar in terms of the nature of the programming support. The nature of these optimization strategies show clearly that today's HLS tools are primarily aimed at designers with strong hardware knowledge and skills, which many application software teams lack.

Software teams deal with large complex applications, with stringent constraints on development cost, source code portability, code reuse, software security, and rapid development cycles, in addition to performance. These constraints make it difficult to invest extensive time and effort to tune application components for specific hardware targets. In many cases, this translates to an *acceptable trade-off of raw performance for improved programmability*. As such, software designers would greatly benefit from an end-to-end compiler framework that supports **hardware-agnostic programming** of heterogeneous devices, including FPGAs.

We informally define *hardware-agnostic programming* as follows: the entire process, including the program itself (which is usually part of a larger application) and the iterative development and tuning steps, should not be specific to a particular hardware target. Programming tools must automatically optimize the code for a target FPGA, instead of requiring detailed hardware understanding from the developer. More specifically, we assume that the input source program must adhere to the following constraints:

(C1) *No manual code restructuring*: The code should not be explicitly rewritten in non-trivial ways for the explicit purpose of achieving (or enabling) a performance improvement on the intended target hardware.

(C2) *No hardware-specific tuning parameters*: The code should not specify tuning parameters aimed at optimizing performance for a specific hardware target, e.g., unroll factors, buffer sizes, memory banking strategies, etc. Hardware-independent program properties, such as the `restrict` keyword for declaring pointers are not aliased, or size attributes of arrays, are acceptable.

(C3) *No explicit platform-specific coding*: The application code should not include platform-specific code that handles host-device communication like copying memory, launching kernels, setting arguments, etc.

We also assume that the development flow must be hardware-agnostic from the programmer's perspective:

(C4) *No hardware-specific optimization tuning by the programmer:* The process of picking a combination of optimizations, and the specific configurations of each optimization, for optimizing a specific program on a target FPGA, must be transparent to the programmer. Any hardware-aware decisions or tuning must be done automatically by the programming tools.

Hardware-agnostic programming of FPGAs *remains a holy grail* in the FPGA community. We believe that the requirements to achieve this goal are the following: a) an **end-to-end compiler and autotuning system** that tunes *hardware-agnostic* kernels by automatically selecting FPGA-specific optimizations, and transparently handles host code generation, b) a compiler intermediate representation that captures different kinds of parallelism (task, data, pipeline) while identifying units of acceleration, and c) a runtime system that transparently interfaces between host and device. This kind of end-to-end flow is largely missing in the FPGA design community, and state-of-the-art tools today lack one or more of these components (see Section V, Table II). Commercial HLS tools from Intel and Xilinx are too narrowly focused on hardware designers, expecting the user to manually implement (with limited guidance) a variety of non-trivial, hardware-specific optimizations needed for high performance [4]. State-of-the-art research compilers are either domain specific [5], or still require some form of hardware-specific tuning/understanding [6], [7]. Finally, some recent projects attempt to solve this problem by compiling hardware-agnostic code [8]–[10], but they use simplistic design space exploration, if any, and do not perform inter-kernel optimizations.

We propose HPVM2FPGA, a novel *end-to-end compiler and autotuning system* that can automatically tune hardware-agnostic programs for FPGAs. HPVM2FPGA uses a hardware-agnostic abstraction of parallelism as a compiler intermediate representation (IR), building on the HPVM compiler IR [11], an explicitly parallel extension of LLVM IR [12] designed for heterogeneous parallel systems. HPVM2FPGA adds a powerful optimization framework that uses sophisticated compiler optimizations (both FPGA-specific and generic) and design space exploration (DSE) to automatically tune a hardware-agnostic program for a given FPGA. HPVM2FPGA is able to support software programmers effectively by eliminating the need for hardware-specific details from the code, and shifting the burden of performing host-device glue code generation, and hardware-specific optimizations to the compiler and DSE.

HPVM2FPGA's optimization framework uses parameterized compiler optimizations and design space exploration (DSE) to automatically tune programs (at the HPVM IR level) for a target FPGA. We use HyperMapper [13] as our DSE engine since it has been shown to optimize hardware design problems [6], [14]. To make DSE practical for FPGAs, we must estimate performance of sample designs without generating a full design, including the (slow) "back-end" synthesis stage of AOC. We developed an analytical performance model that estimates the execution time of the optimized input program on the target FPGA, using a loop pipeline latency calculation and a critical path analysis to account for inter-kernel parallelism. This performance model is independent of specific optimizations, derives its required inputs from a static analysis of the HPVM IR after optimizations and performance metrics generated by the (fast) "front-end" RTL generation stage of AOC. Moreover, our optimization framework is modular and extensible, so that more optimizations can be added relatively easily, and other DSE engines or performance models can be plugged in with minimal effort.

The key contributions of our work are:

1) A *modular and extensible end-to-end compiler framework*, HPVM2FPGA, that generates optimized FPGA kernels from hardware-agnostic input programs, along with the host code. The framework can continuously improve with new compiler optimizations and cost estimation for new hardware targets.
2) HPVM2FPGA's optimization framework, which optimizes the *application kernels* using compiler optimizations and design space exploration (DSE). DSE is guided by an analytical performance model using inputs from static analysis and the AOC front-end.
3) A variety of compiler optimizations designed to automate some of the manual tuning that is required for FPGAs. These are a combination of loop-level, memory-level, and multi-kernel HPVM DFG optimizations. To our knowledge, at least four of these optimizations have not been used in FPGA compilers before.
4) An experimental evaluation of our compiler, which shows that our framework: a) can optimize hardware-agnostic, multi-kernel benchmarks achieving up to $33\times$ speedup on an Arria 10 GX FPGA compared to unoptimized baselines, and b) can match hand-tuned FPGA designs in three out of four cases. Additional support for generating NDRange OpenCL Kernels instead of Single Work Item Kernels and channels for pipelining would enable other cases to match hand-tuned designs.

The rest of the paper is organized as follows: Section II provides background on the HPVM IR and the HyperMapper DSE framework. Section III describes our overall compiler workflow, including our optimization framework. Section IV describes our experimental evaluation and results. Section V presents the related work, and we conclude in Section VI.

II. Background

A. HPVM

Heterogeneous Parallel Virtual Machine (HPVM) [11] is a parallel program representation for heterogeneous hardware that is designed to be used as a Virtual ISA, compiler IR, and runtime representation. Built on top of LLVM [12], HPVM benefits from all the optimization and code generation capabilities of LLVM for scalar and vector code, and adds support

for parallel programs and heterogeneous systems. HPVM represents parallelism using a static, hierarchical dataflow graph (DFG) with side effects. DFG nodes can represent the units of parallelism, or can contain an entire ("nested" or "child") dataflow graph. This nesting allows HPVM to capture multiple levels of parallelism in heterogeneous systems. DFG edges describe explicit, "logical" data transfer between nodes. Each static node in the graph can have multiple independent dynamic instances – or threads – specified as a *replication factor*. These types of nodes usually correspond to parallel loops. In this paper we refer to a leaf node without dynamic replication as a *task*. This structure allows HPVM to capture loop-level data parallelism, fine-grain (vector) data parallelism, task parallelism between concurrent nodes, and pipelined parallelism, in a single parallel program representation. HPVM provides front ends for hardware-agnostic input programs (HeteroC++ and HPVM-C), and supports back end code generation for CPUs and GPUs. Additionally, HPVM provides a run-time scheduler that invokes the corresponding device runtime to launch the kernel and copy data in and out of the device. Our work extends this infrastructure with a new FPGA back end and runtime extensions, in addition to the HPVM2FPGA optimizer (our primary contribution).

B. HyperMapper

HyperMapper (HM) [13] is a software framework used in a number of applications for computer systems design space exploration, including hardware-software co-design [15], FPGA design [6], [14], and automated machine learning [14]. HM aims to find a global minimizer of a user-provided (black-box) objective function $f : \mathbb{X} \rightarrow \mathbb{R}$ (e.g., execution time of an application's kernels on FPGA) under a set of provided (black-box) constraint functions (e.g., whether the kernels fit on the FPGA).

HM tackles the problem using Bayesian Optimization (BO) [13], a machine learning optimization method which approximates an optimal design $\boldsymbol{x}^* \in \mathbb{X}$ that maximizes a utility metric, with each new $\boldsymbol{x}_{t+1}$ depending on the previous sequence of function values $f(x_1), \ldots, f(x_t)$ at $\boldsymbol{x}_1$, ..., $\boldsymbol{x}_t$. BO achieves this by building a probabilistic surrogate model on f based on the set of evaluated designs $\boldsymbol{x}_t$. At each iteration, a new design $\boldsymbol{x}_{t+1}$ is selected and evaluated based on the surrogate model, and this model is updated to include the new data point $(\boldsymbol{x}_{t+1}, f(\boldsymbol{x}_{t+1}))$.

The designs explored by BO are dictated by a utility metric, which attributes a utility to each $\boldsymbol{x} \in \mathbb{X}$ by balancing the predicted value and uncertainty of the prediction for each $\boldsymbol{x}_t$. The utility function is maximized at designs where the predicted function value is low or the uncertainty of the prediction is high. HM adopts the Expected Improvement (EI) utility function criterion. By evaluating the design with the maximal acquisition function value, BO efficiently explores $\mathbb{X}$, and it is an efficient optimization approach in terms of the number of function evaluations. This is especially well-suited to expensive functions, such as FPGA design optimization.

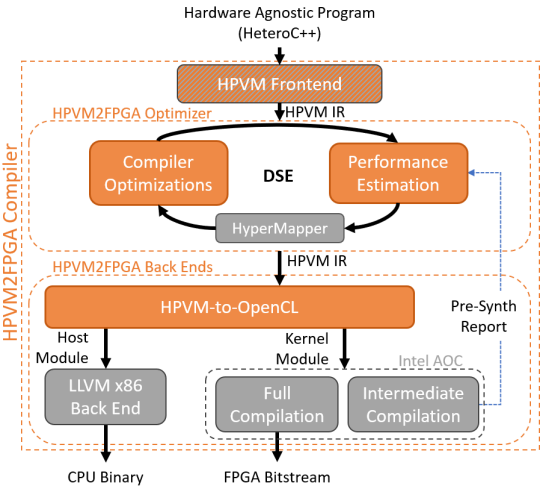

Fig. 1: HPVM2FPGA Compiler Flow. Grey boxes represent existing components. Orange boxes represent components we added.

III. HPVM2FPGA

A. HPVM2FPGA Overview

HPVM2FPGA is an open-source[1] end-to-end compiler infrastructure for hardware-agnostic programming of FPGAs. Our tool flow builds on the existing HPVM infrastructure described in Section II by adding the HPVM2FPGA optimization framework, which includes seven compiler optimizations and DSE, the HPVM2FPGA back end, and runtime extensions. We chose the HPVM IR for two main reasons: 1) Its hardware agnostic nature allows us to easily represent general, parallel, hardware-agnostic programs, and (in future) target them to systems containing a mix of FPGAs and other devices; and 2) Its ability to represent the different levels of parallelism is crucial for optimizing hardware-agnostic codes for hardware targets like FPGAs.

The HPVM2FPGA compiler, shown in Figure 1, comprises two main components: the HPVM2FPGA optimizer, which uses compiler optimizations and DSE to automatically tune programs at the IR level, and the HPVM2FPGA back end, which uses the Intel FPGA SDK for OpenCL (AOC) to synthesize the FPGA bitstream. The compilation process goes as follows: starting with an input program in an HPVM-compatible language (Hetero-C++, HPVM-C, or any other language that can be compiled to HPVM in the future), the HPVM front end lowers the source code into a hardware-agnostic HPVM IR representation of the program (C1). Next, an optimization step, described below, optimizes the HPVM leaf nodes, which will become FPGA kernels; the framework supports both inter-node and intra-node optimizations to achieve the best possible designs (C2,C4). Then, the HPVM-to-OpenCL back end generates an OpenCL file containing the optimized kernels, and generates the required runtime code that launches and manages these kernels (i.e. creates OpenCL buffers, sets the arguments, copies the memory, etc.) into the host LLVM module (C3). Finally, the OpenCL kernels get

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

synthesized using AOC, and the host module gets compiled using LLVM's x86 back end to generate a binary.

We currently have seven optimizations in HPVM2FPGA, implemented as either HPVM-DFG or LLVM transformations; these are described in III-B, and they complement the standard LLVM optimizations, and those performed by AOC (C2). The optimizer can either run a predefined set of optimizations that the compiler (or user) selects from the list of available ones, or it can use DSE to automatically select and tune the optimizations that would provide the best possible performance on the FPGA (C4). DSE is guided by a performance model, which we calculate using a static analysis of the HPVM IR and information that we extract from the AOC pre-synthesis report, as described in III-C. We use HyperMapper as our DSE engine to explore the design space.

Note that we always generate Single Work Item (SWI) Kernels from the HPVM leaf nodes, as they are recommended by Intel's manuals over NDRange Kernels. As such, for leaf nodes that have dynamic replication factors (i.e. parallel loops), we added an HPVM-DFG transformation pass, Node Sequentialization, that transforms these nodes into "task" nodes with loops in their body. No special consideration is required for nodes without dynamic replication.

HPVM2FPGA is designed to be a cornerstone for future hardware-agnostic FPGA compilers by providing a fully extensible framework. Every component described above is modular, making it easy to add more powerful optimizations, add more advanced code-generation options to lower-level hardware IRs, add more device back ends for different FPGAs, and improve on the performance estimation with more accurate or faster performance modeling, all within a unified, extensible framework.

B. Implemented Optimizations

We currently have seven optimizations implemented in our compiler as either HPVM-DFG or LLVM transformations: Automatic Input Buffering (BI), Guided Argument Privatization (AP), Loop Unrolling (LU), Greedy Loop Fusion (LF), Automatic `ivdep` Insertion (IV), Automatic Task Parallelism (TP), and Automatic Node Fusion (NF). These were chosen to implement some of the key optimization techniques that are required for achieving good FPGA designs, and are usually applied manually. These optimizations are used in our optimization framework in addition to standard LLVM optimizations and those that AOC applies internally. To our knowledge, no other FPGA high-level-design-compiler combines all these seven optimizations together, and at least four of these optimizations (AP, LF, IV, and TP) have not been used in this context before.

Most of our optimizations leverage HPVM's DFG representation. These optimizations can be run either as standalone passes or as part of DSE. When used with DSE, the compiler will determine which optimizations to apply on a given application, and how to apply them. Table I lists these optimizations and how they are parameterized for DSE. Each optimization is parameterized for the appropriate granularity of code, enabling a finer-grain design space exploration. When the optimizer is used without DSE, heuristics are used to determine how to apply them (e.g. Node Fusion fuses all fusible nodes instead of making pairwise decisions as in DSE).

TABLE I: Current optimizations and their use in DSE.

	IB	**AP**	**LU**	**LF**	**IV**	**TP**	**NF**
Inter-Kernel	No	No	No	No	No	Yes	Yes
DSE Param	Categorical (`bool`)	Categorical (`bool`)	Ordinal (`int`)	Categorical (`bool`)	N/A	Categorical (`bool`)	Categorical (`bool`)
Param Granularity	Kernel Argument	Kernel Argument	Loop	Kernel Function	N/A	Application	Pair of Kernels

Automatic Input Buffering (IB): This transformation automatically finds read-only kernel (i.e., leaf node function) pointer arguments with constant size, and copies them from *Global* memory to *Local* memory buffers. The constant size is determined using a DFG traversal to determine the size of the argument as allocated in the host code. In DSE, one boolean parameter is created per pointer argument that determines whether or not the argument will get buffered. For example, if a given argument has locality and fits on the FPGA local memory, then DSE may select it for buffering in the final design if that improves performance.

Guided Argument Privatization (AP): Privatizable arguments are ones that are completely generated and then read within the kernel. We require that they must be of fixed size. Such arguments may arise, e.g., when allocating all memory objects in `main()`. This transformation finds such arguments and creates private copies, converting all reads and writes to access *Local* instead of *Global* memory. Since finding privatizable variables is a difficult problem requiring interprocedural pointer analysis and array section analysis [16], we instead make this a guided optimization requiring the programmer to mark `private` arguments and their size in HeteroC++, using a special keyword. *This annotation is completely hardware-agnostic.* In DSE, one boolean parameter is created per privatizable argument that determines if the arguments gets privatized. As such, only the ones that improve performance will be privatized in the final design.

Automatic `ivdep` Insertion (IV): The `ivdep` pragma (with specified pointer parameters) informs AOC that a loop does not have loop-carried dependencies on the specified pointers. This may allow AOC to achieve lower initiation intervals when pipelining the loop. HPVM guarantees that dynamic node instances are parallel, i.e., there are no cross-instance dependencies due to pointer arguments. As such, we insert the pragma for any loop generated by Node Sequentialization, and pass in all pointer input parameters of the node as arguments to the `array` clause of `ivdep`. Since `ivdep` never hurts performance, it is always enabled in DSE.

Loop Unrolling (LU): This transformation unrolls the loops of a leaf node function with known trip counts. For loops with variable trip counts, we added a HeteroC++ marker function (`isNonZeroLoop`) which the programmer can use to specify the run-time trip count of each loop (obtained perhaps from profiling information). This marker function is not hardware specific, and its insertion can be automated in the future using an instrumentation pass. In DSE, an integer parameter

is created for the unroll factor of each loop, including ones generated by Sequentialization. As such, DSE would unroll each loop the appropriate number of times to achieve the best performance.

Greedy Loop Fusion (LF): This transformation fuses all the fusible loops of a leaf node function at each nesting level, going from the outermost nesting level to the innermost. At each level, we find the fusible loops by checking for matching loop bounds and no fusion-preventing dependencies [17]. When `ivdep` is present, we also disallow a forward loop-carried dependence that is otherwise legal, since `ivdep` will no longer hold. Selectively removing these pointers from `ivdep` is part of future work. In DSE, one boolean parameter is created for every function with more than one loop, turning LF on or off for the entire function.

In general, LU and LF can extract pipeline parallelism together by unrolling outerloops and fusing innerloops. This is how the optimizer applies them when used without DSE.

Automatic Node Fusion (NF): Node Fusion fuses multiple leaf nodes in the DFG into one node, merging their function bodies in what is effectively kernel fusion. Having the DFG representation makes it easy to identify nodes with source-sink relationships, allowing the compiler to focus on a subset of possible fusions that are likely to be profitable. In DSE, one boolean parameter is created for each pair of nodes that can legally be fused and are connected by an edge. As such, DSE tries to find the best combination of fused nodes that would maximize performance in the final design.

Automatic Task Parallelism (TP): This optimization analyzes the DFG to determine if there exist any nodes that can run in parallel (i.e. have no connecting path in the DFG), and accordingly guides code generation to enable launching them in parallel on the FPGA. In DSE, one boolean parameter is created for the entire application, enabling task parallelism if it is profitable.

C. Design Space Exploration and Performance Model

Our optimization framework sets up the parameter space for DSE by performing a static analysis of the HPVM IR and extracting the corresponding parameters for every possible optimization. Then, it interacts with HyperMapper (HM), where in every iteration HM sends a chosen design sample (i.e. set of values for the parameters) and waits for a response. For each sample, we apply the optimizations with the provided parameter values, and then estimate the execution time.

To estimate execution time, we devised an analytical model that uses the loop Initiation Intervals (II), latencies of basic blocks (in cycles) (LAT), and kernels' estimated frequency ($Fmax$), which we extract from AOC's pre-synthesis report, in addition to the profiled loop trip counts (TC) provided using `isNonZeroLoop`. The total execution time is calculated as follows:

$$CC_L = \begin{cases} TC_L \times II_L & L \in ImL \quad (1) \\ TC_L \times \sum_{i \in IL(L)} CC_i & L \in OL \quad (2) \\ \left(TC_L \times \sum_{i \in IL(L)} CC_i\right) + \sum_{i \in \{L, IL(L)\}} LAT_i & L \in OmL \quad (3) \end{cases}$$

$$CC_N = \sum_{L_i \in OmL(N)} CC_{L_i} \quad (4)$$

$$CC_{Total} = CriticalPath(CC_N),\ \forall N \in DFG \quad (5)$$

$$T_{exe} = CC_{Total}/F_{max} \quad (6)$$

First, within a leaf node (kernel), the cycle count (CC_L) of a pipelined loop nest is calculated recursively such that: a) for every innermost loop it is the loop's trip count times its initiation interval (Eq. 1); b) for every outer loop it is the loop's trip count times the sum of all its inner loops' cycle counts (Eq. 2); and c) for the outermost loop in the loop nest, it would be the same as (b) plus the total latency of all the basic blocks in all the loop nest bodies (i.e. time to drain the pipeline) (Eq. 3). If a loop nest is not pipelined, the LAT_i term should instead be added to every loop's own cycle count. This is done in our estimation model, but omitted from the equations for simplicity.

Next, the cycle count of the DFG node (kernel) N is calculated as the sum of all the cycle counts of all the loop nests in that node (Eq. 4). The total cycle count of the accelerated portion of the application (i.e. the HPVM DFG) is calculated using a critical path analysis through the HPVM DFG (Eq. 5). Finally, the total execution time (in seconds) is calculated using Eq. 6.

We use the *feasibility* feature of HM which allows it to learn when a combination of parameter values may result in infeasible designs as DSE progresses. We determine the feasibility of a design point (i.e. whether or not it fits on the FPGA) by extracting the estimated resource utilization from the AOC reports. The estimated execution time and validity of the sample are then sent back to HM in our response.

Finally, we performed a set of meta-experiments to tune HM's hyperparameters for our specific use case, i.e., HPVM2FPGA with an Arria 10 GX target FPGA. This involved studying the type of random sampling to use to initialize the Bayesian optimization phase (random sampling versus latin hypercube sampling), whether or not to use batching (which has the effect of speeding up the evaluations by running them in parallel but has the effect of reducing the statistical efficiency of the optimization procedure), for how many iterations to perform DSE, etc. These settings were tuned so that users won't have to tune them themselves; however they can be modified as command-line arguments to our system. We used Random Forests as HM's surrogate model since it is recognized to be flexible for small data, discrete and discontinuous design spaces, which is generally true for computer system designs.

D. HPVM2FPGA Back End and Runtime Extensions

The HPVM2FPGA back end comprises of: 1) HPVM-to-OpenCL code generation, 2) Intel FPGA OpenCL Compiler (AOC) for kernel synthesis and bit-stream generation, and 3) the LLVM x86 back end for host-code binary generation. Out of these three components, the HPVM-to-OpenCL code-gen is our contribution.

HPVM-to-OpenCL code-gen happens in two steps, an HPVM-to-LLVM transformation and an LLVM-to-OpenCL transformation. The HPVM-to-LLVM transformation is performed by a set of back end passes. It starts by creating a new LLVM module that will house all the FPGA kernel code (i.e. the Kernel Module). Then, a bottom-up traversal of the DFG is performed. For each HPVM leaf node that is targeted to the FPGA (i.e. FPGA kernel), the transform starts by running Node Sequentialization if the node hasn't been already sequentialized. Then, a new LLVM function is created from the original leaf node function with the following extra changes: all pointer arguments are marked to be in the *global memory address space* by inserting the LLVM `addrspace` attribute and an LLVM Metadata entry is created to mark that this function corresponds to a kernel. The newly created kernel function is inserted into the Kernel Module. A special map is used to record that the node's parent will be responsible for launching this node. For each internal node, the transform traverses the node's children in topological order, and if the child is an FPGA kernel, generates the required HPVM runtime calls that would set up and launch that kernel as described below. All the tasks in an internal node are launched first, and then they are all waited on. This ensures that concurrent tasks can run in parallel if Task Parallelism is enabled. To make sure kernels are executed in the correct order, the transform generates an "*event list*" for each kernel, which gets used by the run-time to synchronize them. Additionally, the transform generates the remaining host code which handles the DFG and any leaf nodes that are targeted to the CPU into the Host Module. The output of the pass will be the Kernel Module and the Host Module.

The LLVM-to-OpenCL transformation takes the LLVM kernel Module that is generated by the previous transform, and generates an OpenCL file that contains the kernels. This transformation is based on the deprecated LLVM "C" back end. We extended the deprecated back end with a few key features to function as a usable OpenCL code generator. First, the deprecated back end generated loops using infinite while loops and `go to` statements to handle all types of loops and conditionals. This would not allow AOC to detect the loops in the kernels and optimize them. As such, we added the necessary functionality that correctly generates `for` loops and `if-else` statements in the kernel body. Next, we added functionality that parses our metada, extra attributes and intrinsic instructions to correctly generate the necessary qualifiers and keywords (`__global`, `__kernel`) and the `ivdep` pragma. Lastly, we updated the deprecated back end to use the version of LLVM that we use in HPVM2FPGA.

The HPVM2FPGA run-time, which is an extension of the HPVM run-time, provides necessary functions that interface with the OpenCL runtime system of the Intel FPGA SDK for OpenCL. These include calls that set up an *OpenCL Context* for the platform, create *OpenCL Kernel* objects, create *OpenCL Buffers* for kernel arguments, set the kernel arguments, and launch the kernels. In order to support concurrent execution of kernels for Task Parallelism, the run-time can use separate command queues for each kernel, depending on which calls are generated by the back end. OpenCL events are used to synchronize the kernels based on their dependencies in the DFG. Additionally, the HPVM2FPGA uses a memory tracker which keeps track of the location of each pointer (on the host or on the device). This memory tracker is checked when memory locations are queried (either directly using the HeteroC++ API, or when setting the kernel argument) to make sure that memory is only copied when needed. Effectively, HPVM2FPGA transparently handles host and FPGA code-gen as well as automatically generating host-device communication boiler-plate code, without any input from the programmer.

IV. Experimental Evaluation

A. Methodology

We evaluate our framework on a selection of benchmarks, ranging from real-world applications with multiple kernels, to computational kernel benchmarks. Our applications include a 3D spatial audio encoder (*Audio*) from the Illinois Extended Reality (ILLIXR) testbed [18], a camera vision pipeline (*CAVA*), and an image processing edge detection pipeline (*Edge*) [11]. For computational kernels, we selected four multi-kernel benchmarks from the Rodinia benchmark suite [19], which have been used for multiple heterogeneous system studies and some also hand-tuned for the Arria 10 FPGA [20]. These benchmarks are breadth-first search (BFS), backpropagation (BP), and two algorithms of computational fluid dynamics: euler (Euler) and euler with precomputed fluxes (Pre-Euler). We also selected eight single-kernel benchmarks from MachSuite [21], which we used in an experiment to study the individual contributions of our optimizations. We ported each application and unoptimized computational kernel benchmark to hardware-agnostic code in HPVM-C or Hetero-C++, adding the necessary hardware-agnostic annotations where applicable (`restrict`, `private`, and `isNonZeroLoop`).

Our evaluation setup uses an Arria 10 GX FPGA Development Kit with 2GB on-board memory, connected over PCIe to an Intel Xeon W-2775 host CPU with 256 GB main memory. For synthesis, we use the Intel FPGA SDK for OpenCL 19.3.

We performed three main experiments to evaluate our compiler. First, an evaluation of our optimization framework with DSE, second, a comparison of HPVM2FPGA optimized code to hand-tuned OpenCL kernels, and finally, a study of the contribution of each optimization to the overall speedups.

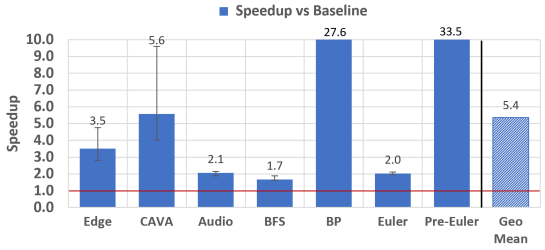

Fig. 2: HPVM2FPGA DSE vs unoptimized baselines.

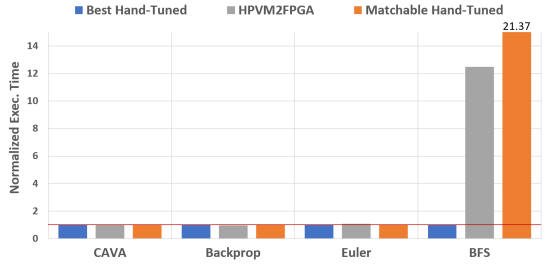

Fig. 3: HPVM2FPGA vs hand-tuned kernels. BFS, BP, and Euler were tuned by Zohouri in [22] and CAVA was tuned by us in [4].

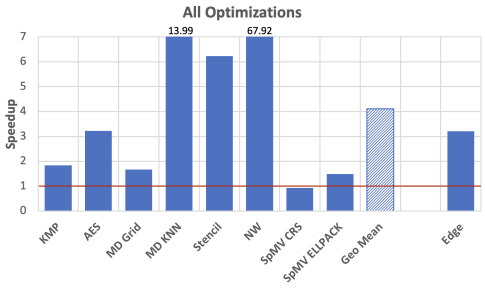

Fig. 4: Speedups on MachSuite and Edge Detection with all optimizations applied (without DSE) compared to unoptimized baselines.

B. DSE Evaluation

We compiled all our multi-kernel benchmarks using HPVM2FPGA with DSE enabled. Each DSE run was repeated five times to account for statistical variation caused by the randomness in HM. The number of DSE iterations was tuned along with the other HM hyperparameters, as described in III-C. At the end of DSE, we collected the best version generated by each repetition, and ran that on the FPGA to measure its performance, averaged over five runs.

To study the performance that our optimization framework can achieve, we compared the DSE-generated designs to a version compiled using HPVM2FPGA without applying any optimizations. Our results are shown in Figure 2, where each column shows the speedup compared to baseline, and the error bars show the variation over the five DSE runs. The figure's y-axis is cut-off at 10 for clarity, and the error bars for the cut-off columns (BP and Pre-Euler) were insignificant.

Our framework was able to achieve a geometric mean speedup of $5.4\times$ compared to baseline across all our benchmarks. For Edge Detection and CAVA, we respectively see a $3.5\times$ and $5.6\times$ speedup compared to the baseline. These applications each include 5 separate kernels, with complex loop structures as well as potential for task parallelism, where manual tuning to achieve similar speedups requires significant restructuring and effort by the programmer. Note that we see larger variation between the five DSE runs for these benchmarks compared to the others due to the large number of categorical parameters, which tend to have more variability with Bayesian Optimization. Audio Encoder achieves a modest speedup of $2.1\times$, due to memory indirection in the kernels that prevents our optimizations from ruling out loop-carried dependencies. For the Rodinia benchmarks, our speedups range from $1.7\times$ to $33.5\times$, depending on the effectiveness of our optimizations on each benchmark.

Given that these benchmarks represent different workloads, with different characteristics, this shows that our framework is effective on a wide variety of workload types, and we expect these speedups to improve as HPVM2FPGA matures with more optimizations.

C. Comparison to hand-tuned code

Next, to study how the code that we generate compares to hand-tuned FPGA kernels, we synthesized the versions of those Rodinia benchmarks that were hand-optimized by Zohouri in [20], [22] and compared them to the versions that were optimized using HPVM2FPGA. These benchmarks were BP, Euler, and BFS (Pre-Euler lacked an optimized version in the repository). We also compared CAVA optimized using HPVM2FPGA to a version that we manually tuned in [4]. Figure 3 shows our results. For each benchmark, we took the *best hand-tuned* version (i.e. the best out of all the available versions in the Rodinia repository, and the best version we tuned by hand in [4]; blue bar) and compared that against the one generated by HPVM2FPGA with DSE by synthesizing and running both on the same FPGA. We found that HPVM2FPGA is able to match the performance of the best version in cases where the hand tuning did not require a different programming model (e.g. NDRange Kernels instead of SWI Kernels) or a significant optimization not currently supported (e.g. using channels for pipelining). This was case for CAVA, Backprop, and Euler. In the case of BFS, the best hand-tuned version used NDRange Kernels, and therefore HPVM2FPGA was unable to match its performance since it can only generate SWI Kernels.

To evaluate how well HPVM2FPGA was able to do given this limitation, we took the best BFS version in the repo that uses SWI Kernels and compared that against HPVM2FPGA. This is the *matchable hand-tuned* version (orange bar) in the figure. The comparison shows that HPVM2FPGA was able to outperform the"matchable" version. This is due to the more exhaustive search on optimizations that HPVM2FPGA can perform with DSE, compared to hand-tuning.

Importantly, HPVM2FPGA was able to achieve these results automatically from unmodified code, whereas hand-optimizing for FPGAs takes significant time and effort. We expect HPVM2FPGA to match hand-tuned designs in more cases as it matures with more optimizations and more support for more backends (e.g. NDRange Kernel generation).

D. Contribution of Optimizations

To analyze the impact of each optimization separately, and in the presence of all the other optimizations, we compiled eight MachSuite benchmarks and the edge detection pipeline using HPVM2FPGA with different combinations of optimizations applied. In this case the optimization framework was used without DSE, and the selected optimizations were applied by the framework (automatically) using heuristics as we described earlier. Our heuristics always consider Loop Unrolling and Loop Fusion together and always fuse all fusible nodes with Node Fusion, as described earlier. Note that the MachSuite benchmarks are single kernel benchmarks, so Node Fusion and Task Parallelism do not apply to them. Also, Input Buffering and `ivdep` Insertion were not included in this study because they did not provide any considerable speedup on these benchmarks[2].

Figure 4 shows the speedup with all the optimizations applied. Figures 5(a-d) show the results where each optimization is applied alone. Figures 5(e-h) show the results where the corresponding optimization is removed, while the rest are all applied. The light blue bars in these figures show the maximum speedup from Figure 4 for ease of comparison. Note that we only show edge detection when studying the inter-kernel optimizations (i.e. Figures 5(c,d,g,h)) because these optimizations don't apply to MachSuite.

Argument Privatization (AP) (Figure 5(a)) provides speedups ranging from $2\times$ to $66\times$ on MachSuite. This optimization tends to help more in cases where the privatized argument is larger and accessed more frequently, which is clearly evident in the NW benchmark. This is also evident when AP is removed from the version with all the optimizations (Figure 5(e)). The figure indicates that all the speedup of NW is coming from AP[3].

Loop Unrolling (LU) and Loop Fusion (LF) (Figure 5(b)) provide speedups ranging from $2\times$ to $14\times$. Together, these two optimizations increase pipeline parallelism and spatial parallelism by unrolling inner loops, or unrolling outer loops and fusing the resulting inner loops. Looking at Figure 5(f), and comparing that to (e), we can see that LU and LF are responsible for the majority of the speedups for most benchmarks, except NW and AES whose speedups are primarily coming from AP.

Node Fusion (NF) (Figure 5(c)) fused all nodes of Edge together, and did not provide speedup on its own. However, when it is removed (Figure 5(g)), the speedup is less than the maximum. This indicates that NF is providing more opportunities for the other optimizations (like LF and LU), thus enabling higher speedups, which we expect.

Finally, Task Parallelism (TP) seems to only provide a small speedup of 10% (Figure 5(d)) when applied to Edge Detection, since the kernels that can run in parallel constitute a small percentage of the total execution time. We note that removing TP does not decrease the speedups (Figure 5(h)) because NF is fusing all the nodes of Edge together, removing the potential for TP. We expect TP to provide more substantial speedups in applications where the parallel tasks constitute a larger portion of the total execution time.

[2]The MachSuite experiment was performed before `ivdep` insertion was added to our framework. The experiment will be repeated for the final version of the paper.

[3]The slowdown in NW is due to IB, which reduced the overall bandwidth, causing other memory accesses to stall more often.

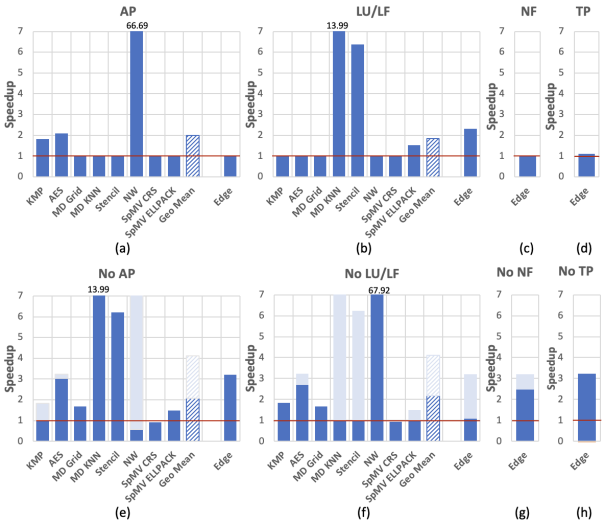

Fig. 5: Speedups on MachSuite and Edge Detection achieved by applying different combinations of optimizations. Light blue bars show the speedup when all opts are enabled (Figure 4)

V. Related Work

A. HLS Tools

The most relevant related work is presented in Table II.

HeteroCL [7], and Spatial [6] provide higher-level abstractions and sophisticated methods for specifying optimizations and low-level hardware details. While they do utilize compiler optimizations, their languages do not aim to abstract the hardware details, but rather help expert hardware designers easily specify the detailed hardware optimizations they need for high-performance designs. Additionally, HeteroCL does not perform DSE, and while Spatial's DSE is also based on HyperMapper, the parameters have to be explicitly defined by the programmer, whereas our framework automatically extracts the parameters from the program. Unlike HPVM2FPGA, neither compiler performs inter-kernel optimizations. Finally, HeteroCL does not generate host code, while Spatial does, although it does not automatically infer all host-device communication details.

ScaleHLS [8] is a new framework that does hardware-agnostic code-generation for FPGAs by relying on compiler optimizations and DSE. However, unlike HPVM2FPGA, they do not perform inter-kernel optimizations or support host-device partitioning and host code generation. Additionally, their DSE optimization engine is based on a simple hill-climbing strategy compared to ours, which uses HyperMapper's Bayesian Optimization framework. Also, they only tune parameters globally, while we can tune them at a fine-grain level for every loop, function, and argument.

Artisan [10] and Pylog [9] support hardware-agnostic input programs and support host-device code partitioning. However, unlike HPVM2FPGA, PyLog's DSE has a very limited scope, and Artisan performs very simple tuning of parameters that is limited to unroll factors. Also, both do not perform any inter-kernel optimizations.

TABLE II: HPVM2FPGA vs state-of-the art HLS compilers.

Feature	Hetero CL [7]	Spatial [6]	Artisan [10]	PyLog [9]	Scale HLS [8]	HPVM2 FPGA
HW Agnostic	✗	✗	✓	✓	✓	✓
Inter-Kernel Opts	✗	✗	✗	✗	✗	✓
Fine-grain DSE	✗	✓	✗	✗	✗	✓
Host Code	✗	✓	✓	✓	✗	✓
Automatic Host-Device Comm	✗	✓	✓	✓	✗	✓

B. Design Space Exploration in HLS

There is significant work in the literature about design space exploration techniques in HLS [23]–[27]. However, most existing work focuses on using DSE for tuning HLS parameters, rather than using it to select compiler optimizations [23], [24], [27]. We view these related works as orthogonal and complementary to our work. Our goal is not the design space exploration framework itself, but rather using DSE as a component in an end-to-end system that enables hardware-agnostic programming of FPGAs, with a focus on compiler techniques and optimizations. HPVM2FPGA's modular infrastructure allows a compiler designer to easily replace the existing DSE framework with relatively little effort.

VI. Conclusion

We presented HPVM2FPGA, a *novel and extensible end-to-end system* that enables more powerful **hardware-agnostic programming** of FPGAs. HPVM2FPGA uses a suitable hardware-agnostic abstraction of parallelism (HPVM IR), and introduces a powerful optimization framework that uses sophisticated compiler optimizations and design space exploration (DSE) to automatically tune a hardware-agnostic program for a given FPGA. Our framework accepts full hardware-agnostic applications parallelized via a HPVM-supported language, and transparently handles the host-FPGA interaction using our runtime system. Our goal is to provide the research community with a framework that can act as a basis for hardware-agnostic FPGA programming research, and that would keep on improving with more compiler optimizations, back end code-generation techniques, and performance estimation models/DSE.

References

[1] A. De La Piedra, A. Braeken, and A. Touhafi, "Sensor systems based on fpgas and their applications: A survey," *Sensors*, vol. 12, no. 9, pp. 12 235–12 264, 2012.

[2] R. Ricart-Sanchez, P. Malagon, P. Salva-Garcia, E. C. Perez, Q. Wang, and J. M. Alcaraz Calero, "Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks," *Journal of Network and Computer Applications*, vol. 124, pp. 80–93, 2018.

[3] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 513–517, 2016.

[4] A. Ejjeh, V. Adve, and R. A. Rutenbar, "Studying the potential of automatic optimizations in the intel fpga sdk for opencl," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[5] J. Pu *et al.*, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[6] D. Koeplinger *et al.*, "Spatial: a language and compiler for application accelerators," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018.

[7] Y.-H. Lai *et al.*, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019.

[8] H. Ye *et al.*, "Scalehls: Scalable high-level synthesis through mlir," *arXiv preprint arXiv:2107.11673*, 2021.

[9] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. W. Hwu, "Pylog: An algorithm-centric python-based fpga programming and synthesis flow," *IEEE Transactions on Computers*, 2021.

[10] J. Vandebon, J. G. Coutinho, W. Luk, E. Nurvitadhi, and T. Todman, "Artisan: a meta-programming approach for codifying optimisation strategies," in *Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020.

[11] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[13] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019.

[14] A. Souza, L. Nardi, L. B. Oliveira, K. Olukotun, M. Lindauer, and F. Hutter, "Bayesian optimization with a prior for the optimum," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2021.

[15] B. Bodin *et al.*, "Integrating algorithmic parameters into benchmarking and design space exploration in 3d scene understanding," in *International Conference on Parallel Architectures and Compilation*, 2016.

[16] P. Tu and D. Padua, "Automatic array privatization," in *Compiler optimizations for scalable parallel systems*. Springer, 2001.

[17] J. Warren, "A hierarchical basis for reordering transformations," in *SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984.

[18] M. Huzaifa *et al.*, "Exploring extended reality with illixr: A new playground for architecture research," 2021.

[19] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2009.

[20] H. R. Zohouri, "High performance computing with fpgas and opencl," Ph.D. dissertation, Tokyo Institute of Technology, Tokyo, Japan, 2018.

[21] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2014.

[22] H. R. Zohouri, "fpga-opencl-benchmarks/rodinia_fpga." [Online]. Available: https://github.com/fpga-opencl-benchmarks/rodinia_fpga

[23] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Auto-tuning high-level synthesis for fpgas using opentuner and legup," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017.

[24] L. Ferretti, A. Cini, G. Zacharopoulos, C. Alippi, and L. Pozzi, "A graph deep learning framework for high-level synthesis design space exploration," 2021.

[25] N. Wu, Y. Xie, and C. Hao, "Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning," in *Great Lakes Symposium on VLSI*, 2021.

[26] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2021.

[27] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.