Programming Heterogeneous Systems from an Image Processing DSL

JING PU, STEVEN BELL, XUAN YANG, JEFF SETTER, and STEPHEN RICHARDSON,
Stanford University
JONATHAN RAGAN-KELLEY, University of California, Berkeley
MARK HOROWITZ, Stanford University

Specialized image processing accelerators are necessary to deliver the performance and energy efficiency required by important applications in computer vision, computational photography, and augmented reality. But creating, “programming,” and integrating this hardware into a hardware/software system is difficult. We address this problem by extending the image processing language *Halide* so users can specify which portions of their applications should become hardware accelerators, and then we provide a compiler that uses this code to automatically create the accelerator along with the “glue” code needed for the user’s application to access this hardware. Starting with Halide not only provides a very high-level functional description of the hardware but also allows our compiler to generate a complete software application, which accesses the hardware for acceleration when appropriate. Our system also provides high-level semantics to explore different mappings of applications to a heterogeneous system, including the flexibility of being able to change the throughput rate of the generated hardware.

We demonstrate our approach by mapping applications to a commercial Xilinx Zynq system. Using its FPGA with two low-power ARM cores, our design achieves up to 6× higher performance and 38× lower energy compared to the quad-core ARM CPU on an NVIDIA Tegra K1, and 3.5× higher performance with 12× lower energy compared to the K1’s 192-core GPU.

CCS Concepts: • **Computing methodologies** → **Image processing**; • **Hardware** → **Hardware accelerators**; **Emerging languages and compilers**; *Hardware-software codesign*; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Image processing, domain specific languages, FPGAs, high-level synthesis

ACM Reference format:

Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (August 2017), 25 pages.
<https://doi.org/10.1145/3107953>

New article, not an extension of a conference paper.

This work was supported by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

Authors’ addresses: J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, and M. Horowitz, Department of Electrical Engineering, Stanford University, Stanford, California 94305; J. Ragan-Kelley, Department of Electrical Engineering & Computer Sciences, University of California, Berkeley, Berkeley, California 94720; email: jingpu@cs.stanford.edu.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/08-ART26 \$15.00

<https://doi.org/10.1145/3107953>

1 INTRODUCTION

The performance and energy efficiency of image processing tasks are increasingly important as cameras become ubiquitous, and as our ability to extract information from images improves. These tasks are extremely computationally intensive, requiring, for example, 120 gigaops/sec to process 1080p/60fps raw video (Hegarty et al. 2014). To efficiently process so many pixels, designers historically built custom hardware engines specialized to the task. For example, a typical image signal processor (ISP) in a mobile SoC operates at 1.2 gigapixels/sec (Qualcomm Inc. 2016), and hardware video codecs perform an equally immense amount of processing, all with power budgets low enough for a smartphone to run for hours on a small battery. This efficiency is possible because the applications have extreme data locality, and matching the hardware to their computation patterns can yield enormous energy savings.

The problem with these platforms is the brittle nature of the functions provided by the accelerators. These functions are relatively fixed, in order to keep their performance and efficiency high, so application programmers are limited to predefined library calls. However, application demands are rapidly evolving, and a more flexible approach is needed. Configurable hardware, using either coarse-grained reconfigurable arrays (CGRAs) (Mei et al. 2003; Govindaraju et al. 2011; Gao and Kozyrakis 2016) or FPGAs (Hauck and DeHon 2010), is one approach to providing a flexible machine that can be configured to match the dataflow of different algorithms. Although these architectures promise much better energy efficiency compared to CPUs or GPUs, programming and integrating them into complete real-world systems remains a formidable task for application developers.

To help address the programming part of this challenge, C-based *high-level synthesis* (HLS) has been widely studied in past decades (Martin and Smith 2009; Cong et al. 2011; Zhang et al. 2008). C-HLS tools raise the design level by decoupling clock timing and automatically scheduling pipelines and other resources. However, designers still need to create a good microarchitecture in their C-code in order to develop high-performance implementations, which requires hardware expertise. Moreover, microarchitecture and functionality are conflated in HLS applications. Thus, optimizing the microarchitectures usually requires global restructuring of the application code, which is difficult, time-consuming, and expensive.

So, to further reduce the hardware knowledge a designer needs and to automate design and optimization, researchers created *domain-specific languages* (DSLs), which can embed microarchitecture knowledge for a specific application domain in the compiler. These DSL systems, including Darkroom (Hegarty et al. 2014), HIPAcc (Reiche et al. 2014), and PolyMage (Chugh et al. 2016) for imaging, can generate efficient FPGA and ASIC designs from high-level image processing code. *Halide*, an image processing DSL, solves the problem of finding an optimal joint implementation for CPU and GPU by decoupling the *algorithms* from the *schedules*: the computation itself is described in the algorithm, while the computation order and storage are defined by the schedule (Ragan-Kelley et al. 2012, 2013). The Halide algorithms represent architecture-independent specifications for applications, and its scheduling language provides a high-level interface for separately specifying architecture-related optimizations, which enables fast exploration for optimal implementation by human programmers or automatic optimizers (Mullapudi et al. 2016).

This article builds on these DSL-to-FPGA systems and extends them in three important ways. First, rather than creating a specialized new language for our DSL, we use an existing widely popular open-source DSL, *Halide*, to describe our applications. In addition to giving us a large collection of existing image-related algorithms, it forces us to extend the microarchitecture template and compiler techniques used in previous systems to support these real-world applications. For example, we include support for affine indices and data-dependent reduction, often found in kernels like *downsampling* or *histogram*.

Fig. 1. Energy savings from accelerating a large application composed of one large kernel computing depth from stereo plus 19 small convolution kernels on a synthetic CPU/FPGA system with different FPGA resource constraints. The number of CPU cores is fixed, while the FPGA resources scale linearly along the x-axis. The scale factor is the size of a Xilinx XC7Z020. When accelerators are constrained to run at unit rate (or not at all), it takes significant FPGA resources to get energy savings (black). By contrast, our variable-rate system provides benefits even from very small FPGA fabrics (white).

Second, in addition to the one-pixel/cycle pipelines of prior systems, we can specify and generate kernels with variable throughput rates in the DSL, exploiting the space-time tradeoffs often needed to map real applications on FPGAs with finite hardware. Figure 1 dramatically illustrates the advantage of this flexibility for accelerating large applications. At the previous one-pixel/cycle fixed throughput (black), no significant energy savings are seen until the FPGA is large enough to accelerate the application’s largest kernel, *stereo*. Accelerating *stereo* at a much lower throughput rate requires significantly less area, so the savings are seen even with very small FPGAs (white “variable” line). As resources increase, our system gracefully tunes the throughput and includes more stages for acceleration.

Finally, our system generates not only the FPGA kernels but also all the software needed to connect that hardware to the user’s application. Thus, in addition to the FPGA configuration file, our system creates the CPU portion of the algorithm, the Linux kernel drivers for the accelerator, and the software glue that maps user Halide calls onto kernel driver calls that access the hardware. The automatic CPU/FPGA integration greatly helps to explore the workload partitioning between CPU and FPGA for system-level optimization, as we saw in Figure 1. Among our contributions:

- We demonstrate that the popular image DSL *Halide* is sufficiently restrictive such that much of its computation can be “compiled” into efficient FPGA implementations. We also show that *Halide*’s scheduling language is powerful enough that we needed to add only two new commands to define and control the hardware generation.
- We extend the line buffer pipeline template of prior image-DSL-to-hardware systems to suit the variety of computation possible in *Halide*. This includes creating pipelines of different throughputs and dealing with higher-dimension input and output stencils. In addition, our compiler implements a new loop transformation optimization, called loop perfection, to support these features.
- We create the first end-to-end system that takes *Halide* user code and creates an FPGA bit-stream along with a multithreaded software program that controls the new hardware. This end-to-end system, coupled with *Halide*’s schedule language, allows a user to seamlessly explore the effects of moving function execution between the CPU and the FPGA. We have used this system to implement a range of applications on a Xilinx Zynq platform.

Fig. 2. A line buffer (the red box) captures the intermediate working set for the convolution functions using minimum storage.

The next section reviews the tools and techniques of image processing and domain-specific languages that we leverage in this work. Sections 3 and 4 describe our extensions to the Halide language, and our compiler system that implements these extensions to produce blended CPU/FPGA designs. Finally, we present our test platform in Section 5, followed by an experimental comparison with other methods, and a quantitative evaluation of potential optimizations.

2 IMAGE PROCESSING BACKGROUND AND PRIOR WORK

Most image processing algorithms consist of kernels that read a small window of the image data to generate each output. For example, sharpening and blurring operations can be expressed as convolutions, which use a fixed window of pixels to compute each result pixel. Likewise, corner detection, edge enhancement, image sensor demosaicking, and color transformation algorithms all calculate their output pixels using small nearby regions of data. The composition of these kernels can be expressed as a directed acyclic graph (DAG), where the result of one computation is fed forward into the next. This computational model holds even for modern compute vision methods, such as convolutional neural networks (CNNs) (LeCun et al. 2015).

This data locality, combined with the fact that image processing algorithms work on millions of pixels, makes image processing a fertile area for code optimization and hardware acceleration. All the data necessary to compute a result pixel fits in a small (and therefore near and low-power) memory block. Moreover, because pixels are typically computed in sequence, shared stencil data can be reused from one pixel to the next.

For key applications in this domain, such as camera pipelines, people have built custom hardware to achieve higher-performance and orders of magnitudes better energy efficiency than CPU or GPU solutions. Custom hardware maximizes data reuse and minimizes memory traffic by carefully designing pipeline stages with balanced throughput and placing specialized buffers between kernels. This organization keeps all the intermediate data locally and ensures that the global image data is fetched just once at the beginning of the pipeline and written back once at the end of the pipeline (Ramanath et al. 2005; ON Semiconductor 2015).

While each kernel must process an entire image, the buffers between kernels can be much smaller than the image size. For example, in a 3×3 convolution function, if the rate of the pixels produced by the upstream kernel equals the rate of convolution kernel, the intermediate working set can be reduced to its minimum size of approximately two rows of pixels, as in Figure 2. Older pixels, shown in gray, are never accessed again and can be dropped from the buffer. A specialized buffer that captures this working set using the minimum required storage is called a *line buffer* (Ruetz and Brodersen 1986; Kamp et al. 1990). The hardware can be viewed as a pipeline of stages computing different intermediate images concurrently and separated by line buffers.

FPGA-based designs can accelerate image processing applications and greatly improve energy efficiency, allowing programmers to design and tune the configurable memory storage to suit the application. In particular, an FPGA can implement a line-buffered image processing pipeline very efficiently, so we use this as an architectural template for creating FPGA accelerators. Flexible FPGA platforms address the long-standing tension between efficiency and flexibility in image processing hardware. One early approach started with such a platform and then “programmed” the right hardware circuit onto that substrate (Athanas and Abbott 1995; Draper et al. 2003). We follow this approach in our work, and the next section reviews some of the rich work that has already been done in this area.

2.1 Hardware Generation from High-Level Languages and DSLs

Despite its potential for high performance and energy efficiency, the difficulty of creating an efficient FPGA-based hardware accelerator and connecting it to an application has limited its use. This section describes prior work in easing the FPGA programming task, and the following section addresses prior work in system integration.

To facilitate FPGA use, researchers have raised the abstraction level for describing hardware implementation. Early efforts used some subset of C as the description language, often referred to as high-level synthesis or HLS (Martin and Smith 2009; Cong et al. 2011; Cardoso and Diniz 2011). To abstract away even more hardware details, higher-level DSLs were used. The early systems focused on describing the computation of a single kernel, since FPGA resources were limited, but as FPGA capabilities grew, systems needed to create deep imaging pipelines, with multiple kernels and their associated line buffer memories.

HLS translates an untimed specification in a high-level language, like C/SystemC/Java, into a fully timed register-transfer level (RTL) implementation, greatly improving the productivity of hardware development by decoupling the input description from low-level issues like register allocation, clock-level timing, and pipelining. The HLS concept was proposed in the 1980s, and recently many commercial HLS tools by FPGA and EDA vendors have become available, including Vivado HLS (Xilinx 2016b), Catapult HLS (Mentor Graphics 2016), Altera OpenCL (Altera 2016), and MaxCompiler (Maxeler Acceleration Technology 2011). Because of their high-level design abstraction and production-level quality, these HLS tools are seeing increasing use, especially for FPGAs (Chen et al. 2016; Zhang et al. 2015; Moreau et al. 2015; Cattaneo et al. 2015). For the same reason, our DSL framework compiles high-level application specifications into HLS code and uses a commercial HLS tool to generate RTL designs.

Although HLS makes programming FPGAs much easier by moving the development toward higher-level language descriptions, writing high-performance hardware in HLS shares the same issues as using C to develop high-performance software. Users of these tools still need a fairly good understanding of the hardware they want to create, since the system cannot (yet) start from any generic code and automatically determine how to partition the algorithm for optimal locality and parallelism. Also, the high-performance code in these languages is difficult to understand and arduous to port to other platforms, since functionalities and optimizations are conflated in the application code.

To address this limitation, we can restrict the hardware target to a single domain like image processing, and, by examining the characteristics of applications in that domain, we can create reasonable microarchitectural templates for automatic hardware generation. As an early effort to incorporate image-processing-specific optimizations in hardware generation, SA-C proposed a window generator extension to the C language to create implicit parallel loops and used a hardware template tuned for the loop generation (Draper et al. 2000; Najjar et al. 2003). ROCCC performed code analysis in the window-based operations and synthesized “smart buffers” to reuse data among

adjacent input windows (Guo et al. 2005, 2008). Probably due to the resource limitation of the time, both projects focused on generating FPGA datapaths for a single kernel, streaming all input and output data from and to DRAM. With increased FPGA resources, and to further improve energy efficiency, a pipeline of kernels could be implemented in FPGA in the form of a line-buffered pipeline, with all intermediate data kept on-chip. Generating such pipelines requires a compiler to extract coarse-grain dataflow from input programs, which is difficult for programs specified in imperative languages like C (Koeplinger et al. 2016).

To eliminate this obstacle, DSLs have become a popular approach. DSL systems further restrict application specifications (often in some functional representation), allowing compilers to easily extract the parallelism and the locality and to generate efficient implementations using domain-specific templates. Spiral synthesized signal processing applications from mathematical descriptions, initially for generating high-performance x86 code, and later for creating efficient hardware (Milder et al. 2012). George et al. (2014), and (Prabhakar et al. 2016), and (Koeplinger et al. 2016) used OptiML (Sujeeth et al. 2011), a machine-learning DSL, to compile applications into FPGA instances. Gorilla++ generates custom hardware for applications with irregular dataflow (Lavasani 2015).

For the image processing application domain, researchers have created Darkroom (Hegarty et al. 2014; Brunhaver 2015), HIPAcc (Reiche et al. 2014), and PolyMage (Chugh et al. 2016) DSLs and compilers. These compilers use a line-buffered pipeline microarchitectural template to guide their hardware generation. They demonstrated it was possible to take a function coded in a DSL and implement it as an FPGA or ASIC. To improve the performance on high-end FPGAs with sufficient memory bandwidth, PolyMage exploits coarse-grain parallelism by creating multiple pipelines that process image tiles in parallel, while recent HIPAcc follow-up work (Özkan et al. 2016) vectorizes the whole pipeline at a fine-grain level and creates stages processing vectors of pixel data. Like our system, both HIPAcc and PolyMage compilers emit HLS-C and feeds that output to Vivado HLS to generate the hardware. However, to efficiently support both large and small programs, we allow the user to set the throughput rate of each pipeline stage individually, either greater than one pixel/cycle (vectorization) or less than one pixel/cycle (time multiplexing). Moreover, to support a wider application class, we generalize the line-buffered pipeline microarchitecture to include higher-dimension input stencils ($> 2D$), affine indexing into these arrays, and data-dependent reductions.

Rigel Hegarty et al. (2016), a follow-on to Darkroom, supports a multirate line buffer pipeline similar to the microarchitecture used in our FPGA targets. Rigel provides a set of multirate modules for explicitly instantiating pipeline stages running at throughputs other than one pixel/cycle. As a result, these rate decisions are embedded into the Rigel program. By contrast, inspired by Halide's choice of separating algorithm from schedule, we provide high-level scheduling primitives that set the data throughput and automatically lower high-level specifications to multirate pipeline instances. As a result, in our system, exploring the possible hardware mapping onto FPGAs involves optimizing the application's schedule, just as is done for CPUs and GPUs today. It does not require any change of application algorithm code.

It is worth noting that the separation of functions and schedules has been widely used in designing architecture simulators. Asim split the functional partition from the timing partition of an architecture specification, focusing on modularity and reusability of the framework (Emer et al. 2002). Interestingly, HAsim took such a separated model and generated an FPGA implementation to accelerate the simulation (Pellauer et al. 2011). For DSLs, the formerly mentioned Gorilla++ separated functions from microarchitecture templates that can implement the functions in different forms for performance optimization. Spiral had separate mapping functions to direct the efficient implementation of signal processing operations for a particular target architecture.

2.2 End-to-End Systems

Generating efficient FPGA instances solves only one-half of the challenge in mapping applications to heterogeneous systems made of general-purpose processors and reconfigurable FPGA fabrics. It is also important to have an end-to-end system to orchestrate the new hardware and to assist the exploration of workload partitioning and scheduling to balance the usage of various hardware engines in the system.

Leveraging the success of HLS, researchers have explored the feasibility of using heterogeneous computing languages like OpenCL (Stone et al. 2010) and NVIDIA's CUDA (Nickolls et al. 2008) to program CPU/FPGA systems. Some research projects in this field include OpenCL-to-FPGA (Czajkowski et al. 2012), SOpenCL (Owaid et al. 2011), and FCUDA (Papakonstantinou et al. 2009). These languages provide a C-based programming environment that is similar for host and device code, along with a runtime API to manage memory transfers and synchronization. However, tuning an application on a CPU/FPGA platform is still hard, as programmers need to manually explore and implement a good workload partitioning and scheduling scheme.

LegUp (Canis et al. 2013) and Warp (Vahid et al. 2008) identified and synthesized FPGA hardware for critical regions in a standard C program based on software profiling, and compiled the program to a hybrid architecture with FPGA and a soft processor. However, the decision of workload partitioning is not exposed to the programmer. Lime (Auerbach et al. 2012) had a unified language for CPU, GPU, and FPGA, with semantics to delineate boundaries between computation blocks. The blocks compile to one or more target architectures, and the runtime system selects a set of implementations, automatically handling data transfers and synchronization at those block boundaries.

We adopt one of the key strengths of these systems: if the compiler knows the boundary between CPU and accelerator code, it can automatically fill the gap between them. Our new scheduling primitives extend Lime's concept of "relocation brackets," enabling the developer to easily specify whether code should run on the CPU or accelerator, separate from the algorithm itself. However, the scheduling primitives also provide more control over the generated hardware, as described in the following sections.

3 LANGUAGE

As discussed in the previous section, the design space for implementing an image processing algorithm is huge. Finding the most efficient implementation for an application is difficult and time-consuming. Halide helps by decoupling the computation to be performed (the *algorithm*) from the order in which it is done (the *schedule*).

Halide represents algorithms in a pure functional form. In an algorithm, images are *functions* mapping pixel coordinates to their values. Functions are side-effect-free expressions defined over an infinite multidimensional integer domain. For example, a separable 3×3 box blurring filter can be expressed as a chain of two functions in x, y as follows:

```
blurry(x, y) = (input(x, y-1) + input(x, y) + input(x, y+1)) / 3;
blurred(x, y) = (blurry(x-1, y) + blurry(x, y) + blurry(x+1, y)) / 3;
```

In this form, functions are data parallel by construction, and the dataflow of functions can be extracted statically.

In Halide, the order and the range of functions to be evaluated are treated as optimization choices and are specified using the language of schedules. Halide's language of schedules is based on a set of loop transformation concepts (loop splitting, fusion, reordering, tiling, etc.). The language provides *scheduling primitives* for applying these transformations on each function and defining the granularity with which to interleave the computation of each function. The explicit separation

Fig. 3. (a) Algorithm and schedule code for the unsharp function, and (b) its corresponding DAG. The accelerate primitive defines the accelerator scope, from *in* to *unsharp*.

between algorithms and schedules makes it easy to experiment with various tradeoffs among locality, parallelism, and redundant recomputation without changing the functionality of the application.

Although Halide algorithms present functional specifications that are architecture independent, its language of schedules is insufficient for describing the efficient hardware architecture for an FPGA target and the orchestration of the generated FPGA accelerator on a CPU/FPGA platform. Our task is to extend Halide to cover this new target, which mostly involves mapping Halide algorithms onto a specialized hardware engine. Specifically, the schedule should include:

- The scope and interface of the hardware accelerator pipeline
- The granularity of the accelerator launch task, that is, the portion of the output image block the hardware produces per launch
- The amount of parallelism implemented in the hardware datapath, which affects the throughput of each pipeline stage
- The allocation of buffers, specifically line buffers, which optimally trades storage resources for less recomputation
- The number of delay register slices needed to match varying computation latencies

Many hardware scheduling choices have analogs in CPU scheduling, and Halide already has primitives to describe them. For example, both CPU and hardware schedules must describe computation order and memory allocation. In such cases, we reuse as many of the existing primitives as possible. Ultimately, we were able to achieve efficient hardware mapping and hybrid CPU/accelerator execution using only two new primitives and a bit of syntactic sugar.

The language of scheduling is best explained in the context of an example. Figure 3(a) shows a simple unsharp mask filter implemented in Halide. Unsharp masking is an image sharpening technique often used in digital image processing. We will use this as a running example throughout the article, as it demonstrates many important features of our system. The code first computes a

blurred grayscale version of the input image using a chain of three functions (gray, blurry, and blurx), and then amplifies the input based on the difference between the original image and the blurred image.

The hardware schedule begins on line 14. `unsharp.tile` is a standard Halide operation, which breaks an ordinary row-major traversal (defined by vars `x` and `y`) into a blocked computation over tiles (here, 256×256 pixels). The variables `xi` and `yi` represent the inner loops of the blocked computation, which works pixel by pixel, while `x` and `y` then become the outer loops for iterating over blocks.

With the image now broken into constant-sized pieces, we can apply hardware acceleration. Our first new primitive is `f.accelerate(inputs, innerVar, blockVar)`, which defines both the scope and the interface of the accelerator and the granularity of the accelerator task. The first argument, `inputs`, specifies a list of Funcs for which data will be streamed in. The accelerator will use these inputs to compute all intermediate Funcs to produce the result `f`. In this example, this is the sequence of computation through gray, blurry, blurx, sharpen, and ratio that produces unsharp from `in` (Figure 3(b)).

The block loop variable `blockVar` defines the granularity of the computation: the hardware will compute an entire tile of the size that `blockVar` counts—in this case, 256×256 pixels. The inner loop variable `innerVar` controls the throughput: `innerVar` will increment each cycle, in this case producing one pixel each time. To create higher-throughput hardware, we could use Halide's `split` primitive to split the `innerVar` loop into two, and accelerate with the outer one as the hardware stride size.

Our second new primitive is `src.fifo_depth(dest, n)`. It specifies a FIFO buffer with a depth of `n`, instantiated for the direct edge from function `src` and function `dest`. This primitive helps to balance the latencies of different paths between two vertices in a DAG. Unbalanced paths in the DAG will cause pipeline stalls, which results in substantial performance degradation or even deadlock. In the unsharp example, both `in` and `gray` are consumed by multiple functions. Without a FIFO from `in` to unsharp, for example, the pipeline stalls after one pixel of `in` is fetched because the channel between `in` and unsharp is full, while there is no value of `ratio` available for stage unsharp to proceed. In order to eliminate the pipeline stalls, we add a FIFO buffer of depth 512 along the path from `in` to unsharp, which matches the latency of the other paths. The optimal FIFO depths in the DAG can be solved automatically as an integer linear programming problem (Hegarty et al. 2014), so we can eventually automate this decision, but for now we specify and tune it by hand.¹

`f.linebuffer()`, our syntactic sugar for a combination of existing Halide primitives, is designed to instantiate a line buffer for function `f`.² Without these primitives, functions would be fused directly into other downstream functions, potentially causing recomputation when their values were reused. In the unsharp example, by default, function `gray` will be fused into functions `sharpen`, `ratio`, and `unsharp`, causing each value of `gray` to be evaluated three times. Instead, we use the `linebuffer` primitive to buffer the value and avoid the recomputation, as shown on line 18 in Figure 3(a). Therefore, the `linebuffer` primitive helps explore the tradeoff between storage and computation resources available in the hardware.

¹Another issue with automatically solving for optimal FIFO depths is that some latencies of hardware blocks are unknown before HLS compilation. In the future, if an HLS tool provides an API for querying these latencies, the `fifo_depth` primitive can be opted away in favor of automatic solving in the DSL compiler.

²Halide's native analog of the line buffer, the *sliding window* pattern, is achieved by specifying different compute and storage levels with `compute_at` and `store_at` primitives, and letting the compiler apply a *storage folding* optimization. We overload the semantics of this same pattern when it is used in an accelerated portion. As `accelerate` already defines the compute and storage levels, we add the `linebuffer` sugar, which needs no additional arguments.

Fig. 4. Compilation flow. Blue circles are new. Green rectangles are unchanged/existing Halide compilation passes. Yellow hexagons are inputs and outputs of the compiler.

The existing Halide primitive `f.unroll(var, factor)` is overloaded in the hardware context for specifying variable-rate pipeline stages and exploring the space-time tradeoff. Previously, for the CPU target, `unroll` was used to eliminate short loops and to enable optimizations on cross-iteration sharing of data. However, in terms of hardware, since the HLS tool schedules resources for one loop iteration, having a larger loop body through unrolling also increases the parallelism of the datapath. It effectively duplicates the compute units in the pipeline, potentially scaling up the throughput. In the example, `unroll` on `unsharp` causes three multipliers to be instantiated for computing three color channels simultaneously, which scales the throughput of the pipeline from one-third pixel/cycle to one pixel/cycle.

All other existing Halide primitives (e.g., `tile`, `vectorize`, `parallel`) remain unchanged for the portion of the program mapped to software, where Halide already provides state-of-the-art performance on ARM and x86 CPUs. In our example, the `parallel` primitives on line 16 schedule multiple tile processing tasks concurrently onto multiple CPU cores and multiple FPGA pipelines. Further details about parallel execution are discussed in Section 5.

4 COMPILER IMPLEMENTATION

Figure 4 describes our compiler design. The inputs to our system are an application’s *algorithms* and *schedules* written in Halide. An analysis pass extracts parameters for the architecture template. A transformation pass rewrites the hardware parts of the Halide original imperative IR into a dataflow style. After the new loop perfection optimization and some common scalar optimization passes, like constant propagation, common subexpression elimination, and so forth, the final IR is passed to an HLS code generator and an LLVM code generator, which produce the hardware designs and the software programs, respectively.

4.1 Architecture Parameter Extraction

Our system generates specialized hardware accelerators by instantiating architectural templates from a scheduled program. The architectural template approximates the *line-buffered pipeline* of Darkroom (Hegarty et al. 2014), with extensions to support a wider range of algorithm and performance targets.

In this template, an accelerator is a DAG whose edges are streams of windows of pixels, or *stencil streams*, and whose nodes are *stencil kernels*. Each kernel is a Halide function scheduled for

Fig. 5. Architecture template parameters for *unsharp*. The blue circles are stencil kernels corresponding to line-buffered functions (*blurx* and *sharpen* were fused into *ratio*). Middle and right columns show parameters of selected kernels and stencil streams. The domain of the output stencil stream *gray* is the union of the domains of the stencil *gray* in kernels *blurry* and *ratio*.

a line buffer, into which one or more non-line-buffered functions can be fused. A stencil stream is parameterized by the window size, the sliding stride, and the range of the image domain. Note that because an output stencil may serve as input to more than one downstream kernel, the producer kernel must compute a union of the stencils required by all consumers. Figure 5 shows some of the architectural parameters for the *unsharp* application of Figure 3(a).

Since the scope of the pipeline and the line-buffered functions are defined in a high-level language, composing the DAG is straightforward. To extract the parameters of each stencil stream in the template, we apply bounds inference analysis recursively back from the output. This is similar to the original Halide compiler (Ragan-Kelley et al. 2013), except that we now have hardware line buffers between stencil kernels that capture data reuse, so the output stencil size of upstream kernels doesn't cumulatively increase. For example, for a pipeline of three cascaded 3×3 convolution functions, in order to compute one final value, the original bound analysis will infer a 5×5 output of the first function and a 3×3 output of the second function for the worst case (i.e., at image boundaries), while our new analysis is aware of the line buffers that are specialized to handle the boundary conditions and gives 1×1 output bounds for all functions.

4.2 IR Transformation

Given a schedule, the Halide compiler generates an architecture-independent imperative representation of the algorithm, with loop nests and storage allocations injected for each function. Figure 6(a) shows part of the Halide loop IR for function *gray*. Because the schedule specifies a sliding-window order for computing *gray*, the storage for *gray* is allocated outside the loop nest that iterates over the 256×256 block of *unsharp*, while the computation of *gray*, along with other functions, is interleaved inside the loop nest.

```

1 alloc gray [258, 258]
2 for (unsharp.y, 0, 256)
3   for (unsharp.x, 0, 256)
4     // compute function "in" here...
5     for (y, gray.y.loop_min, gray.y.loop_extent)
6       for (x, gray.x.loop_min, gray.x.loop_extent)
7         gray(x,y) = 0.3*in(0,x,y) + 0.6*in(1,x,y) + 0.1*in(2,x,y)
8     // compute function "blur" here...

```

(a) Halide IR of function *gray*

```

1 // kernel "in" here...
2 def_stream gray_stream [1, 3]
3 def_stream gray_step_stream [1, 1]
4 for (scan_y, 0, 258)
5   for (scan_x, 0, 258)
6     alloc in [3, 1, 1]
7     alloc gray [1, 1]
8     pop (in, in_stream)
9     for (y, 0, 1)
10      for (x, 0, 1)
11        gray(x,y) = 0.3*in(0,x,y) + 0.6*in(1,x,y) + 0.1*in(2,x,y)
12      push (gray, gray_step_stream)
13 linebuffer(gray_step_stream, gray_stream, [258, 258])
14 dispatch(gray_stream, [-1:256, -1:256],
15          "blur", 1, [-1:256, -1:256],
16          "ratio", 8, [0:255, -1:256])
17 // kernel "blur" here...

```

(b) Dataflow IR of kernel *gray*

Fig. 6. IR transformation allocates additional storage for local stencils and separates pipeline stages with data streams.

To map this scheduled loop IR into the line buffer pipeline shown in Figure 5, we further lower it into our own dataflow IR, using the architecture template parameters previously extracted. First, additional storage for input and output stencils are allocated locally to a kernel computation, and references to the original storage are replaced with references to stencils. Then, the original storage allocation is replaced with declarations of stencil streams, and stream operations (pop and push) are inserted before and after the kernel computation. Finally, loops iterating over the image domain are inserted for each compute stage, along with calls to `linebuffer` and `dispatch` IR primitives if a line buffer or a stream dispatcher is needed.

Figure 6(b) shows the IR of kernel *gray* after the transformation. For each scanning step (here, each `scan_x` loop iteration), the inner loop nest computes a 1×1 *gray* stencil and pushes it to `gray_step_stream`, which is later line buffered and dispatched to the kernels *blur* and *ratio*. The unit length loops `x` and `y` will be eliminated in a later simplification pass.

After the IR transformation, the dataflow IR presents an untimed, bit-accurate representation of the pipeline, with different stages explicitly separated by data streams. For example, the *unsharp* pipeline is composed of four kernel stages, two line buffers, and two stream dispatchers, as shown in Figure 7. Many HLS tools, such as Vivado HLS (Xilinx 2016b) and Catapult HLS (Mentor Graphics 2016), can infer coarse-grained pipelined designs from such code structure. However, the throughput of stages producing less than one pixel/cycle is not optimal, due to limits on the automatic loop pipelining of the current HLS tool. We address this problem next.

4.3 Loop Perfection Optimization

Figure 8(a) shows the IR of a five-point convolution kernel scheduled at the rate of one-fifth pixel/cycle. Loop pipelining in Vivado HLS only applies to a *perfect loop nest* such that only the

Fig. 7. Pipeline implementation for *unsharp*. Processing elements *PE* implement kernel DAGS. Line buffers *LB* capture the data reuse in the stencil pattern. Stream dispatchers *DP* fork a stencil stream to multiple consumers.

```

1 for (scan_x, 0, 256)
2   pop (in, in_stream)
3   out(0) = 0
4   for (d, 0, 5)
5     out(0) += mask(d)*in(d)
6   push (out, out_stream)

```

(a) Original IR.

```

1 for (scan_x, 0, 256)
2   for (d, 0, 5)
3     if (d == 0)
4       pop (in, in_stream)
5       out(0) = 0
6       out(0) += mask(d)*in(d)
7     if (d == 4)
8       push (out, out_stream)

```

(b) IR after the optimization.

Fig. 8. Loop perfection optimization creates a larger perfect loop nest by pushing operations from the outer loop body into the innermost loop.

innermost loop has operations. Here, loops *scan_x* and *d* do not form a perfect loop nest due to the instructions on lines 2, 3, and 6, so the *pop*, *initialization*, *accumulation*, and *push* operations run sequentially and cannot be pipelined in the generated hardware. Moreover, it is expensive in hardware to enter and exit a loop due to pipeline flushing. In this case, the pipeline for the innermost loop (loop *d*) needs to be flushed every five iterations.

In order to create a fully pipelined convolution stage with one-fifth pixel/cycle throughput, the content in loop *scan_x* must be pushed into the innermost loop, as shown in Figure 8(b). The perfect loop nest after the transformation not only creates a longer hardware pipeline containing all the operations, but also eliminates the pipeline flushing overhead of entering an intermediate loop level. In hardware, the “if” statements are usually implemented using multiplexers that select the results from true branches. Nevertheless, for IO operations, such as the push and pop operations in the example, the multiplexers select values of the control signals of the IO ports, so the real IO operations only happen when the predicates are true.

We apply this restructuring automatically in the accelerated region of IR through a recursive descent algorithm. The impact of the optimization is analyzed in Section 6.3. Note that loop perfection is an inverse operation of loop peeling (Wolfe 1992), which moves predicates out of the innermost loop to improve performance on traditional processors, as it reduces the number of dynamic instructions.

4.4 Code Generation

After some common optimizations, the final IR of the pipeline is passed to two different code generator back-ends, HLS and LLVM. The HLS code generator translates the hardware accelerator portions of IR into HLS-synthesizable C code, and the rest of the IR is translated into a C++ test-bench wrapper. The code generator inserts HLS directives (pragmas) automatically to assist the

Fig. 9. Multidimensional line buffer. 1D LB uses shift register. 2D LB uses RAM to buffer rows of pixels, pushes a column to 1D LBs, and outputs 2D stencils. 3D LB instantiates RAMs and 2D LBs similarly.

HLS compiler in applying loop pipelining and array partitioning. To simplify the code generator, we built an HLS-synthesizable C++ template library implementing an abstract line buffer interface:

```
template<int IMG_SIZE_0, int IN_SIZE_0, int OUT_SIZE_0, int IMG_SIZE_1, int IN_SIZE_1, int OUT_SIZE_1, ...>
void linebuffer(stream<IN_SIZE_0, IN_SIZE_1, ...> &in, stream<OUT_SIZE_0, OUT_SIZE_1, ...> &out);
```

We designed the multidimensional (up to 4D) line buffer templates hierarchically, as shown in Figure 9.

As a further design optimization, the compiler can also statically evaluate constant functions (e.g., lookup tables) and generate code that later synthesizes to ROMs.

Generating machine code for the CPU portion of the software program is left to the LLVM compiler infrastructure. We largely use the existing Halide ARM back-end, which includes a highly optimized ARM NEON SIMD vectorizer, thread-pool-based parallel runtime, and so forth. The final IR describes a complete pipeline with both CPU and accelerator, from which the code generator emits platform-specific device driver calls to access the hardware when it visits the boundary of the hardware pipeline during IR traversal. Moreover, the tool recognizes all the data buffers accessed by any hardware pipeline, and thus can emit special allocation routines for these buffers, inserting data transfers where required by the platform setup.

5 PLATFORM DEVELOPMENT

We implemented our system on a Xilinx Zynq-7000 SoC platform (Xilinx 2016d), which contains a pair of ARM Cortex A9 cores and FPGA fabric. The low power of Zynq (total system power of 2W) makes it an ideal platform for running many image processing applications in mobile and battery-powered environments. We run Linux on the ARMs, giving us the many conveniences of an operating system (e.g., a file system, networking, and core utilities), and also realistically modeling the challenges of integrating an accelerator into a larger heterogeneous system. The generated Halide program appears to the user as a single C ABI function, callable from ARM CPU userland; all interaction with the FPGA fabric is automatically managed within that function.

We use Xilinx’s AXI DMA (Xilinx 2016a) to connect the streaming interface of the accelerators to the CPU cache hierarchy through an accelerator coherency port (ACP), which lets us DMA data without cache flushing. The DMA engines use 2D transfer mode to access data row by row with a fixed offset between the start of each row. By adjusting the DMA configuration, we can send a subimage to the accelerator without having to copy or move any data.

Table 1. Applications Used in This Article

Application	Description
gaussian	9-point 2-D Gaussian filter
harris	Harris corner detector
unsharp	unsharp masking filter
stereo	stereo depths using block matching
bilateral grid	fast bilateral filtering algorithm
camera	camera pipeline from Frankencamera

We built a parameterized Linux kernel module to drive the DMA engines and the accelerators. The driver provides a simple interface used by the generated Halide CPU code, including an asynchronous launch for the hardware pipeline and a synchronization barrier to wait for the hardware to complete.

We overlap the execution of CPU cores and accelerators using Linux threads. After the output image has been tiled into smaller blocks, the processing pipeline of the small block is wrapped by a loop iterating over the tiles. A user can schedule the loop to run in parallel using Halide's `parallel` primitive, as shown in line 16 of Figure 3(a). After this, a typical software program looks like:

```

1 parallel foreach tile_index:
2   retrieve pinned_buffers
3   compute values in pinned_buffers...
4   task_id = launch_accelerator(pinned_buffers)
5   wait_sync(task_id)
6   consume values in pinned_buffers...
7   release pinned_buffer

```

During execution, a thread pool is created for launching the workers that run the loop body. Some threads use CPU cores to compute values in the buffer, launch the accelerator task, and then quickly get blocked in the `wait_sync` calls. While these threads sleep and the accelerator is running, other active threads can use the CPU cores.

6 EVALUATION

We now describe our experimental setup, followed by evaluation on five separate fronts: (1) *hardware generation*, comparing our generated hardware versus optimized kernels from an HLS library and generated hardware from the HIPAcc compiler; (2) impact of the *loop perfection optimization*; (3) *heterogeneous system performance*, where we describe our efforts to optimize the hardware and software running on the target platform; (4) *efficient programmability*, where we show that we can generate competitive hardware and software for a variety of Halide applications; and (5) *extensibility* of the framework for generating ASIC designs.

6.1 Experimental Setup

Compiler. Our compiler is based on open-source Halide built using GCC 4.8 and LLVM 3.6. The evaluated implementations for the Zynq platform, ARM CPU, and CUDA GPU are all generated by our compiler, although the CPU and GPU code is the same as produced by the original Halide compiler. We also compared our results to those obtained using the open-source HIPAcc-Vivado compiler (Membarth and Reiche 2016).

Applications. Table 1 lists the applications we use in this article, including *Gaussian*, a basic stencil kernel; *Harris corner detector*, a pipeline of stencil kernels; *unsharp mask*, an example of a DAG

of kernels; *stereo*, a compute-intensive algorithm calculating stereo correspondence using block matching; *bilateral grid*, a fast bilateral filtering algorithm (an edge-preserving smoothing filter) containing rich patterns of histogram, downsampling, and data gathering (Paris and Durand 2009); and a simple *camera* pipeline from Frankencamera (Adams et al. 2010). The last two applications are adapted from Halide’s open-source repository.

Platforms. We use a Xilinx ZC702 evaluation board holding a Zynq XC7Z020 SoC as the target heterogeneous platform, running a Linux 4.0 kernel built from Xilinx Open Source Linux 2015.4 (Xilinx 2016c) along with our custom-built device driver module. We use Xilinx Vivado Design Suite 2015.4 for synthesizing HLS, RTL simulation, and generating an FPGA bitstream, since it officially supports our Xilinx Zynq part.

For comparison to a CPU+GPU platform, we use an NVIDIA Jetson TK1 board with JetPack 2.0. The Tegra K1 SoC is fabricated in the same 28nm technology as the Zynq SoC. We evaluate both ARM and CUDA implementations on TK1 for each application using the same Halide algorithm but different schedules, optimized by Halide experts. For *stereo*, where Halide-generated CUDA performs poorly, we compared to the Tegra-optimized OpenCV CUDA kernel shipped by NVIDIA with JetPack.

Measuring Power and Performance. We pull statistics from Texas Instruments UCD9248 power controllers on the ZC702, which reports the power of each subsystem including FPGA, DRAM, and CPU. For the TK1 board, we measure the current on the 12V DC supply. To derive the energy efficiency for TK1, we subtract the board idle power (about 2 watts) to exclude the board’s uncore components, which should give the TK1 an advantage as it also excludes SoC and DRAM static power. We use `gettimeofday` to measure software program execution time and report the average over 20 runs. For the CUDA target, we exclude the data transfer time between CPU and GPU (again giving the TK1 an advantage).

6.2 Hardware Generation

To evaluate the generated FPGA implementations, we develop two applications in Halide, a 5×5 two-dimensional convolution (*conv2d*)³ and *harris*, both of which are also available in both the Xilinx HLS video library (Xilinx 2016b) and HIPAcc-Vivado’s examples. We find that our system can generate hardware quality similar to that of these other two frameworks. While it is hard or even impossible to code all the applications in Table 1 using the other frameworks, *conv2d* (a single-stage kernel) and *harris* (a pipeline of many kernels) suffice to showcase the hardware building blocks and their compositions, respectively.

Generated FPGA designs for the three frameworks achieve similar peak frequencies of around 180MHz for *conv2d* and 150MHz for *harris*. The *harris* pipelines operate slower because they are bounded by a floating-point-to-integer conversion. Figure 10 shows the FPGA resource utilization of our generated designs versus library and HIPAcc-Vivado designs on a 1080p image. We generated designs at different pixel rates using the same Halide algorithm code but different unrolling schedules, while the library and HIPAcc only provide designs at a single-pixel-per-cycle rate.⁴

Our line buffer components use less BRAM for two reasons. First, the line buffer instance from the library is not optimal in terms of storage usage: in 5×5 convolution, the library design buffers

³*conv2d* is similar to *gaussian* except with configurable filter weights instead of constant weights.

⁴The recent HIPAcc Altera-OpenCL back-end (Özkan et al. 2016) can generate greater than one pixel/cycle pipelines through kernel vectorization, but cannot generate less than one pixel/cycle designs. We did not evaluate this system as it is not open source.

Fig. 10. Resource utilization (area) comparison of *conv2d* (top) and *harris* (bottom) in our system at various pixel rates versus HLS library and HIPAcc implementations. Our system’s designs can target multiple different throughputs, while HIPAcc and library kernels are only optimized at a one pixel/cycle rate.

five rows, whereas the minimum required is four, so both our and HIPAcc’s designs start with 20% less BRAM. Second, the library and HIPAcc both instantiate per-kernel line buffers for each input stencil, while we place a line buffer for each *output* stencil stream, which can be shared among kernels consuming the same stream. In *harris*, our design instantiates fewer line buffer instances thanks to this buffer sharing, for an extra 6% BRAM savings as compared to HIPAcc.

In *harris*, our design and HIPAcc use fewer DSPs and LUTs because the code generation (meta-programming) approach is more flexible and does better constant propagation and simplification as compared to the C++ template solution used by the library. Our unit-rate *conv2d* design has fewer LUTs because the library solution creates four separate streams for the RGBA channels, whereas we package RGBA as a single structure and pass it through a wider stream along the pipeline, which simplifies the control logic for managing data streams.

Multirate designs need higher compute throughput and buffer bandwidth. Fortunately, the BRAM banks of the line buffers provide more than enough bandwidth for 1080p images. Therefore, the multirate designs use more LUTs and DSPs for the arithmetic datapath, while the line buffer resources do not change significantly.

One source of overhead in our design comes from unnecessary FIFOs inserted between pipeline stages. We use `hls::stream` objects to connect different stages in the generated HLS code, and the current HLS compiler creates a hardware FIFO for each stream object. However, in our design, stages can be directly connected through a handshake interface (e.g., AXI4-stream) because the latencies in the pipeline are already balanced. The extra FIFO adds 30% FF and 20% LUT overheads. Future optimizations in the HLS compilation could eliminate these unnecessary FIFOs.

Table 2 summarizes the code length of *conv2d* and *harris* using different systems. Moving from less to more domain specific, Halide and HIPAcc DSLs are orders of magnitude more compact than both HLS C library and our generated HLS code. Because Halide uses functional representation, its application code is $2\times$ shorter than HIPAcc’s.

6.3 Impact of Loop Perfection

As discussed in Section 4.3, the loop perfection optimization helps fully pipeline the stages that have sequential loops (running at <1 pixel/cycle rate). Such pipeline stages can be found in

Table 2. Lines of Code (LoC) for *conv2d* and *harris* in HLS Library, HIPAcc, Halide, and Generated HLS Code from Halide. HLS Lib Excludes Basic Data Structure Code; HIPAcc Counts re DSL Parameter+Algorithm (Membarth et al. 2016); Halide Counts are Algorithm+Schedule; Generated HLS Counts Exclude 900 LoC in the Line Buffer Template Library

	HLS Lib	HIPAcc	Halide	Generated HLS
Conv2D	209	5+5	2+2	885
Harris	520	47+26	23+11	619

Table 3. Scheduled and Measured Throughput and Resource Utilization of *histogram* and *stereo* With and Without the Loop Perfection Optimization

Application	Sched. Rate (pix/cyc)	Meas. Rate (pix/cyc)	Resource (LUT+FF)
histogram	0.016	0.015	1.1%+0.6%
histogram+opt	0.016	0.016	2.0%+1.0%
stereo	0.25	0.062	55%+25%
stereo+opt	0.25	0.25	55%+32%

applications that require data-dependent reductions (e.g., the *histogram* stage in *bilateral grid*) or applications that are intentionally scheduled at a slower rate due to FPGA resource constraints (e.g., *stereo*). This optimization does not affect the results of other applications that do not have such loop patterns in the IR. Therefore, we show results for *histogram* and *stereo*.

Table 3 lists the scheduled and measured throughput and resource utilization of *histogram* and *stereo* with and without the loop perfection optimization. The optimization improves performance by 7% and 300% with low resource overhead for *histogram* and *stereo*, respectively. *Stereo* gains more because its innermost loop is short (four iterations) and the overhead of entering and exiting the short loop is relatively large if not pipelined with the outer loops.

6.4 Heterogeneous System Performance

System performance can be greatly improved if the CPU and the accelerator run concurrently while processing a heterogeneous pipeline. Figure 11 illustrates the effectiveness of our multi-threading technique from Section 5, overlapping CPU and accelerator workloads for *stereo*. *Stereo* is chosen because, for optimal partitioning, its processing pipeline contains similar CPU and FPGA workloads in terms of runtime, and so it best illustrates the benefit of the concurrent processing feature.

The figure plots the program runtime using the same accelerator to process different-size images with and without multiple threads. For each launch, the accelerator processes a 600×400 image tile in 7.63ms. The software program breaks the image into such tiles, prepares an input image tile for the accelerator including padding the boundary of the original image, launches and waits for the accelerator, then repeats for the next tile.

When running the program sequentially with one thread, each tile takes 12.8ms, indicating a CPU workload of about 5ms/tile. However, using three threads to process three tiles concurrently, the per-tile processing cost reduces to 7.96ms, as most of the CPU workload now overlaps with accelerator execution. With this overlap, the system is bounded by its slowest part, which in this

Fig. 11. Runtime of *stereo* with different sizes of input image. All runs use a hardware accelerator that processes a 600×400 image tile in 7.93ms. Our multithreading technique helps overlap CPU and accelerator execution time, so that per-tile processing cost plus CPU overhead becomes just 7.96ms instead of 12.76ms.

Table 4. Performance and Energy Cost Measurements of the *Gaussian* Accelerators for Different Block Sizes and Pixel Rates Running at 100MHz. Larger Blocks Cause L2 Cache Misses to DRAM, Increasing DRAM Dynamic Energy

Rate (px/cy)	Read BW (MB/s)	Block Size (KB)	—HW Time—		DRAM E_{dyn} (nJ/px)
			Sim. (μ s)	Measured (μ s)	
1	300	48	193	199	0.0
		192	710	721	2.5
		768	2,730	2,764	3.3
2	600	48	99	111	0.0
		192	360	510	2.3
		768	1,378	2,444	3.3

case is the accelerator portion of the workload. When the CPU workload dominates, we may not see such nice behavior, as we find out later.

On our target platform, the accelerators are cache coherent and share the L2 cache with the CPU cores. If intermediate buffers passing data between CPU kernels and accelerators fit in the L2, the number of DRAM accesses will decrease, and the memory access latency will improve as well. To this end, blocking the image via the tile primitive effectively reduces the size of the intermediate buffers.

Table 4 summarizes the performance and energy cost of *gaussian* compiled for different block sizes and pixel rates. (Note that the choice of application—in this case *gaussian*—is arbitrary as hardware computation does not affect the results and only the memory traffic on the interface matters.) We show the accelerator execution time for both Verilog simulation and real time as measured when running the accelerator repeatedly on the Zynq. Memory accesses start to miss in the L2 when the block size exceeds 48KB, causing the DRAM dynamic energy to increase from zero to around 3.3nJ/pixel (i.e., 34pJ/bit-access).⁵

⁵Computing an RGB pixel in gaussian requires 6 bytes of memory access. If the memory access misses in the L2, we assume it causes two DRAM accesses.

Table 5. Specifications of Generated Accelerators for the Evaluated Applications. The Resource Utilization is Reported as Percentage of LUT, FF, BRAM, and DSP used by the Given Application on the Zynq XC7Z020

Application	Rate (pix/cyc)	Read BW (MB/s)	Resource (%) (LUT/FF/BRAM/DSP)			
gaussian	2	200	9,	5,	3,	33
harris	2	284	22,	19,	7,	30
unsharp	1	375	7,	5,	7,	31
stereo	0.25	62	55,	32,	5,	0
bilateral	1	182	23,	20,	14,	13
camera	2	568	9,	6,	5,	5
camera+unsharp	1	250	14,	10,	11,	34

When we compare simulated to measured accelerator performance (*sim* vs. *measured* in Table 4), we see that, for low rate configurations, the measured runtime does not degrade when large block sizes cause L2 misses, since DMA buffers can hide the latency. However, for high rate configurations, although the accelerators are twice as fast as the unit-rate designs in simulation, peak speed is only achieved for the 48KB block configuration (i.e., when memory accesses all hit in the L2). Because of cache misses, the actual runtime is 77% longer than the simulated accelerator runtime for the much larger 768KB block configuration.

However, breaking images into many small blocks introduces two problems of its own: first, more overlapping block boundaries cause more recomputation; and second, the host CPU has to schedule more accelerator launches through the device driver interface. On the current platform, we observe scheduling overhead around 100 to 200 microseconds per launch in the device driver, mostly caused by context switches and synchronizations between the user thread and kernel background threads responsible for managing accelerator launch and completion queues. As a result of these overheads, for most of the evaluated applications, fine-grained blocking to fit in the L2 is not an efficient strategy overall. Instead, we chose to block the image in larger sizes (e.g., 480×640) for our full system evaluation.

With further engineering of our existing system, it would be possible to increase memory bandwidth for streaming large blocks. However, if we were to design a new platform, we would choose a faster CPU core (the Zynq’s 667MHz A9 was slow even at its introduction in 2012), a larger shared L2 cache, and a hardware launch engine that pulls accelerator tasks directly from a memory buffer without CPU intervention, as opposed to having a background thread pushing tasks to the accelerator.

6.5 Programmability and Efficiency

We implemented six individual applications plus an application that combined the camera pipeline with an unsharp mask, all in Halide, and generated the hardware and software for Zynq using our compiler.

Table 5 lists the specifications of generated accelerators for these applications. In the evaluation, we manually explored the scheduling space of each application and used the best Halide schedule found in terms of the performance. The ARM cores in the target platform are relatively weak as compared to the FPGA fabric, so for most of the applications, the CPU does not implement any computation kernels but rather controls tiling, that is, calculating the coordinates of each

Fig. 12. Throughput comparison of the four ARM cores, CUDA GPU on TK1, our Zynq platform implementation, and an ideal accelerator. The accelerator RTL simulation throughput represents the theoretical for the accelerator in isolation, if it were never bottlenecked by other parts of the system.

Fig. 13. Energy efficiency comparison of the four ARM cores, CUDA GPU on TK1, and the Zynq platform.

image tile and scheduling the accelerator for processing tiles. In *stereo*, the CPU also computes padding using a repeat edge condition, and in *bilateral grid*, it shuffles data into an 8×8 grid. We run 8-megapixel images through each application, but the BRAM usage for internal buffering in each accelerator is kept low thanks to the image tiling.

Figures 12 and 13 show the throughput and energy efficiency of the Zynq implementation versus TK1's ARM and GPU cores, respectively. On average, applications on Zynq achieve $2.6\times$ and $1.9\times$ higher throughput than CPUs and GPU, respectively, and $14.2\times$ and $6.1\times$ better energy efficiency. *Harris* achieves the most energy reduction of $38\times$ and $12\times$, as well as the highest throughput speedups of $6\times$ and $3.5\times$, compared to the TK1 CPUs and GPU. The energy efficiency is achieved by high locality and data reuse exploited in the line-buffered accelerator pipeline and the greatly reduced memory requests as compared to programmable cores. In addition, the applications mostly use low-precision fixed-point arithmetic, which can be efficiently implemented using the LUTs and DSPs on an FPGA fabric.

All Zynq-based applications except *camera* achieve the peak throughput of the accelerator in Verilog simulation. The *camera* accelerator requires much higher read bandwidth (568MB/s), and thus suffers the most from bandwidth problems caused by reading data that misses in the L2 cache (see Section 6.4).

Once the application fits in the FPGA fabric, the throughput of Zynq implementations is generally bound by memory bandwidth and clock frequency. Therefore, speedups compared to the ARM CPUs or GPU on TK1 are proportional to the number of operations accelerated on the FPGA (approximately proportional to the number of LUTs and DSPs used). For this reason, *harris*,

Fig. 14. Energy cost breakdown and compute intensity in terms of operations-per-byte access to DRAM for different applications on Zynq. The energy cost reduces with more operations accelerated onto FPGA.

stereo, and *camera+unsharp* get the most speedup. In *bilateral grid*, the accelerator also uses a lot of LUTs, but most of them implement control logic and multiplexers for building histograms and data gathering for interpolation, which do relatively little real computation. Moreover, parallelism in the kernels of the *bilateral grid* is limited by data dependencies, making it more favorable to execute on high-frequency and high memory bandwidth processors (i.e., the Tegra CPUs and GPU).

An important takeaway is that acceleration using the FPGA becomes more effective as the image processing pipeline grows deeper, which matches the trend of new applications from computational photography and computer vision. For example, on CPU or GPU, the execution time of the *camera+unsharp* combination is the sum of the execution times of the two individual applications. However, pipelining two applications onto the FPGA fabric simultaneously doesn't increase the required memory bandwidth or slow the clock frequency. Therefore, the throughput for any composition of pipelines that fit on the FPGA is bounded only by the slowest app in the chain (in this case, *unsharp*, which is around 110 megapixels/second).

Figure 14 plots the energy cost per operation and the compute intensity for different applications on Zynq. The FPGA energy ranges from 2.6pJ/op to 35pJ/op depending on the types of operations, while the CPU and DRAM energy scales strongly with the compute intensity. *Stereo* achieves the lowest energy, 5.8pJ/op, with the highest compute intensity of 4,220op/byte. On the other hand, the 26op/byte application, *bilateral grid*, consumes 311pJ per operation, 186pJ of which it spends on DRAM.

7 CONCLUSION

The Halide image processing DSL enables programmers to quickly create and optimize new image processing applications. The availability of complex SoC chips with large FPGA fabrics provides a potential platform for exploring new heterogeneous architectures for these applications, but implementing the hardware accelerator and the interface software is a huge barrier for many designers. We extended Halide to remove this barrier, allowing it to generate both the design of the accelerator and the software that communicates with the hardware. Our results demonstrate significant gains in both performance and energy efficiency, which increase as the imaging computation becomes more complex. We will open-source our system to share with the community for others to use and build on.⁶

⁶The source is available at <https://github.com/jingpu/Halide-HLS>.

We plan to extend our system in several ways. First, we will incorporate more automatic optimization, to help the designer find optimal low-level buffering and blocking. Next, given the generality of the underlying computational model, we want to generate “code” for different underlying image processing engines including custom hardware and/or specialized SIMD and coarse-grained reconfigurable array architectures optimized for image processing appearing in new SoCs. This system will help the community develop and use efficient programmable ISPs.

ACKNOWLEDGMENTS

This work was supported by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

REFERENCES

- A. Adams, E.-V. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy. 2010. The frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics* 29, 4 (July 2010), 29:1–29:12.
- Altera. 2016. Intel FPGA SDK for OpenCL. Retrieved from <https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html>.
- P. M. Athanas and A. L. Abbott. 1995. Real-time image processing on a custom computing platform. *Computer* 28, 2 (Feb. 1995), 16–25.
- J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. 2012. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference (DAC’12)*. ACM, New York, 271–276.
- J. S. Brunhaver. 2015. *Design and Optimization of a Stencil Engine*. Ph.D. Dissertation. Stanford.
- A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems* 13, 2, Article 24 (Sept. 2013), 27 pages.
- J. M. P. Cardoso and P. C. Diniz. 2011. *Compilation Techniques for Reconfigurable Architectures*. Vol. 81. Springer Science & Business Media.
- R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio. 2015. On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article 53 (Dec. 2015), 26 pages.
- W. Chen, Z. Wang, Q. Wu, J. Liang, and Z. Chai. 2016. Implementing dense optical flow computation on a heterogeneous FPGA SoC in C. *ACM Transactions on Architecture and Code Optimization* 13, 3, Article 25 (Aug. 2016), 25 pages.
- N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT’16)*. ACM, New York, 327–338.
- J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (April 2011), 473–491.
- T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL’12)*. IEEE, 531–534.
- B. Draper, W. Najjar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins. 2000. Compiling and optimizing image processing algorithms for FPGAs. In *Proceedings of the 5th IEEE International Workshop on Computer Architectures for Machine Perception*. 222–231.
- B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe. 2003. Accelerated image processing on FPGAs. *IEEE Transactions on Image Processing* 12, 12 (Dec. 2003), 1543–1551.
- J. Emer, P. Ahuja, E. Borch, A. Klauser, Chi-Keung Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. 2002. Asim: A performance model framework. *Computer* 35, 2 (Feb. 2002), 68–76.
- M. Gao and C. Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA’16)*. IEEE, 126–137.
- N. George, H.-J. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. 2014. Hardware system synthesis from domain-specific languages. In *Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL’14)*. IEEE, 1–8.

- V. Govindaraju, C.-H. Ho, and K. Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE, 503–514.
- Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. 2005. Optimized generation of data-path from C codes for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (DATE'05)*. IEEE Computer Society, Washington, DC, 112–117.
- Z. Guo, W. Najjar, and B. Buyukkurt. 2008. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1, Article 6 (May 2008), 26 pages.
- S. Hauck and A. DeHon. 2010. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Vol. 1. Morgan Kaufmann, Burlington, MA.
- J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics* 33, 4, Article 144 (July 2014), 11 pages.
- J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics* 35, 4, Article 85 (July 2016), 11 pages.
- W. Kamp, R. Kunemund, H. Soldner, and R. Hofer. 1990. Programmable 2D linear filter for video applications. *IEEE Journal of Solid-State Circuits* 25, 3 (June 1990), 735–740.
- D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 115–127.
- M. Lavasani. 2015. *Generating Irregular Data-stream Accelerators: Methodology and Applications*. Ph.D. Dissertation. The University of Texas at Austin.
- Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
- G. Martin and G. Smith. 2009. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers* 26, 4 (2009), 18–25.
- Maxeler Acceleration Technology. 2011. MaxCompiler White Paper. Retrieved from <https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf>.
- B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*. Springer, Berlin, 61–70.
- R. Membarth and O. Reiche. 2016. Fork of HIPAcc generating code for Vivado HLS. <https://github.com/hipacc/hipacc-vivado>. (2016).
- R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. 2016. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 210–224.
- Mentor Graphics. 2016. Catapult High-Level Synthesis. Retrieved from <https://www.mentor.com/hls-lp/catapult-high-level-synthesis/>. (2016).
- P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 2 (2012), 15.
- T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmailzadeh, L. Ceze, and M. Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 603–614.
- R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics* 35, 4, Article 83 (July 2016), 11 pages.
- W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. 2003. High-level language abstraction for reconfigurable computing. *Computer* 36, 8 (Aug. 2003), 63–69.
- J. Nickolls, I. Buck, M. Garland, and K. Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (2008), 40–53.
- ON Semiconductor. 2015. 1/4-Inch 5 Mp System-on-a-Chip (SOC) CMOS Digital Image Sensor. Retrieved from <http://www.onsemi.cn/PowerSolutions/document/MT9P111-D.PDF>.
- M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos. 2011. Synthesis of platform architectures from OpenCL-programs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. IEEE, 186–193.
- M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. 2016. FPGA-based accelerator design from a domain-specific language. In *Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL'16)*. IEEE, 1–9.
- A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the IEEE 7th Symposium on Application Specific Processors, 2009 (SASP'09)*. IEEE, 35–42.
- S. Paris and F. Durand. 2009. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision* 81, 1 (2009), 24–52.

- M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. 2011. HASim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 406–417.
- R. Prabhakar, D. Koeplinger, K. J. Brown, H.-J. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun. 2016. Generating configurable hardware from parallel patterns. *SIGOPS Operating System Review* 50, 2 (March 2016), 651–665.
- Qualcomm Inc. 2016. Snapdragon 800 Series Mobile Processors. Retrieved from <https://www.qualcomm.com/products/snapdragon/processors/800-series>.
- J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 32.
- J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. 2013. 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)*. ACM, New York, 519–530.
- R. Ramanath, W. E. Snyder, Y. Yoo, and M. S. Drew. 2005. Color image processing pipeline. *IEEE Signal Processing Magazine* 22, 1 (Jan. 2005), 34–43.
- O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. 2014. Code generation from a domain-specific language for C-based HLSof hardware accelerators. In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'14)*. IEEE, 1–10.
- P. Ruetz and R. Brodersen. 1986. A realtime image processing chip set. In *1986 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, Vol. XXIX. 148–149.
- J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 1–3 (2010), 66–73.
- A. Sujeeth, H.-J. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*. JMLR, Bellevue, WA, 609–616.
- F. Vahid, G. Stitt, and R. Lysecky. 2008. Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer* 41, 7 (2008), 40–46.
- M. Wolfe. 1992. Beyond induction variables. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, New York, NY, 162–174.
- Xilinx. 2016. AXI DMA v7.1 LogiCORE IPProduct Guide. Retrieved from http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.
- Xilinx. 2016. Vivado High-Level Synthesis. Retrieved from <http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- Xilinx. 2016. Xilinx Wiki - Open Source Linux. Retrieved from <http://www.wiki.xilinx.com/Open+Source+Linux>.
- Xilinx. 2016. Zynq-7000 All Programmable SoC Overview. Retrieved from http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
- C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 161–170.
- Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. 2008. *AutoPilot: A Platform-Based ESL Synthesis System*. Springer Netherlands, Dordrecht, 99–112.

Received December 2016; revised May 2017; accepted June 2017