RIPL: A Parallel Image Processing Language for FPGAs

ROBERT STEWART, KIRSTY DUNCAN, GREG MICHAELSON, and PAULO GARCIA,
Heriot-Watt University
DEEPAYAN BHOWMIK, Sheffield Hallam University
ANDREW WALLACE, Heriot-Watt University

Specialized FPGA implementations can deliver higher performance and greater power efficiency than embedded CPU or GPU implementations for real-time image processing. Programming challenges limit their wider use, because the implementation of FPGA architectures at the register transfer level is time consuming and error prone. Existing software languages supported by high-level synthesis (HLS), although providing a productivity improvement, are too general purpose to generate efficient hardware without the use of hardware-specific code optimizations. Such optimizations leak hardware details into the abstractions that software languages are there to provide, and they require knowledge of FPGAs to generate efficient hardware, such as by using language pragmas to partition data structures across memory blocks.

This article presents a thorough account of the Rathlin image processing language (RIPL), a high-level image processing domain-specific language for FPGAs. We motivate its design, based on higher-order algorithmic skeletons, with requirements from the image processing domain. RIPL's skeletons suffice to elegantly describe image processing stencils, as well as recursive algorithms with nonlocal random access patterns. At its core, RIPL employs a dataflow intermediate representation. We give a formal account of the compilation scheme from RIPL skeletons to static and cyclostatic dataflow models to describe their data rates and static scheduling on FPGAs.

RIPL compares favorably to the Vivado HLS OpenCV library and C++ compiled with Vivado HLS. RIPL achieves between 54 and 191 frames per second (FPS) at 100MHz for four synthetic benchmarks, faster than HLS OpenCV in three cases. Two real-world algorithms are implemented in RIPL: visual saliency and mean shift segmentation. For the visual saliency algorithm, RIPL achieves 71 FPS compared to optimized C++ at 28 FPS. RIPL is also concise, being 5x shorter than C++ and 111x shorter than an equivalent direct dataflow implementation. For mean shift segmentation, RIPL achieves 7 FPS compared to optimized C++ on 64 CPU cores at 1.1, and RIPL is 10x shorter than the direct dataflow FPGA implementation.

CCS Concepts: • **Computing methodologies → Image processing**; • **Computer systems organization → Data flow architectures**; • **Hardware → Reconfigurable logic and FPGAs**; • **Software and its engineering → Domain specific languages**;

We acknowledge the support of the Engineering and Physical Research Council, grant references EP/K009931/1 (Programmable Embedded Platforms for Remote and Compute Intensive Image Processing Applications), EP/N014758/1 (The Integration and Interaction of Multiple Mathematical Reasoning Processes), and EP/N028201/1 (Border Patrol: Improving Smart Device Security Through Type-Aware Systems Design).
Authors' addresses: R. Stewart, K. Ducan, and G. Michaelson, Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK, EH14 4AS; emails: {R.Stewart, krd1, G.Michaelson}@hw.ac.uk; P. Garcia and A. Wallace, Engineering and Physical Sciences, Heriot-Watt University, Edinburgh, UK, EH14 4AS; emails: {P.Garcia, A.M.Wallace}@hw.ac.uk; D. Bhowmik, Department of Computing, Sheffield Hallam University, Sheffield, UK, S1 1WB; email: Deepayan.Bhowmik@shu.ac.uk.

ACM 1936-7406/2018/03-ART7 $15.00
https://doi.org/10.1145/3180481

Additional Key Words and Phrases: Domain specific languages, Parallel processing, Dataflow, FPGA, Hardware accelerators, Image processing, RIPL, Semantics, Cyclo static dataflow, High level synthesis, OpenCV

ACM Reference format:
Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. 2018. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 1, Article 7 (March 2018), 24 pages.
https://doi.org/10.1145/3180481

1 INTRODUCTION

Video analytics has witnessed a major growth in applications including surveillance, vehicle autonomy and driver assistance, marketing, entertainment, intelligent domiciles, and medical diagnosis. These domains need high performance, which can be achieved with parallel processing. Parallel image processing is most commonly performed on multicore CPUs and GPUs. This is because the data parallel nature of image processing algorithms maps well to multicore architectures, and because mature programming frameworks for them are widely adopted. Array-based image processing computations can be chunked into smaller single instruction, multiple data (SIMD) work items, then mapped across threads at each processor core and compiled to vectorized machine instructions at each core. The implementations of array/image processing languages for CPUs and GPUs (e.g., Chakravarty et al. (2011)) often store image data into memory close to processor cores, where the memories match complete images. Although compilers for these languages try to achieve good data locality, cache misses can cost idle clock cycles, which is critical because bandwidth between a processor and memory is often the most significant factor in determining overall performance (Wulf and McKee 1995).

As the volume of image data collected from sensors grows, there is a strong need to process the data close to the sensor, to considerably reduce the amount of data to be transferred between sensors or to perform real-time processing, and to maximize the lifetime of nonpowered energy sources.

Hence, two factors make FPGAs more suitable than CPUs and GPUs for real-time image processing:

(1) *Predictability.* Real-time image processing performance must be predictable—that is, it must not fluctuate below a frames per second (FPS) threshold, because that may compromise algorithmic robustness. Cache misses on fixed CPU/GPU architectures may result in dropped frames during off-chip accesses. FPGAs are not restricted by fixed architectural choices, so FPGA programs do not suffer from fetch-decode-execute instruction latencies or from cache misses.
(2) *Power.* CPUs and GPUs consume far more energy than FPGAs. CPUs and GPUs lag significantly behind FPGAs when comparing the performance per watt for different classes of applications (Brodtkorb et al. 2010; Thomas et al. 2009). For example, for an image processing sliding window benchmark in Fowers et al. (2012), the CPU required 130 watts, the GPU 145 watts, and the FPGA just 20 watts. This makes FPGAs most attractive for remote smart camera deployments (e.g., Bhowmik et al. 2017), where access to power may be scarce.

Hardware resource availability is often the bottleneck when using FPGAs. The CPU and GPU dynamic memory allocation approach for image storage is prohibitive for FPGA implementations, because limited on-chip memory is limited—up to 68Mb of on chip block RAM (BRAM) (Xilinx 2015).

The FPGA on a $1.7k Xilinx Kintex 7 can accommodate five 1024×768 image buffers, whereas the $475 Xilinx Zedboard cannot accommodate any (Stewart et al. 2016). Newer-generation FPGAs such as the UltraScale+ offer improved memory density thanks to UltraRAM technology (Ahmad et al. 2016), although still below the memory requirements of complex image processing pipelines. The global shared memory model does not scale when used with hardware pipelines on FPGAs, as each computation in a pipeline would have to contend for the bandwidth to off-chip frame buffers—only one access request can be performed in each clock cycle. New memory technologies such as Hybrid Memory Cube (HMC) (Jeddeloh and Keeth 2012) and high-bandwidth memory (HBM) (Marinissen and Zorian 2017) might alleviate this problem in the near future, thanks to much higher bandwidth, if integration technologies are developed to enable their use across programming environments. Regardless, efficient use of local memory remains a first-order design concern. In contrast to shared off-chip memory, on-chip block RAM (BRAM or UltraRAM) blocks are distributed across the FPGA fabric, and with the right programming model, separate computations can efficiently be assigned their own local contention-free image region buffers. This is the approach taken with the Rathlin image processing language (RIPL).

Hardware description languages (HDLs) such as Verilog (Thomas and Moorby 1996) are the common tool that FPGA engineers use to specify hardware circuits. Designers think about their implementation in terms of connecting IP blocks or, if IP blocks required by an algorithm do not exist, very low level building blocks such as gates, registers, and multiplexers. Software engineers are seldom familiar with HDL concepts such as clocks, control signals, and combinational/sequential logic, limiting the dissemination of FPGA technology in the software industry.

To improve programmer productivity and to reduce hardware implementation errors, high-level synthesis (HLS) supports a subset of C/C++. Compiling these languages to FPGAs often suffers from the drawback of being compiled to larger and slower FPGA configurations than those generated by a structural description (Compton and Hauck 2002). Knowledge of digital hardware is required for generating high-quality hardware from C/C++ with HLS (Schafer and Mahapatra 2014), such as using pragmas to partition data structures across memory blocks, and these vary across different tools (Nane et al. 2016). Moreover, the dynamic heap memory model in C/C++ cannot be used on FPGAs since there is no software operating system there to provide this automatic memory map to off-chip memory. Therefore, new high-level programming abstractions that generate efficient structural descriptions are needed to encourage the wider adoption of FPGAs.

This article makes the following contributions:

— A presentation of RIPL, a high-level FPGA language with a collection of image processing skeletons for elementwise operations, sliding 1D/2D stencils, image reduction operations, recursive algorithms, and random access into 2D images and *N* dimensional arrays (Section 2).
— A formal presentation of RIPL's dataflow intermediate representation (IR) and the FSM transitions that describe the data rates and static scheduling behavior of each skeleton (Section 3).
— A comparison of the expressiveness and performance of RIPL versus the Vivado HLS OpenCV library using four synthetic benchmarks (Section 4.1). We also assess the performance of RIPL with a visual saliency algorithm comprising stencils and a discrete wavelet transform (DWT), compared to a C++ equivalent with Vivado HLS (Section 4.2). The expressivity of RIPL is demonstrated with mean shift segmentation, which requires recursion and random access (Section 4.3).
— RIPL is validated with a Zynq-based FPGA smart camera architecture (Section 4.4).

Section 5 describes related image processing languages for FPGAs, and we conclude in Section 6.

Table 1. Skeleton Functionality

Data Access Pattern	Nonoverlapping Image Traversal	Overlapping Image Traversal
Point	*map, zipWith*	
1D/2D window		*stencil*
Image to image	*scale, scan, splitX, splitY*	
Image reduction	*fold*	
Random access	*fold*	

```
1   image2 = map image1 (\p -> min 255 (p + 50));
2
3   image3 = zipWith image1 image2 (\p1 p2 -> p1 + p2 / 2);
4
5   image4 = zipWith image1 [maxPixel..]
6       (\x maxPixel -> if x > (maxPixel-50) then pixel else 0);
7
8   biggerImage = scale (2,3) image1;
9
10  image5 = stencil (3,1) image1 (\[.] (x,y) -> ([.-1] + [.] + [.+1])/3);
11
12  image6 = stencil (3,3) image1
13     (\p1 p2 p3 p4 p5 p6 p7 p8 p9 (x,y) ->
14        abs ((p1 + (2*p2) + p3) - (p7 + (2*p8) + p9))
15        + abs ((p3 + (2*p6) + p9) - (p1 + (2*p4) + p7)));
```

Fig. 1. Examples of image processing with stream-based skeletons.

2 RIPL DESIGN

2.1 RIPL Overview

RIPL is a DSL for describing FPGA designs for image processing algorithms at a very high level. RIPL's design is inspired by stream-based functional programming languages (e.g., McGraw et al. (1985)) and libraries (e.g., Kiselyov (2012)), such as *map* and *zipWith* over streams. Such primitives are sometimes called *skeletons* (Cole 1991), and this is how we refer to RIPL's language primitives. RIPL skeletons capture many low- and medium-level image signal processing operations, such as 1D and 2D filters, image transformations, and global image reductions.

RIPL programs are nonterminating, repeatedly executed for every frame coming from a source. RIPL therefore must be able to process frames, ideally at the rate of capture. To achieve this, hardware pipelines are generated from RIPL programs to hide latency. Synthesizing intermediate data structures from high-level languages on FPGAs would quickly use up all available on-chip BRAM memory. Therefore, to maximize the use of BRAMs, RIPL skeletons are compiled to memory-efficient data-localized functions, such as pixel-to-pixel or region-to-region actor functions, rather than image-to-image actor functions, because the latter would incur high memory costs.

2.2 Image Processing With RIPL Skeletons

RIPL skeletons are reusable generalized image processing patterns, to which the user supplies functions and values. Their distinctions in terms of data access patterns are shown in Table 1, with examples given in Figure 1. Point operations, such as with *map*, compute a single pixel value from an input pixel, and the processing does not depend on pixel neighbors (line 1 in Figure 1). Examples include arithmetic, logical, and threshold operations. The *zipWith* skeleton (line 3) merges two data structures—for example, two images or combining an image with a stream of a constant value (line 5).

$$
\begin{aligned}
imread_{M,N} &: ColourType \rightarrow (M : Int) \rightarrow (N : Int) \rightarrow I_{(M,N)} \\
map_{M,N,A,B} &: I_{(M,N)} \rightarrow ([P]_A \rightarrow [P]_B) \rightarrow I_{(M*(B/A),N)} \\
stencil_{M,N,A,B} &: I_{(M,N)} \rightarrow (A : Int, B : Int) \rightarrow ([P]_{(A*B)} \rightarrow (Int, Int) \rightarrow P) \rightarrow I_{(M,N)} \\
zipWith_{M,N,A} &: I_{(M,N)} \rightarrow I_{(M,N)} \rightarrow ([P]_A \rightarrow [P]_A \rightarrow [P]_A) \rightarrow I_{(M,N)} \\
scale_{M,N,A,B} &: (A : Int, B : Int) \rightarrow I_{(M,N)} \rightarrow I_{(M*A,N*B)} \\
splitX_{M,N} &: Int \rightarrow I_{(M,N)} \rightarrow (I_{(M/2,N)}, I_{(M/2,N)}) \\
splitY_{M,N} &: Int \rightarrow I_{(M,N)} \rightarrow (I_{(M,N/2)}, I_{(M,N/2)}) \\
scan_{M,N} &: I_{(M,N)} \rightarrow Int \rightarrow (P \rightarrow Int \rightarrow Int) \rightarrow I_{(M,N)} \\
fold_{M,N} &: State \rightarrow I_{(M,N)} \rightarrow (State \rightarrow Element \rightarrow [Statement]) \rightarrow State \\
fold &: State \rightarrow Range \rightarrow (State \rightarrow Index \quad \rightarrow [Statement]) \rightarrow State
\end{aligned}
$$

Fig. 2. RIPL skeletons.

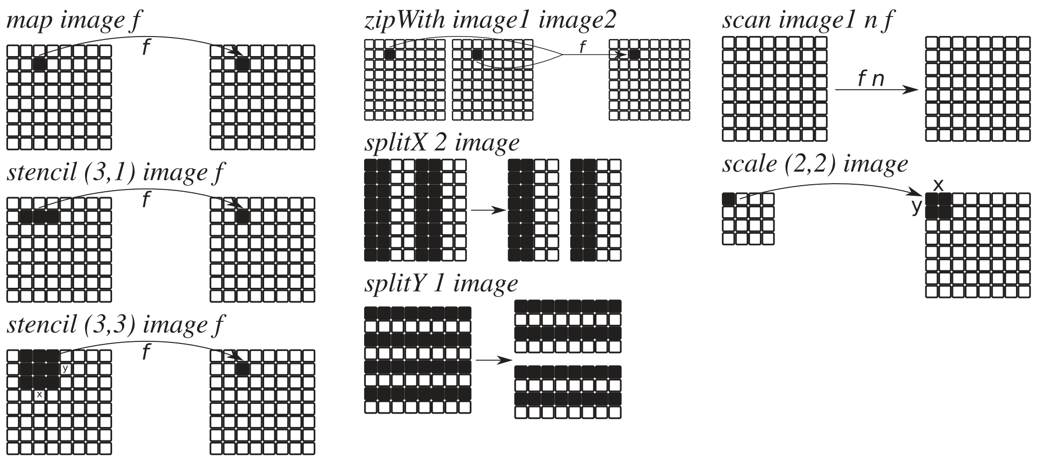


Fig. 3. Visualizing image processing with RIPL's stream-based skeletons.

Local operations, such as with *stencil*, depend on small 1D or 2D subregions of neighboring pixels (e.g., convolution, filtering, smoothing, image enhancements), upsampling an image with *scale* (line 8) with an interpolation filter with *stencil*, or horizontally splitting an image with *splitX*. For example, *splitX 2 img1* would return two images, where the first is constructed from columns 1, 2, 5, 6, 9, 10 . . ., and the second image has columns 3, 4, 7, 8 . . ., and so on.

RIPL's skeleton API is shown in Figure 2. The user supplies functions and values, using standard notation for function type signatures. For example, *map* is a skeleton that takes two arguments, an $M \times N$ image and function from a pixel to a pixel, and returns an $M \times N$ image. Images are read with *imread* in row major order from left to right, then top to bottom. The type P represents pixel integer values. An image $I_{(M,N)}$ is M pixels in width and N pixels in height. Processing images with RIPL's stream-based skeletons is visualized in Figure 3.

2.3 Image Stencils, Recursion, and Random Access

2.3.1 Image Stencils. RIPL has a primitive for applying 1D and 2D stencils called *stencil.* It captures a way to express many common image processing kernels, such as a blur kernel and edge detection. The *stencil* skeleton provides the user with the pixel values, and also the (x, y) position of the center pixel. Line 10 in Figure 1 is an implementation of a 1D blur stencil. For 1D stencils, the syntax *[.]* points to the indexed columnwise position of that pixel. Indexing pixels on either side is done with *+/-*. For example, *[.-1]* points to the pixel to the left of the *[.]* pixel. When applying a 2D kernel, such as Sobel edge detection on line 12, the *stencil (M,N) ..* skeleton provides $M * N$ pixels to the user-defined function. When the user-defined kernel is applied at the edges of the image, the closest adjacent pixel is mirrored over the image's edge. The (x, y) position argument can be used to make a choice about how to compute a kernel result for each position (e.g., applying different kernels for odd and even columns). This functionality is used for separating the low and high bands in a DWT in Section 4.2.3:

```
img2 = stencil (3,1) img1 (\[.] (x,y) ->
    if x % 2 == 0
    then ([.] - (([.-1] + [.+1]) >> 1))
    else ([.] + (([.-1] + [.+1]) >> 2)));
```

2.3.2 *Stateful Recursion.* The *fold* skeleton is designed to meet algorithmic requirements that do not fit into RIPL's stream combinator model. This skeleton supports the following:

—Accumulating state while traversing *N* dimensional data structures including images.
—Random access into *N* dimensional data structures.
—Recursion with the *while* construct.

The *fold* skeleton takes three arguments: (1) an initial state, (2) an iterable data structure, and (3) a user-defined function that takes a state and the next element from the structure, then executes a sequence of statements. The initial state can be a Boolean or integer literal, a tuple of literals, or *N* dimensional arrays created with *genarray*, where *genarray(50,10)* creates a 50 × 10 array. An iterable data structure can either be a previously computed value, such as an image region, or an iterable range with *range*, where *range(10,5)* provides the user-defined function *(i,j)* loop counters.

Two examples using *fold* are the following:

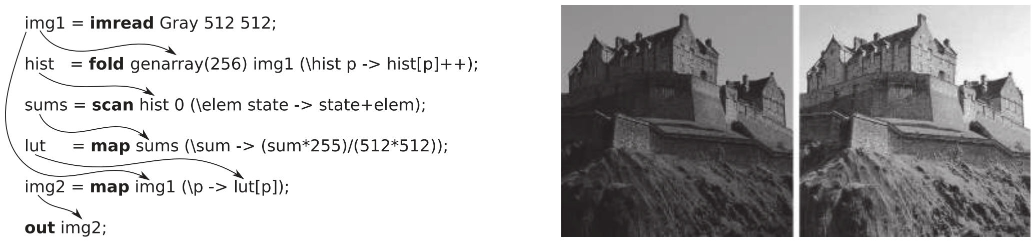
```
maxPixel  = fold 0 img1 (\maxP pixel -> maxP[0]=max maxP[0] pixel; );
histogram = fold genarray(256) image1 (\hist p -> hist[p]++; );
```

The first computes the maximum pixel in an image. The user-defined function that applies the *max* binary operator is folded over *image1*, starting from initial state 0. The second example instantiates a 1D array of size 256, to serve the purpose of a histogram data structure. The function folds over the *image1*, incrementing the bin value positions according to each grayscale pixel value.

2.4 A Dataflow Intermediary for RIPL

RIPL programs are compiled to dataflow process networks (DPNs) (Lee and Parks 2002) as an IR of image processing computations. A RIPL program is shown in Figure 4, which normalizes an image's grayscale distribution with its sum histogram. It reads image pixels with *imread* into a histogram with *fold*. A lookup table containing the scale factor is created with *scan* and *map*. An output image is created from this lookup table. The arrows in Figure 4(a) show the dataflow analysis that the RIPL compiler performs, to generate pipelined FPGA designs.

RIPL's dataflow IR comprises static and cyclostatic properties to support the required scheduling behaviors of the RIPL skeletons. Synchronous dataflow (SDF) (Lee and Messerschmitt 1987) actors produce and consume the same number of image pixels on every firing. Skeletons *map* and *zipWith* are compiled to SDF actors. All other skeletons are compiled to cyclostatic dataflow (CSDF) (Bilsen et al. 1996) actors. They have cyclically changing state transition sequences and have an internal actor state for pixel/line buffers to support 1D/2D *stencil*, *scale*, *scan*, *splitX*, *splitY*, and *fold*. The scheduling behavior of each CSDF actor for RIPL skeletons is fixed, and the pixel/line



```
img1 = imread Gray 512 512;
hist  = fold genarray(256) img1 (\hist p -> hist[p]++);
sums = scan hist 0 (\elem state -> state+elem);
lut    = map sums (\sum -> (sum*255)/(512*512));
img2 = map img1 (\p -> lut[p]);
out img2;
```

(a) Dataflow through RIPL code (b) Input (c) Output

Fig. 4. RIPL dataflow analysis of histogram normalization.

buffer sizes and the functions to produce data output values are derived from the user's RIPL code. The dataflow IR exploits FPGA parallelism and good data locality.

Parallelism. The dataflow model of independent computational blocks are a natural fit for parallel FPGA circuits, to process infinite image streams. Parallelism is implicit in RIPL, without parallel primitives or pragmas. Generating hardware pipelines hides latency. The RIPL compiler preserves lazy evaluation in the tail of image streams to exploit pipelined parallelism. Otherwise, actors would consume more data than needed to compute their function, which would reduce parallelism and increase memory costs for intermediate storage. There are two forms of automatic RIPL parallelism:

— *Temporal parallelism.* This form of parallelism is introduced when image data is streamed through a pipeline of producer-consumer kernels (e.g., the output from a *stencil* is the input to a *zipWith* operation).
— *Spatial parallelism.* RIPL produces computational code inside actors to exploit *spatial* parallel FPGA logic. Parallelism is exploited within expressions in user-defined functions, in the absence of dataflow dependencies within expressions.

Data locality. There is no global shared memory in the dataflow model. Instead, the dataflow memory model is that of isolated memory within actors, where computations and their data are co-located. This maps naturally to on-chip contention-free (i.e., parallel accesses) BRAM memory and registers distributed across FPGA fabric.

3 RIPL IMPLEMENTATION

Each RIPL skeleton is compiled to a dataflow actor that encapsulates a finite state machine (FSM), responsible for the required scheduling behavior, and a corresponding datapath. These actors are connected into deep parallel pipelines as multiple skeletons are used in larger programs. RIPL enforces single assignment semantics (i.e., a variable is assigned a value with a skeleton instance exactly once). This makes data dependencies easy to extract to dataflow wiring, to facilitate the generation of hardware pipelines. The RIPL compiler is open source,[1] and the RIPL, CAL, and C++ implementations for the evaluation in Section 4 are available as an open dataset (Stewart 2018).

3.1 Skeletons to Dataflow FSMs

3.1.1 Dataflow Semantics for RIPL Skeletons. The compilation of each RIPL skeleton is based on a framework for describing rule-based dataflow actors (Janneck 2003). It describes the data rates and static scheduling for each RIPL skeleton in terms of transitions on an abstract machine, which

[1] https://github.com/robstewart57/ripl.

is later compiled to hardware specified in Verilog (Section 3.2). An actor is a sequential process, communicating values by transitioning between states in the actor's FSM. The state machine receives tokens and reacts to them, possibly entering another state, and possibly producing tokens. A state transition from σ to σ' consumes a tuple s of token sequences and produces a tuple s' of token sequences. An actor has a set of m input ports, written S^m, and n output ports, written S^n. The value m is dictated by the arity of the corresponding RIPL skeleton. For example, *zipWith* takes two images as inputs, and hence m is 2 for this skeleton. If the output of one skeleton is used as an input argument of multiple other skeletons, then those actors are connected to the same input port and the stream is automatically duplicated into each connection.

Let U be the universe of all values, and let $S = U^*$ be the set of all finite sequences in U. A tuple $s \in S^m$ contains one or more token sequences consumed from p ports, where $1 \leqslant p \leqslant m$. We write s_A^p to describe a token sequence of length A at port p. For example, a transition from σ to σ' that consumes sequences from input ports 1 and 2 of lengths A and B, to produce a sequence of length C to an output port 1, is written as follows:

$$\sigma \xrightarrow{s_A^1 \times s_B^2 \mapsto s'^1_C} \sigma'$$

For some scheduling phases of a RIPL skeleton, a transition rule may not read from or write to any ports. We use $\emptyset$ as port descriptions for these transitions. To support the stateful RIPL skeletons such as *stencil*, an internal actor state is needed. We add $\mathcal{S}$ as a notation to transition rules for the internal actor state. State transition examples are written as follows:

$$\langle \sigma_0, \mathcal{S} \rangle \xrightarrow{s_3^1 \mapsto s'^1_6} \langle \sigma_1, \mathcal{S}' \rangle$$

This transition moves from FSM state σ_0 with a set of internal variable states in $\mathcal{S}$ to FSM state σ_1 with internal variable states $\mathcal{S}'$, and consumes three input tokens from input port 1 and emits six output tokens to output port 1. In the mapping from RIPL to dataflow graphs, the vertices (actors) represent image operations and the edges (wires) represent dataflow between composed skeletons.

3.1.2 Compilation Scheme.

Pointwise skeletons. The skeleton to dataflow FSM compilation scheme is shown in Figure 5. Each skeleton is compiled to an actor with one or more transitions. The *map* and *zipWith* skeletons are each compiled to a single transition rule in an SDF actor with no internal state—that is, transition rules *(map_stream)* and *(zipWith_stream)* map from an empty internal state {} to an empty internal state, and the output values from these transitions are determined by the user-defined function f.

Image transforms. The *scale* skeleton is compiled to three transition rules. The *(scale_pop_buffer)* consumes a row and outputs each pixel in turn by the width scale factor M. The *(scale_output_row)* outputs this scaled row $N - 1$ times, where N is the height scale factor, then the FSM is reset to σ_0 with transition *(scale_reset)*. The *scan* skeleton is compiled to three transition rules. The *(scan_init_state)* rule initializes the actor buffer with the user-defined initial state. The *(scan_output)* rule performs the accumulation operation, outputting intermediate values until a data structure is consumed, then *(scan_reset)* reset back to the initial state. The *splitX* skeleton has two rules, one each for outputting to output ports 1 and 2. Considering *splitX 2 image*, *(splitX_output1)* consumes two tokens and then outputs them to output port 1, then *(splitX_output2)* does the same, outputting to output port 2. This alternating schedule repeats forever. The same FSM schedule is used for *splitY*, this time consuming entire rows in each transition.

Stencils. A 1D image blur stencil in Figure 6 demonstrates how the RIPL compiler minimizes the memory requirements of an FPGA. The compiler infers the range of the indexed pixel either side

$$[\![map_{M,N,A,B}\ f]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{f\ :\ s^1_A \mapsto s'^1_B} \langle \sigma_0, \{\} \rangle \qquad \text{(map\_stream)}$$

$$[\![zipWith_{M,N,A}\ f]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{f\ :\ s^1_A \times s^2_A \mapsto s'^1_A} \langle \sigma_0, \{\} \rangle \qquad \text{(zipWith\_stream)}$$

$$[\![fold\ state\ f]\!] =$$
$$\langle \sigma_0, state \rangle \xrightarrow{f\ :\ s^1_1 \mapsto \emptyset} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(fold\_init\_state)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{f\ :\ s^1_1 \mapsto \emptyset} \left\langle \sigma_1, \mathcal{S}' \right\rangle \qquad \text{(fold\_accum)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{\emptyset \mapsto s'^1_1} \langle \sigma_2, \mathcal{S} \rangle \qquad \text{(fold\_output)}$$
$$\langle \sigma_2, \mathcal{S} \rangle \xrightarrow{\emptyset \mapsto \emptyset} \langle \sigma_0, state \rangle \qquad \text{(fold\_reset)}$$

$$[\![scan_{M,N}\ state\ f]\!] =$$
$$\langle \sigma_0, state \rangle \xrightarrow{f\ :\ s^1_1 \mapsto \emptyset} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(scan\_init\_state)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{f\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_1, \mathcal{S}' \right\rangle \qquad \text{(scan\_output)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{\emptyset \mapsto \emptyset} \langle \sigma_0, state \rangle \qquad \text{(scan\_reset)}$$

$$[\![splitX_{M,N,A}]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{s^1_A \mapsto s'^1_A} \langle \sigma_1, \{\} \rangle \qquad \text{(splitX\_output1)}$$
$$\langle \sigma_1, \{\} \rangle \xrightarrow{s^1_A \mapsto s'^2_A} \langle \sigma_0, \{\} \rangle \qquad \text{(splitX\_output2)}$$

$$[\![splitY_{M,N,A}]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{s^1_{A*M} \mapsto s'^1_{A*M}} \langle \sigma_1, \{\} \rangle \qquad \text{(splitX\_output1)}$$
$$\langle \sigma_1, \{\} \rangle \xrightarrow{s^1_{A*M} \mapsto s'^2_{A*M}} \langle \sigma_0, \{\} \rangle \qquad \text{(splitX\_output2)}$$

$$[\![stencil_{M,N,A}\ (A,1)\ f]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{s^1_{(A-1)} \mapsto \emptyset} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(stencil1D\_pop\_buffer)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{f\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_1, \mathcal{S}' \right\rangle \qquad \text{(stencil1D\_stream)}$$

$$[\![stencil_{M,N,A,B}\ (A,B)\ k]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{s^1_{((B-1)*M+A)} \mapsto \emptyset} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(stencil2D\_pop\_buffer)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_2, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_top\_left)}$$
$$\langle \sigma_2, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_{(M-2)} \mapsto s'^1_{(M-2)}} \left\langle \sigma_3, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_top\_row)}$$
$$\langle \sigma_3, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_4, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_top\_right)}$$
$$\langle \sigma_4, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_5, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_mid\_left)}$$
$$\langle \sigma_5, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_{(M-2)} \mapsto s'^1_{(M-2)}} \left\langle \sigma_6, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_mid)}$$
$$\langle \sigma_6, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_7, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_mid\_right)}$$
$$\langle \sigma_7, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \left\langle \sigma_8, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_bottom\_left)}$$
$$\langle \sigma_8, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_{(N-2)} \mapsto s'^1_{(M-2)}} \left\langle \sigma_9, \mathcal{S}' \right\rangle \qquad \text{(stencil2D\_bottom\_row)}$$
$$\langle \sigma_9, \mathcal{S} \rangle \xrightarrow{k\ :\ s^1_1 \mapsto s'^1_1} \langle \sigma_0, \{\} \rangle \qquad \text{(stencil2D\_bottom\_right)}$$

$$[\![scale_{M,N,A,B}]\!] =$$
$$\langle \sigma_0, \{\} \rangle \xrightarrow{s^1_A \mapsto s'^1_{M*A}} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(scale\_pop\_buffer)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{\emptyset \mapsto s'^1_{M*(B-1)}} \langle \sigma_1, \mathcal{S} \rangle \qquad \text{(scale\_output\_row)}$$
$$\langle \sigma_1, \mathcal{S} \rangle \xrightarrow{\emptyset \mapsto \emptyset} \langle \sigma_0, \{\} \rangle \qquad \text{(scale\_reset)}$$

Fig. 5. Compilation scheme from RIPL skeletons to actor FSMs.

```
stencil (3,1) image (\[.] (x,y) -> ([.-1] + [.] + [.+1]) / 3)
```

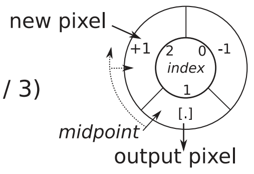


Fig. 6. Circular buffer to support *stencil (3,1)* .. for a 1D blur kernel.

of [.], which in this case is three pixels between [. − 1] and [. + 1]. The actor contains a three-element array that acts as a circular buffer, which is partially filled with two elements by firing rule *(stencil1D_pop_buffer)* to reach state σ_1 and a mid position pointer is set to 1. Each time the *(stencil1D_stream)* rule is fired, the next incoming token is pushed to the front of the buffer, and the RIPL function is computed on the buffer's contents as the output token value, and the mid position pointer is rotated forward one position. This stencil actor needs memory for just four pixels, the three-element circular buffer and the incoming token, to apply a 1D blur over an entire image.

The *stencil* skeleton for 1D stencils is compiled to two transition rules. The internal state for 1D stencils is populated with the *(stencil1D_pop_buffer)* rule before transitioning to stream-based rules *(stencil1D_stream)*. The *stencil* skeleton for 2D stencils is compiled to 10 transition rules. The *(stencil2D_populate_buffer)* rule consumes the first two rows and A pixels required to start processing an image from the top left corner—that is, by consuming $((B - 1) * M + A)$ tokens where

zipWith image1 image2 (λp1 p2 → (p1 + p2) / 2)

$$\begin{aligned}
\langle \sigma_0, \{\} \rangle &\xrightarrow{[3]\times[43]\mapsto[23]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}\\
\langle \sigma_0, \{\} \rangle &\xrightarrow{[4]\times[8]\mapsto[6]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}\\
\langle \sigma_0, \{\} \rangle &\xrightarrow{[16]\times[8]\mapsto[12]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}\\
\langle \sigma_0, \{\} \rangle &\xrightarrow{[62]\times[48]\mapsto[55]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}\\
\langle \sigma_0, \{\} \rangle &\xrightarrow{[30]\times[10]\mapsto[20]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}\\
\langle \sigma_0, \{\} \rangle &\xrightarrow{[12]\times[92]\mapsto[52]} \langle \sigma_0, \{\} \rangle && \text{(zipWith\_stream)}
\end{aligned}$$

fold 0 image1 (λmaxPixel pixel → maxPixel[0] = max maxPixel[0] nextPixel)

$$\begin{aligned}
\langle \sigma_0, 0 \rangle &\xrightarrow{[8]\mapsto\emptyset} \langle \sigma_1, 8 \rangle && \text{(fold\_init\_state)}\\
\langle \sigma_1, 8 \rangle &\xrightarrow{[5]\mapsto\emptyset} \langle \sigma_1, 8 \rangle && \text{(fold\_accum)}\\
\langle \sigma_1, 8 \rangle &\xrightarrow{[9]\mapsto\emptyset} \langle \sigma_1, 9 \rangle && \text{(fold\_accum)}\\
\langle \sigma_1, 9 \rangle &\xrightarrow{[16]\mapsto\emptyset} \langle \sigma_1, 16 \rangle && \text{(fold\_accum)}\\
\langle \sigma_1, 16 \rangle &\xrightarrow{\emptyset\mapsto[16]} \langle \sigma_2, 0 \rangle && \text{(fold\_output)}\\
\langle \sigma_2, 0 \rangle &\xrightarrow{\emptyset\mapsto\emptyset} \langle \sigma_0, 0 \rangle && \text{(fold\_reset)}
\end{aligned}$$

Fig. 7. RIPL skeleton transition execution examples.

A and B are the width and height of the kernel window and M is the width of the entire image. The remaining 9 rules carry out three tasks: (1) consume one pixel into the internal state, (2) compute the application of the user-defined 2D kernel function f on the window around the central pixel, and (3) output that computed value. They are defined as separate transition rules to index the internal state to handle boundary conditions at image edges, where the edge pixel is mirrored over the boundary.

Reductions and stateful recursion. The *fold* skeleton is compiled to four transition rules. The internal actor state is persisted throughout the consumption of each input data structure (e.g., an image region). The internal state buffer is initialized with the user-defined *state* value before being modified each firing with *(fold_accum)*. Once an entire data structure is consumed, the *(fold_output)* rule begins firing to output the final state. The *(fold_reset)* resets the internal actor buffer to the initial state and readies the actor to consume the next data structure.

3.1.3 Example Transitions. The execution of RIPL skeletons is now given showing sequences of FSM rule firings for two examples, shown in Figure 7. The first example combines two images with *zipWith*, with the mean of six pixels at the same position in each image. The second example uses *fold* to return the maximum pixel value in a 2×2 image region, by repeatedly folding the current highest pixel value through the image until all pixels have been consumed. The initial internal state takes the second argument to the skeleton, in this case 0. This is used to retain the largest current pixel value while consuming, then immediately discarding, each pixel from the image. Once all pixels have been consumed, the internal state is output with *(fold_output)* before the FSM is reset to $\langle \sigma_0, 0 \rangle$ for the next image frame.

3.2 Dataflow to FPGAs

The CAL dataflow language (Eker and Janneck 2003) is the target language of RIPL's dataflow IR backend. There are three phases to compile RIPL to FPGAs: (1) compiling RIPL to CAL, (2) compiling the dataflow graph to Verilog, and (3) synthesizing the Verilog to a bitfile. The dataflow intermediary provides optimization opportunities (e.g., Stewart et al. (2017)) before being compiled to hardware components such as registers, memories, and actor communication handshake protocols. Xronos (Bezati 2015), an FPGA backend plugin for the Orcc compiler, maps each actor into a language-independent model (LIM) and connects each LIM block to a clock. Xronos is designed to support dynamic asynchronous dataflow and therefore adds a FIFO-based data communication mechanism to support token passing. Decoupling actor communication via these FIFOs results in two clock cycles latency per pixel—that is, for a 512×512 image at 100MHz, the highest theoretical performance is $\frac{100000000}{512 \times 512 \times 2} = 191$ FPS for a single actor performing elementwise

```
hls::Mat<512,512, HLS_8UC1> img_0(rows,cols);
hls::Mat<512,512, HLS_8UC1> img_1(rows,cols);
hls::Mat<512,512, HLS_8UC1> img_2(rows,cols);
hls::Mat<512,512, HLS_8UC1> img_3(rows,cols);

// explicit depth for img_2 to prevent deadlock
#pragma HLS stream depth=262144 variable=img_2.data_stream

// convert AXI4 stream data to hls::mat format
hls::AXIvideo2Mat(INPUT_STREAM1, img_0);

// duplicate the img_0 stream
hls::Duplicate(img_0,img_1,img_2);

// find the maximum pixel of img_1 duplicate
int maxP, minP;
hls::Point p1,p2;
hls::MinMaxLoc(img_1,&minP,&maxP,p1,p2);

// threshold the img_2 duplicate using the max pixel - 50
int threshold = maxP - 50;
hls::Threshold(img_2,img_3,threshold,255,HLS_THRESH_TOZERO);
```

Fig. 8. Thresholding with a maximum pixel value in HLS OpenCV.

operations. The Xilinx OpenForge compiler allocates hardware: memory, arithmetic datapaths, registers, and actor interconnects, then generates HDL modules for each actor and for defining dataflow connectivity between these modules. This HDL code is synthesized and deployed to FPGAs using commercial tools.

RIPL's code generation strategy maximizes spatial parallelism to maximize the number of operations per clock cycle. By default, OpenForge uses BRAM blocks for CAL arrays beyond a certain size. Our experiments with OpenForge show that a BRAM block is instantiated for CAL arrays requiring more than 2KB of storage (e.g., about a 4×512 row buffer at 8 bits per pixel). To use a BRAM in the combinational setting, at most two read/write accesses to a CAL array may occur. When more than two reads/writes are performed on the same array in an actor's action, OpenForge instead instantiates LUTs for CAL arrays with three-plus reads/writes to preserve zero latency accesses.

4 EVALUATION

4.1 RIPL Versus HLS OpenCV

4.1.1 Expressivity. Vivado HLS OpenCV (Neuendorffer et al. 2015) is a collection of 34 predefined functions, a subset of the 2,500+ algorithm functions in OpenCV for CPUs. An example is shown in Figure 8. In contrast, RIPL provides algorithmic templates, whose functionality is entirely user defined. For example, the HLS OpenCV *hls::Max* function for combining two images can be expressed with RIPL's *zipWith*. HLS OpenCV's *hls::Filter2D* is a convolution filter. RIPL's *stencil* supports any stencil function, including convolution. This demonstrates the inflexibility of libraries of fixed kernels. Moreover, the library approach prohibits optimizations across library call boundaries. Other expressivity differences are the following:

— *Sharing*. Since the *Mat* type is just a synonym for *hls::stream* in the HLS OpenCV library, a *hls::Duplicate* call must be performed on an image to ensure that the underlying data stream is not consumed in one program location to deadlock in another location. RIPL supports automatic image sharing, where an image is processed in two places in a program.

Table 2. Microbenchmark Results

		RIPL				HLS OpenCV			
	Benchmark	DSP	BRAM	LUTs	FPS	DSP	BRAM	LUTs	FPS
1	Image brighten	0	0 (0%)	118 (0%)	191	0	0	880 (1%)	126
2	Sobel 2D edge detection	0	1 (0%)	12,273 (23%)	54	0	3 (1%)	1,675 (3%)	125
3	Threshold with max pixel	0	64 (45%)	280 (0%)	191	0	64 (45%)	9,172 (1%)	75
4	Histogram normalization	0	64 (45%)	799 (1%)	191	1	2 (1%)	7,265 (13%)	126

— *Image shape inference*. OpenCV programming requires explicit dimension in image declarations. For example, $hls :: Mat\langle 512, 512, HLS\_8UC1\rangle$ defines a 512×512 single channel image, using eight unsigned bits per pixel. In contrast, the RIPL compiler infers the dimension of every intermediate image, by following dimension transformations performed by skeletons through the implicit dataflow paths starting from *imread*, where dimensions are specified.

— *Parallel pipelines*. Pipelining in RIPL is automatic. For example, streaming the result of a *map* in a *stencil* will generate a hardware pipeline. In HLS OpenCV, the programmer must specify *#pragma HLS dataflow* above the function calls intended to be pipelined over the image stream.

4.1.2 Performance Comparison. We use four benchmarks to compare the space performance of RIPL and OpenCV compiled to FPGAs using Vivado HLS (Xilinx 2017b). Programs are compiled for the Xilinx Zedboard XC7Z020 for 512×512 single-channel images. The post place-and-route results in Table 2 are taken from Stewart et al. (2016). To quantify computational image processing performance, we use RTL simulation to calculate latency in clock cycles per 512×512 frame and report FPS according to the clock frequency. All experiments were performed using a 100MHz clock, which is the default frequency for Vivado HLS, and we did not perform any speed optimizations on Vivado HLS or RIPL-generated code.

RIPL achieves a higher FPS performance compared to HLS OpenCV, achieving 191 FPS for image brightening, max-thresholding, and histogram normalization. This is the maximum achievable FPS result for RIPL at 100MHz (i.e., $\frac{100000000}{512 \times 512 \times 2}$), due to FIFO-based token passing latency introduced by RIPL's FPGA backend. OpenCV achieves 126, 75, and 126 FPS, respectively. HLS OpenCV outperforms RIPL for a Sobel 2D edge detection, 125 FPS versus 54. This may be caused by the amount of runtime scheduling caused by the 10 dataflow transition rules in the implementation of *stencil* (Section 3.1.1) to handle mirroring pixels over boundaries. Reducing runtime scheduling by reducing the number of transition rules for *stencil* is future work. Resource utilization is similar, except for Sobel, where RIPL uses two BRAM blocks less and a lot more LUTs (23%). Because CAL code generated by RIPL implements a combinational filter (Section 3.2), accessing up to nine different data at the same time to compute Sobel, OpenForge is unable to map the data into BRAMs, which are limited to two simultaneous read/writes. Hence, data storage is mapped to LUTs.

HLS OpenCV uses less BRAM than RIPL for histogram normalization. The dataflow graph generated from RIPL computes two passes of the image: one to compute the histogram and a second to normalize the color distribution with it. In contrast, *hls::EqualizeHist()* normalizes a frame using a histogram computed for the previous frame. This approximation optimization means that BRAM is not needed to store an image for the second pass to normalize with the computed histogram.

4.2 Higher Level Case Study 1: Visual Saliency

This section uses a pyramidal visual saliency algorithm to compare the performance of RIPL versus variants of a C++ equivalent implementation compiled with Vivado HLS. The algorithm comprises three components: (1) an average filter (Section 4.2.1), (2) a DWT (Section 4.2.2), and (3) a weighted linear summation of wavelet subbands (Section 4.2.3).

4.2.1 Low-Level Image Processing: Average Filtering. Average filtering is one of the most commonly used filtering methods used in image preprocessing that helps reduce the amount of intensity variation between a pixel and its neighbors. Each pixel value is replaced with a mean of its neighbors including the pixel itself and commonly realized by convolving with a predefined mask (González and Woods 1992). An equation for such filters is expressed as follows:

$$I' = W * I = \sum_{n=1}^{N} \sum_{m=1}^{M} w(m,n) \cdot I(m,n), \tag{1}$$

where I is the input image of size $M \times N$, I' is the filtered image, W is the filter kernel, m and n are the pixel locations, and $*$ is a convolution operator. The process is repeated using a sliding window to receive filter output for the entire image. Average filtering with filter coefficients $w(m,n) = \frac{1}{9}$ can be implemented as a RIPL function using the *stencil* skeleton:

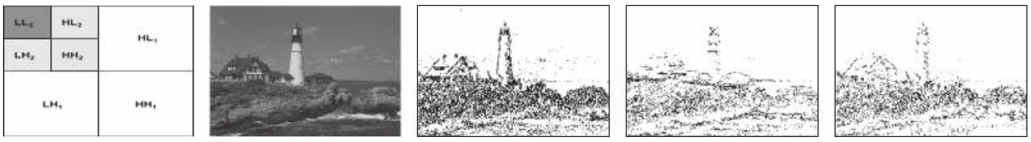
```
let averageFilter image =
  blurredImage = stencil (3,3) image
   (\p1 p2 p3 p4 p5 p6 p7 p8 p9 (x,y) -> (p1+p2+p3+p4+p5+p6+p7+p8+p9)/9);
  blurredImage;
```

4.2.2 Intermediate-Level Image Processing: Multiresolution 2D Wavelet Transform. DWT is used for many image processing applications, such as compression, denoising, and texture analysis. Multiorientation and multiresolution analysis using DWT closely resembles human vision. DWT decomposes an image into independent frequency subbands of multiple orientations at multiple scales demonstrating details and structures. Due to its popularity in the JPEG2000 image compression standard (Taubman and Marcellin 2012), we use the 5/3 wavelet kernel as our RIPL example, using a lower-complexity lifting-based approach. The filters are realized by decomposing the signal into lifting steps by factoring its polyphase matrix using the Euclidean factoring algorithm (Daubechies and Sweldens 1998). The input signal, lowpass subband signal, and highpass subband signal are denoted as $a[n]$, $s[n]$, and $d[n]$, respectively. DWT is expressed in Equation (2), where $s_0[n] \triangleq a[2n]$ and $d_0[n] \triangleq a[2n+1]$:

$$5/3 \begin{cases} d[n] &= d_0[n] - \frac{1}{2}(s_0[n+1] + s_0[n]), \\ s[n] &= s_0[n] + \frac{1}{4}(d[n] + d[n-1]). \end{cases} \tag{2}$$

A 2D DWT is computed by separately filtering rows and columns leading to one approximation (LL) subband and three detailed subbands in the decomposition level $i \in \mathbb{N}_1$ emphasizing vertical (LH_i), horizontal (HL_i), and diagonal (HH_i) contrasts within an image, portraying prominent edges in various orientations as shown in Figure 9. This process is repeated on the LL subband to get multiresolution decomposition. The 5/3 DWT is implemented as a RIPL function in Figure 10. The L and H components of the input *image* are computed with *stencil* on line 2 and then separated to L and H using *splitX* on line 5. The LL, LH, HL, and HH subcomponents are computed in the vertical direction in the same fashion on lines 6 and 8, and separated on lines 10 and 11.

4.2.3 High-Level Image Processing: Visual Saliency Modeling. The human visual system is sensitive to many salient features that lead to attention being drawn toward specific regions in a scene. A visual attention model identifies the salient regions in an image as perceived by human



(a) DWT illustration (b) LL_1 (c) LH_1 (d) HL_1 (e) HH_1

Fig. 9. Example of multiresolution wavelet decomposition, where (b), (c), (d), and (e) are level 1 approximation (LL_1), vertical (LH_1), horizontal (HL_1), and diagonal (HH_1) subbands, respectively.

```
let waveletDecompose image =
 img2 = stencil (3,1) image (\[.] (x,y) ->
   if x % 2 == 0 then ([.] - (([.-1] + [.+1]) >> 1))
                 else ([.] + (([.-1] + [.+1]) >> 2)));
 (L,H)  = splitX 1 img2;
 img3 = stencil (3,3) L (\p1 p2 p3 p4 p5 p6 p7 p8 p9 (x,y) ->
     if y % 2 == 0 then (p2 + p6) >> 2 else (p2 - p6) >> 1);
 img4 = stencil (3,3) H (\p1 p2 p3 p4 p5 p6 p7 p8 p9 (x,y) ->
     if y % 2 == 0 then (p2 + p6) >> 2 else (p2 - p6) >> 1);
 (LL,LH) = splitY 1 img3;
 (HL,HH) = splitY 1 img4;
 (LL,LH,HL,HH);
```

Fig. 10. 5/3 DWT in RIPL.

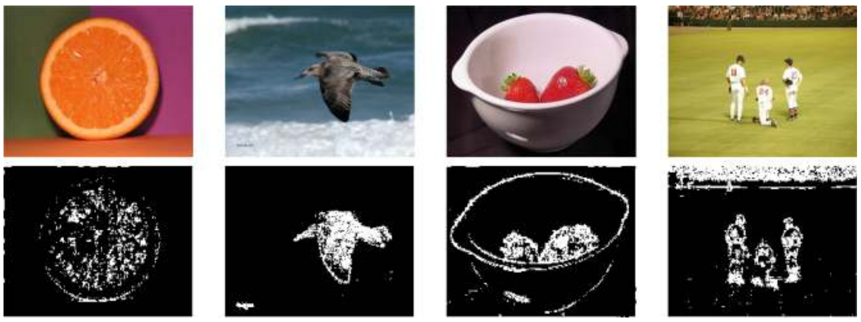


Fig. 11. Example saliency maps computed by the original model: row 1 shows original images, and row 2 shows the thresholded saliency maps generated by our implementation.

vision and has applications in many domains including computer vision (Borji and Itti 2013). We have adopted the visual attention model from Bhowmik et al. (2016), which demonstrates superior performances in joint saliency detection and low computational complexity. The algorithm uses the 5/3 DWT in Section 4.2.2 as its core decomposition. Example algorithm results are shown in Figure 11.

DWT captures horizontal, vertical, and diagonal contrasts within an image, respectively, portraying prominent edges in various orientations. A complete visual saliency RIPL implementation is shown in Figure 12, which uses the *waveletDecompose* RIPL function from Figure 10. It processes the *Y* channel of the *YUV* color spectral space, because this luminance channel exhibits prominent intensity variations and has significant structural information of the scene and carries maximum weight in the original algorithm. The RIPL implementation applies three levels of wavelet decomposition. Due to the dyadic nature of the multiresolution wavelet transform, the image resolutions are decreased after each wavelet decomposition iteration from lines 11 to 13. This is useful in cap-

```
 1  /* RIPL function that blurs a tile then scales it up */
 2  let blurResize region scaleFactor =
 3    blurred = averageFilter region; /* from Section 4.2.1 */
 4    resized = scale (scaleFactor,scaleFactor) blurred;
 5    resized;
 6
 7  /* The RIPL program starts here */
 8  image1 = imread Gray 512 512;
 9
10  /* 3 levels of decomposition using the RIPL function from Fig. 10 */
11  (LL1,LH1,HL1,HH1) = waveletDecompose image1;
12  (LL2,LH2,HL2,HH2) = waveletDecompose LL1;
13  (LL3,LH3,HL3,HH3) = waveletDecompose LL2;
14
15  /* blur and scale wavelet quartiles */
16  scale_LH1 = blurResize LH1 2;      scale_HL1 = blurResize HL1 2;
17  scale_HH1 = blurResize HH1 2;      scale_LH2 = blurResize LH2 4;
18  scale_HL2 = blurResize HL2 4;      scale_HH2 = blurResize HH2 4;
19  scale_LH3 = blurResize LH3 8;      scale_HL3 = blurResize HL3 8;
20  scale_HH3 = blurResize HH3 8;
21
22  mapVer = zipWith scale_LH1 scale_LH2 scale_LH3 (\x y z -> 4*x+8*y+4*z);
23  mapHor = zipWith scale_HL1 scale_HL2 scale_HL3 (\x y z -> 4*x+8*y+4*z);
24  mapDia = zipWith scale_HH1 scale_HH2 scale_HH3 (\x y z -> 4*x+8*y+4*z);
25  mapFinal = zipWith mapVer mapHor mapDia (\x y z -> x + y + z);
26  finalThresholded = map mapFinal (\x -> if x > 150 then x else 0);
27  out finalThresholded;
```

Fig. 12. Visual saliency in RIPL.

turing both short and long structural information at different scales. The *blurResize* function on line 2 is called on lines 16 through 20, which applies an average filter to each subband to remove unnecessary finer details, then scales up the result back to a full resolution output. The interpolated subband feature maps, lh_i (horizontal), hl_i (vertical), and hh_i (diagonal), $i \in \mathbb{N}_1$, for all decomposition levels L 1 to 3 are combined by a weighted linear summation as illustrated in Equation (3):

$$lh_{1\cdots L_Y} = \sum_{i=1}^{L} lh_i * \tau_i \qquad hl_{1\cdots L_Y} = \sum_{i=1}^{L} hl_i * \tau_i, \qquad hh_{1\cdots L_Y} = \sum_{i=1}^{L} hh_i * \tau_i\ , \tag{3}$$

where τ_i is the subband weighting parameter and $lh_{1\cdots L_Y}$, $hl_{1\cdots L_Y}$ and $hh_{1\cdots L_Y}$ are the subband feature maps for the spectral channel Y, on lines 22 through 24.

Last, S_Y, the saliency map for the Y spectral channel, is on line 25 and is computed as follows, before being thresholded with a binary threshold to emphasize salient regions:

$$S_Y = \overline{lh}_Y + \overline{hl}_Y + \overline{hh}_Y. \tag{4}$$

4.2.4 Visual Saliency Results.

RIPL versus C++ for hardware. The RIPL visual saliency implementation is compared to variants of an equivalent C++ implementation. The C++ implementation separates the three algorithmic components into C++ functions: the blur filter, the FWT, and the weighted linear summation. C++ templates are used to size array arguments to these functions, and these arrays are updated in-place in the body of each function. These template parameters are propagated through the program at compile time from 512 and 512 height and width image dimensions in the test bench—for example,

```
template<int h, int w> void blurFilter(uchar image[h][w]);
```

Table 3. Visual Saliency on a Virtex 7

Benchmark	FPS	Unroll Factor	BRAM (/2060)	DSP (/2800)	FF (/607200)	LUT (/303600)
RIPL	71	—	1%	0%	2%	18%
C++	21	—	90%	1%	1%	1%
C++ *with array reuse*	23	4	12%	1%	1%	1%
C++ *with array reuse*	24	8	12%	2%	1%	6%
C++ *with array reuse*	28	16	12%	4%	3%	12%
C++ *with array reuse*	28	32	12%	9%	6%	23%
C++ *with array reuse*	27	64	12%	18%	45%	60%
C++ *with array reuse*	25	96	12%	25%	94%	*(103%)*

The C++ variants are the following:

(1) *C++ with array reuse*. Each function contains declared 2D arrays with element values initialized to zero. For example, *lowBand[h][w/2]* and *highBand[h][w/2]* are declared in the *waveletDecompose* function, which can both be populated in a single loop statement to separate the low and high bands in one iteration over the image. This version contains 27 arrays.

(2) *C++ with extensive reuse of arrays*. This version reuses some arrays for different purposes. Refactoring has been done manually—for example, renaming *lowBand[h][w/2]* to *band[h][w/2]* and populating it with low band FWT values to compute LL and LH, before repopulating it with high band values to compute HL and HH. This approach also sequentializes the summation of weighted LH, HL, and HH contrasts because one array is repopulated with LH, then HL, then HH, after each sum accumulation into an output array. This version contains 13 arrays.

(3) *C++ with array reuse and loop unrolling*. Loop unrolling with *#pragma AP unroll* is employed to create multiple parallel independent operations from originally sequential loops. This loop unrolling pragma is used in eight places, and the unrolling factor is varied to measure the trade-off between space and latency.

Hardware designs for RIPL and C++ are clocked at 100MHz for the Virtex 7 VC707 and synthesized using Vivado 2016.2. Results are displayed in Table 3. Vivado HLS does not apply fusion optimizations to eliminate intermediate *for* loops over successive arrays, and hence the 90% BRAM use for 27 image buffers. Manually eliminating intermediate arrays down to 13 reduces BRAM use to 12%, which runs at 23 FPS. Unrolling the eight *for* loops in the C++ code up to an unroll factor of 32 increases FPS to 28, and increases DSP, FF, and LUT resource use. FPS performance degrades as more space is needed with a 64 unroll factor, and an unroll factor of 96 requires too many LUTs. The RIPL program achieves 2.5x higher FPS performance than the best C++ variant at 71 FPS, using fewer BRAMs and a similar number of LUTs.

RIPL versus dataflow in software. The RIPL program has also been compiled for an Intel i5 3.2GHz CPU, using the Orcc dataflow compiler's C backend. This CPU is a high-end processor compared to embedded processors usually used for remote image processing. The CPU visual saliency performance is 9 FPS. The FPGA performance is 8.3x faster than the CPU.

Code size. Visual saliency is 29 lines of RIPL code. The C++ visual saliency is 160 lines of code, and 162 with manual array reuse—that is, RIPL is 5x shorter than the C++ equivalents. The 29 lines of RIPL are compiled to 44 actors amounting to 3.1k lines of dataflow actor code generated by the

RIPL compiler. The high-level RIPL skeleton abstractions, and the ability to reuse RIPL functions, results in a program that is 111x shorter than its direct dataflow program equivalent. The Orcc compiler generates approximately 26k lines of C and 356k lines of Verilog. These program sizes are much larger because it combines both algorithm code and scheduling code to implement DPN actor firing semantics and actor synchronization on dataflow wires.

4.3 Higher-Level Case Study 2: Mean Shift Segmentation

The mean shift clustering algorithm (Fukunaga and Hostetler 1975) is a method to cluster data. This has been applied to text detection in images (Kim et al. 2003) and real-time video tracking of nonrigid objects (Comaniciu et al. 2000). Because the algorithm is used to distinguish between objects based on their color rather than shape, nonrigid objects can be tracked. The applications in image segmentation were proposed in Comaniciu and Meer (1999).

4.3.1 Mean Shift Segmentation Algorithm. Each pixel is mapped into an RGB color space, where each pixel has a vector position according to its red, green, and blue values. This feature space can be regarded as the probability density function of color. Dense regions of this space correspond to the local maxima of the probability density function. The value of a density function at a point is estimated from the values observed around that point. The multivariate kernel density estimator at the point $\mathbf{x}$ estimates the value using points within radius h of $\mathbf{x}$ and is given by

$$\hat{f}_{h,k}(\mathbf{x}) = \frac{c_k}{nh^d} \sum_{i=1}^{n} k\left(\left\| \frac{\mathbf{x} - \mathbf{x_i}}{h} \right\|^2 \right). \tag{5}$$

The density gradient estimator at a point can be estimated similarly from the values within a small window around the point. The modes of the density estimator are found at the points with zero gradient $\triangledown f(\mathbf{x}) = 0$. The mean shift vector given by

$$\mathbf{m}_{h,k}(\mathbf{x}) = \frac{1}{2} h^2 c \frac{\hat{\triangledown} f_{h,k}(\mathbf{x})}{\hat{f}_{h,-k'}(\mathbf{x})} = \frac{\sum_{i=1}^{n} \mathbf{x}_i k'(||\frac{\mathbf{x}-\mathbf{x_i}}{h}||^2)}{\sum_{i=1}^{n} k'(||\frac{\mathbf{x}-\mathbf{x_i}}{h}||^2)} - \mathbf{x} \tag{6}$$

is the difference between the weighted mean of points within the bandwidth parameter h and $\mathbf{x}$, the center of the window. It points in the direction of a normalized maximum density gradient estimate; it is zero at the point where the gradient of the probability density function is zero. The mean shift vector can therefore be used to define a path toward the local maximum of the estimated density for each point in the feature space. To ensure that mean shift clustering is applied only to neighboring image pixels of similar RGB colors, the mean shift vector is rewritten as follows:

$$\mathbf{m}_{h_r,h_s,G}(\mathbf{x}) = \frac{\sum_{i=1}^{n} \mathbf{x}_i^{r,s} k'(||\frac{\mathbf{x}^r-\mathbf{x}_i^r}{h_r}||^2) k'(||\frac{\mathbf{x}^s-\mathbf{x}_i^s}{h_s}||^2)}{\sum_{i=1}^{n} k'(||\frac{\mathbf{x}^r-\mathbf{x}_i^r}{h_r}||^2) k'(||\frac{\mathbf{x}^s-\mathbf{x}_i^s}{h_s}||^2)} - \mathbf{x}, \tag{7}$$

where $\mathbf{x}^{r,s}$ is the 5D position in the joint space, $\mathbf{x}^r$ is the 3D component of the joint space vector corresponding to the range, and $\mathbf{x}^s$ is the 2D component of the joint space vector corresponding to the position in the image. Using the Epanechnikov kernel, the required value,

$$-k'\left(\frac{x - x'}{h}\right) = \begin{cases} 1 & 0 \leq x - x' \leq h \\ 0 & x - x' > h, \end{cases} \tag{8}$$

determines that all points x' within a radius h of x are considered in the calculation, and all points outside are not. The algorithm is shown in Algorithm 1. For each point in the joint space:

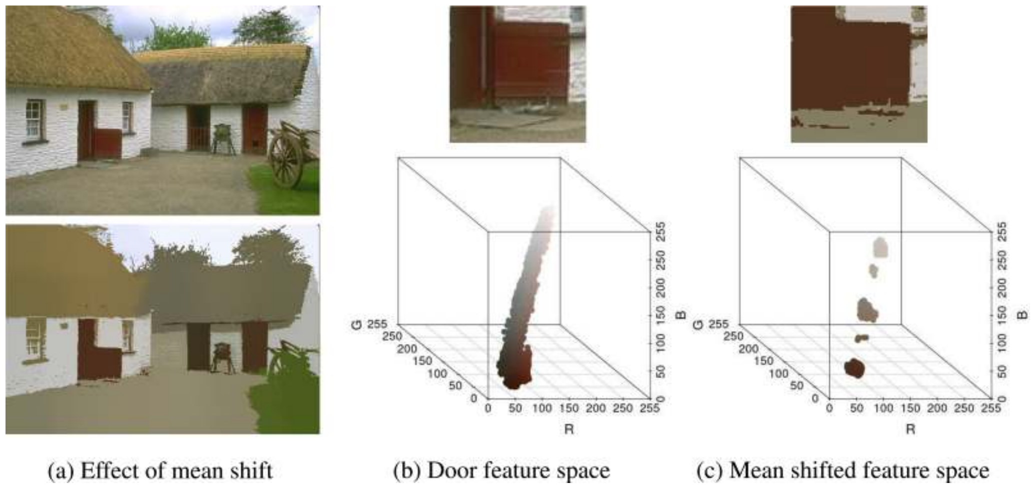


Fig. 13. Mean shift segmentation visualized.

(1) Define a spherical window of radius h_r around the three color dimensions of this point and a circular window of radius h_s around the spatial dimensions, and calculate the mean shift vector (Equation (7)) from this point.
(2) If the mean shift vector is nonzero, add it to the current position and go back to step 1.
(3) If the mean shift vector is zero at this point, define this point as the peak of the original point.

All points with the same peak are considered to belong to the same cluster.

ALGORITHM 1: Mean Shift Clustering Algorithm Pseudocode

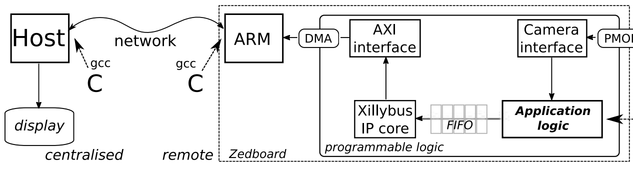
```
input: Image file
output: Image with each cluster shaded the color of its peak
for each index in joint space array do
    while peak has not been found & iteration limit not exceeded do
        Collect points within the spatial window
        Collect points within the range window
        Calculate mean shift vector using Equation (7)
        if mean shift vector is zero then
            Peak has been found;
        end
        Add mean shift vector to current point in joint space;
    end
    Store value of peak in equivalent index of output image;
end
```

Mean shift segmentation of image #385028 from the Berkeley segmentation training dataset (Martin et al. 2001) is shown in Figure 13. Figure 13(b) shows the feature space for the red door, and Figure 13(c) shows the segmented colors in the feature space after mean shift.

```
/* phase 1: map RGB image into a 5D array for RGB + feature space data */
img1 = imread RGB 512 512;
peaksRGB = fold genarray(512,512,5) range(512,512) (\(peaks) (i,j) ->

/* phase 2: the mean shift kernel over the RGB features space */
while ((count < recurseLimit) and (not peakFound)) {
  /* for each point within chosen window, find center of mass */
  for k in range((-1)*spatialWindow, spatialWindow) {
   for l in range((-1)*spatialWindow, spatialWindow) {
   /* if the point is within a circle of the center */
   if (l*l+k*k <= spatialWindow*spatialWindow){
    /* if the point is within the image */
    if ((k+peaks[i,j,3] < width) and (k+peaks[i,j,3] >= 0)
     and (l+peaks[i,j,4] < height) and (l+peaks[i,j,4] >= 0)) {
      /* if point is within RGB window */
      if ( (peaks[i,j,0]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,0])
         * (peaks[i,j,0]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,0])
         + (peaks[i,j,1]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,1])
         * (peaks[i,j,1]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,1])
         + (peaks[i,j,2]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,2])
         * (peaks[i,j,2]-img1[peaks[i,j,3]+k,peaks[i,j,4]+l,2])<=20*20) {
     /* update values of 5-vector */
     rVal += img1[peaks[i,j,3]+k,peaks[i,j,4]+l,0]-peaks[i,j,0];
     gVal += img1[peaks[i,j,3]+k,peaks[i,j,4]+l,1]-peaks[i,j,1];
     bVal += img1[peaks[i,j,3]+k,peaks[i,j,4]+l,2]-peaks[i,j,2];
     xVal += k;    yVal += l;     norm++;
   }}}}}

   if (norm != 0 ) { /* update value of each peak */
     peaks[i,j,0] += (rVal/norm);    peaks[i,j,3] += (xVal/norm);
     peaks[i,j,1] += (gVal/norm);    peaks[i,j,4] += (yVal/norm);
     peaks[i,j,2] += (bVal/norm);
   }
   /* check if current point is the peak */
   peakFound = (rVal==0) and (gVal==0) and (bVal==0);
count++; } );

/* phase 3: project the RGB values into a new image for output */
img2 = fold rgb(512,512) range(512,512) (\(image) (i,j) ->
                 image[i,j,0] = peaksRGB[i,j,0];
                 image[i,j,1] = peaksRGB[i,j,1];
                 image[i,j,2] = peaksRGB[i,j,2]; );
out img2;
```

Fig. 14. Mean shift segmentation in RIPL.

4.3.2 Mean Shift Segmentation in RIPL. The RIPL implementation is split into three phases, shown in Figure 14. The first phase reads an RGB image into the first three elements in the third dimension of a *peaksRGB* array. The fourth and fifth elements store the 2D image space coordinates. The second phase performs the mean shift kernel, which recurses with *while* either until mean shift has converged to a peak or until the recursion limit is reached. The *fold* over *range(512,512)* iterates *(i,j)* counters, and the kernel creates a window around this point. A center of mass is computed for each point in the window, and each peak value is updated. If the current point is the peak (*peakFound*), the *while* loop exits. The third phase projects the mean shifted RGB values from elements 1 to 3 of the third dimension of *peaksRGB* as output of the program.



(a) FPGA-based processing architecture

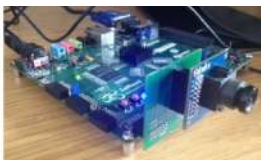

(b) RIPL on a Zedboard

Fig. 15. Deploying RIPL to image processing architectures.

4.3.3 Mean Shift Segmentation Results. Mean shift in RIPL is synthesized at 100MHz for the Virtex 7 VC707. The mean shift convergence limit is 5, the spatial window size is 10, the range window is 20, and test image size is 512×512. The RIPL is 70 lines of code. This compiles to 700 lines of dataflow IR code, which compiles to 427k lines of HDL code. The HDL uses 2% of available LUTs, 1% FFs, 1% DSPs, and 91% BRAMs. The BRAMs are used for the 3D array that stores the RGB values and the 2D image space coordinates. The RIPL mean shift program achieves approximately 7 FPS. This compares to our equivalent C++ that achieves 1.1 FPS for the mean shift kernel, or 0.7 FPS when factoring in image file IO, parallelized with OpenMP on 64 cores of an AMD Opteron 1.4GHz CPU.

4.4 Evaluation Platform: RIPL to an FPGA-Based Smart Camera

To evaluate RIPL on real-time architectures, the Xilinx Zedboard in Figure 15(b), a platform FPGA based around the Zynq-7020 chip, is used. Only the result of RIPL programs are transmitted over Ethernet or Wifi, rather than raw camera frames, avoiding pressure on network bandwidth and power needed for data transmission. Our experimental setup in Figure 15(a) integrates camera control and image acquisition hardware, interfacing an off-chip OmniVision OV7670 sensor, and a host processor interface. Xillybus[2] is used to abstract the interface between hardware and software (Andrews et al. 2004), ensuring that our FPGA subsystem is portable across Zynq platforms. The interface converts hardware FIFO channels to file descriptors in software, allowing software to use standard libraries to receive data from the FPGA for further processing, or to transmit RIPL results over a network connection. A complete description of the architecture is given in Bhowmik et al. (2017).

5 RELATED WORK

In addition to RIPL, two other high-level image processing FPGA languages have been developed in recent years: Darkroom (Hegarty et al. 2014) and Rigel (Hegarty et al. 2016). A comparison of their compilation to FPGAs is shown in Figure 16.

The expressivity of Darkroom is constrained to functions from (x, y) coordinates to pixel values (i.e., elementwise operations) and stencils that use pixels neighboring an (x, y) position. Darkroom is compiled to a line buffer IR; however, its line buffer IR to Verilog compiler is not publicly available. Darkroom is limited to processing one pixel per clock (Hegarty et al. 2016), a limitation overcome by its successor, Rigel. Rigel is a line buffer compiler IR language rather than a user facing language like RIPL or Darkroom. It presents a space/time trade-off to the user and overcomes a limitation of Darkroom by supporting the construction of image processing pyramids. Image pyramids are also supported by RIPL, as shown in the three-level DWT example in Section 4.2.2. Like RIPL, Rigel supports SDF dataflow semantics, with an extension that supports dynamic clock cycle latency and dynamic scheduling to check the validity of inputs. A contribution of our

[2] http://xillybus.com.

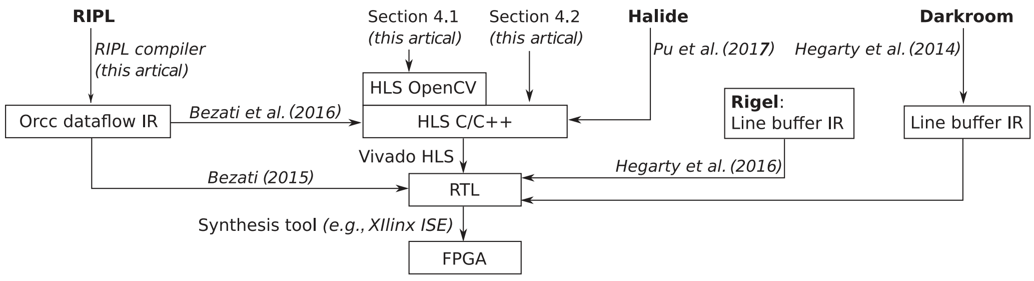


Fig. 16. Compilation of high-level FPGA image processing languages.

article in Section 3.1 is a formal account of the FSMs in the SDF and CSDF models that support RIPL's skeletons. Rigel supports data-dependent early kernel termination, which Hegarty et al. (2016) call *sparse computations*. RIPL also supports data-dependent latency from image processing kernels with the *while* construct.

A key difference between RIPL and Darkroom/Rigel is RIPL's support for multiple passes. For example, the histogram normalization RIPL program in Figure 4 folds over an image to compute an image histogram before normalizing pixels of the original image with that histogram. This would require two passes in Darkroom/Rigel, with a handwritten driver to combine the designs.

Another approach is FPGA support for the Halide (Pu et al. 2017) stencil language, with a backend targeting C that is synthesizable with Vivado HLS. The language separates computation from scheduling. To achieve FPGA parallelism, loops should be manually unrolled with Halide's *unroll* by increasing the degree of parallelism of the FPGA datapath. In contrast, parallelism is automatically derived from pipelines of RIPL skeletons, and data locality is achieved with RIPL's stream combinator programming style. To support FPGAs, the Halide programming model differs slightly from its software origins. The parallelism primitives (e.g., *tile* and *vectorize*) are not supported by the FPGA backend. As such, existing Halide code may have to be modified to run on FPGAs.

Darkroom, Rigel, and Halide are all restricted to expressing image stencil computations. RIPL supports elementwise stencils with stream combinators *map*, *zipWith*, and *scale*, and 1D/2D window stencils with the *stencil* skeleton. RIPL's *fold* skeleton adds expressivity needed for image reduction operations, and recursive algorithms with nonlocal access patterns, as demonstrated with mean shift segmentation in Section 4.3. The Rigel work (Hegarty et al. 2016) suggests that Rigel's higher-order *Reduce* module supports binary reduction only on windowed kernels residing in line buffers and not on entire images. RIPL's fold skeletons can reduce image regions and also entire images.

A different approach to FPGA design is the use of graphical interfaces, where IP blocks are configured and connected to generate the target system. Most notable is the Matlab/Simulink HDL Coder (MathWorks 2017), which is supported through vendor FPGA-specific technologies, namely Xilinx System Generator for DSP (Xilinx 2017a) and Altera DSP Builder (Altera 2017). This approach is used for rapid prototyping with fast development cycles, as IP blocks are ready to use. However, the black box nature of most IP blocks limits functionality customization, so algorithm changes are prohibited if the desired modified functionality does not exist in IP block toolboxes. In contrast, textual languages such as RIPL provide the opportunity for programmers to define custom functionality *within* generated IP blocks (e.g., as user-defined skeleton functions in RIPL).

6 CONCLUSION

This article presents RIPL, an image processing language for FPGAs. The language has high-level algorithmic skeleton primitives that capture image processing requirements including 1D/2D stencils, as well as random data access and recursion. The skeletons are abstractions that represent small programmable IP block templates that form building blocks for constructing higher-level algorithms. We show RIPL's expressivity for filters, a wavelet transform, and a pyramidal visual saliency algorithm in Section 4.2, and mean shift segmentation in Section 4.3. RIPL is more abstract than OpenCV for small microbenchmarks because parallelism, data types, FIFO depths, and data copying are inferred. Despite this, RIPL outperforms in three of four benchmarks. RIPL also outperforms a C++ equivalent for visual saliency compiled with Vivado HLS.

There are many compilation routes for high-level FPGA languages (Section 5), such as compiling generic dataflow or domain-specific IRs, and direct routes to RTL. This presents a trade-off between compile-time reasoning about throughput and fine-grain pixel processing pipelines, and real-world expressivity. RIPL's generic dataflow IR underpinnings allow it to support random global data access, recursion, and automatic parallelism, in addition to standard image stencil pipelines. Lowering image processing information from RIPL into its IR will enable fine-grain scheduling (e.g., adapting line buffer pipelines as in Rigel's IR). Likewise, adding recursion, dataflow feedback, and global data access to Rigel's IR, likely at the cost of compile-time pipeline scheduling, will expand its expressivity for complex vision algorithms. Tying flexible dataflow semantics (for expressivity) with image processing IRs (for optimization) could enable user-driven rewrite systems (e.g., Jones et al. (2001)) to expose space versus time FPGA trade-offs guided by acceptable domain-specific approximations.

REFERENCES

S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig. 2016. A 16-nm multiprocessing system-on-chip field-programmable gate array platform. *IEEE Micro* 36, 2, 48–62.

Altera. 2017. DSP Builder for Intel FPGAs. Retrieved February 4, 2018, from https://www.altera.com/products/design-software/model---simulation/dsp-builder/overview.html.

David L. Andrews, Douglas Niehaus, Razali Jidin, Michael Finley, Wesley Peck, Michael Frisbie, Jorge L. Ortiz, Ed Komp, and Peter J. Ashenden. 2004. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro* 24, 4, 42–53.

Endri Bezati. 2015. *High-Level Synthesis of Dataflow Programs for Heterogeneous Platforms: Design Flow Tools and Design Space Exploration.* Ph.D. Dissertation. School of Engineering, Ecole Polytechnique Federale de Lausanne, Switzerland.

Endri Bezati, Simone Casale Brunet, Marco Mattavelli, and Jörn W. Janneck. 2016. High-level synthesis of dynamic dataflow programs on heterogeneous MPSoC platforms. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'16).* IEEE, Los Alamitos, CA, 227–234.

Deepayan Bhowmik, Paulo Garcia, Andrew M. Wallace, Robert J. Stewart, and Greg Michaelson. 2017. Power efficient dataflow design for a heterogeneous smart camera architecture. In *Proceedings of the 2017 Conference on Design and Architectures for Signal and Image Processing (DASIP'17).* IEEE, Los Alamitos, CA, 1–6.

Deepayan Bhowmik, Matthew Oakes, and Charith Abhayaratne. 2016. Visual attention-based image watermarking. *IEEE Access* 4, 8002–8018.

G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. 1996. Cycle-static dataflow. *IEEE Transactions on Signal Processing* 44, 2, 397–408.

Ali Borji and Laurent Itti. 2013. State-of-the-art in visual attention modeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1, 185–207.

André Rigland Brodtkorb, Christopher Dyken, Trond Runar Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. 2010. State-of-the-art in heterogeneous computing. *Scientific Programming* 18, 1, 1–33.

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming (DAMP'11).* ACM, New York, NY, 3–14.

Murray Cole. 1991. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA.

Dorin Comaniciu and Peter Meer. 1999. Mean shift analysis and applications. In *Proceedings of the 7th IEEE International Conference on Computer Vision.* IEEE, Los Alamitos, CA, 1197–1203.

Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. 2000. Real-time tracking of non-rigid objects using mean shift. In *Proceedings of the 2000 Conference on Computer Vision and Pattern Recognition (CVPR'00).* IEEE, Los Alamitos, CA, 2142.

Katherine Compton and Scott Hauck. 2002. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys* 34, 2, 171–210.

I. Daubechies and W. Sweldens. 1998. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications* 4, 3, 245–267.

Johan Eker and Jorn W. Janneck. 2003. *CAL Language Report Specification of the CAL Actor Language*. Technical Report UCB/ERL M03/48. EECS Department, University of California, Berkeley.

Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays (FPGA'12)*. ACM, New York, NY, 47–56.

Keinosuke Fukunaga and Larry Hostetler. 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory* 21, 1, 32–40.

Rafael C. González and Richard E. Woods. 1992. *Digital Image Processing*. Addison-Wesley, Reading, MA.

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics* 33, 4, 144:1–144:11.

James Hegarty, Ross Daly, Zachary DeVito, Mark Horowitz, Pat Hanrahan, and Jonathan Ragan-Kelley. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics* 35, 4, 85:1–85:11.

Jörn W. Janneck. 2003. Actors and their composition. *Formal Aspects of Computing* 15, 4, 349–369.

J. Jeddeloh and B. Keeth. 2012. Hybrid Memory Cube new DRAM architecture increases density and performance. In *Proceedings of the 2012 Symposium on VLSI Technology (VLSIT'12)*. IEEE, Los Alamitos, CA, 87–88.

S. Peyton Jones, A. Tolmach, and T. Hoare. 2001. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Proceedings of the ACM SIGPLAN Haskell Workshop*. ACM, New York, NY, 203–233.

Kwang In Kim, Keechul Jung, and Jin Hyung Kim. 2003. Texture-based approach for text detection in images using support vector machines and continuously adaptive mean shift algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25, 12, 1631–1639.

Oleg Kiselyov. 2012. Iteratees. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS'12)*. 166–181.

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow: Describing signal processing algorithm for parallel computation. In *Proceedings of the 32nd IEEE Computer Society International Conference (COMPCON'87)*. IEEE, Los Alamitos, CA, 310–315.

Edward A. Lee and Thomas M. Parks. 2002. Dataflow process networks. In *Readings in Hardware/Software Co-Design*, G. De Micheli, R. Ernst, and W. Wolf (Eds.). Kluwer Academic Publishers, Norwell, MA, 59–85.

Erik Jan Marinissen and Yervant Zorian. 2017. Guest editors introduction: Design and test of a high-volume 3-D stacked graphics processor with high-bandwidth memory. *IEEE Design and Test* 34, 1, 6–7.

David R. Martin, Charless C. Fowlkes, Doron Tal, and Jitendra Malik. 2001. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings of the 8th IEEE International Conference on Computer Vision (ICCV'01)*. IEEE, Los Alamitos, CA, 416–425. DOI: http://dx.doi.org/10.1109/ICCV.2001.937655

MathWorks. 2017. FPGA Design and SoC Codesign. Retrieved February 4, 2018, from https://uk.mathworks.com/solutions/fpga-design.html.

J. McGraw, S. Skedzielewski, S. Allan, Oldehoeft Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. 1985. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence-Livermore-National-Laboratory, Livermore, CA.

R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10, 1591–1604.

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 Transactions on Architecture and Code Optimization* 14, 3, 26:1–26:25.

B. C. Schafer and A. Mahapatra. 2014. S2CBench: Synthesizable SystemC benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters* 6, 3, 53–56.

Stephen Neuendorffer, Thomas Li, and Devin Wang. 2015. *Accelerating OpenCV Applications With Zynq-7000 All Programmable SoC Using Vivado HLS Video Libraries (v3.0)*. Technical Report. Xilinx. https://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf.

Robert Stewart. 2018. Open dataset for *"RIPL: A Parallel Image Processing Language for FPGAs." ACM Transactions on Reconfigurable Technology and Systems*. Forthcoming. DOI: http://dx.doi.org/10.17861/ca09418a-cbc2-4d28-98a1-746267a26f9d

Robert Stewart, Greg J. Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. 2016. A dataflow IR for memory efficient RIPL compilation to FPGAs. In *Algorithms and Architectures for Parallel Processing*. Lecture Notes in Computer Science, Vol. 1194. Springer, 174–188.

Robert J. Stewart, Deepayan Bhowmik, Andrew M. Wallace, and Greg Michaelson. 2017. Profile guided dataflow transformation for FPGAs and CPUs. *Signal Processing Systems* 87, 1, 3–20.

David Taubman and Michael Marcellin. 2012. *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Vol. 642. Springer Science & Business Media, Berlin, Germany.

David B. Thomas, Lee W. Howes, and Wayne Luk. 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA 17th International Symposium on Field Programmable Gate Arrays (FPGA'09)*. ACM, New York, NY, 63–72.

Donald E. Thomas and Philip Moorby. 1996. *The Verilog Hardware Description Language (3rd ed.)*. Kluwer, Boston, MA.

William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1, 20–24.

Xilinx. 2015. *7 Series FPGAs Overview, DS180 (v1.17) Product Specification*. Technical Report. Xilinx.

Xilinx. 2017a. System Generator for DSP. Retrieved February 4, 2018, from https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html.

Xilinx. 2017b. Vivado High-Level Synthesis. Retrieved February 4, 2018, from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

Received February 2017; revised November 2017; accepted December 2017