# HPVM: *Heterogeneous Parallel Virtual Machine*

Maria Kotsifakou[*]
Department of Computer Science
University of Illinois at
Urbana-Champaign
kotsifa2@illinois.edu

Prakalp Srivastava[*]
Department of Computer Science
University of Illinois at
Urbana-Champaign
psrivas2@illinois.edu

Matthew D. Sinclair
Department of Computer Science
University of Illinois at
Urbana-Champaign
mdsincl2@illinois.edu

Rakesh Komuravelli
Qualcomm Technologies Inc.
rakesh.komuravelli@qti.qualcomm.
com

Vikram Adve
Department of Computer Science
University of Illinois at
Urbana-Champaign
vadve@illinois.edu

Sarita Adve
Department of Computer Science
University of Illinois at
Urbana-Champaign
sadve@illinois.edu

## Abstract

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

***CCS Concepts*** • **Computer systems organization → Heterogeneous (hybrid) systems**;

***Keywords*** Virtual ISA, Compiler, Parallel IR, Heterogeneous Systems, GPU, Vector SIMD

## 1 Introduction

Heterogeneous parallel systems are becoming increasingly popular in systems ranging from portable mobile devices to high-end supercomputers to data centers. Such systems are attractive because they use specialized computing elements, including GPUs, vector hardware, FPGAs, and domain-specific accelerators, that can greatly improve energy efficiency, performance, or both, compared with traditional homogeneous systems. A major drawback, however, is that programming heterogeneous systems is extremely challenging at multiple levels: algorithm designers, application developers, parallel language designers, compiler developers and hardware engineers must all reason about performance, scalability, and portability across many different combinations of diverse parallel hardware.

At a fundamental level, we believe these challenges arise from three root causes: (1) diverse hardware parallelism models; (2) diverse memory architectures; and (3) diverse hardware instruction sets. Some widely used heterogeneous systems, such as GPUs, partially address these problems by defining a *virtual instruction set* (ISA) spanning one or more families of devices, e.g., PTX for NVIDIA GPUs, HSAIL for GPUs from several vendors and SPIR for devices running OpenCL. Software can be shipped in virtual ISA form and then translated to the native ISA for execution on a supported device within the target family at install time or runtime, thus achieving portability of "virtual object code" across the

[*] The first two authors contributed equally to the paper.

*PPoPP '18, February 24–28, 2018, Vienna, Austria*

ACM ISBN 978-1-4503-4982-6/18/02.
https://doi.org/10.1145/3178487.3178493

corresponding family of devices. Except for SPIR, which is essentially a lower-level representation of the OpenCL language, these virtual ISAs are primarily focused on GPUs and do not specifically address other hardware classes, like vector hardware or FPGAs. Moreover, *none of these virtual ISAs* aim to address the other challenges, such as algorithm design, language design, and compiler optimizations, across diverse heterogeneous devices.

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

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

*No previous system we know of can achieve all three purposes, and most can only achieve one of the three.* None of the existing virtual ISAs — PTX, HSAIL, SPIR — can serve as either compiler IRs (because they are designed purely as executable instruction sets, not a basis for analysis or transformation) or as a basis for flexible runtime scheduling across a heterogeneous system (because they lack sufficient flexibility to support this). No previous parallel compiler IR we know of (for example, [9, 19, 29, 32, 36]) can be used as a virtual ISA for shipping programs for heterogeneous systems (as they are not designed to be fully executable representations, though some, like Tapir [32] for homogeneous shared memory systems, could be extended to do so), and none can be used as a parallel program representation for runtime scheduling (because they are not retained after static translation to native code, which is a non-trivial design challenge).

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

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

We describe a prototype system (also called HPVM) that supports all three capabilities listed earlier. The system defines a compiler IR as an extension of the LLVM IR [30] by adding HPVM abstractions as a higher-level layer describing the parallel structure of a program.

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

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

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

Overall, our contributions are as follows:

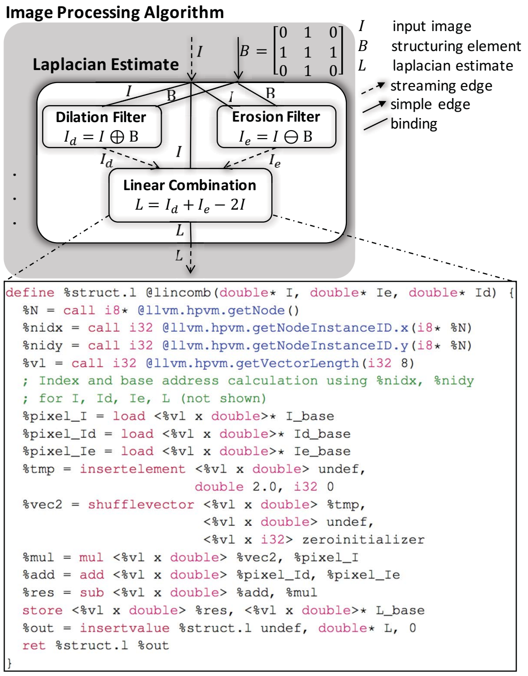
- We develop a parallel program representation (HPVM) for heterogeneous parallel systems based on a hierarchical dataflow graph with side effects, which captures essentially all the important kinds of task- and data-parallelism on heterogeneous systems.
- We show that HPVM can be used as an effective parallel compiler IR, that can support important optimizations like node fusion and tiling.
- We show that HPVM can be used to create an effective parallel virtual ISA for heterogeneous systems by (a) using HPVM as a *persistent* representation of programs, and (b) by implementing translators from HPVM to three different classes of parallel hardware: GPUs, vector SIMD, and multicore CPUs.
- We show that HPVM dataflow graphs can be used to support flexible static and dynamic scheduling policies, that take full advantage of the ability to translate individual HPVM graph nodes to multiple hardware.
- Finally, we implement HPVM on top of a widely used compiler infrastructure, LLVM, which historically has lacked any *explicit* support for heterogeneous parallel systems in the LLVM IR, potentially contributing a valuable new capability for the LLVM community.

## 2 HPVM Parallelism Abstractions

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

### 2.1 Dataflow Graph

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



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

**Figure 1.** Non-linear Laplacian computation in HPVM

Nodes may access globally shared memory through load and store instructions ("side-effects"), since hardware shared memory is increasingly common across heterogeneous systems. These operations may result in *implicit* data transfers, depending on the mapping of nodes to hardware compute units and on the underlying memory system. Because of these side effects, HPVM is not a "pure dataflow" model. Figure 1 shows the HPVM version of a Laplacian estimate computation of a greyscale image, used as part of image processing filters. This will be used as a running example. The figure shows the components of the Laplacian as separate dataflow nodes – `Dilation Filter (DF)`,`Erosion Filter (EF)` and `Linear Combination (LC)` – connected by edges. The figure also shows the code for node `LC`, which is standard LLVM IR except for the new intrinsic functions named `llvm.hpvm.*`, explained later. Load/store instructions access shared memory, using pointers that must be received explicitly from preceding nodes.

#### 2.1.1 Dynamic Instances of Nodes and Edges

The dataflow graphs in HPVM can describe varying (data-dependent) *degrees of parallelism* at each node. In particular, a single static dataflow node or edge represents multiple dynamic instances of the node or edge, each executing the same code with different index values. The dynamic instances of a node are required to be independent of each other, so that each time a (static) node is executed on receiving a new set of data items, all dynamic instances of the node may execute in parallel, similar to invoking a parallel loop. The dynamic instances form an n-dimensional grid, with integer indexes in each dimension, accessed via the `llvm.hpvm.getNodeInstanceID.*` functions. Our implementation allows up to three dimensions. For example, the `LC` node in the example is replicated to have $dim_X \times dim_Y$ instances, where $dim_X$ and $dim_Y$ are computed at runtime. Similarly, a static edge between two static nodes may represent multiple dynamic edges between dynamic instances of the two nodes, as explained further in Section 2.1.3.

### 2.1.2 Dataflow Node Hierarchy

Each dataflow node in a DFG can either be a *leaf node* or an *internal node*. An internal node contains a complete dataflow graph, called a *child graph*, and the child graph itself can have internal nodes and leaf nodes. In Figure 1, the node `Laplacian Estimate` is an internal node, and its child graph comprises the leaf nodes `DF`, `EF`, and `LC`.

A leaf node contains scalar and/or vector code, expressing actual computations. Dynamic instances of a leaf node capture independent computations, and a single instance of a leaf node can contain fine-grain vector parallelism. Leaf nodes may contain instructions to query the structure of the underlying dataflow graph, as described in Section 3.

Internal nodes describe the structure of the child graph. The internal nodes are traversed by the translators to construct a static graph and generate code for the leaf nodes and edges (Section 4). One restriction of this model is that the dataflow graph cannot be modified at runtime, e.g., by data-dependent code, dynamically spawning new nodes; this enables fully-static optimization and code generation at the cost of some expressivity.

The grouping and hierarchy of parallelism has several advantages. It helps express several different kinds of parallelism in a compact and intuitive manner: coarse-grain task (i.e., pipelined) parallelism via top-level nodes connected using dataflow edges; independent coarse- or fine-grained data parallelism via dynamic instances of a single static node; and fine-grained data parallelism via vector instructions within single instances of leaf nodes. It provides a flexible and powerful mechanism to express tiling of computations for memory hierarchy in a portable manner (Section 6.3). It enables efficient scheduling of the execution of the dataflow graph by grouping together appropriate sets of dataflow nodes. For example, a runtime scheduler could choose to map a single top-level (internal) node to a GPU or to one core of a multicore CPU, instead of having to manage potentially large numbers of finer-grain nodes. Finally, it supports a high-degree of modularity by allowing separate compilation of parallel components, represented as individual dataflow graphs that can be composed into larger programs.

### 2.1.3 Dataflow Edges and Bindings

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

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

Figure 1 shows the dataflow edges describing the data movement of input image $I$, dilated image $I_d$, eroded image $I_e$, and matrix $B$ between dataflow nodes. Some edges (e.g., input $B$ to node `Laplacian Estimate`) are "*fixed*" edges: their semantics is as if they repeatedly transfer the same data for each node execution. In practice, they are treated as a constant across node executions, which avoids unnecessary data transfers (after the first execution on a device).

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

## 2.2 Vector Instructions

The leaf nodes of a dataflow graph can contain explicit vector instructions, in addition to scalar code. We allow parametric vector lengths to enable better performance portability, i.e., more efficient execution of the same HPVM code on various vector hardware. The vector length for a relevant vector type need not be a fixed constant in the HPVM code, but it must be a *translation-time constant* for a given vector hardware target. This means that the parametric vector types simply get lowered to regular, fixed-size vector types during native code generation. Figure 1 shows an example of parametric vector length (`%vl`) computation and use.

Evaluating the effect of parametric vector lengths on performance is out of the scope of this paper because we only support one vector target (Intel AVX) for now. Moreover, in all the benchmarks we evaluate, we find that vectorizing

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

**Table 1.** How HPVM represents major parallel hardware features

across dynamic instances of a leaf node is more effective than using explicit vector code, as explained in Sections 4.2.2 and 7.2. More complex vector benchmarks, however, may benefit from the explicit vector instructions.

### 2.3 Integration with Host Code

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

### 2.4 Discussion

An important consideration in the design of HPVM is to enable efficient mapping of code to key features of various target hardware. We focus on three kinds of parallel hardware in this work: GPUs, vectors, and multithreaded CPUs. Table 1 describes which HPVM code constructs are mapped to the key features of these three hardware families. This mapping is the role of the translators described in Section 4. The table is a fairly comprehensive list of the major hardware features used by parallel computations, showing that HPVM is effective at capturing different kinds of hardware.

## 3 HPVM Virtual ISA and Compiler IR

We have developed a prototype system, also called HPVM, including a Compiler IR, a Virtual ISA, an optimizing compiler, and a runtime scheduler, all based on the HPVM representation The compiler IR is an extension of the LLVM IR, defined via LLVM intrinsic functions, and supports both code generation (Section 4) and optimization (Section 6) for heterogeneous parallel systems. The virtual ISA is essentially just an external, fully executable, assembly language representation of the compiler IR.

We define new instructions for describing and querying the structure of the dataflow graph, for memory management and synchronization, as well as for initiating and terminating execution of a graph. We express the new instructions as function calls to newly defined LLVM "*intrinsic functions*." These appear to existing LLVM passes simply as calls to unknown external functions, so no changes to existing passes are needed.

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

The intrinsics for describing graphs can only be "executed" by internal nodes; *all these intrinsics are interpreted by the compiler at code generation time and erased*, effectively fixing the graph structure. (Only the number of dynamic instances

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

**Table 2.** Intrinsic functions used to implement the HPVM internal representation. i$N$ is the $N$-bit integer type in LLVM.

of a node can be varied at runtime.) All other intrinsics are executable at run-time, and can only be used by leaf nodes or by host code.

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

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

## 4 Compilation Strategy

We only briefly describe the key aspects of the compilation strategy for lack of space.

### 4.1 Background

We begin with some background on how code generation works for a virtual instruction set, shown for HPVM in Figure 2. At the developer site, front-ends for one or more source languages lower source code into the HPVM IR. One or more optimizations may be optionally applied on this IR, to improve program performance, while retaining the IR structure and semantics. The possibly optimized code is written out in an object code or assembly language format, using the IR as a virtual ISA, and shipped to the user site (or associated server). A key property of HPVM (like LLVM [21]) is that the compiler IR and the virtual ISA are essentially identical. Once the target hardware becomes known (e.g., at the user site or server), the compiler backend is invoked. The backend traverses the Virtual ISA and uses one or more target-ISA-specific code generators to lower the program to executable native code.

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

In this paper, we focus on using HPVM for efficient code generation (this section) and optimizations (section 6). We leave front ends for source languages for future work. Note that we do rely on a good dataflow graph (representing parallelism, not too fine-grained nodes, good memory organization) for good code generation. This need can be met with a combination of parallelism information from suitable parallel programming languages (such as OpenMP or OpenCL), combined with the graph optimizations at the HPVM level, described in Section 6. We do *not* rely on precise static data

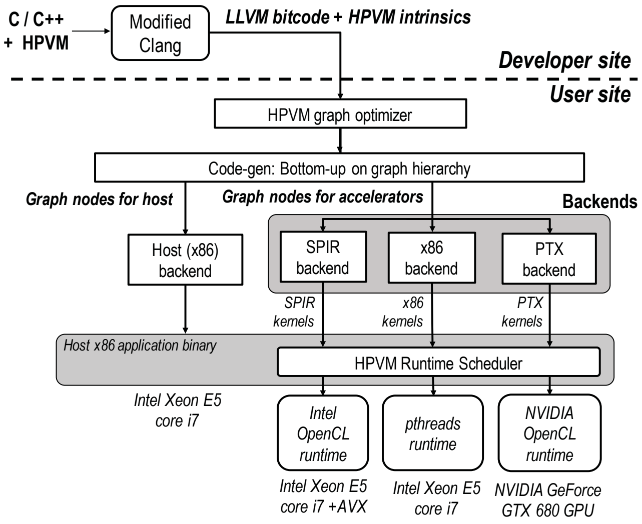


**Figure 2. Overview of compilation flow for HPVM.**

dependence analysis or precise knowledge of data transfers or memory accesses, which is important because it means that we can support irregular or data-dependent parallelism and access patterns effectively.

## 4.2 HPVM Compilation Flow

The HPVM compilation flow follows the structure shown in Figure 2. The compiler invokes separate back-ends, one for each target ISA. Each back end performs a depth-first traversal of the dataflow graph, maintaining the invariant that code generation is complete for all children in the graph hierarchy of a node, $N$, before performing code generation for $N$. Each back end performs native code generation for selected nodes, and associates each translated node with a host function that implements the node's functionality on the target device.

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

### 4.2.1 Code Generation for PTX

The PTX [27] backend builds on the existing NVPTX backend in LLVM. This back end translates an extended version of the LLVM IR called NVVM (containing PTX-specific intrinsic functions) [28] into PTX assembly.

A node annotated for GPU will usually contain a two-level or three-level DFG, depending on whether or not the computation must be tiled, as shown in Table 1 and explained in Section 6.3. Our translator for PTX takes as input the internal node containing this DFG. It generates an NVVM kernel function for each leaf node, which will execute the dynamic instances of the leaf node. If the DFG is a three-level graph, and the second (thread block) level node contains an *allocation node* (defined as a leaf node that allocates memory using the `llvm.hpvm.malloc` intrinsic), the allocated memory is assigned to scratchpad memory, as explained in Section 6.3. All other memory is allocated by the translator to GPU global memory or GPU constant memory. The generated NVVM kernel is translated to PTX by the NVPTX back-end. Our translator also generates code to use the NVIDIA OpenCL runtime to load and run the PTX assembly of the leaf node on the GPU. This code is the host function associated with the input dataflow node on the GPU.

### 4.2.2 Code Generation for AVX

Dynamic instances of leaf nodes are independent, making it possible to vectorize *across* node instances. We leverage Intel's translator from SPIR [20] to AVX, which is part of Intel's OpenCL runtime system, for two reasons: it recognizes and utilizes the independence of SPIR work items to produce vector parallelism, and it is well tuned to produce efficient code for the AVX instruction set. Instead of writing our own AVX code-generator directly from HPVM with these sophisticated capabilities, we instead wrote a translator that converts HPVM code to SPIR, in which the dynamic instances of leaf nodes become SPIR work items. The SPIR code is then vectorized for AVX by Intel's translator. Our translator also creates the necessary host function to initiate the execution of the SPIR kernel.

### 4.2.3 Host Code Generation

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

- Native code for all nodes annotated as CPU nodes. We build upon the LLVM X86 backend for regular LLVM IR, adding support for HPVM query intrinsics. We translate `createNode` operations to loops enumerating the dynamic instances, and dataflow edges to appropriate data transfers (section 4.2.4).
- For nodes with multiple native versions, i.e. annotated with more than one target, a wrapper function that invokes the HPVM runtime scheduler (section 5) to choose which target function to execute on every invocation.
- Host-side coordination code, enforcing the order of execution dictated by the dataflow graph.
- Code to initiate and terminate execution of each dataflow graph.

### 4.2.4 Data Movement

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

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

It is important to avoid unnecessary data copies between devices for good performance. To that end, we allow explicit attributes `in`, `out`, and `inout` on node arguments, and only generate the specified data movement. Achieving the same effect without annotations would require an interprocedural May-Mod analysis [1] for pointer arguments, which we aim to avoid as a requirement for such a key optimization.

# 5 HPVM Runtime and Scheduling Framework

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

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

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

1. *Static Node Assignment*: Always schedule a dataflow node on a fixed, manually specified target, so the target depends only on the node identifier.
2. *Static Data Item Assignment*: Schedule all nodes of a graph for a particular input data item on a single target, so the target depends only on the data item number.
3. *Dynamic*: A dynamic policy that uses the node identifier as in policy #1 above, plus instantaneous availability of each device: when a specified device is unavailable, it uses the CPU instead.

We leave it to future work to experiment with more sophisticated scheduling policies within the framework. In this paper, we simply aim to show that we offer the flexibility to support flexible runtime scheduling decisions. For example, the second and third policies above could use a wide range of algorithms to select the target device per data item among all available devices. The key to the flexibility is that HPVM allows individual dataflow graph nodes to be compiled to any of the targets.

# 6 Compiler Optimization

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

## 6.1 Node Fusion

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

Two nodes $N1$ and $N2$ are valid node fusion candidates if: (1) they both are (a) leafs, or (b) internal nodes containing an optional allocation node (see Section 4.2.1) and a single other leaf node (which we call the *compute node*); (2) they have the same parent, target, dimensions and size in each dimension, and, if they are internal nodes, so do their compute nodes and their optional allocation nodes; and (3) they are either concurrent (no path of edges connects them), or they are connected directly by 1-1 edges and there is no data transfer between $N1$'s compute and $N2$'s allocation node, if any.

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

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

### 6.2 Mapping Data to GPU Constant Memory

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

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

### 6.3 Memory Tiling

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

Memory can be allocated for each tile using the `llvm.hpvm.malloc` intrinsic in a single allocation node (see Section 4.2.1), which passes the resulting pointer to all instances of the leaf node representing the tile. This memory would be assigned to scratchpad memory on a GPU or left in global memory and get transparently cached on the CPU.

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

We have used this mechanism to create tiled versions of four of the seven Parboil benchmarks evaluated in Section 7. The tile sizes are determined by the programmer in our experiments. For the three benchmarks (`sgemm, tpacf, bfs`) for which non-tiled versions were available, the tiled versions achieved a mean speedup of 19x on GPU and 10x on AVX, with `sgemm` getting as high as 31x speedup on AVX.

## 7 Evaluation

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

### 7.1 Experimental Setup and Benchmarks

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

For the performance portability and hand-coded comparisons, we used 7 OpenCL applications from the Parboil benchmark suite [33]: Sparse Matrix Vector Multiplication (spmv), Single-precision Matrix Multiplication (sgemm), Stencil PDE solver (stencil), Lattice-Boltzmann (lbm), Breadth-first search (bfs), Two Point Angular Correlation Function (tpacf), and Distance-cutoff Coulombic Potential (cutcp).

In the GPU experiments, our baseline for comparison is the best available OpenCL implementation. For spvm, sgemm, stencil, lbm, bfs and cutcp, that is the Parboil version labeled `opencl_nvidia`, which has been hand-tuned for the Tesla NVIDIA GPUs [22]. For tpacf, the best is the generic Parboil version labeled `opencl_base`. We further optimized the codes by removing unnecessary data copies (bfs) and global barriers (tpacf, cutcp). All the applications are compiled using NVIDIA's proprietary OpenCL compiler.

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

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

### 7.2 Portability and Comparison with Hand Tuning

Figures 3 and 4 show the execution time of these applications on GPU and vector hardware respectively, normalized to the baselines mentioned above. Each bar shows segments for the time spent in the compute kernel (Kernel), copying data (Copy) and remaining time on the host. The total execution time for the baseline is shown above the bar.

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

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

### 7.3 Evaluation of Flexible Scheduling

We used a six-stage image processing pipeline, Edge Detection in grey scale images, to evaluate the flexibility that HPVM provides in scheduling the execution of programs consisting of many dataflow nodes. The application accepts a stream of grey scale images, $I$, and a fixed mask $B$ and computes a stream of binary images, $E$, that represent the edges of $I$. We feed 1280x1280 pixel frames from a video as the input and measure the frame rate at the output. This pipeline is natural to express in HPVM. The streaming edges and pipeline stages simply map to key features of HPVM, and the representation is similar to the code presented in Figure 1. In contrast, expressing pipelined streaming parallelism in OpenCL, PTX, SPIR or HSAIL, although possible, is extremely awkward, as explained briefly in Section 8.

Expressing this example in HPVM allows for flexibly mapping each stage to one of three targets (GPU, vector or a CPU thread), for a total of $3^6 = 729$ configurations, all generated from a single HPVM code. Figure 5 shows the frame rate of 7 such configurations. The figure shows that HPVM can capture pipelined, streaming computations effectively with good speedups. More importantly, however, the experiment shows that HPVM is flexible enough to allow a wide range of *static* mapping configurations with very different performance characteristics from a single code.

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

In this environment, we execute the Edge Detection pipeline using the three different scheduling policies described in Section 5. Figure 6 shows the *instantaneous frame rate* for each policy. Green and red sections show when the GPU is available or not respectively. We truncate the Y-axis because the interesting behavior is at lower frame rates; the suppressed peak rates are about 64 frame/s.

Static node assignment policy makes no progress during the intervals when the GPU is not available. The other two policies are able to adapt and make progress even when the GPU is unavailable, though neither is perfect. Static data item assignment policy may or may not continue executing when the GPU is unavailable, depending on when the data items that will be issued to the GPU are processed. Also, it may have low frame rate when the GPU is available, if data items that should be processed by the CPU execute while the GPU is available. Dynamic policy will not start using the GPU to execute a dataflow node for a data item until the node is done for the previous data item. That is why the frame rate does not immediately increase to the maximum when the GPU becomes available. The experiment shows HPVM enables flexible scheduling policies that can take advantage of static and dynamic information, and these policies are easy to implement directly on the HPVM graph representation.

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

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

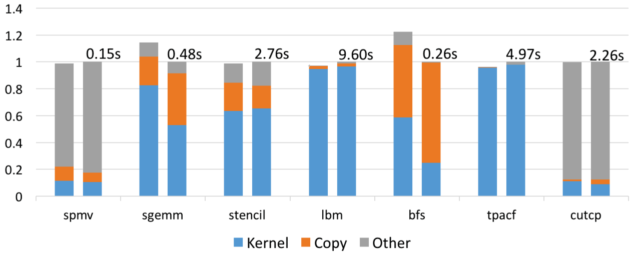


**Figure 3.** GPU Experiments - Normalized Execution Time. For each benchmark, left bar is HPVM and right bar is OpenCL.

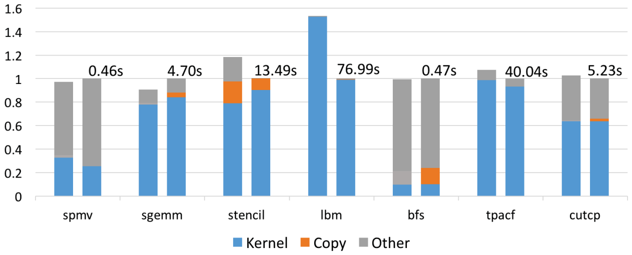


**Figure 4.** Vector Experiments - Normalized Execution Time. For each benchmark, left bar is HPVM and right bar is OpenCL.

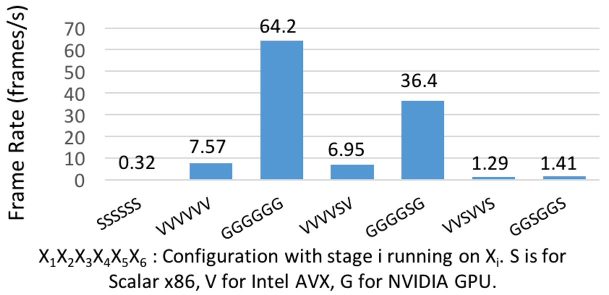


**Figure 5.** Frame rates of different configurations of Edge Detection six stage pipeline through single HPVM object code.

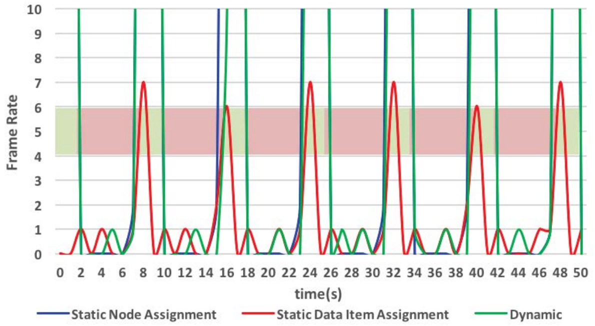


**Figure 6.** Edge Detection Frame rate with different scheduling policies. The green and red band in the graph indicates when the GPU is available or not respectively.

### 7.4 Node Fusion Optimization Evaluation

We evaluated the benefits of Node Fusion using two widely used kernels, Laplacian Estimate ($L$) and Gradient Computation ($G$). Most benchmarks we examined have been hand-tuned to apply such transformations manually, making it hard to find Node Fusion opportunities (although they may often be more natural to write without manual node fusion). The two kernels' dataflow graphs have similar structure, shown for $L$ in Figure 1. We compiled the codes to run entirely on GPU and fed the same video frames as before. Fusing just the two independent nodes gave a speedup of 3.7% and 12.8% on $L$ and $G$ respectively. Fusing all three nodes yielded a speedup of 10.6% and 30.8% on $L$ and $G$ respectively. These experiments show that Node Fusion can yield significant gains, but the benefits depend heavily on which nodes are fused.

## 8 Related Work

There is a long history of work on dataflow execution models, programming languages, and compiler systems for homogeneous parallel systems [3, 12, 15, 16, 18, 26, 32, 38]. HPVM aims to adapt the dataflow model to modern heterogeneous parallel hardware. We focus below on programming technologies for heterogeneous systems.

**Virtual ISAs:** NVIDIA's PTX virtual ISA provides portability across NVIDIA GPUs of different sizes and generations. HSAIL [14] and SPIR [20] both provide a portable object code distribution format for a wider class of heterogeneous systems, including GPUs, vectors and CPUs. All these systems implement a model that can be described as a "grid of kernel functions," which captures individual parallel loop nests well, but more complex parallel structures (such as the 6-node pipeline DAG used in our Edge Detection example) are only expressed via explicit, low-level data movement and kernel coordination. This makes the underlying model unsuitable for use as a retargetable compiler IR, or for flexible runtime scheduling. Finally, it is difficult, at best, to express some important kinds of parallelism, such as pipelined parallelism (important for streaming applications), because all buffering, synchronization, etc., must be implemented explicitly by the program. In contrast, pipelined parallelism can be expressed easily and succinctly in HPVM, in addition to coarse- or fine-grain data-parallelism.

**Compiler IRs with Explicit Parallel Representations:** We focus on parallel compilers for heterogeneous systems. The closest relevant compiler work is OSCAR [25, 29, 36], which uses a hierarchical task dependence graph as a parallel program representation for their compiler IR. They do *not* use this representation as a virtual ISA, which means they cannot provide object code portability. Their graph edges represent data and control dependences, *not dataflow* (despite the name), which is well suited to shared memory systems but not as informative for non-shared memory. In particular, for explicit data transfers, the compiler must infer automatically what data must be moved (e.g., host memory to accelerator memory). They use hierarchical graphs only for the (homogeneous) host processors, not for accelerators, because they do not aim to perform parallel compiler transformations for code running within an accelerator nor runtime scheduling choices for such code. KIMBLE [8, 9] adds a hierarchical parallel program representation to GCC, while SPIRE [19] defines a methodology for sequential to parallel IR extension. Neither KIMBLE nor SPIRE make any claim to, or give evidence of, performance portability or parallel object code portability.

**Runtime Libraries for Heterogeneous Systems:** Parallel Virtual Machine (PVM) [17] enables a network of diverse machines to be used cooperatively for parallel computation. Despite the similarity in the names, the systems have different goals. What is virtualized in PVM are the application programming interfaces for task management and communication across diverse operating systems, to achieve portability and performance across homogeneous parallel systems. HPVM virtualizes the parallel execution behavior and the parallel hardware ISAs, to enable portability and performance across heterogeneous parallel hardware, including GPUs, vector hardware, and potentially FPGAs.

Several other runtime systems [4, 7, 24, 31] support scheduling and executing parallel programs on heterogeneous parallel systems. Habanero-Java [37] and Habanero-C [23], provide an abstraction of heterogeneous systems called *Hierarchical Place Trees*, which can be used to express and support flexible mapping of parallel programs. None of these systems provide program representations that can be used to define a compiler IR or a virtual ISA.

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

PENCIL [5] is a programming language defined as a restricted subset of C99, intended as an implementation language for libraries and a compilation target for DSLs. Its compiler uses the polyhedral model to optimize code and is combined with an auto-tuning framework. It shares the goals of source code portability and performance portability with HPVM. However, it is designed as a readable language with high-level optimization directives rather than as a compiler IR, per se, and it also does not address object code portability.

StreamIt [35] and CnC [10] are programming languages with a somewhat more general representation for streaming pipelines. They, however, focus on stream parallelism, whereas HPVM supports both streaming and non-streaming parallelism. This is crucial when defining a compiler IR or a virtual ISA for parallel systems (of any kind), because most parallel languages (e.g., OpenMP, OpenCL, CUDA, Chapel, etc.) are used for non-streaming parallel programs.

Legion [6] is a programming model and runtime system for heterogeneous architectures. It provides abstractions for describing the structure of program data in a machine independent way. Similarly, Sequoia [13] provides rich memory abstractions to enable explicit control over movement and placement of data at all levels of a heterogeneous memory hierarchy. HPVM lacks these features, but does express tiling effectively and portably using the hierarchical graphs. In future, we aim to add richer memory abstractions to HPVM.

Petabricks [2] explores the search space of different algorithm choices and how they map to CPU and GPU processors. In Tangram [11], a program is written in interchangeable, composable building blocks, enabling architecture-specific algorithm and implementation selection. Exploring algorithm choices is orthogonal to, and can be combined with, our approach.

Delite [34] is a library for developing compiled, embedded DSLs inside the programming language Scala. The Delite execution graph encodes the dependencies between computations. To provide flexibility to run these computations on different hardware devices, Delite relies on the DSL developers to provide Scala, CUDA, OpenCL implementations of these computations as necessary for efficiency. HPVM on the other hand relies on the hardware vendors to provide platform specific implementation of computations in HPVM IR. The broader Delite approach can be combined with HPVM approach to ease burden on the DSL developers.

## 9 Conclusion

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

We conclude that HPVM is a promising basis for achieving performance portability and for implementing parallelizing compilers and schedulers for heterogeneous parallel systems.

## Acknowledgments

This work was supported in part by the National Science Foundation through grants CCF 13-02641 and CCF 16-19245, by C-FAR – one of the six SRC STARnet Centers sponsored by MARCO and DARPA, and by Intel Corporation.

## References

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

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice *(PLDI).*

[3] E. A. Ashcroft and W. W. Wadge. 1977. Lucid, a Nonprocedural Language with Iteration. *Commun. ACM* (1977).

[4] CÃĺdric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-AndrÃĺ Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* (2011).

[5] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15).* IEEE Computer Society, Washington, DC, USA, 138–149.

[6] Michael Bauer, Sean Treichler, Elliot Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions *(SC).*

[7] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17).* ACM, New York, NY, USA, 235–248.

[8] Nicolas Benoit and Stéphane Louise. 2010. Extending GCC with a Multi-grain Parallelism Adaptation Framework for MPSoCs. In *2nd Int'l Workshop on GCC Research Opportunities.*

[9] Nicolas Benoit and Stéphane Louise. 2016. Using an Intermediate Representation to Map Workloads on Heterogeneous Parallel Systems. In *24th Euromicro Conference.*

[10] Zoran Budimlic, Michael Burke, Vincent CavÃĺ, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. 2010. Concurrent Collections. *Scientific Programming* 18, 3-4 (2010), 203–217.

[11] Li-wen Chang, Abdul Dakkak, Christopher I. Rodrigues, and Wen mei Hwu. 2015. Tangram: a High-level Language for Performance Portable Code Synthesis *(MULTIPROG 2015).*

[12] D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. Voneicken. 1993. TAM - A Compiler Controlled Threaded Abstract Machine. *Parallel and Distributed Computing.*

[13] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy *(SC).*

[14] HSA Foundation. 2015. HSAIL. (2015). Retrieved January 17, 2018from http://www.hsafoundation.com/standards/

[15] Vladimir Gajinov, Srdjan Stipic, Osman S. Unsal, Tim Harris, Eduard Ayguadé, and Adrián Cristal. 2012. Integrating Dataflow Abstractions into the Shared Memory Model. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing.* 243–251.

[16] Vladimir Gajinov, Srdjan Stipic, Osman S. Unsal, Tim Harris, Eduard Ayguadé, and Adrián Cristal. 2012. Supporting Stateful Tasks in a Dataflow Graph. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12).* ACM, New York, NY, USA, 435–436.

[17] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. 1994. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT press.

[18] Google. 2013. Google Cloud Dataflow. (2013). Retrieved January 17, 2018from https://cloud.google.com/dataflow/

[19] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. 2012. SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension *(CPC).*

[20] Khronos Group. 2012. SPIR 1.2 Specification. https://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf. (2012).

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Conf. on Code Generation and Optimization.* San Jose, CA, USA, 75–88.

[22] Li-wen Chang. 2015. Personal Communication. (2015).

[23] D. Majeti and V. Sarkar. 2015. Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors *(IPDPS Workshop).*

[24] Tim Mattson, Romain Cledat, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Bala Seshasayee, Wijngaart Rob van der, and Vivek Sarkar. 2015. *OCR: The Open Community Runtime Interface.* Technical Report.

[25] Takamichi Miyamoto, Saori Asaka, Hiroki Mikami, Masayoshi Mase, Yasutaka Wada, Hirofumi Nakano, Keiji Kimura, and Hironori Kasahara. 2008. Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications.* IEEE.

[26] Rishiyur S. Nikhil. 1993. The Parallel Programming Language Id and its compilation for parallel machines *(IJHSC).*

[27] NVIDIA. 2009. PTX: Parallel Thread Execution ISA. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html. (2009).

[28] NVIDIA. 2013. NVVM IR. http://docs.nvidia.com/cuda/nvvm-ir-spec. (2013).

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

[30] LLVM Project. 2003. LLVM Language Reference Manual. (2003). Retrieved January 17, 2018from http://llvm.org/docs/LangRef.html

[31] Qualcomm Technologies, Inc. 2014. *MARE: Enabling Applications for Heterogeneous Mobile Devices.* Technical Report.

[32] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation *(PPoPP).*

[33] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. *Parboil: A revised benchmark suite for scientific and commercial throughput computing.* Technical Report.

[34] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages *(ACM TECS).*

[35] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications *(International Conference on Compiler Construction).*

[36] Yasutaka Wada, Akihiro Hayashi, Takeshi Masuura, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, and Hironori Kasahara. 2011. *A Parallelizing Compiler Cooperative Heterogeneous Multicore Processor Architecture.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[37] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. 2009. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09).* Springer-Verlag, Berlin, Heidelberg, 172–187.

[38] Jin Zhou and Brian Demsky. 2010. Bamboo: A Data-centric, Object-oriented Approach to Many-core Software. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10).* ACM, New York, NY, USA, 388–399.