A Dataflow IR for Memory Efficient RIPL Compilation to FPGAs

Robert Stewart[1(✉)], Greg Michaelson[1], Deepayan Bhowmik[2], Paulo Garcia[2], and Andy Wallace[2]

[1] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK
R.Stewart@hw.ac.uk
[2] School of Engineering and Physical Sciences, Heriot-Watt University, Edinburgh, UK

Abstract. Field programmable gate arrays (FPGAs) are fundamentally different to fixed processors architectures because their memory hierarchies can be tailored to the needs of an algorithm. FPGA compilers for high level languages are not hindered by fixed memory hierarchies. The constraint when compiling to FPGAs is the availability of resources.

In this paper we describe how the dataflow intermediary of our declarative FPGA image processing DSL called RIPL (Rathlin Image Processing Language) enables us to constrain memory. We use five benchmarks to demonstrate that memory use with RIPL is comparable to the Vivado HLS OpenCV library without the need for language pragmas to guide hardware synthesis. The benchmarks also show that RIPL is more expressive than the Darkroom FPGA image processing language.

Keywords: Domain specific languages · FPGAs · Data locality

1 Introduction

1.1 Memory Costs of High Level FPGA Languages

General Purpose Languages. Programming with C++ for FPGAs often relies heavily on the programmer's use of language pragmas to control how data structures should be implemented in hardware. For example when using Xilinx Vivado HLS [13], if a 3×3 window for applying a 2D filter is needed, the programmer must use an array partition pragma to partition the 3×3 pixel window array into individual scalar elements, to avoid its implementation using BRAM.

Image Processing Languages and Libraries. Domain specific languages (DSLs) offer potential for clearer syntax, stronger semantic checks, type-system-based guarantees and compiler optimisation for improved code execution. Compared to compiling C/C++ with HLS tools, DSLs can capture domain knowledge to abstract hardware templates that encapsulate common data access patterns that can more easily be analysed, *e.g.* for FIFO depth and bitwidth requirements.

J. Carretero et al. (Eds.): ICA3PP 2016 Workshops, LNCS 10049, pp. 174–188, 2016.
DOI: 10.1007/978-3-319-49956-7_14

A DSL may be an existing collection of language primitives ported to FPGAs, *e.g.* the Vivado HLS support [10] for a subset of the OpenCV [2] library. OpenCV C++ library code is not synthesisable directly, instead OpenCV function calls in existing software code must be replaced with corresponding function calls from the HLS library. In this restricted setting, it is not possible to use dynamic memory allocation *e.g.* in the construction of image whose dimensions are decided at runtime. For good performance using this library, the programmer must use explicit pragmas to guide hardware generation.

Alternatively, DSLs may be embedded within the programming model of an existing language, *e.g.* the Darkroom [5] FPGA image processing DSL is embedded within Terra [4]. Darkroom is compiled to line-buffered pipelines, with all intermediate values in local line-buffer storage. Images at each stage of computation are specified as pure functions from 2D coordinates to the values at those coordinates.

Our RIPL DSL for FPGAs is implemented as a standalone language, *i.e.* it has its own syntax and its stream processing based programming model is not hindered by a programming model of any general purpose host language. The memory performance and expressivity of HLS OpenCV, Darkroom and RIPL is compared in Sect. 5.

1.2 Data Locality

Fixed memory architectures comprise very fast cache access, off chip DDR memory access, or slow disk storage. Each application must fit into a fixed memory architecture representing a single large hierarchical memory space. A common approach is to build data locality aware compilers [11], *e.g.* locality aware scheduling of OpenMP tasks on multicore CPUs [9] and mapping nested access patterns on GPUs [8]. Minimising cache misses involves profiling cache traces, moreover trading function inlining with executable size, and managing memory pressure. Minimising memory requirements is a particular problem for close to sensor real-time image processing on FPGAs, where hard choices must be made in trading off memory and processing.

1.3 FPGA Memory

An important FPGA language implementation choice is whether on chip or off chip memory should be used to store data structures. Utilising off chip memory is sometimes unavoidable depending on the data transforms an algorithm requires, *e.g.* transposing or rotating an image, both of which require an image frame buffer. However, frequently using off chip memory from different parts of FPGA circuits does not scale, because only one memory read from an on chip circuit can be performed in each clock cycle. This can sequentialise execution and hence hurt throughput performance. Moreover, off chip memory access can take multiple clock cycles compared to latency-free LUT RAM or one cycle to access BRAM. On chip memory provides contention free local buffer access for different parts of the application specific circuit, because it is distributed across the FPGA's fabric.

Compilers of high level real-time languages should therefore prioritise wholly on chip memory implementations. However, the scarcity of BRAM introduces its own set of constraints for programming language designers to consider.

Memory layout on FPGA chips is fundamentally different to fixed processor architectures. Instead of compiling a program to map efficiently to fixed memory hierarchies, synthesis of high level languages builds a custom memory architecture on chip tailored for the needs of an algorithm. The constraint when compiling high level languages to FPGAs is the available resources, *e.g.* on chip memory ranges from 4 Mb to 68 Mb. The challenge for HLS compilers is therefore to minimise memory use from algorithms expressed with high level software languages. Synthesis tools can choose to implement memory using registers, lookup tables (LUTs), or block RAM (BRAM). Unlike cache contention issues on multicore CPUs, there is no contention to access BRAM memory because it is distributed across the fabric of an FPGA.

2 FPGA Memory Constraints

2.1 Image Buffer Capacity

The main FPGA resource for implementing memory is BRAM blocks. For example, the Z-7020 chip on the Zedboard has 140 36 Kb BRAM blocks amounting to 4.4 Mb. The XC7K480T chip on the Kintex-7 board has 1,910 18 Kb blocks and 995 36 Kb blocks amounting to 34 Mb. The XC7VX1140T chip on the Virtex-7 board has 3,760 18 Kb blocks and 1,880 36 Kb blocks amounting to 68 Mb. A single channel image pixel is 8 bits, or 1 byte. A 320×240 image with a single colour channel is 76,800 bytes, a 512×512 image is 262,144 bytes, a 1024×768 image is 786,432 bytes, and a 1920×1080 image is 2,073,600 bytes.

Storing entire image frame buffers on FPGAs does not scale. The cost of buffering entire image frames on chip is shown in Fig. 1. The Zedboard can store up to seven 320×240 frame buffers and just two 512×512 frame buffers. The Kintex-7 can store up to five 1024×768 buffers and two 1920×1080 buffers, and the Virtex-7 is able to store four 1920×1080 buffers. Localised pixel, row and region buffers should instead be generated by high level language compilers.

2.2 Eliminating Intermediate Buffers with Compiler Optimisation

When compiling high level programs to FPGAs, it is important to eliminate intermediate data structures because on chip BRAM is a scarce resource. A motivating example is shown in Fig. 2. This C++ code applies a Sobel edge detection filter with a nested *for* loop, and then brightens the result with another nested *for* loop. It uses the OpenCV *Mat* class for two intermediate images, *image2* and *image3*. The *xGradient()* and *yGradient()* functions are omitted for brevity. Whilst these intermediate image structures could be offloaded to off chip DDR memory, this would result in a latency of multiple clock cycles for every memory access, compared to a single cycle for on chip access. There is a need for

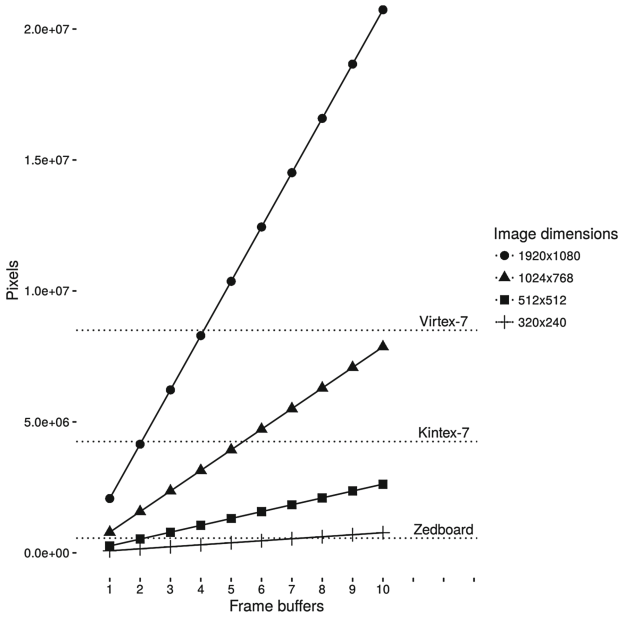


Fig. 1. Storing multiple frame buffers with on chip FPGA Memory

```
Mat image2;
/* Sobel filter */
for(int y = 1; y < image1.rows - 1; y++) {
  for(int x = 1; x < image1.cols - 1; x++) {
     gx = xGradient(image1, x, y);
     gy = yGradient(image1, x, y);
     sum = abs(gx) + abs(gy);
     image2.at<uchar>(y,x) = sum;
  }
}

Mat image3;
/* brighten image2 */
for(int y = 1; y < image2.rows - 1; y++) {
  for(int x = 1; x < image2.cols - 1; x++) {
     newPixel = image2.at<uchar>(y, x) + 50;
     image3.at<uchar>(y,x) = newPixel > 255 ? 255 : newPixel;
  }
}
```

Fig. 2. Intermediate images using OpenCV's *Mat* class

FPGA language compilers to avoid wasteful memory resources on intermediate images *image2* and *image3*.

One data locality approach in data parallel language compilers is to start from an imperative language with loops, and fuse the successive loops over the input *image1* into an expression tree in a single loop, to improve cache locality and on chip register locality *e.g.* [6,12]. For CPU or GPU scheduling, this expression tree can be duplicated to apply the same fused computation on image chunks in

a data parallel fashion. However for pipelined FPGA scheduling, where different computations are applied to separate regions of an image stream, a compiler would apply loop fusion optimisations, and then expression pipelining in the body of those loops to create hardware pipelines of fine grained operator dataflow graphs.

3 RIPL: An FPGA DSL for Maximising Data Locality

We take a different approach with RIPL. The language design is inspired by streaming libraries *e.g.* [7], which provides stream combinators like *map*, *fold* and *sum*. Composition of these RIPL primitives is a natural way of expressing pipelines of low and medium level image processing kernels. These pipelines are preserved during compilation and mapped into hardware as concurrent circuits.

RIPL has a declarative non-terminating programming model that is constrained for processing infinite image stream, a programming model from which minimal memory costs can more easily be extracted. It represents a high programming abstraction when compared to direct hardware design with HDLs. We term RIPLs stream combinator primitives as *algorithmic skeletons* [3]. They capture the common pattern of many low and medium level image signal processing operations such as 1 dimensional (1D) and 2D filters, combining images, and global operations such as finding the maximum pixel value. Intermediate images in RIPL programs are transformed to streams that are shared through parallel hardware pipelines, rather than copying whole images for each pipeline phase to process. The RIPL implementation is available online[1].

3.1 RIPL Skeletons

RIPL skeletons abstract common data access patterns, to which the user supplies functions and values. They have been designed such that dataflow analysis can be performed on their composition, and to extract the minimal memory requirements of their use. The RIPL program in Fig. 3 broadly corresponds to the OpenCV C++ in Fig. 2, though RIPLs *convolve* and *filter2D* skeletons also mirrors edge pixels over image boundaries to apply the kernel function to edge pixels. When compiled to hardware, the image stream *image1* is incrementally processed, first by hardware logic that computes Sobel edge detection and then by logic that brightens each pixel in the stream.

Skeleton API. The RIPL skeletons are shown in Fig. 4, using a standard notation for function type signatures, *e.g. map* is a skeleton that takes two arguments: an $M \times N$ image, and function from a vector of A pixels to a vector of B pixels. It returns an $M \times N$ image. Each skeleton is repeatedly applied over an image stream. Types in Fig. 4 are annotated with pixel major order, vector lengths and image dimensions. For example, $[P]_A$ is a vector of pixels of length A, so an argument in a function of the form $\lambda[a, b]$ has an implicit type $[P]_2$.

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

```
image1 = imread 512 512;
/* Sobel filter */
image2 = filter2D image1 (3,3)
   (\p1 p2 p3 p4 p5 p6 p7 p8 p9 ->
      abs ((p1 + (2*p2) + p3) - (p7 + (2*p8) + p9))
      + abs ((p3 + (2*p6) + p9) - (p1 + (2*p4) + p7)));

/* brighten image2 */
image3 = map image2 (\[pixel] -> [min 255 (pixel + 50)]);
```

Fig. 3. RIPL equivalent of the OpenCV C++ in Fig. 2

$$
\begin{aligned}
imread_{M,N} &: (M : Int) \to (N : Int) \to I^{R}_{(M,N)} \\
map_{M,N,A,B} &: I^{R}_{(M,N)} \to ([P]_A \to [P]_B) \to I^{R}_{(M*(B/A),N)} \\
map_{M,N,A,B} &: I^{C}_{(M,N)} \to ([P]_A \to [P]_B) \to I^{C}_{(M,N*(B/A))} \\
imap_{M,N,A} &: I_{(M,N)} \to ([P]_A \to P) \to I_{(M,N)} \\
convolve_{M,N,A,B} &: I_{(M,N)} \to (A, B) : (Int, Int) \to [K]_{(A*B)} \to I_{(M,N)} \\
filter2D_{M,N,A,B} &: I_{(M,N)} \to (A, B) : (Int, Int) \to ([P]_{(A*B)} \to P) \to I_{(M,N)} \\
zipWith_{M,N,A} &: I_{(M,N)} \to I_{(M,N)} \to ([P]_A \to [P]_A \to [P]_A) \to I_{(M,N)} \\
zipWithScalar_{M,N,A} &: I_{(M,N)} \to P \to (P \to P \to P) \to I_{(M,N)} \\
zipWithVector_{M,N,A,B} &: I_{(M,N)} \to [P]_A \to ([P]_A \to P \to P) \to I_{(M,N)} \\
unzip_{M,N,A} &: I^{R}_{(M,N)} \to ([P]_A \to P) \to ([P]_A \to P) \to (I^{R}_{(M/2,N)}, I^{R}_{(M/2,N)}) \\
unzip_{M,N,A} &: I^{C}_{(M,N)} \to ([P]_A \to P) \to ([P]_A \to P) \to (I^{C}_{(M,N/2)}, I^{C}_{(M,N/2)}) \\
scan_{M,N} &: I_{(M,N)} \to Int \to (P \to Int \to Int) \to I_{(M*N)} \\
foldScalar_{M,N} &: I_{(M,N)} \to Int \to (P \to Int \to Int) \to Int \\
foldVector_{M,N,A} &: I_{(M,N)} \to Int \to (A : Int) \to (P \to [Int]_A \to [Int]_A) \to [Int]_A \\
transpose_{M,N} &: I^{R}_{(M,N)} \to I^{C}_{(M,N)} \\
transpose_{M,N} &: I^{C}_{(M,N)} \to I^{R}_{(M,N)}
\end{aligned}
$$

Fig. 4. RIPL skeletons

An image $I^{R}_{(M,N)}$ is M pixels wide and N pixels high, and whose pixels are in row (R) major order. RIPLs *map* and *unzip* skeletons are implicitly directional, sliding linearly either row wise or column wise over a one dimensional vector of pixels. This meta information about stream order, image dimensions and vector lengths is not specified by the programmer; it is inferred by the RIPL compiler.

Skeletons with Non-overlapping Sliding Windows. The *map* skeleton slides over an image and applies the user defined function with a non-overlapping 1D vector on each execution. The *zipWith* skeleton is similar, though it slides a vector window of the same length over two images in lock step. The *map* and *zipWith* skeletons are stateless and do not carry state between executions. Their memory costs are therefore solely determined by the length of the sliding vector that they consume. The *zipWithScalar* skeletons combines every pixel and a

scalar value with a user defined function, and similarly *zipWithVector* allows the programmer to use a random access vector to modify an image.

As an example of non-overlapping sliding windows, the following RIPL assignment combines two images using *zipWith* with a mean average combinator. The memory cost is 2 8 bit integers, one each for pixels *p1* and *p2* from images *image1* and *image2* respectively.

```
image3 = zipWith image1 image2 (\[p1] [p2] -> [(p1+p2)/2]);
```

Skeletons with Overlapping Sliding Windows. The *imap* skeleton is useful for applying 1D filters to an image. It is an *indexed* map that slides over contiguously positioned pixels in a non-discrete fashion. The *imap* syntax differs from *map*, because *imap* applies a function from a pixel *position [.]* to a new value for that position, using the current pixel value and its neighbouring pixels using *+/-*, *e.g. [.-1]* points to the pixel to the left of *[.]* in an I^R image. The difference in how *map* and *imap* traverses an image is depicted in Fig. 5, which is labelled with repeated execution counts show the difference in their data processing rates of an image row. Figure 6 shows the expression of a 1D blur filter in RIPL, along with its memory cost. The hardware implementation of this *imap* consumes pixels into a 3 element circular buffer, updating the mid point index for *[.]*, before executing the user defined blur function.

(a) traversing images with *map* (b) traversing images with *imap*

Fig. 5. Comparison of *map* with *indexed map*

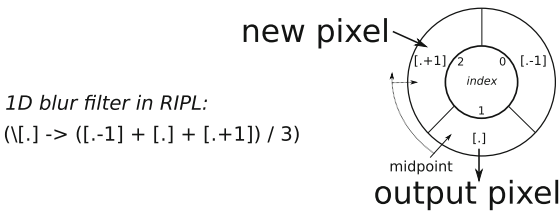


Fig. 6. Memory requirements of 1D blur with *imap* satisfied with a circular buffer

RIPLs *unzip* skeleton is for splitting apart an image into two images, and shares the pixel position syntax with *imap*. The hardware memory generated from *unzip* is similar to *imap*, the difference being its scheduling – the hardware for *unzip* creates two image streams, which are produced by alternating the execution of the two user defined functions.

Skeletons for 2D Filters. Many 2D filters can be implemented by combining the results of two 1D filters, one in the horizontal direction and one in the vertical direction. This implementation approach is possible in RIPL by applying a 1D horizontal filter with *imap*, transposing the result with *transpose*, then applying a vertical 1D filter with *imap*. However, this is a very memory costly composition, because *transpose* generates a frame buffer. For better stream data locality, RIPL has two 2D filters *convolve* and *filter2D*. The *convolve* skeletons modifies each pixel by applying a convolution of its neighbours using a small user defined $M\ PLH\ N$ kernel. The following example applies 3×3 kernel to sharpen an image.

```
image2 = convolve image1 (3,3) {0,-1,0,-1,5,-1,0,-1,0};
```

The *filter2D* skeleton provides more expressivity than *convolve*. When using *filter2D* with a 3×3 window, the programmer is provided 9 pixel values that can be used in their own function body, as shown earlier in Fig. 3 which computes the approximate magnitude $|G| = |Gx| + |Gy|$ for Sobel edge detection.

The memory requirements for *convolve* and *filter2D* is shown in Fig. 7. This has the capacity to store two rows and a further three pixels. Stream based processing begins once one row and two pixels are streamed into the corresponding buffer, which is when the top left pixel can be processed.

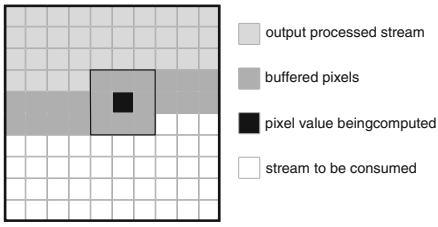


Fig. 7. Memory cost for *convolve* and *filter2D*

Stateful Skeletons. Stateful programming is achieved with RIPL using two skeletons, *foldScalar* and *foldVector*. They apply user defined reduction operations on images or image regions. Reducing an image to a scalar value is done using *foldScalar*, *e.g.* finding the maximum pixel value. The *scan* skeleton is similar to *foldScalar*, but returns a stream of intermediate successive reduced values. Reducing an image to a vector is done using *foldVector*, *e.g.* computing a colour histogram with each bin initialised to 0. Maximum pixel and histogram calculations are expressed as:

```
maxValue  = foldScalar image1 0 (\p currMax -> max p currMax);
histogram = foldVector image1 0 255 (\p hist -> hist[p]++);
```

4 RIPL Memory Costs

4.1 Memory Costs for Computation

RIPL programs are compiled to a dataflow intermediary of small computational actors and FIFOs. The memory costs for each RIPL skeleton in bytes is shown

Table 1. Memory costs for RIPL skeletons

		Example		
Skeleton	buffer size	RIPL code	$M \times N$	buffer size
$map_{M,N,A,B}$	A	**map** image1 ($\lambda[a,b,c] \rightarrow ...$)	n/a	3
$imap_{M,N,A}$	$A+1$	**imap** image1 ($\lambda[.] \rightarrow ([.-1]+[.]+[.+1])/3$)	n/a	4
$zipWith_{M,N,A}$	$A*2$	**zipWith** image1 ($\lambda[a,b]\ [c,d] \rightarrow ...$)	n/a	4
$zipWithScalar_{M,N,A}$	$A+1$	**zipWithScalar** image1 ($\lambda[a,b]\ x \rightarrow ...$)	n/a	3
$zipWithVector_{M,N,A,B}$	$A+B+1$	**zipWithVector** image1 ($\lambda[a,b]\ vect \rightarrow ...$)	n/a	3 + B
$unzip_{M,N,A}$	$A+1$	**unzip** image1 ($\lambda[a,b] \rightarrow ...$) ($\lambda[c,d] \rightarrow ...$)	n/a	3
$convolve_{M,N,A,B}$	$M*2+3$	**convolve** image1 (3,3) kernel	512×512	1027
$filter2D_{M,N,A,B}$	$M*2+3$	**filter2D** image1 (3,3) ($\lambda... \rightarrow ...$)	512×512	1027
$scan_{M,N}$	2	**scan** image1 0 ($\lambda.. \rightarrow ..$)	n/a	2
$foldScalar_{M,N}$	2	**foldScalar** image1 0 ($\lambda.. \rightarrow ..$)	n/a	2
$foldVector_{M,N,A}$	$A+1$	**foldVector** image1 255 0 ($\lambda.. \rightarrow ..$)	n/a	256
$transpose_{M,N}$	$M*N$	**transpose** image1	512 × 512	262144

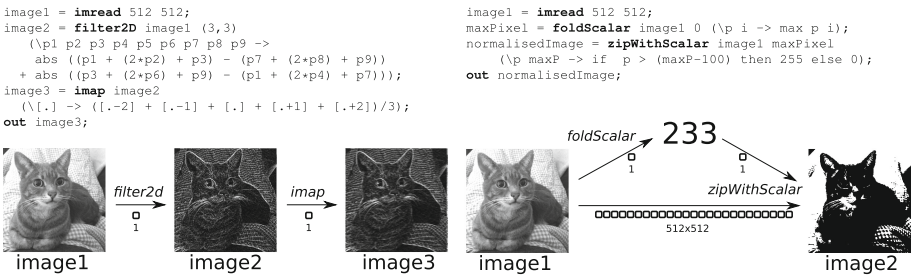
in Table 1. The *map*, *imap*, *zipWith* and *unzip* skeletons are implemented with either overlapping or non-overlapping sliding vectors, and hence their memory requirements are not determined by an image's dimensions. These costs are calculated from their offsets in stream access, analogous to array access offset analysis in *for* loops in imperative languages. The *map* and *zipWith* memory costs are solely determined by the vector length of the λ argument in the user defined function. The memory cost of *zipWithScalar* and *zipWithVector* is the stored scalar or vector, and the next incoming pixel value. The memory costs for *imap* are determined by the biggest X in *[.+X]* occurrences in the output expression, minus the biggest Y in *[.-Y]* occurrences.

The memory cost for the *foldScalar* and *scan* skeletons is the folded scalar and the next pixel from the image stream. The *foldVector* skeleton's memory requirements are determined by the programmer's choice of output vector length which is folded through each execution, and the next pixel from the image stream. The *convolve* and *filter2D* skeleton's memory requirements are determined by the processed image's width. The most costly skeleton is *transpose*, because it

requires an entire image to be stored in a buffer before being outputted with a transpose index.

4.2 Memory Costs for Communication

An addition memory cost is the depth of dataflow wires to ensure deadlock free RIPL execution. Dataflow wires are derived by data dependencies in RIPL programs, *i.e.* if the output of one skeleton is used as an input to another, then a FIFO point-to-point connection is created in hardware to support that data sharing. When the output of one RIPL skeleton is used in just one place in a program, only one output FIFO will be connected from the hardware implementing that skeleton, shown in Fig. 8a. In these cases, the required FIFO depth is determined by the vector length of the λ argument in the receiving skeleton. For example, if a *map* takes $\lambda[a, b, c]$ then the required depth is 3. The overall memory cost for FIFOs in Fig. 8a is 2 8 bit integers.



(a) Edge filter then 1D blur (b) Image threshold

Fig. 8. Memory costs for dataflow FIFOs

If the output image of one skeleton is used in multiple places, then depth requirements can increase, shown in Fig. 8b. This RIPL program finds the biggest pixel value of 233 with *foldScalar*, which is used to threshold the original image using *zipWithScalar* with a threshold of $233 - 100$. The generated hardware duplicates image *image1* over two FIFOs, one to the dataflow actor for computing the maximum value, and the other to threshold the image. Pixel tokens are transmitted to both FIFOs in lock step. Therefore in order for $maxP$ to be computed, the actor executing *foldScalar* needs to receive all tokens, so the FIFO to the threshold actor needs capacity to buffer the entire 512×512 image for the $maxP$ value to be computed. The overall memory cost for FIFOs in Fig. 8b is $(2 + 512 \times 512)$ 8 bit integers.

4.3 FPGA Memory Implementation

Once RIPL programs are compiled to dataflow graphs, actor computation code and dataflow wires are compiled to HDL using an open source dataflow

compiler [1], which makes choices about how to implement memory. For scalar integer values it uses FPGA slice registers used as memory. For small arrays that the RIPL compiler generates to support *convolve*, *filter2D* and *foldVector*, the dataflow compiler may also use slice registers depending on the overall memory requirements of the complete hardware design. The benefit of implementing these memories with slice registers is that the larger BRAM blocks are available for other parts of an algorithm, and because BRAM access is one clock cycle whilst LUTs RAM can be accessed without any latency. BRAM is used to support larger arrays generated by the RIPL compiler to support *convolve* and *filter2D* on big images, and for *transpose* which needs an entire image buffer.

FIFOs are compiled to HDL as generic memories, leaving the FPGA synthesis tools to choose how to implement them. Rendevous single token FIFOs will be implemented using registers or LUTs. Small FIFOs, *e.g.* to buffer a single row, are likely to be implemented with LUTs, whilst large FIFO depths, *e.g.* to support duplicating image streams in Fig. 8b, will likely be implemented using BRAM.

5 Evaluation

5.1 Expressivitiy

We next compare RIPL with the Vivado HLS OpenCV library. A key difference is that RIPL supports used defined functions to be expressed, whilst HLS OpenCV is a collection of predefined functions. For example, the HLS OpenCV *hls::Max* function combines two images by retaining the brighter of the pixels at each point, which can be expressed using RIPL's *zipWith* and *max* in the function body. Another example is RIPLs *filter2D*, which enables the programmer to define the mid pixel point with any function, whereas *hls::Filter2D* only supports convolution of a user defined kernel, equivalent to RIPL's *convolve* skeleton.

Another difference between RIPL and HLS OpenCV is image sharing. When an image is used in two places in a RIPL program, the image stream is automatically duplicated and shared to both consuming skeletons. With the HLS OpenCV model, an equivalent program would deadlock because the first function that uses the image will consume its pixels, emptying the FIFO. The *hls::Duplicate* function must be used explicitly to avoid this.

OpenCV programming requires explicit dimension and bitwidth information for each image declaration. The $hls::Mat<>$ template class is used to initialise an image, *e.g.* $hls::Mat < 512, 512, HLS\_8UC1 >$ defines a 512×512 single channel image, using 8 unsigned bits per pixel. In contrast, the RIPL compiler infers the dimension of every image, by following dimension transformations performed by skeletons through the implicit dataflow paths starting from *imread*, the only place where dimensions are explicit. The compiler also infers pixel bitwidths automatically, by calculating maximum upper bounds on bitwidth requirements as image data flows through arithmetic operators. Another difference is the inference of FIFO depths. The default FIFO depth in both RIPL and HLS OpenCV is 1. However, when an image is used in multiple places the RIPL compiler increases the FIFOs automatically to frame buffers (Sect. 4.2).

```
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. 9. Thresholding with HLS OpenCV using a maximum pixel value

The HLS compiler does not make this inference, leaving the programmer to use FPGA co-simulation to identify deadlocks. The user then programmatically uses *#pragma HLS stream depth* $=< N >$ to specify the FIFO depth to avoid deadlock. Figure 9 shows a HLS OpenCV example that demonstrates the need for explicit image duplication and explicit FIFO depths. The RIPL compiler infers both of these properties automatically.

The final difference is how image processing pipelines are constructed. Pipelined parallelism in RIPL is automatic. When two skeletons are composed in sequence over an input image, they will execute in parallel over different regions of the image stream. In HLS OpenCV, the programmer must specify *#pragma HLS dataflow* above the function calls intended to be pipelined over the image stream.

One similarity between HLS OpenCV and RIPL is the implementation of image data structures. An OpenCV image is a *hls::Mat*. The Vivado HLS FPGA implementation of *hls::Mat* images uses *hls::stream* internally, so OpenCV images on FPGAs are FIFOs, which is also true for RIPL. Hence random image access is not possible in either case.

Darkroom can be used to express benchmarks 1 and 2 (✓), but not 3, 4 or 5 (✗). Global reductions are not supported, because Darkroom's line buffers cannot be used to store values beyond a traversing a single line. Such a buffer is required to compute the maximum pixel value (3) and the histogram (4). RIPL is the only language of the three compared that supports image transposition, which again requires a frame buffer that uses 64 BRAMs.

5.2 Space Performance

We use five benchmarks to compare the space performance of RIPL and OpenCV compiled to FPGAs using Vivado HLS. The benchmarks are (1) brighten each pixel in an image by 50, (2) 2D Sobel edge detection, (3) find the maximum pixel $maxPixel$ value then threshold the image with $(maxPixel - 50)$, (4) compute a sum histogram for an image then normalise the image using the histogram, and (5) transpose an image. All programs are compiled for the Xilinx Zedboard XC7Z020 for 512×512 single channel images. The memory use performance of Darkroom cannot be compared because the line buffer to Verilog compiler backend is not publicly available.

Table 2. Memory implementation and expressivity results

	Benchmark	RIPL		HLS OpenCV		Darkroom
		BRAM	LUTs	BRAM	LUTs	
1	Image brighten	0 (0 %)	118 (0 %)	0	450 (0 %)	✓
2	Sobel 2D edge detection	1 (0 %)	12273 (23 %)	3 (1 %)	713 (1 %)	✓
3	Threshold with max pixel	64 (45 %)	280 (0 %)	64 (45 %)	9172 (1 %)	✗
4	Histogram normalisation	64 (45 %)	799 (1 %)	3 (1 %)	2918 (5 %)	✗
5	Image transposition	64 (45 %)	321 (0 %)	✗		✗

The synthesis results in Table 2 are for RIPL and HLS OpenCV. RIPL and OpenCV occupy very similar memory resources for image brightening (1) and image thresholding (3). For Sobel edge detection (2), RIPL uses 2 BRAMs less than OpenCV by instead using more LUTs. Thresholding and histogram normalisation (4) in RIPL require a FIFO depth equal to the number of pixels in the image in RIPL's hardware backend. To support 8 bit pixels, the synthesis tools use BRAMs in 32Kb mode, so storing a 512×512 image requires 64 BRAM blocks for these two benchmarks. The same is true for HLS OpenCV for thresholding, but not histogram normalisation. This is because of an optimisation built into *hls::EqualizeHist()*, which normalises frame $N + 1$ using the histogram computed for the previous frame N. This results in more efficient BRAM use compared to RIPL. We plan this optimisation for RIPL.

6 Conclusion

Memory resources on FPGAs can be tailored to the needs of an algorithm, so FPGA compilers are not hindered by fixed memory hierarchies such as those on CPUs and GPUs. They are however constrained by the limited amount of on chip BRAM and LUT memory resources. This paper describes the memory efficiency aspects of RIPL, our image processing DSL for FPGAs. RIPL is more concise than Vivado HLS OpenCV, because it automatically infers upper bounds on bitwidths and the required FIFO depths for image streams, and image streams

are automatically duplicated when necessary. Despite these abstractions, RIPL memory use is competitive on three of the four benchmarks expressible with the Vivado HLS OpenCV. RIPL is more expressive than the Darkroom image processing DSL, because Darkroom compiles to line buffers so does not support global image reductions. Future work will explore temporal video processing capabilities in RIPL, where new opportunities for dataflow analysis for data locality may arise. We also wish to explore the applicability of RIPL for FPGA acceleration of other stream based domains beyond image processing.

Acknowledgements. We acknowledge the support of the Engineering and Physical Research Council, grant reference EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications).

References

1. Bezati, E.: High-level synthesis of dataflow programs for heterogeneous platforms. Ph.D. thesis, STI, EPFL, Switzerland (2015)
2. Bradski, G.R., Kaehler, A.: Learning OpenCV - Computer Vision with the OpenCV library: Software that Sees. O'Reilly, Beijing (2008)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1991)
4. DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P., Vitek, J.: Terra: a multi-stage language for high-performance computing. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, USA, June 16–19, 2013, pp. 105–116. ACM (2013)
5. Hegarty, J., Brunhaver, J., DeVito, Z., Ragan-Kelley, J., Cohen, N., Bell, S., Vasilyev, A., Horowitz, M., Hanrahan, P.: Darkroom: compiling high-level image processing code into hardware pipelines. ACM Trans. Graph. **33**(4), 1–11 (2014)
6. Kennedy, K., McKinley, K.S.: Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) LCPC 1993. LNCS, vol. 768, pp. 301–320. Springer, Heidelberg (1994). doi:10.1007/3-540-57659-2_18
7. Kiselyov, O.: Iteratee IO: Safe, Practical, Declarative Input Processing. In: 11th International Symposium on Functional and Logic Programming. LNCS, vol. 7294, pp. 166–181 (2012)
8. Lee, H., Brown, K.J., Sujeeth, A.K., Rompf, T., Olukotun, K.: locality-aware mapping of nested parallel patterns on GPUs. In: 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, UK, December 13–17, 2014, pp. 63–74. IEEE (2014)
9. Muddukrishna, A., Jonsson, P.A., Brorsson, M.: Locality-aware task scheduling and data distribution for openmp programs on NUMA systems and manycore processors. Sci. Program. **2015**, 981759: 1–981759: 16 (2015)
10. Stephen Neuendorffer, T.L., Wang, D.: Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries. Technical report, Xilinx, June 2015
11. Tate, A., et al.: Programming abstractions for data locality. In: Workshop on Programming Abstractions for Data Locality, Swiss National Supercomputing Center, Lugano, Switzerland, April 2014

12. Wieser, V., Grelck, C., Haslinger, P., Guo, J., Korzeniowski, F., Bernecky, R., Moser, B., Scholz, S.: Combining high productivity and high performance in image processing using Single Assignment C on multi-core CPUs and many-core GPUs. J. Electron. Imaging **21**(2), 21116 (2012)
13. Xilinx: Implementing Memory Structures for Video Processing in the Vivado HLS Tool. Technical report, Xilinx, September 2012