HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration

Jiajie Li*
li-jj16@mails.tsinghua.edu.cn
Tsinghua University

Yuze Chi
chiyuze@cs.ucla.edu
University of California, Los Angeles

Jason Cong
cong@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

The domain-specific language (DSL) for image processing, Halide, has generated a lot of interest because of its capability of decoupling algorithms from schedules that allow programmers to search for optimized mappings targeting CPU and GPU. Unfortunately, while the Halide community has been growing rapidly, there is currently no way to easily map the vast number of Halide programs to efficient FPGA accelerators. To tackle this challenge, we propose HeteroHalide, an end-to-end system for compiling Halide programs to FPGA accelerators. This system makes use of both algorithm and scheduling information specified in a Halide program. Compared to the existing approaches, flow provided by HeteroHalide is significantly simplified, as it only requires moderate modifications for Halide programs on the scheduling part to be applicable to FPGAs. For part of the compilation flow, and to act as the intermediate representation (IR) of HeteroHalide, we choose HeteroCL, a heterogeneous programming infrastructure which supports multiple implementation backends (such as systolic arrays and stencil implementations). By using HeteroCL, HeteroHalide can generate efficient accelerators by choosing different backends according to the application. The performance evaluation compares the accelerator generated by HeteroHalide with multi-core CPU and an existing Halide-HLS compiler. As a result, HeteroHalide achieves 4.15× speedup on average over 28 CPU cores, and 2 ~ 4× throughput improvement compared with the existing Halide-HLS compiler.

ACM Reference Format:
Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373087.3375320

1 INTRODUCTION

Image processing plays a significant role in lots of applications today, including medical imaging [8], autonomous driving [10], augmented reality [18], computational photography [12], etc. However, due to possibly large number of different processing stages and the structured data dependency among them, implementing image processing pipelines efficiently is not an easy job. It is difficult and time-consuming for a designer to write image processing algorithms while parallelizing and optimizing for data locality and performance. Halide [15], a widely-used image processing domain-specific language (DSL), partially solves this problem by decoupling the algorithm and scheduling to allow programmers to search for optimized mappings of the resulting pipelines to various parallel architectures and complex memory hierarchies.

Computation and energy consumption are often the bottlenecks of image processing tasks due to great amounts of pixels and complex algorithms. For example, it requires 120 gigaops/sec to process 1080p/60fps raw video [9]. Considering there is a lot of data locality in image processing applications, appropriate customized hardware designs could result in enormous computation and energy saving, which is not available to CPU and GPU due to their fixed architecture. Therefore, field programmable gate arrays (FPGAs) are a great acceleration platform to provide a configurable computation fabric to match a different data flow and computation pattern for different applications [2, 9, 14, 16].

While the Halide community has been growing rapidly in recent years (received over 3,000 stars on GitHub [7]), there is no way to easily migrate the vast number of Halide programs to FPGA accelerators. The direct and traditional way to design FPGA accelerators is to rewrite programs to register-transfer level (RTL) code. This is very time-consuming. Although C-based high-level synthesis (HLS) raises the design abstraction level to untimed specification by automated scheduling, pipelines, and resource sharing [5], it still requires expertise on the microarchitecture to get efficient designs that result in a high threshold for software programmers. The complicated rules of using scheduling primitives for HLS bring programmers a greater workload as well.

Another approach is to rewrite Halide programs to hardware-oriented DSLs, such as Darkroom [9], HIPAcc [16], PolyMage [3], SODA [2], HeteroCL [11], etc. But it still takes time to learn these DSLs, not to mention other limitations that exist in these DSLs. For example, Darkroom supports only 1 pixel/cycle pipelines, which may not be acceptable for many image applications. SODA can achieve great performance, and implement lots of optimizations for stencil programs, but it does not support non-stencil programs, such as convolution and matrix multiplication. HeteroCL is a promising heterogeneous programming language inspired by Halide, but it takes time for Halide programmers to learn. Some conventions and behaviors of HeteroCL and Halide are not the same. This may cause confusion to programmers who try to manually migrate from Halide to HeteroCL.

The only prior work from Halide to FPGA is Halide-HLS [14]. Halide-HLS is a Halide-to-FPGA compiler and allows programmers

*Work mainly done at UCLA during the research internship in Summer 2019.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375320

to design FPGA accelerators without many modifications. It provides a simple way for Halide programmers to implement their programs on a Xilinx Zynq FPGA. However, due to the lack of active maintenance, the HLS code it generates is no longer supported by the latest FPGA vendor tools, which means no one can actually make use of it now. Even if someone is willing to make it up-to-date, a significant amount of engineering effort would be required because their code generator and the generated architecture is tightly and directly coupled with the backend HLS tool. Nevertheless, this work is an important motivating factor of this study.

In this paper, we propose HeteroHalide, an end-to-end system for compiling Halide programs to FPGA accelerators, making use of both algorithm and scheduling information specified in a Halide program. HeteroHalide not only significantly simplifies the migration effort, but also enables efficient accelerator designs via its flexible backend choices. More concretely, this paper makes the following contributions:

- HeteroHalide provides an easy-to-use flow from Halide to FPGA. It only requires moderate modifications for Halide programs on the scheduling part, instead of algorithm. Compared to other existing approaches, including rewriting in HLS / RTL, this solution of migrating Halide greatly reduces the migration effort.
- HeteroHalide generated FPGA accelerators outperform both CPU with 28 cores and the Halide-HLS FPGA compiler. Accelerators generated by HeteroHalide achieve 4.15× speedup on average over CPU, and has 2 ~ 4× peak performance as Halide-HLS, when tested on the same Zynq-7020 board.
- We develop a Halide-to-HeteroCL code generator, which can automatically generate HeteroCL [11] code, with both algorithms and schedules. We choose HeteroCL as the intermediate representation (IR) in HeteroHalide, because of its great hardware customization capability, using the idea of separating algorithms and schedules. HeteroCL supports multiple heterogeneous backends (spatial architectures), including a stencil backend [2], a systolic array backend [6], and the general Merlin compiler backend [4, 17]. Therefore, it is able to generate efficient hardware code according to the type of the applications.
- When Halide is compiled, the scheduling is applied directly at IR level using *immediate transformation*. We make extensions on Halide schedules, allowing some schedules to be lowered with annotations, using *lazy transformation*. By adding this extension to Halide, HeteroHalide can generate specific scheduling primitives at the HeteroCL backend level, thus emitting more efficient accelerators.

The remainder of this paper is organized as follows. In Section 2, we introduce the Halide DSL and its intermediate representation (IR) system. Section 3 describes our extensions to the Halide language, which allows the scheduling primitives to be mapped to FPGA. We report our evaluation in Section 4 and compare with related work in Section 5. Section 6 concludes this work and outlines future research directions.

2 BACKGROUND

Many image processing applications have bottlenecks in their computation and energy efficiency, due to the high pixel count and the complex algorithm. Considering that there are plenty of data locality and parallelism in these applications, exploiting these features with optimized execution strategies is important for programmers to write high-performance code. This, however, greatly increases coding complexity and requires hardware knowledge from programmers. Domain-specific languages (DSL), e.g., Halide [15], are developed to reduce the amount of detailed knowledge required for application creation.

2.1 Halide

Halide [15] is an open-source DSL for fast and portable computation on images and tensors. It is designed to help programmers write high-performance image processing code easily on modern machines. One of the biggest advantages of Halide is that it decouples the algorithm description of the program from the scheduling — its execution strategy. When trying to optimize Halide code, programmers can simply modify the code of the scheduling part without changing the algorithm part to change how the program is executed. For equivalent C or C++ code, programmers have to change the whole loop of their code. Halide scheduling primitives include loop transformations like `split`, `fuse`, `reorder`, `tile`, etc., and parallelization primitives like `unroll`, `parallel`, `vectorize`, etc. This allows programmers to explore the design space and search for the optimal schedule for their targeting machines. Halide currently targets CPU and GPU, which are both software platforms without hardware customization capability.

2.2 Halide IR System

The intermediate representation (IR) of Halide connects Halide source code and the corresponding target code of CPU or GPU architectures. Halide lowers the source code to the IR level first, applies optimization passes, and then the target's code generator emits code for the resulting pipeline. Halide analyzes the syntax and semantics of the Halide program and transforms them into an abstract syntax tree (AST). Every node in the AST represents an operation to the variables, such as `Add` and `Store`. In the process of analyzing syntax, a node may point to other nodes. Therefore, Halide builds the connections between operations and variables, and constructs this AST. The AST is used as the IR in Halide.

2.3 HeteroCL

HeteroCL [11] is a Python-based domain-specific language (DSL). Similar to Halide, HeteroCL separates the algorithms and the schedules. One major difference between HeteroCL and Halide is that HeteroCL supports a compilation flow targeting FPGA with multiple backends, including SODA [2], PolySA [6], and Merlin Compiler [4, 17]. Therefore, the customized schedule can be migrated to the hardware design in subsequent HLS / RTL code generated. HeteroCL categorizes the hardware customization into three types: compute, data types, and memory architectures, which allows programmers to explore performance / area / accuracy trade-offs.

The heterogeneous backend supported by HeteroCL generates HLS code, which then is synthesized to RTL code using vendor tools. These backends target different types of programs and achieves decent performance. SODA [2] targets stencil computation, which performs local computation over a sliding window of the input

array(s). Stencil computation is widely-used in image processing applications, e.g., the Gaussian blur filter and the Harris corner detector [1]. PolySA [6] targets systolic arrays, an architecture consisting of a group of identical processing elements (PE). This architecture is applicable to a wide range of applications, including convolution computation and matrix multiplication, etc. Apart from the above, Merlin Compiler [4, 17] is a more general backend that can generate optimized HLS code for both Intel and Xilinx platforms and greatly enhance the generality of HeteroCL.

Therefore, we choose HeteroCL as the IR in our flow from Halide to FPGA to fully utilize the heterogeneous feature of HeteroCL and to enable easy extensions to the flow in the future. Connecting the Halide-to-HeteroCL code generator with multiple backends, we are able to generate efficient hardware code according to the type of applications. If new and more efficient backends are developed for HeteroCL, one only needs to change the schedule generated by HeteroHalide to make use of them.

3 COMPILATION FLOW

HeteroHalide provides an automated flow from Halide to FPGA accelerators. It consists of a Halide-to-HeteroCL code generator, HeteroCL, and multiple backends of HeteroCL. More specifically, HeteroCL generates corresponding hardware domain-specific languages (DSL) to these heterogeneous backends, and then those backends generate hardware code from their hardware DSLs.

Both Halide [15] and HeteroCL [11] separate algorithms and schedules in the code. However, there are still some semantic gaps between them in the scheduling part while the algorithm part is basically consistent during code generation. When Halide is compiled, the scheduling primitives are applied at the IR level and are hard to recover, but HeteroCL often needs explicit scheduling information to generate efficient accelerators. To tackle this challenge, we propose two schedule lowering methods and extend Halide by changing the scheduling primitives accordingly. The rest of this section introduces the process of code generation for algorithms and schedules, with examples, and our extensions on schedule transformation.

3.1 Algorithm Transformation

This step is straightforward. We use the blur filter as an example to show the compilation from Halide algorithm code to HeteroCL algorithm code. The blur filter consists of two stages. Each stage is described by a Halide function and represents a 3×1 (or 1×3) filter, as shown in Listing 1.

```
1 Func blur_x("blur_x");
2 blur_x(x, y) = (input(x, y) + input(x+1, y) + input(x+2, y))/3;
3 Func blur_y("blur_y");
4 blur_y(x, y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2))/3;
```

Listing 1: Halide algorithm.

The corresponding HeteroCL algorithm code is shown in Listing 2. The generated HeteroCL code uses a top function to describe the overall algorithm. The sub-stage and the following for loop correspond to each Func in Halide. Other than the syntax and convention differences, we can see that Halide and HeteroCL share very similar code structures for the algorithms. This makes the transformation relatively simple. We use HeteroCL's imperative programming APIs (`heterocl.Stage`, `heterocl.for_`) instead of the compute API (`heterocl.compute`). In this way, we explicitly represent the computing for loop in the HeteroCL source code that is close to both Halide IR and HeteroCL IR. This improves the scalability and stability of the Halide-to-HeteroCL code generator.

```
1  def top(input_hcl):
2      with heterocl.Stage("blur_x"):
3          with heterocl.for_(y_min, y_max) as y:
4              with heterocl.for_(x_min, x_max) as x:
5                  tensor_blur_x[x, y] = (input_hcl[x, y]
6                  + input_hcl[x+1, y] + input_hcl[x+2, y]) / 3
7
8      with heterocl.Stage("blur_y"):
9          with heterocl.for_(y_min, y_max) as y:
10             with heterocl.for_(x_min, x_max) as x:
11                 tensor_blur_y[x, y] = (tensor_blur_x[x, y]
12                 + tensor_blur_x[x, y+1] + tensor_blur_x[x, y+2]) / 3
13
14     return tensor_blur_y
```

Listing 2: HeteroCL algorithm.

3.2 Schedule Transformation

Unlike the algorithm part, effort is needed for code generation for the scheduling part. While Halide implements the schedules directly at the IR level, HeteroCL needs explicit scheduling (customization) information in order to generate efficient FPGA accelerators. Therefore, we propose two methods for schedule lowering: immediate transformation (Section 3.2.1), and lazy transformation (Section 3.2.2). These two different methods are further explained with examples as follows.

3.2.1 Immediate Transformation. It summarizes that Halide schedules are directly implemented at the IR level. As an example, we apply the Halide `unroll` schedule to the blur filter shown in Listing 1 to demonstrate the process of immediate transformation. Line 2 in Listing 1 represents a computation stage. Without any customized schedule, its default loop nest is shown in Listing 3.

```
1 for y [min=...; extent=...; stride=1]:
2     for x [min=...; extent=...; stride=1]:
3         blur_x(y, x) = ...
```

Listing 3: Halide IR of the first loop nest in the blur filter without schedules.

Then we apply the Halide schedule `blur_x.unroll(x,4)` to unroll the `x` loop with a factor of 4. It is directly implemented into the Halide IR, and the loop nest is transformed to Listing 4.

```
1 for y [min=...; extent=...; stride=1]:
2     for x [min=...; extent=...; stride=4]:
3         blur_x(y, x) = ...
4         blur_x(y, x + 1) = ...
5         blur_x(y, x + 2) = ...
6         blur_x(y, x + 3) = ...
```

Listing 4: Halide IR of the first loop nest in the blur filter with `blur_x.unroll(x,4)` applied using immediate transformation.

Via immediate transformation, the schedule is directly implemented into the Halide IR, and the explicit scheduling information is lost at the IR level in the process of lowering.

3.2.2 Lazy Transformation. In this case, Halide schedules are stored explicitly at the IR level as an annotation. The scheduling annotation is further transferred to the subsequent steps of the flow and implemented by the backends. To have a clear comparison with *immediate transformation*, we apply the same unrolling schedule to the blur filter again, but this time we use `lazy_unroll`, which is added as an extension to the existing Halide schedules. With `blur_x.lazy_unroll(x,4)` applied, the corresponding loop nest is shown in Listing 5.

```
1 for y [min=...; extent=...; stride=1]:
2     for x [min=...; extent=...; stride=1; unrolled; factor=4]:
3         blur_x(y, x) = ...
```

Listing 5: Halide IR of the first loop nest in the blur filter with `blur_x.lazy_unroll(x,4)` applied using lazy transformation.

In Line 2 of Listing 5, the `unrolled` annotation and the corresponding unroll factor are stored in the `For` IR node corresponding to the `x` loop. Thus, explicit scheduling information is maintained at the IR level and can be implemented in subsequent steps of the flow. This is necessary because the HeteroCL backends sometimes need to apply their unique primitives to direct HLS schedules, using the information in the annotations.

These unique scheduling primitives for HeteroCL backends are essential for emitting efficient FPGA code. As HeteroCL supports all those backends, the best way to support the scheduling transformation from Halide to heterogeneous hardware DSLs is to fully utilize HeteroCL and to generate explicit scheduling code for HeteroCL. We keep using the `lazy_unroll` schedule as an example and demonstrate the subsequent compilation flow for this schedule.

```
1 schedule = heterocl.create_schedule([input_hcl], top)
2 stage_blur_x = top.blur_x
3 schedule[stage_blur_x].unroll(stage_blur_x.axis[1], 4)
```

Listing 6: HeteroCL scheduling code of the first loop nest in the blur filter with `blur_x.lazy_unroll(x,4)` applied.

Listing 6 shows the corresponding HeteroCL schedule. First, a HeteroCL API is called to create a default schedule based on algorithm code of HeteroCL (Line 1). The function defining the algorithm (`top`) and its input tensor(s) (`[input_hcl]`) are passed to the `heterocl.create_schedule` API. Then, in Line 2, our target stage is identified with the algorithm `top` and the stage's name. Line 3 applies the `unroll` scheduling primitive to `stage_blur_x`. `axis[1]` corresponds to loop `x`, and the unroll factor is 4. The HeteroCL scheduling code in Listing 6 is then transformed to different backend scheduling codes using different HeteroCL backend code generators.

```
1 for (int y = ...; y < ...; y++)
2 #pragma ACCEL parallel factor=4 flatten
3     for (int x = ...; x < ...; x++)
4         blur_x[y][x] = ...
```

Listing 7: Merlin C code generated from the HeteroCL code.

Here, we show two HeteroCL backend schedules as examples. Listing 7 shows the loop nest generated by the Merlin C backend and its scheduling primitives. Merlin C is an OpenMP-like programming model used by the Merlin compiler [4] from Falcon Computing Solutions [17]. Similar to the Halide IR loop nest with lazy transformation in Listing 5, the explicit scheduling information is stored as an annotation (Line 2 in Listing 7 and Line 2 in Listing 5). Another example is the SODA [2] DSL, which can be synthesized into efficient accelerators with scalable parallelism in addition to fully pipelined communication reuse buffer with the least possible buffer size. SODA's unrolling primitive is relatively simple: `unroll factor: 4`, because the SODA microarchitecture inherently unrolls the inner loop of every stage with the same unroll factor.

Table 1: Schedule primitives and the corresponding transformation methods supported by Halide vs HeteroHalide.

Primitive	Description	Halide	HeteroHalide
`reorder`	Switch the order of sub-loops in the same nested loop.	Immediate	Immediate
`split`	Split a loop into a two-level nested loop given the extent of the inner loop.	Immediate	Immediate
`fuse`	Fuse a two-level nested loop into a single-level loop.	Immediate	Immediate
`tile`	Split an iteration domain into smaller tiles and iterate over each tile separately.	Immediate	Immediate
`unroll`	Unroll a loop with given factor.	Immediate	Lazy
`parallel`	Schedule a loop in parallel.	Immediate	Lazy

3.3 Extensions on Halide Schedules

In this section, we summarize our extensions on Halide schedules and the design methodology of schedule transformation for different Halide schedules.

To obtain efficient FPGA accelerators, we need to generate scheduling primitives (e.g., Line 2 in Listing 7) in the process of compilation. As HeteroCL [11] is a heterogeneous programming platform, maintaining schedule explicitly in the process of Halide-to-HeteroCL is essential. Therefore, we change the lowering method for some Halide schedules (e.g., `lazy_unroll` introduced in Section 3.2.2) as extensions. Similar schedule extensions are used for other backend targets as well (e.g., `gpu_tile`) [7].

Table 1 lists the schedules supported by Halide and HeteroHalide, and the corresponding schedule lowering methods. By default, Halide uses immediate transformation to implement the schedule directly into the IR. Loop transformation schedule primitives, e.g., `reorder`, `split`, do not have special semantics in HeteroCL, and, therefore, there is no need to create new scheduling primitives for them. However, for the parallelization scheduling primitives, e.g., `unroll`, `parallel`, the explicit scheduling information is required in HeteroCL to generate efficient backend code. For example, if the unrolling schedule is applied immediately and the Halide IR is in the form of Listing 4, HeteroCL will not be able to generate SODA DSL and leverage its highly-efficient spatial architecture. Therefore, we create lazily-applied Halide schedules `lazy_unroll` and `lazy_parallel` to transfer scheduling information explicitly to generate efficient FPGA accelerators. Note that not all scheduling primitives are applicable to both Halide and HeteroHalide. For example, `vectorize` is only applicable to Halide, whereas pipelining

Table 2: Applications used in the evaluation.

Application	Description
Harris	Harris corner detector.
Gaussian	3 × 3 Gaussian filter.
Unsharp	Unsharp masking filter.
Blur	Average over 3 × 3 window.
Linear Blur	Blur with two linear transformations.
Stencil Chain	3 × 3 kernel chained 3 times.
Dilation	Maximum over 3 × 3 window.
Erosion	Minimum over 3 × 3 window.
Median Blur	Median over 3 × 3 window.
Sobel	Sobel edge detector.
GEMM	General matrix multiplication.
K-Means	K-means clustering.

Table 3: LoC at different levels in the flow. HeteroHalide and HeteroCL counts are algorithm + schedule. Numbers in parentheses are ratios over HeteroHalide.

Application	HeteroHalide	HeteroCL Generated	HLS Generated
Harris	26 + 14	72 + 22 (2.4×)	14224 (355.6×)
Gaussian	8 + 3	23 + 8 (2.8×)	17181 (1561.9×)
Unsharp	13 + 5	46 + 12 (3.2×)	21383 (1187.9×)
Blur	2 + 4	9 + 4 (2.2×)	1455 (242.5×)
Linear Blur	11 + 10	22 + 10 (1.5×)	1072 (51.0×)
Stencil Chain	15 + 10	14 + 8 (0.9×)	9061 (362.4×)
Geo. Mean	—	(2.0×)	(378.6×)

is implicitly inferred in HeteroHalide (and thus no explicit scheduling primitive is required nor provided).

4 EVALUATION

We now present our experiments, followed by the evaluation on two parts: 1) *programming efficiency*, where we show the simplified migration effort from Halide to FPGA accelerators via the lines of code (LoC) comparison, and 2) *accelerator performance*, comparing the throughput of the FPGA code generated by HeteroHalide and the throughput of 28 CPU cores with several real-world applications. This section also compares the peak performance between the accelerators generated by HeteroHalide and those reported by Halide-HLS [14]. The applications we use in the evaluation and the corresponding description are listed in Table 2.

4.1 Programming Efficiency

In this section, we compare the lines of code (LoC) of the same program at different levels in the flow to demonstrate the different workload required when compiling Halide to FPGA accelerators.

The compilation flow from HeteroHalide consists of the following steps: Halide to HeteroCL, HeteroCL to HLS code, and finally to an FPGA accelerator. It is possible to obtain the same FPGA accelerator by manually writing the corresponding code at any level in the flow, but the required programming effort is significantly different. Table 3 shows the LoC comparison among the code at different levels. For most applications, Halide code is more compact than HeteroCL code. Both of them are orders of magnitude more compact than our generated HLS code. The partial reason of this significant difference is HLS code generated by a compiler is redundant compared to optimized HLS code.

Table 4: LoC comparison between HeteroHalide and Xilinx xfOpenCV Library [19]. HeteroHalide counts are algorithm + schedule. xfOpenCV counts include only the core functions, not the utility libraries. Numbers in parentheses are ratios over HeteroHalide.

Application	HeteroHalide	xfOpenCV
Harris	26 + 14	117 (2.9×)
Gaussian	8 + 3	104 (9.5×)
Dilation	2 + 1	80 (26.7×)
Erosion	2 + 1	79 (26.3×)
Median Blur	2 + 1	81 (27.0×)
Sobel	3 + 2	208 (41.6×)
Geo. Mean	—	(16.7×)

Table 5: Performance comparison with Halide-HLS [14].

Benchamrk	Data Sizes & Type	Halide-HLS	HeteroHalide	Speedup
Harris	640 × 640, UInt8	2 pixel/cycle	4 pixel/cycle	2
Gaussian	640 × 640, UInt8	2 pixel/cycle	8 pixel/cycle	4
Unsharp	640 × 640 × 3, UInt8	1 pixel/cycle	4 pixel/cycle	4
Geo. Mean	—	—	—	3.175

Table 4 summarizes the LoC comparison between the Halide code and the Xilinx xfOpenCV library [19]. The kernels in the xfOpenCV library are optimized for Xilinx FPGAs and SoCs, based on the OpenCV computer vision library. Compared with the HLS code optimized by experts, Halide code is still more concise and compact. In summary, for both approaches without HeteroHalide, i.e., rewriting Halide in HeteroCL and HLS respectively, the workload for programmers increases, not to mention the additional knowledge required. HeteroHalide greatly simplifies the process of migrating Halide to FPGA accelerators.

4.2 Accelerator Performance

In this section, we first evaluate the accelerators generated by HeteroHalide. The experiments for CPU are performed on an Ubuntu 16.04 server with two Intel Xeon 2680v4 CPU (28 cores in total) and 64 GiB DDR4 memory. Our target FPGA is the state-of-the-art Xilinx VU9P FPGA, whose default target frequency is 250 MHz.

Table 6 shows the applications and overall evaluation results of each application with certain data sizes and type, including the speedup over CPU, the energy efficiency gain, the accelerator's maximum throughput for stencil applications, and the resource utilization given by the post-synthesis report. Since Halide originated as an image processing DSL and stencil kernels are extensively used, we mainly focus on those in the experiments. To demonstrate the capability of using multiple backends via leveraging HeteroCL [11], two other applications, GEMM and K-Means, that are not in the image processing domain are included as well. The energy efficiency gain is the accelerator-to-CPU ratio and is calculated based on the thermal design power. The throughput of accelerators is memory-bounded and is calculated based on the total bandwidth and the data type width of the application. For the CPU execution of the Halide [15] programs, we use the same scheduling strategies as the examples provided in the open-source Halide repository [7]. The specific parameters such as the unroll factor and tiling size are manually fine-tuned in our testing environment. We leverage the auto-scheduling feature of Halide [13] to compare our manual fine-tuned schedules with the optimal strategies generated by Halide

Table 6: Evaluation results of accelerators generated by HeteroHalide. The speedup is HeteroHalide over 28 CPU cores.

Benchmark	Data Size & Type	#LUT	#FF	#DSP	#BRAM	Throughput	Energy Efficiency	Speedup	Pattern	Back End
Harris	2448 × 3264, UInt8	55198	64427	264	80	32 pixel/cycle	29.11	10.31	Stencil	SODA [2]
Gaussian	2160 × 3840, UInt8	67298	41496	768	0	32 pixel/cycle	17.17	6.08	Stencil	SODA
Unsharp	2448 × 3264 × 3, UInt8	47683	33114	400	24	32 pixel/cycle	9.57	3.39	Stencil	SODA
Blur	648 × 482, UInt16	6821	8209	32	0	16 pixel/cycle	10.98	3.89	Stencil	SODA
Linear Blur	768 × 1280 × 3, Float32	31049	39369	536	16	8 pixel/cycle	12.65	4.48	Stencil	SODA
Stencil Chain	1536 × 2560, UInt16	61230	46174	48	192	16 pixel/cycle	4.29	1.52	Stencil	SODA
Dilation	6480 × 4820, UInt16	13046	12114	0	64	32 pixel/cycle	4.69	1.66	Stencil	SODA
Median Blur	6480 × 4820, UInt16	14388	10066	0	64	32 pixel/cycle	12.51	4.43	Stencil	SODA
GEMM	1024 × 1024 × 1024, Int16	454492	800283	2507	932	—	9.97	3.53	Systolic Array	PolySA [6]
K-Means	320×32, k=16, Int32	212708	235011	1536	32	—	29.00	10.27	General	Merlin Compiler [4, 17]
Geo. Mean	—	—	—	—	—	—	11.71	4.15	—	—

auto-scheduler on CPU. The geometric mean of CPU speed ratio between manual and auto scheduling tested on several benchmarks is 1.15, which shows that our manual optimizations are on par with the highly optimized schedules after extensive design-space exploration.

Note that the CPU code generated by Halide is capable of utilizing all 28 cores available on the server. The experimental results show that the accelerators generated by HeteroHalide achieves 4.15× average speedup and 11.71× energy efficiency gain over CPU.

Table 5 lists the peak performance of the accelerators generated by HeteroHalide and Halide-HLS [14] for three applications. Since we were unable to reproduce the results using the current synthesis tools, for Halide-HLS we use the reported numbers in their paper [14]. The throughput of HeteroHalide is obtained using the same Zynq 7020 device as Halide-HLS. The results show that for various applications, HeteroHalide achieves 2 ~ 4× speedup over Halide-HLS. Clearly, HeteroHalide is more efficient to migrate Halide programs to FPGA accelerators.

5 RELATED WORK

There exist projects that can compile image domain-specific languages (DSL) into hardware code. Darkroom [9], HIPAcc [16], and PolyMage [3] create their own image DSLs and provide compilation flow from the DSLs to hardware code, generating efficient FPGA and / or ASIC designs. All these compiler tools use a line buffered pipeline microarchitecture template to help emit efficient hardware designs. Different from these image DSLs, HeteroHalide leverages the existing Halide [15] infrastructure and keeps algorithms decoupled from schedules, which greatly improves the portability and composability of code. The moderate modifications on the scheduling part of existing Halide programs enables fast adoption of FPGA accelerators for the vast number of Halide programs.

Similarly, Halide-HLS [14] presents a Halide-to-HLS compiler, providing an automatic pass to implement Halide programs to hardware designs. However, the Halide-HLS compiler is not designed in a composable and hierarchical way, i.e., its scheduling primitives and the corresponding code generators are tightly and directly coupled with the underlying microarchitecture template. This makes it difficult to leverage state-of-the-art accelerator microarchitectures for the best performance and adapt to behavior changes in the vendor tools. Moreover, Halide-HLS does not have the option to delay loop transformations and does not handle the scheduling semantics in the most efficient way. As a result, it does not scale as well as HeteroHalide, which is shown in Table 5.

In comparison, HeteroHalide chooses HeteroCL [11] as the intermediate representation (IR) of the whole flow. With HeteroCL as an actively developing heterogeneous programming infrastructure that supports multiple backends and the corresponding efficient spatial architectures, any behavior changes in the vendor tools would be handled by HeteroCL and its backends, minimizing the maintenance burden. Furthermore, if hardware experts find a more efficient microarchitecture for image processing applications, Halide programmers would be able to leverage it as soon as HeteroCL provides a backend. Note that since HeteroCL is designed to play a role like what LLVM does for connecting the frontend source code (e.g., C, C++, Go, Rust, etc.) and the backend machine architecture (e.g., x86, x64, ARM, PowerPC, etc.), maintaining a HeteroCL-to-DSL backend enables much more code reuse than a direct Halide-to-DSL backend.

6 CONCLUSION

In this paper, we present HeteroHalide, an end-to-end system for migrating Halide to efficient FPGA accelerators. Compared to existing approaches, HeteroHalide greatly simplifies the flow from Halide to hardware. We extend the existing Halide schedules in order to generate efficient code for the backend tools. With only moderate modifications on the scheduling part of Halide programs, HeteroHalide is able to generate accelerators with 4.15× average speedup over 28 CPU cores and 2 ~ 4× speedup over existing work. Moreover, by choosing HeteroCL, a heterogeneous programming infrastructure that supports multiple backends, as the intermediate representation (IR) of HeteroHalide, the frontend and backend of HeteroHalide is decoupled with high extensibility. The evaluation demonstrates significant gains in both performance and programming efficiency for programmers.

ACKNOWLEDGMENTS

This work is supported by the Intel and NSF joint research programs for Computer Assisted Programming for Heterogeneous Architectures (CAPA), Tsinghua Academic Fund for Undergraduate Overseas Studies, and Beijing National Research Center for Information Science and Technology (BNRist). We thank Prof. Zhiru Zhang (Cornell) and his research group for their help on HeteroCL and Prof. Mark Horowitz (Stanford) and his research group for their help on Halide-HLS. We also thank Amazon for providing AWS F1 credits.

REFERENCES

[1] Alan C. Bovik. *The Essential Guide to Image Processing*. Academic Press, 2009.

[2] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA : Stencil with Optimized Dataflow Architecture. In *ICCAD*, 2018.

[3] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs. In *PACT*, 2016.

[4] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers. In *ISLPED*, 2016.

[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*, 2011.

[6] Jason Cong and Jie Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *ICCAD*, 2018.

[7] Halide developers. Halide. https://github.com/halide/Halide, 2019.

[8] Hayit Greenspan, Bram Van Ginneken, and Ronald M Summers. Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique. *IEEE Transactions on Medical Imaging*, 35(5):1153–1159, 2016.

[9] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. In *SIGGRAPH*, 2014.

[10] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[11] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *FPGA*, 2019.

[12] Rastislav Lukac. *Computational photography: methods and applications*. CRC Press, 2016.

[13] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically Scheduling Halide Image Processing Pipelines. In *SIGGRAPH*, 2016.

[14] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *TACO*, 14(3), 2017.

[15] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. In *SIGGRAPH*, 2012.

[16] Oliver Reiche, M. Akif Ozkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-based Image Processing Accelerators with Hipacc. In *ICCAD*, 2017.

[17] Falcon Computing Solutions. https://www.falconcomputing.com, 2019.

[18] Erik Todeschini. Augmented-reality signature capture, February 2 2016. US Patent 9,251,411.

[19] Xilinx. Xilinx xfOpenCV Library. https://github.com/Xilinx/xfopencv, 2019.