

*h*VISC: A Portable Virtual Instruction Set for Heterogeneous Parallel Systems

Author Name

Affiliation

Email

Abstract

Heterogeneous computing is widely used in the System-on-chip (SoC) processors that power modern mobile devices in order to reduce power consumption through specialization. However, programming such systems can be extremely complex as a single SoC combines multiple different parallelism models, instruction sets, and memory hierarchies, and different SoCs use *different combinations* of these features. We propose *hVISC*, a new Virtual Instruction Set Architecture (ISA) that aims to address both functional portability and performance portability across mobile heterogeneous SoCs by capturing the wide range of different parallelism models expected to be available on future SoCs. Our virtual ISA design uses only two parallelism models to achieve this goal: *a hierarchical dataflow graph with side effects* and *parametric vector instructions*. *hVISC* is more general than existing ones that focus heavily on GPUs, such as PTX, HSAIL and SPIR, e.g., it can capture both streaming pipelined parallelism and general dataflow parallelism found in many custom and semi-custom (programmable) accelerators. We present a compilation strategy to generate code for a diverse range of target hardware components from the common virtual ISA. As a first prototype, we have implemented backends for GPUs that use nVidia's PTX, vector hardware using Intel's AVX, and host code running on X86 processors. Experimental results show that code generated for vectors and GPUs from a single virtual ISA representation achieves performance that is within about a factor of 2x of separately hand-tuned code, and much closer in most cases. We further demonstrate qualitatively using a realistic example that our virtual ISA abstractions are also suited for capturing pipelining and streaming parallelism.

1. Introduction

In computing contexts where energy is an important consideration, such as in mobile devices like smartphones, tablets, and e-book readers, or where power and heat dissipation are important, such as in data centers, traditional homogeneous multicore processors can be quite inefficient. These contexts are increasingly seeing the advent of heterogeneous computing systems, which use specialized computing elements that can deliver much greater efficiency in

performance-per-Joule or performance-per-Watt. For example, the “application processor” on a modern smartphone or tablet is a heterogeneous System-on-chip (SoC) that often includes not just a multicore host CPU, but also a GPU, a DSP, and several more specialized processors for tasks such as audio and video decoding, image processing, digital photography, and speech recognition.

Programming applications for hardware that uses such diverse combinations of computing elements is extremely challenging. The challenges include developing portable algorithms, writing efficient yet portable source-level programs, producing portable object code, and tuning the programs. At a more fundamental level, these challenges arise from three root causes: (1) diverse parallelism models; (2) diverse memory architectures; and (3) diverse hardware instruction sets. To make use of the full range of available hardware to maximize performance and energy efficiency, the programming environment needs to provide common abstractions for all the available hardware compute units in heterogeneous systems. Not only are these abstractions required at the level of source-code, but also at object-code level to make the object-code portable across the same and different manufacturer's devices, thus allowing application vendors to be able to ship a single software version across a broad range of devices.

We believe that these issues are best addressed using a virtual instruction set layer that abstracts away most of the low-level details of different hardware components, but provides a small number of abstractions of parallelism that can be mapped down (or “translated”) effectively to all the different kinds of parallel hardware on a wide range of SoCs. The (virtual) object code is translated down to specific hardware components available on a particular device, at install time, load time or run-time. This general approach, which we call Virtual Instruction Set Computing (VISC), has been used very successfully for GPGPU computing, e.g., through the PTX virtual ISA for several generations of nVidia GPUs, and more recently HSAIL [1] and SPIR [8] for other classes of hardware. Although HSAIL and SPIR can be mapped down to non-GPU hardware, their design has been heavily influenced by the SIMT parallelism model of GPUs, which supports both GPU and vector hardware well but limits their effectiveness for other kinds of parallelism. This is discussed in more detail in Sections 5.3 and 6.

In this paper, we propose a virtual ISA design that abstracts away the wide range of parallelism models and the disparate instruction sets used within and across SoCs. (In this work, we do not consider the different memory hierarchy architectures used across compute units or devices, but it is a subject of our ongoing work.) In fact, we can represent these different parallelism models using only *two abstractions of parallelism*:

- Hierarchical dataflow graphs with side effects, and
- Short-vector SIMD (Single Instruction Multiple Data) instructions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CONF 'yy, Month d-d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-xxxx-xxxx-n/yy/mm...\$15.00.
<http://dx.doi.org/10.1145/nnnnnnnn.nnnnnnn>

Dataflow graphs are a very general model of data parallelism and, when extended to allow shared memory accesses (side effects), can capture many forms of parallel computing over data elements, including vector SIMD parallelism, the SIMT (Single Instruction Multiple Threads) parallelism model used in general-purpose GPUs, streaming or pipelined-dataflow parallelism, and fine-grained data parallelism, which may be synchronous or asynchronous. Although dataflow graphs can capture vector parallelism too, vector instructions, when applicable, provide a representation that is far more compact, efficient, and much easier to reason about and transform; for this reason, we include explicit vector instructions in our model.

We make the dataflow graphs hierarchical to express multiple granularities of parallelism in a natural manner, e.g., coarse-grain parallelism across different compute units vs. fine-grain parallelism within a single compute unit. In particular, a dataflow graph node is either an *internal node* or a *leaf node*. An internal node itself contains another dataflow graph within it. A leaf node contains executable code that is some mixture of scalar and vector instructions. Each leaf node in a dataflow graph includes a parameter value, N , which specifies that the node should be *replicated* N times for independent parallel execution; the value of N may be computed at run-time. This allows the graph to capture fine-grain parallelism, and is similar to how a GPU kernel in CUDA, OpenCL or PTX is replicated across the threads of a GPU device.

One final feature of our representation is that a dataflow graph edge may be either an ordinary edge or a “streaming” edge. An ordinary edge represents a one-time data transfer from a producer node to a consumer node; implicitly the two nodes connected by the edge are executed only once. A streaming edge specifies that the producer and consumer edges execute repeatedly, transferring data items continuously with the semantics of a bounded FIFO buffer.

This code representation can be mapped down and executed effectively on the full range of parallel hardware on a modern SoC, including GPUs, vector hardware, multicore host processors, digital signal processors (DSPs), and semi-custom hardware accelerators. In this work, we describe a first prototype system that translates a single virtual object code program to nVidia GPUs (using PTX), Intel’s AVX vector instructions, and X86 host processors. We present preliminary experimental results comparing the performance of the generated code for a set of benchmarks to hand-tuned code written using OpenCL for the GPU and hand-vectorized for AVX. Our results show that the code generated by *hVISC* is close in performance to the hand-tuned code in many cases, and within about 2x in all cases. These results were obtained with relatively little compiler optimization for either GPU or vector hardware, which gives us confidence that *hVISC* can provide object code portability with relatively low performance cost.

We also present a detailed description of a pipelined streaming benchmark and how it is represented in *hVISC*. Representing this benchmark in PTX, HSAIL or SPIR would be extremely awkward: it would require manually written tiling and buffering, with complicated synchronization to achieve concurrent execution of different pipeline stages. Although we have not yet implemented the buffered message passing required for streaming parallelism, the example shows that *hVISC* can naturally express a broader class of parallelism than can be expressed with the existing virtual ISAs. We also briefly discuss an example class of programmable, custom accelerators for machine learning algorithms, which can be naturally targeted using the parallelism models in *hVISC*, although capturing all the details of the hardware is a subject of future work.

The next section describes the high-level design goals of *hVISC*. Section 3 then presents the detailed design of the *hVISC* virtual ISA, and its implementation as an extension of the LLVM instruction set [10]. Section 4 describes our general compilation strategy,

and our prototype translators for PTX, AVX, and X86. Section 5 presents our experimental results and our qualitative discussion of the pipelined benchmark and our future work on the machine learning accelerator. Section 6 compares our work with the state of the art, and Section 7 concludes.

2. Virtual ISA Design Goals

Previous work [5, 6] has shown that the approach of a virtual ISA can achieve both high performance and be commercially viable. In this work, we aim to design a virtual ISA for the wide range of parallel hardware configurations found in current and future mobile SoCs. We briefly summarize the primary design goals of our virtual ISA:

Object code portability with as good performance as possible:

The key goal of our virtual ISA design is to enable the *virtual object code* to be portable across a wide range of different configurations of heterogeneous parallel SoCs, while obtaining as good performance as possible on each compute unit. We emphasize that we do not necessarily aim to match manually tuned code for individual compute units because such tuning usually comes at the cost of portability, or at the cost of hurting performance on other compute units. Object code portability is an absolute requirement for modern applications running on mobile hardware. Applications that absolutely require hand-tuned performance can already use conditionally compiled code or *fat binaries* or both to achieve such performance, at the cost of significantly greater programming and maintenance effort.

Language independence: The virtual ISA should be able to support a wide range of parallel programming languages, such as OpenCL, Renderscript, and OpenMP 4.0 accelerator features. In particular, the virtual ISA is *not* intended as a source-level programming language, but the parallelism abstractions must be easy to reason about by programmers.

Machine independence: The virtual ISA should be able to support a wide range of hardware instruction sets, application binary interfaces (ABIs).

As few abstractions of parallelism as possible: The virtual ISA must use as few parallelism models as possible to capture the wide range of parallel hardware on a modern SoC. This is important so that programmers can design and tune algorithms without having to become experts in a wide range of different kinds of parallelism. These few abstractions must be able to map down effectively to today’s parallel hardware, such as multicore CPUs, GPUs and vectors, and also to emerging parallel hardware, especially semi-custom, programmable accelerators. (Custom, fixed-function accelerators may have high degrees of internal parallelism but are usually programmed via fairly straightforward library interfaces, which do not require exposing the detailed internal parallelism features.)

Coarse-grain parallelism across compute units: The virtual ISA must capture relatively large-grain parallelism mapped to different compute units, while compiling down to use as efficient data transfer mechanisms as possible between those compute units.

Coarse- and fine-grain parallelism within compute units: The virtual ISA must also capture both coarse- and fine-grain parallelism that can be mapped to a single compute unit, in order to achieve the highest possible parallel performance for each compute kernel on a wide range of compute units.

Representation of both explicit and implicit communication: It must be possible to represent both explicit data copies between compute units, e.g., between a CPU and an accelerator, and im-

PLICIT data transfers through shared memory, e.g., for a shared-memory multicore system or for emerging GPU hardware that allows direct sharing between CPU and GPU. Both kinds of memory transfers need to be under careful control of the programmer because memory accesses are often the primary determining factor in program performance.

Flexible scheduling support across compute units: It must be possible to compile kernels represented in the virtual ISA to multiple different compute units, so that a run-time scheduler can flexibly map a given kernel to different compute units, depending on availability constraints and battery conservation goals.

Offline compilation model: To minimize energy consumption and perceived application startup time, it should be possible to compile the virtual ISA ahead-of-time (AOT) to native machine code. For example, this was one major change from Android’s Dalvik virtual machine, which uses just-in-time (JIT) compilation every time an application is loaded, to the ART system, which uses AOT compilation once at install time.

3. Virtual ISA Design

This section presents *hVISC*, a virtual ISA design that abstracts away differences between parallelism models in hardware by exposing only two models of parallelism: hierarchical dataflow graphs with side effects and vector parallelism.

Figure 1 shows how an example of using *hVISC* for an image processing filter, specifically, a non-linear estimate of the Laplacian of a greyscale image. The estimate is computed by applying a dilation filter and an erosion filter in the input image and then computing a linear combination of the initial, the dilated and the eroded image. This example is used throughout the section to demonstrate the features of *hVISC*.

hVISC is implemented as an extension of the LLVM virtual instruction set [9], and the code fragments in our examples therefore use LLVM syntax [10].

3.1 Dataflow Graph

In *hVISC*, a program is represented as a hierarchical dataflow graph with side effects, where nodes represent units of execution, and dataflow edges describe the explicit data transfer requirements between these units of execution. If a pair of nodes (source and destination) is connected by a dataflow edge, the destination node logically must receive data from the source node before beginning execution.

The dataflow graph is a static representation. However, in order to express data parallelism we may have to represent a statically unknown number of node instances and/or edge instances, depending possibly on the size of the input. To that end, we allow a single static dataflow node to represent multiple dynamic instances of the node, i.e., a static node can be replicated at runtime and the resulting dynamic nodes can be executed independently of each other, subject only to the dependencies imposed by the dataflow edges. As described in Section 3.4.1, nodes may be replicated to form an n -dimensional grid; our current implementation allows up to three dimensions. Similarly, a static dataflow edge between two static dataflow nodes may represent multiple dynamic dataflow edges between dynamic instances of the two dataflow nodes.

For example, for an iterative four-point nearest-neighbor Jacobi solver that computes

$$A_{new}[i, j] = 0.25 * (A_{old}[i - 1, j] + A_{old}[i + 1, j] + A_{old}[i, j - 1] + A_{old}[i, j + 1])$$

on $N \times N$ matrices in each iteration, the static graph node could represent a single element-wise evaluation of the above equation

and would be replicated to create $N \times N$ independent dynamic instances.

Figure 1 demonstrates the components of the non-linear Laplacian estimate as separate dataflow nodes: DilationFilter, ErosionFilter and LinearCombination.

3.1.1 Dataflow Node Hierarchy

To allow for modularity and to capture multiple granularities of parallelism, the dataflow graph is hierarchical, i.e. each dataflow node can either be a *leaf node* or an *internal node*. A leaf node contains plain LLVM IR, expressing actual computations, which may be a mixture of scalar and vector operations. Vector parallelism is the only form of parallelism available in leaf nodes.

An internal node contains a complete dataflow graph, called a *child graph* of the current graph, and the child graph itself can have internal nodes and leaf nodes. This design allows for the programmer to represent logically connected operations performed in several dataflow nodes as a single dataflow node. This enhances the effectiveness of potential analyses by providing hints about closely related operations, and allows for the scheduler to efficiently orchestrate the execution of the dataflow graph by grouping together appropriate sets of dataflow nodes. For example, the run-time scheduler may choose to map a single top-level internal node to a GPU or to each core of a multicore CPU, instead of having to manage potentially large numbers of finer-grain nodes.

Leaf nodes may contain instructions to query about the structure of the underlying dataflow graph, as explained in more detail in Section 3.4.2. Also, they may contain side effects, i.e., load and store instructions accessing global shared memory, which express implicit data movement through a memory hierarchy. Because of these side effects, *hVISC* is not a “pure dataflow” model.

In Figure 1, the nodes comprising the Laplacian computation are children, in the hierarchy, of a top level node, LaplacianEstimate. DilationFilter, ErosionFilter, and LinearCombination are leaf dataflow nodes. The dilation and erosion filters compute the maximum and minimum, respectively, brightness in an area of a pixel defined by the binary structuring element B . The LinearCombination dataflow node performs the final computation. Figure 1 shows the LLVM instructions for this node, demonstrating the use of side effects and instructions querying the structure of the dataflow graph.

Note that the LaplacianEstimate dataflow node, although it is a top level node in this computation, it may itself become a child of a higher level dataflow node performing an image processing computation that requires the operation of a Laplacian. This highlights the importance of hierarchy for providing modularity and code reuse.

3.1.2 Dataflow Edges and Bindings

Explicit data movement between compute nodes is expressed with dataflow edges. A dataflow edge has the semantics of copying the corresponding data from the source to the destination dataflow node. Depending on where the execution of the source and destination is scheduled, the dataflow edge may be translated down to an explicit copy between compute units, or communication through shared memory.

As with dataflow nodes, static dataflow edges also represent multiple dynamic instances of dataflow edges between the dynamic instances of the source and the destination dataflow nodes. A dataflow edge between two static dataflow nodes can be instantiated at runtime using two different replication mechanisms: “all-to-all”, where all dynamic instances of the source node are connected with all the dynamic instances of the destination node, thus expressing a barrier between the two groups of nodes, or “one-to-one” where a single dynamic instance of the source dataflow node is connected with the corresponding instance of the destination node. One-to-one replication requires that the grid structure (number of dimen-

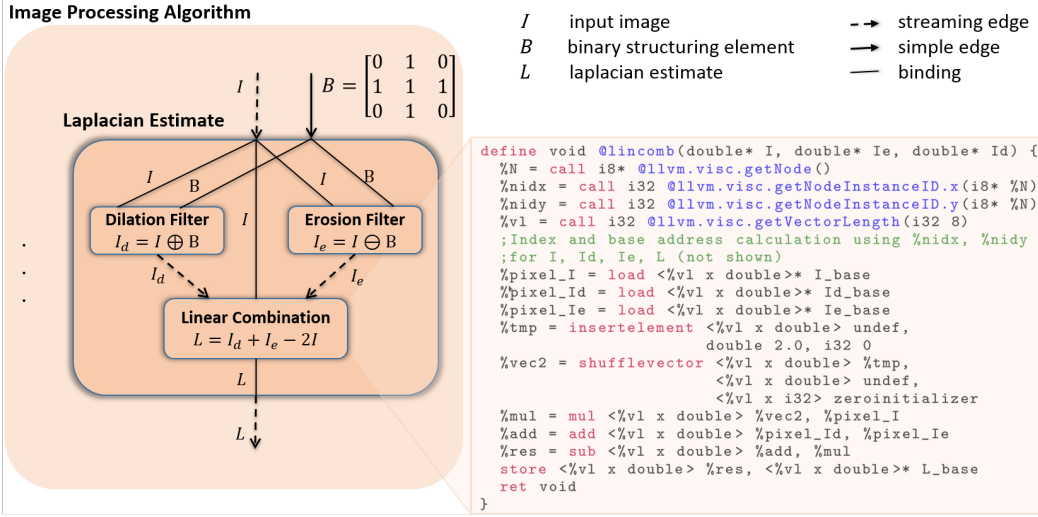


Figure 1. Non-linear Laplacian computation in *hVISC*

sions and the extents in each dimension) of the dynamic instances of the source and destination nodes is identical. One-to-one replication enables various optimizations at the dataflow graph level by expressing the exact dependency between the instances of the source and destination dataflow nodes. For example, an graph transformation pass could chose to merge two consecutive dataflow nodes, since the “one-to-one” replication denotes that a dynamic instance of the second node depends only on data generated from the corresponding instance of the first node.

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.

When an internal (“parent”) graph node contains an inner graph, the incoming edges of the parent node may provide the inputs to the one or more nodes of the child graph, and conversely with the outgoing edges. For example, in Figure 1, the inputs labeled I and B to node Laplacian Estimate provide inputs to the nodes Dilation Filter, Erosion Filter and Linear Combination of the child graph. Similarly, the output labeled L of node Linear Combination provides the output of the parent node. Semantically, these are *not* dataflow edges because no explicit data movement is implied: rather, these simply represent a *binding* between the input of a dataflow node to the input of a node within it, and the same for the outputs. We show these bindings as undirected edges in our diagrams, as in the figure. Dataflow edges always connect two nodes within the *same* graph, representing data transfer between the two nodes. Bindings always connect inputs or outputs of a parent node with those of the nodes in a child graph, and they represent a local assignment or renaming of input and output data.

3.1.3 Streaming Edges

Additionally, *hVISC* defines a special type of dataflow edge which we call a streaming edge, shown as dashed arrows instead of solid ones. Instead of a one-time data transfer that is expressed using ordinary dataflow edges, a streaming edge denotes that data items will be repeatedly transferred though this edge, and thus will need to be processed by the destination dataflow node. This allows the dataflow graph to express pipelining, as all nodes with incoming streaming edges will continue executing until the stream of data is finished. The stream processing is initiated and terminated by the code that sets up and initiates execution of the dataflow graph.

In Figure 1, the node Laplacian Estimate is a stage in an image processing pipeline that operates on a stream of incoming images. The edge I represents this stream. Correspondingly, I_d , I_e and the Laplacian estimate L are all streaming edges: they compute intermediate results and outputs for the Laplacian for each input image.

If a node has both streaming and ordinary input dataflow edges (e.g., I and B to node Laplacian Estimate), the simple edges repeatedly transfer the same data for each node execution, which in practice can be treated as a constant across node executions. This optimization allows unnecessary data transfers to be avoided.

3.2 Vector Instructions

The leaf nodes of a dataflow graph express the single-threaded parts of the computation. They contain ordinary LLVM IR, which includes both scalar and vector instructions. The LLVM virtual instruction set can be translated down for execution on a wide range of hardware, which provides a high degree of retargetability for *hVISC*.

We extend the LLVM vector instruction set with parametric vector lengths to enable better performance portability, i.e., more efficient execution of the same code on various vector hardware. Evaluating the effect of parametric vector length on performance is out of the scope of this paper, as for now we only support one vector target.

The LLVM IR provided for the LinearComputation in Figure 1 contains vector instructions, showing vector parallelism at the leaf level. The vector lengths are parametric, and are computed from the hardware vector length returned by `%llvm.visc.getVectorLength(i32 sz)`, which is a translation-time constant for a given hardware compute unit.

3.3 Integration with Host Code

hVISC is aimed to represent operations whose execution would benefit from executing on data-parallel hardware such as GPUs, vectors, and other accelerators. It is not intended for code that performs operations that are typically executed as host code. The host code contains ordinary LLVM IR for performing operations that cannot or should not be executed in accelerators such as file I/O, operations or calls to external libraries that may contain these operations, as well as initialization, memory allocation, or high level control flow decisions.

To integrate *hVISC*, the host code creates one or more Root dataflow nodes, each with a single dynamic instance, each containing a dataflow graph. Instantiating a root node at runtime translates to launching the execution of the contained dataflow graph. The result of this operation is the result of the dataflow graph execution, and can be accessed by the host code. The launch operation is asynchronous, allowing the host code to continue executing concurrently with the dataflow graph. The host code can also wait on the result of a dataflow graph execution at any point after launching the execution of that graph, ensuring that the computation is complete before accessing the result.

3.4 Implementation

We have implemented *hVISC* as an extension of the LLVM virtual instruction set. We define new instructions for manipulating and querying the structure of the data flow graph, as well as initiating execution of a dataflow graph. To minimize interference with existing LLVM compiler passes, we express the new instructions as function calls to intrinsic functions, a standard LLVM mechanism to extend the instruction set and communicate back end-specific information to a particular back end. A call to an intrinsic function appears to existing LLVM passes as a function call to an external function, i.e., it can only have side effects on externally visible global variables and on memory reachable through pointer arguments. This mechanism ensures that they do not perform any transformations that interfere with these instructions.

The functionality of each dataflow node is described by an explicit LLVM function. Functions describing internal nodes may only contain calls to *hVISC* intrinsics. Functions describing leaf nodes contain LLVM code with scalar and vector instructions and may also contain *hVISC* intrinsics used to query information about the structure of the dataflow graph; in particular, leaf nodes cannot use the intrinsics to define new graphs.

The LLVM dataflow intrinsics must refer to graph nodes and edges, in order to manipulate or query information about them. We represent dataflow nodes with opaque handles (pointers) and input and output edges of a node as integer indices. This allows the backend translator to define the structure and runtime representation of the nodes and edges. The LLVM type `i8*` is used for the opaque node handles. The *hVISC* intrinsics, divided according to their functionality, are described briefly in the following subsections.

3.4.1 *hVISC* Graph Intrinsics

hVISC intrinsics manipulating the structure of the dataflow graph:

- **`i8* llvm.visc.createNode1D(Function* F, int n)`**: Create a dataflow node with `n` dynamic instances, all associated with the function `F`. Returns the opaque handle for the node. There are also 2D and 3D versions of this intrinsic, which take two and three integer arguments instead of one.
- **`void llvm.visc.createEdge(i8* Src, i8* Dst, i32 sp, i32 dp, i1 ReplType)`**: Create a dataflow edge from node `Src` to node `Dst` in the static dataflow graph. The `ReplType` argument specifies the pattern of replication for the static edge: `OneToOne` or `AllToAll`. `sp` and `dp` specify the indices of the output of node `Src` and the input of node `Dst` that are connected by the edge; these connections are the same for all dynamic instances of the nodes, in either pattern of replication.
- **`void llvm.visc.createStreamingEdge(i8* Src, i8* Dst, i32 sp, i32 dp, i1 ReplType)`**: Similarly, but create a streaming dataflow edge.
- **`void llvm.visc.bind.input(i8* N, i32 ip, i32 ic)`**: Map input `ip` of current dataflow node to input `ic` of child node `N`.

- **`void llvm.visc.bind.output(i8* N, i32 ic, i32 ip)`**: Map output `ic` of child node `N` to output `ip` of current dataflow node. `N`.

3.4.2 *hVISC* Query Intrinsics

hVISC intrinsics querying the structure of the dataflow graph:

- **`i8* llvm.visc.getNode()`**: Return a handle to the dataflow graph node associated with the calling function, i.e. the current node.
- **`i8* llvm.visc.getParentNode(i8* N)`**: Return a handle to the hierarchical parent of dataflow graph node `N`.
- **`i32 llvm.visc.getNodeInstanceID.[xyz](i8* N)`**: Return the index of the dynamic node instance of dataflow node `N` with respect to its parent node in dimension `x`, `y` or `z`. (`z` is only valid if node `N` is replicated in 3D, and `y` in 2D or 3D.)
- **`i32 llvm.visc.getNumNodeInstances.[xyz](i8* N)`**: Return the number of dynamic instances of dataflow node `N` in dimension `x`, `y` or `z`.
- **`i32 llvm.visc.getVectorLength(i32 typeSz)`**: Return a symbolic constant representing the vector register length in the underlying architecture for a type of size `typeSz`.

3.4.3 *hVISC* Launch Intrinsics

hVISC intrinsics integrating a dataflow graph in the host code:

- **`i8* llvm.visc.launch(Function* F, argList, struct OutType* out)`**: This is a variation of `i8* llvm.visc.createNode(Function* F)` designed to allow for host variables to be passed to graph node inputs and results to be returned (unlike dataflow edges, which pass node outputs to other node inputs). It creates a single dynamic instance of a Root dataflow node and associates it with the function `F`, using `argList` as arguments. The struct `out` is allocated by the caller and is used to return results from the execution of the Root node; its type `OutType` must match the return type of `F`. The new node is marked as ready for asynchronous execution and control is returned to the host. Returns an opaque handle for the node.
- **`void llvm.visc.wait(i8* N)`**: Block until execution of dataflow node `N` is complete.

4. Compilation Strategy

The goal of our compilation strategy is to generate native code from a single virtual ISA format, allowing parts of an application to map flexibly to different compute units. Our goal, in this paper, is not to develop new optimization techniques on this virtual ISA; we are developing those techniques in our ongoing research. In this paper, we show how the virtual ISA design lends itself to be compiled piecewise to different hardware compute units.

We use simple annotations on the node functions to specify to which compute unit a given graph node should be translated, e.g., the annotation may specify one or more of `{GPU, Vector, None}`. Typically, the annotations would be chosen by a language front-end, a programmer, or (in future) a run-time scheduler that decided when a new version of native code was needed for a given subgraph. If an entire hierarchical graph will be compiled as a single kernel mapped to a single compute unit, then only the parent node of that graph needs to be annotated. The compiler will generate code for each such graph using the compilation flow described below.

Device-specific “translators” use this information to generate native code for a particular compute unit. Once mapping of nodes to different hardware components is done, the code generation for transfer of data between corresponding hardware components is generated. In future, virtual ISA compilers can allow more flexible

```

define {float*, i64} @Laplacian(float* in %I, i64 %sizeI, float* in %B, i64 %sizeB, i32 %dimX, i32 %dimY) {
; Create dataflow nodes in child graph
%erode_node = call i8* @llvm.visc.createNode(@erode)
%dilate_node = call i8* @llvm.visc.createNode(@dilate)
%lincomb_node = call i8* @llvm.visc.createNode2D(@lincomb, i32 %dimX, i32 %dimY)
; Bind inputs of parent node Laplacian with child nodes Dilate, Erode and lincomb
call void @llvm.visc.bind.input(i8* %dilate_node, i32 0, i32 0)
call void @llvm.visc.bind.input(i8* %dilate_node, i32 1, i32 1)
call void @llvm.visc.bind.input(i8* %dilate_node, i32 2, i32 2)
call void @llvm.visc.bind.input(i8* %dilate_node, i32 3, i32 3)
call void @llvm.visc.bind.input(i8* %erode_node, i32 0, i32 0)
call void @llvm.visc.bind.input(i8* %erode_node, i32 1, i32 1)
call void @llvm.visc.bind.input(i8* %erode_node, i32 2, i32 2)
call void @llvm.visc.bind.input(i8* %erode_node, i32 3, i32 3)
call void @llvm.visc.bind.input(i8* %lincomb_node, i32 0, i32 0)
call void @llvm.visc.bind.input(i8* %lincomb_node, i32 1, i32 1)
call void @llvm.visc.bind.input(i8* %lincomb_node, i32 2, i32 6)
call void @llvm.visc.bind.input(i8* %lincomb_node, i32 3, i32 7)
; Create edges between child nodes for sending output of Erode and Dilate to lincomb node
call void @llvm.visc.createEdge(i8* %dilate_node, i8* %lincomb_node, i32 0, i32 2)
call void @llvm.visc.createEdge(i8* %dilate_node, i8* %lincomb_node, i32 1, i32 3)
call void @llvm.visc.createEdge(i8* %erode_node, i8* %lincomb_node, i32 0, i32 4)
call void @llvm.visc.createEdge(i8* %erode_node, i8* %lincomb_node, i32 1, i32 5)
; Bind output of lincomb node with output of parent node Laplacian
call void @llvm.visc.bind.output(i8* %lincomb_node, i32 0, i32 0)
call void @llvm.visc.bind.output(i8* %lincomb_node, i32 1, i32 1)
ret {float*, i64} zeroinitializer
}

```

Listing 1. *hVISC* code for Laplacian node in Figure 1

mapping by generating native code for multiple targets for the same subgraph, and relying on the runtime and scheduler to perform data transfers when mapping of source and destination nodes of a dataflow edge are known at runtime.

Our current compilation strategy does not support cycles in a dataflow graph, although loops within leaf nodes present no problems. Outer-level cycles must be expressed in the host code outside the dataflow graphs, as we do for iterative algorithms (like stencil) and streaming computations (like the image processing pipeline described in Section 5.3).

4.1 Compilation Flow

The compilation flow for a virtual ISA program can be divided into three phases: (1) Mapping and code generation of distinct subgraphs to hardware accelerators, specifically, compute code for the annotated nodes. (2) Calls to the run-time library for data movement for the DFG edges. (3) Generating sequential code for the remaining unmapped parts of the graph. The latter phase – sequential code – is straightforward and is only briefly described in Section 4.2. The other two phases are described below.

The translation to native code is carried out for one annotated node at a time. The compilation requires traversal of the dataflow graph to find the annotated nodes and to translate each of them into native code for the selected compute unit. We use Algorithm 1 to traverse the hierarchical graph at find the annotated nodes. This algorithm is a simple depth-first traversal of the graph, translating each annotated node as it is encountered, as described below. The edges in the hierarchical graph between nodes belonging to the same child graph express dataflow edges that require run-time support for the data transfers.

4.1.1 Mapping Subgraphs to Accelerators

The annotations described earlier identify distinct subgraphs that should be mapped to specific compute units. For example, the subgraph containing Laplacian node in Figure 1 expresses parallelism well suited for a GPU, and assuming it is marked as such, the GPU translator would translate it for execution on an available GPU. It would first collapse the hierarchical graph at the node, N ,

Algorithm 1 Hierarchical Dataflow Graph Traversal

```

procedure VISIT(Node  $N$ )
  if  $N$  was visited before then return
  end if
  if  $N$  is an annotated node then
     $NN = \text{CollapseToLeaf}(N)$ 
     $\text{Translate}(NN)$ 
  else  $\triangleright N$  is an internal node
     $G \leftarrow$  child graph of node  $N$ 
     $L \leftarrow$  list of all nodes of  $G$  in topological order
    while  $L$  is non-empty do
      remove a node  $n$  from  $L$ 
       $\text{VISIT}(n)$ 
    end while
  end if
end procedure

```

into a single leaf node, NN , and then translate node NN to the specified compute unit. Collapsing a graph into a single node is conceptually straightforward, though it involves many steps, and the details are omitted here. To translate the leaf node, the translator isolates the functions associated with the node into a separate LLVM module and generates native code for it. The specific details of the translation are implementation specific, and are described below in Section 4.2. The final result of this phase is a new graph where all leaf nodes have been translated for execution on individual compute units.

4.1.2 Data Movement and Internal Nodes' Code Generation

The input to this phase is a graph where all leaf nodes have been mapped to hardware accelerators and contain target specific code. The compiler performs code generation of all the internal nodes of this graph, and for dataflow edges between nodes. The child graph of any internal node is traversed in topological order and function calls are inserted to the corresponding leaf node. For CPU code (e.g., targeting vector hardware), loops are inserted around a

function call if a static child node maps to multiple instances in the dynamic dataflow graph.

For data flow edges where the source and destination 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 would be enough. Such pointer passing is safe even with copy semantics because a dataflow edge implies that the source node must have *completed* execution before the sink node can begin, so the source code will not overwrite the data once the sink node begins execution. However, several accelerators today have separate memory hierarchy and data needs to be explicitly brought into the accelerator memory before starting the execution. In such cases explicit data copy instructions are generated using calls to the accelerator API. For example, we use OpenCL API calls to move data to and from the GPU.

4.2 Implementation

Our current compiler has functional translators for compiling the *hVISC* virtual ISA to PTX, AVX and host code for x86-64 (host code should also work for other architectures for which an LLVM backend and the OpenCL run-time are available). To reduce implementation effort for our prototype, we leverage existing back-ends in the mainline LLVM infrastructure for PTX (the open source NVPTX back end) and for AVX (the LLVM-to-SPIR back-end with Intel’s OpenCL SPIR-to-AVX translator). Our implementation then mainly has to translate our virtual ISA to the input code expected by each of these back-ends.

4.2.1 Translators

Our PTX translator takes the subgraph where an internal node has a single leaf node in its child graph, which is replicated into several dynamic instances. The PTX translator generates NVVM IR [13] for the leaf node. NVVM IR is a subset of the LLVM IR, together with a set of intrinsic functions, which the open source NVPTX backend can translate into PTX [6] assembly. For the internal node, our translator generates code to load and run the PTX assembly of the leaf node on the target nVidia GPU using the nVidia OpenCL runtime to execute the internal node.

In a similar fashion, our AVX translator generates SPIR [8] code for the leaf node and uses the Intel OpenCL [7] runtime to execute it on multicore CPUs supporting AVX extensions. The Intel SPIR translator to AVX has significant autovectorization capabilities that take advantage of the independence of SPIR kernel instances to produce vector code. Note that it is reasonable for us to reuse Intel’s vectorizer instead of writing our own because our goal is *not* to invent new vectorization and vector code generation technology; rather, our goal is to show that the *hVISC* virtual ISA is a suitable input code representation for enabling effective vectorization, which we can accomplish by feeding Intel’s SPIR translator from our virtual ISA.

OpenCL does not allow dynamic memory allocation inside the kernel. As a result, dataflow nodes which perform dynamic memory allocation cannot be compiled for GPUs. For nodes generating a data array as output, pointers to pre-allocated arrays are passed as inputs to a node. Thus, pointer arguments to a node can be pointers to both input or output data array. The general idiom we use to pass arrays is to provide a pointer to the array and the array size as arguments.

To differentiate between pointers to input/output data arrays, we add attributes *in*, *out*, and *inout* to node arguments as shown for input pointer *I* in Listing 1. These attributes enable us to avoid extra memory copies, when executing on GPUs. For example, in the iterative *stencil* benchmark, the main kernel is executed a fairly large number of times, and only one of the two arrays it operates needs to be copied back to the host every time and the

other one is then copied back from host to the GPU. By marking one of the array arguments as *in* and the other one as *out*, we avoid the extra copy in each direction.

4.2.2 Launch/Wait Intrinsic Code Generation

The *launch* intrinsic is used to asynchronously start a dataflow graph execution from host code. The *wait* intrinsic blocks until the dataflow graph execution is complete. The compiler replaces the *launch* intrinsic with a runtime API call to start the dataflow graph execution in a new thread, using the Posix pthreads library. The *wait* intrinsic is implemented using `pthread_join`.

4.2.3 hVISC Runtime

Previous subsections describe the static code generation of key features of the virtual ISA. Two specific features, however, require runtime support.

First, the virtual ISA design allows a leaf node to query node instance and dimension queries to any ancestor. When such a query can be addressed by hardware registers, the query intrinsic is replaced by the corresponding accelerator API call. However, when it is not supported, the runtime maintains a stack to keep track of the instance ID, and dimension limit of the dynamic instance of the ancestors and responds when a query arrives.

Second, the dataflow graph semantics of the virtual ISA assumes a globally addressable memory model. However, in the present form, many accelerators present in a SoC do not support this model. For example, many of today’s GPUs cannot address CPU memory directly (although this capability is emerging and may be more common in future). In such a scenario, the data has to be explicitly transferred to the accelerator memory before one initiates computation on the accelerator. To perform these data transfers, the translator inserts static API calls to the accelerator runtime in the generated native binary. These data copies are expensive and critical to application performance. It may happen that such a copy is unnecessary because the data is already present on the device. This would happen because the data was brought in the device memory by a prior node executing on the device. Thus, as an optimization, the *hVISC* runtime incorporates a feature we call the “memory tracker,” which keeps track of the latest copy of data arrays to avoid unnecessary copies to and from the accelerator.

5. Evaluation

In our experiments, we evaluate the suitability of the virtual ISA design in two ways. (1) The virtual ISA design should be portable. For this, we use the same virtual ISA binary of an application to compile to different compute units. (2) When compared to current heterogeneous programming technologies such as OpenCL, CUDA, and others, the virtual ISA design should be able to capture the parallelism expressed using these languages, and thus achieve reasonable performance when compiled to target architectures for these source-level languages.

5.1 Experimental Setup and Benchmarks

We modified the OpenCL front-end in the Clang compiler to generate the virtual ISA for OpenCL applications. We use annotations as hints to identify the subgraphs in the virtual ISA that are suitable for accelerators. We then used the compilation strategy described in Section 4 to translate the virtual ISA 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 16 GB RAM.

For our experimental evaluation, we used four applications from the Parboil [18] benchmark suite: Sparse Matrix Vector Multiple

(spmv), Single-precision Matrix Multiply (sgemm), Stencil PDE solver (stencil), and a Lattice-Boltzmann solver (lbm).

In the GPU experiments, our baseline for comparison is the best available OpenCL implementation in Parboil that does not use local memory (since our virtual ISA does not yet support local memory). For spmv and lbm, that is the Parboil version labeled `opencl_nvidia`, which has been hand-tuned for the Tesla NVidia GPUs [11]. For sgemm, the hand tuned version was utilizing local memory, thus preventing us from using it. Instead, using that version as a starting point, we implemented a version that is similar in every way except that the accesses to local memory were replaced by accesses to global GPU memory instead, and that we tuned the work group sizes to achieve the best performance. Finally, for stencil, we use the basic version since following the same practice did not improve the execution time. All the applications are compiled using nVidia’s proprietary OpenCL compiler.

In the vector experiments, our baseline is the same OpenCL implementations that we chose as GPU baselines, but compiled using the Intel OpenCL compiler, as we found that these versions achieved the best performance compared to the other available OpenCL versions on vector hardware as well.

The *hVISC* binaries were also generated using the same versions of OpenCL.

We use two input sizes for each benchmark, labeled ‘Small’ and ‘Large’ below. Each data point we report is an average of ten runs for the small test cases and an average of five runs for the large test cases; we repeated the experiments multiple times to verify their stability.

5.2 Experimental Results

Figures 2 and 3 show the normalized execution time of these applications against GPU baseline for each of the two sizes. Similarly, figures 4 and 5 compare the performance of *hVISC* programs with the vector baseline. The execution times are broken down to segments corresponding to time spent in the compute kernel of the application (kernel), copying data (copy) and remaining time spent on the host side. The total execution time for the baseline is depicted on the corresponding bar to give an indication of the actual numbers.

Comparing *hVISC* code with the GPU baseline, the performance is within about 25% of the baseline in most cases and within a factor of 1.8 in the worst case. We see that the *hVISC* application spends more time in the kernel execution relative to the GPU baseline. However, inspection of the generated PTX files generated by nVidia OpenCL compiler for OpenCL applications and *hVISC* compiler for *hVISC* applications has shown that they are almost identical, with the only difference being a minor number of instructions being reordered. Also, we notice increased, sometimes to a significant factor, data copy times, despite the fact the data copied in both applications are similar and that the *hVISC* runtime makes use of a memory tracking mechanism to avoid unnecessary data copies. We are working on getting a clear picture of the overheads that the *hVISC* representation or compilation may be imposing on the program execution.

In the vector case, we see that the performance of *hVISC* is within about 30% in all cases, and within a factor of 1.6x in the worst case. We again observe the same inefficiencies in kernel and copy time, albeit less pronounced due to the fact that the total running times are generally larger, which minimizes the effect of constant overheads to the total execution time.

Finally, we note that none of our benchmarks made use of vector code at the leaf dataflow nodes. This choice was made after comparing the performance of two *hVISC* versions: (a) the *hVISC* object code as generated from the modified Clang frontend, and (b) the *hVISC* code after altering the number of dynamic instances of the

leaf nodes as well as their code, in order to perform a bigger amount of computation so that vectorization can be achieved. This transformation may have improved the performance in some cases for one of the two targets, but it never achieved reasonable performance on both. This is due to the competing representation required to achieve good performance for GPUs and vector units. In the GPU case, code executing by a thread should perform carefully strided memory accesses in order to achieve coalescing of the memory requests performed by multiple threads, and vector instructions get serialized at the hardware thus no performance gain occurs from their use. In the vector case, a thread aims to access consecutive locations so as to perform vectorized memory operations and computations. Thus, a simple code where all threads perform independent operations and access consecutive locations has the potential to achieve good performance on both targets, by allowing memory coalescing on the GPU side and vectorization across work items in the vector case. To conclude, for simple benchmarks where vectorization across work items can be achieved automatically, our experiment shows that the presence of vector instructions does not improve performance on both targets. We expect the vector instructions to lead to performance gains for more complicated kernels where automatic vectorization will not be effective.

5.3 Expressing parallelism beyond GPUs

hVISC is aimed to be extensible beyond the devices that are most commonly found in today’s accelerators and represent parallelism models in a broad class of available hardware. Apart from data parallelism, many accelerators expose a streaming parallelism model and would benefit greatly by a representation that can capture this feature. *hVISC* presents the unique advantages of representing a program as a dataflow graph, which is a natural way of representing the communication between producers and consumers, as well as describing the repeated transfer of multiple data items via streaming edges. This section uses an image processing pipeline to demonstrate the benefits of expressing a streaming application in *hVISC*.

Figure 6 presents an application for Edge Detection in gray scale images in *hVISC*. At a high level, this application is a dataflow node that accepts a greyscale image I and a binary structuring element B and computes a binary image E that represents the edges of I . The application begins by computing an estimate of the Laplacian L of I , as depicted in figure 6, and proceeds by computing its zero crossings, i.e. points of sign change in L . A different dataflow node computes the gradient G of I , operation that can proceed in parallel with the remaining computations. The final dataflow node uses the output of the Gradient and the ZeroCrossings to perform a thresholding operation that will allow it to reject small variations in the brightness of the image and only detect more significant variations that actually constitute edges.

We implemented this pipeline using OpenCV computer vision library. We used C++ thread library to create threads for each top level node in this example, and implemented fixed size circular buffers for each streaming edge between these nodes to pass data between them. The pipeline, streaming and dataflow parallelism expressed in this example is easy to capture in *hVISC*. The streaming edges, dataflow nodes simply map to key features of *hVISC*. Our current implementation of *hVISC* is only missing the implementation of circular buffers for streaming edges, and thus we do not have a working *hVISC* version of this example.

However, mapping pipeline and streaming parallelism model to SPIR, HSAIL parallelism models of one kernel replicated across several cores, is non-intuitive and difficult to achieve. OpenCL supports concurrent execution of kernels running in two different streams, Expressing concurrency across kernels working on different image sections would require complex synchronization and

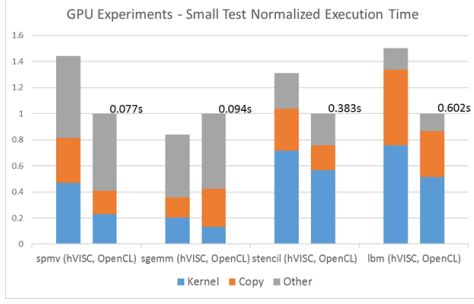


Figure 2. GPU Experiments - Small Test Normalized Execution Time

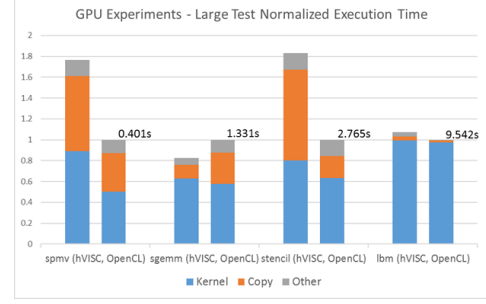


Figure 3. GPU Experiments - Large Test Normalized Execution Time

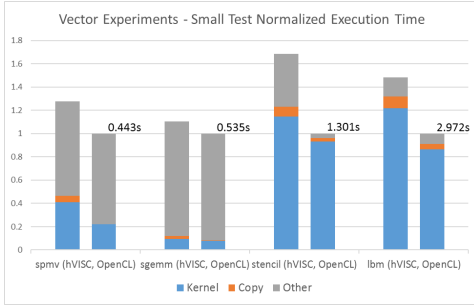


Figure 4. Vector Experiments - Small Test Normalized Execution Time

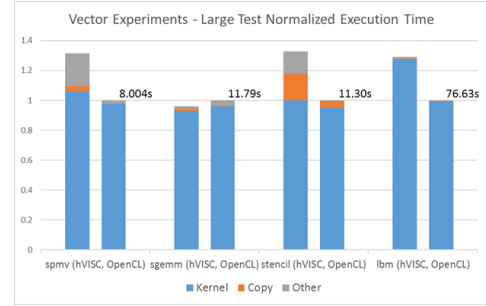


Figure 5. Vector Experiments - Large Test Normalized Execution Time

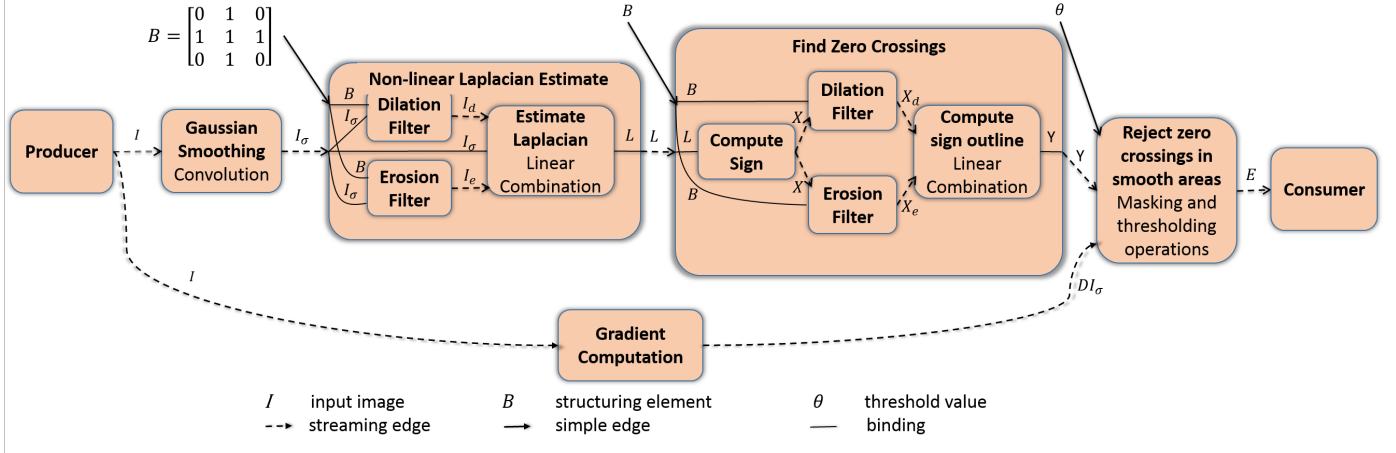


Figure 6. Edge Detection in gray scale images in *hVISC*

an implementation of programmer managed scheduling of nodes. This is a tedious and error-prone task, which is unlikely to scale to bigger and more complex pipelines.

Expressing this example in *hVISC*, would have the added advantage of flexibly mapping computationally heavy parts of the pipeline to accelerators. The Laplacian node is the pipeline bottleneck. Mapping Laplacian to GPU, achieved 2x speedup, as it balances the two branches of the pipeline. However, mapping both Laplacian and Gradient to GPU achieves a modest 1.1x speedup. This further shows the advantage of flexible mapping, which allows the programmer or auto-tuner to easily tune an application.

6. Related Work

Virtual ISAs: PTX virtual ISA was developed by nVidia to provide portability across GPUs of different sizes and across multiple GPU generations. It is however designed to target nVidia GPUs specifically and does not aim to support other hardware. There are currently a few projects with the goal to develop a portable object code distribution format for heterogeneous systems. HSAIL [1] and SPIR [8] are two such standards which map well to GPUs and multicore CPUs. However, they support only a restrictive throughput-oriented SIMT parallelism model, which is not general enough to capture other models of parallelism like pipeline or streaming parallelism (as explained in Section 5.3), whereas these are captured naturally in our dataflow graph model.

Source Languages: Source-level programming languages for heterogeneous systems such as OpenCL [16] and CUDA [12], and the accelerator extensions in OpenACC [2] and in recent versions of OpenMP [17], all support a common programming model where a single-threaded kernel function is replicated across a large number of cores, usually with explicit copying of data between host and device. Intel ISPC [14] is a set of language extensions to C, and an optimizing LLVM-based compiler, that effectively uses the SPMD programming model to deliver performance using both multiprocessor and SIMD vector units. Like PTX and SPIR, all these languages map well to GPUs and vector parallelism. None of them address object code portability. Moreover, they all have the same limitations of being unable to express more general models of data parallelism, like streaming parallelism.

RenderScript [3] aims to provide performance and portability across heterogeneous SoC architectures for Android devices. Like SPIR, it uses LLVM bitcode as its on-device portable object code format. This format, however, does not have well-defined parallelism abstractions, instead using some ad hoc combination of LLVM (scalar and vector) code and run-time operations.

Domain Specific Languages (DSLs) such as Delite [19] and Halide [15] can potentially target different architectures efficiently using tuning based on domain specific knowledge, but the techniques are largely limited to the intended domain.

Compiler and Autotuning Approaches: Besides virtual ISAs and source level languages, a number of autotuning frameworks explore interesting methods to distribute computation between compute units in a heterogeneous system. Petabricks [4] explores the search space of different algorithm choices and how they map to CPU and GPU processors. Similarly, in Tangram [20] a program is written in interchangeable, composable building blocks, which enables architecture-specific algorithm and implementation selection. Exploring algorithm choices is orthogonal to, and can be combined with, our approach. Moreover, these techniques though effective, put a huge burden on compiler and runtime system to explore a potentially large search space to find the correct tuning parameters, and it is not clear how such search strategies will scale up to more realistic applications.

7. Conclusion

We present *hVISC*, a new Virtual ISA that aims to address the functional and performance portability challenges arising in today's SoC's. *hVISC* is designed as a hierarchical dataflow graph with side effects and parametric vector instructions. We argue that these two models of parallelism exposed by *hVISC* successfully capture the diverse parallelism models exposed by a wide range of parallel hardware. We also presented a compilation strategy that uses a single object code to target a wide range of parallel hardware, and implemented backend translators for nVidia's GPUs targeting PTX, vector hardware using Intel's AVX, and host code for X86 processors.

We evaluate our design by (a) using a single *hVISC* representation of four applications from the Parboil Benchmark Suite to generate code for both nVidia's GPUs and vector hardware, and comparing with baselines that are each separately tuned for their respective target device. The achieved performance is within a factor of 2x at the worst case, demonstrating the achieved performance portability from a single *hVISC* representation, and (b) demonstrating that *hVISC* can naturally capture streaming parallelism due to its dataflow representation.

References

- [1] HSAIL. <http://www.hsaoundation.com/standards/>.
- [2] OpenACC-Standard. <http://www.openacc-standard.org/>.
- [3] RenderScript. <http://developer.android.com/guide/topics/renderscript/compute.html>.
- [4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. URL <http://doi.acm.org/10.1145/1542476.1542481>.
- [5] R. L. Bocchino, Jr. and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *Proceedings of the 2nd international conference on Virtual execution environments, VEE '06*, pages 46–56, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. URL <http://doi.acm.org/10.1145/1134760.1134769>.
- [6] N. Compute. Ptx: Parallel thread execution isa version 2.3. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf, 2011.
- [7] Intel. <https://software.intel.com/en-us/intel-opencl>.
- [8] Khronos Group. SPIR 1.0 Specification for OpenCL. http://www.khronos.org/registry/cl/specs/spir_spec-1.0-provisional.pdf.
- [9] C. Lattner and V. Adve. LLVM Language Reference Manual. <http://llvm.org/docs/LangRef.html>.
- [10] C. Lattner and V. Adve. LLVM: A compilation framework for life-long program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.
- [11] Li-wen Chang. Personal communication, Aug. 2015.
- [12] nVidia. CUDA Toolkit Documentation v7.5. <http://docs.nvidia.com/cuda/>.
- [13] NVVM IR Specification 1.2. <http://docs.nvidia.com/cuda/nvvm-ir-spec>.
- [14] M. Pharr and W. R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (In-Par)*, 2012, pages 1–13. IEEE, 2012.
- [15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. URL <http://doi.acm.org/10.1145/2491956.2462176>.
- [16] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science and engineering*, 12(1-3):66–73, 2010.
- [17] E. Stotzer. Tutorial: Openmp accelerator model. Technical report, 2014. URL http://portais.fieb.org.br/senai/iwomp2014/presentations/tutorial_accelerator_model.pdf.
- [18] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.
- [19] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, Apr. 2014. ISSN 1539-9087. URL <http://doi.acm.org/10.1145/2584665>.
- [20] L. wen Chang, A. Dakkak, C. I. Rodrigues, and W. mei Hwu. Tangram: a high-level language for performance portable code synthesis, 2015.