A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs

by

Nitin chugh, Vinay Vasista, Suressh Purini, Uday Bondhugula

in

PACT ’16 - 2016 International Conference on Parallel Architectures and Compilation

Report No: IIIT/TR/2016/-1

Centre for Software Engineering Research Lab
International Institute of Information Technology
Hyderabad - 500 032, INDIA
September 2016
A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs

Nitin Chugh  
International Institute of Information Technology  
Hyderabad 500032 India  
nitin.ch@research.iiit.ac.in

Vinay Vasista  
Dept of CSA  
Indian Institute of Science  
Bengaluru 560012 India  
vinay.vasista@csa.iisc.ernet.in

Suresh Purini  
International Institute of Information Technology  
Hyderabad 500032 India  
suresh.purini@iiit.ac.in

Uday Bondhugula  
Dept of CSA  
Indian Institute of Science  
Bengaluru 560012 India  
uday@csa.iisc.ernet.in

ABSTRACT

This paper describes an automatic approach to accelerate image processing pipelines using FPGAs. An image processing pipeline can be viewed as a graph of interconnected stages that processes images successively. Each stage typically performs a point-wise, stencil, or other more complex operations on image pixels. Recent efforts have led to the development of domain-specific languages (DSL) and optimization frameworks for image processing pipelines. In this paper, we develop an approach to map image processing pipelines expressed in the PolyMage DSL to efficient parallel FPGA designs. Our approach exploits reuse and available memory bandwidth (or chip resources) maximally. When compared to Darkroom, a state-of-the-art approach to compile high-level DFGs to FPGAs, our approach (a) leads to designs that deliver significantly higher throughput, and (b) supports a greater variety of filters. Furthermore, the designs we generate obtain an improvement even over pre-optimized FPGA implementations provided by vendor libraries for some of the benchmarks.

1. INTRODUCTION

An image processing pipeline can be viewed as a directed acyclic graph, where each node performs simple processing on image pixels; the edges of the graph represent producer-consumer relationships between the nodes. Each node is referred to as a stage or a filter and typically involves the application of a point-wise, stencil, or other more complex operations on image pixels. Examples of domains that employ such pipelines include computational photography, embedded vision, and medical imaging. High performance and acceleration is highly desirable either due to the volume of data, the requirement to provide a response in real time, or the growing complexity of algorithms compounded with high resolutions and frame rates.

The advantages offered by Field-Programmable Gate Arrays (FPGAs) in comparison to general-purpose microprocessors are well-known, and so are the programmability issues surrounding them [3]. The difficulty of programming FPGAs has restricted FPGA technology to small and specialized segments. High-level Synthesis (HLS) through C-based languages [6, 27, 37, 25, 13, 24], OpenCL/CUDA-based ones [26, 28], high-level language frameworks [2, 15], and model-based frameworks like NI LabVIEW [10] and MATLAB HDL coder [21] are various efforts that improve FPGA programmability. A comprehensive survey of all these approaches can be found in [3]. In the last decade, HLS suites [6] have significantly advanced and have made it easier for programmers to use a C/C++ like high-level interface to program FPGAs instead of hardware description languages (VHDL or Verilog). Nearly all EDA vendors now provide suites supporting HLS — examples of these include Xilinx Vivado, Cadence C-to-silicon compiler, Synopsys Synphony C compiler, and Mentor Graphics Catapult C.

Domain-specific languages (DSLs) provide a high-level language interface suitable to a domain, to improve programmability as well as deliver performance for both general-purpose processors as well as accelerators like GPUs and FPGAs. In the context of compiling for FPGAs, HLS programming interfaces are an interesting level of abstraction to target, for DSL compilers and code generators. The approach we take in this paper is that of adding support for FPGA compilation in a DSL compiler by targeting the input interface of a HLS suite. There have been several recent efforts on building domain-specific languages and compilers for image processing pipelines [32, 16, 33, 23]. Darkroom [15] is a language embedded in Terra for describing image processing pipelines. Although Darkroom’s approach extracts maximum reuse, it does not exploit degrees of data parallelism that could lead to a higher or maximal utilization of memory bandwidth or chip resources, as the case may be. Extracting data parallelism while allowing reuse of data between stages of the pipeline poses unique challenges, and we address these.

In this paper, we present an FPGA backend for PolyMage [30,
23], a DSL for image processing pipelines, and study the potential for acceleration of image processing pipelines using FPGAs. Our approach first starts by exploiting the pipeline structure evident from the directed graph of image processing stages; it then replicates processing elements to exploit data parallelism in the filters until (a) either all of the memory bandwidth is utilized or (b) FPGA slices or other on-chip resources are exhausted. Irrespective of whether the kernel is compute-bound or memory bandwidth bound, we obtain designs with near-peak throughput in this manner. We take in PolyMage DSL as input and express our design in C++ code that serves as input to the Xilinx Vivado HLS compiler.

Our transformation and optimization approach is tailored to the domain of image processing pipelines considered, in particular, sequences of stencil-like operations on images. We will see that our transformations cannot be realized as a composition of traditional loop transformations – the latter have been considered by several existing FPGA compiler and HLS approaches [13, 12, 24, 11, 36, 5, 1, 31]. Thus, our DSL-based approach provides a level of abstraction much higher than that provided by generic HLS efforts, and automates a specialized optimization strategy.

The rest of this paper is organized as follows. Section 2 presents necessary background on image processing pipelines, FPGAs, and PolyMage. Section 3 and 4 describe our approach in detail. Related work is discussed in Section 6 and conclusions are presented in Section 7.

2. BACKGROUND ON POLYMAGE

PolyMage is a domain-specific language (DSL) and compiler for automatic optimization of image processing pipelines. It supports optimization of pipelines where the filters are either point-wise, stencil, upsampling or downsampling operations. Other data-dependent operations and reductions are also supported but optimizations (fusion and tiling) are limited in the presence of such filters (to program parts that do not comprise such filters). PolyMage DSL is embedded in Python, i.e., it is an internal DSL. Optimized OpenMP C++ code can be generated for shared-memory parallel systems, typically, a multi-socket system with general-purpose multicore processors.

The PolyMage language is functional – each function corresponds to a stage in the pipeline. Functions in PolyMage (with the exception of reductions) are data parallel and apply on the entire set associated with the variables in the function’s domain; the variables have corresponding intervals specified when defining the function. For brevity, we do not describe constructs that are not relevant to the remaining content of this paper. More detail can be found in [23]. Compiler transformation and code generation happens when the Python code is executed; the generated OpenMP/C++ code is compiled on-the-fly, and the compiled code can be used subsequently in the Python program or offline as an optimized routine.

PolyMage DSL code for an image processing algorithm, unsharp mask, is shown in Listing 1. Unsharp mask is an image sharpening technique, very commonly used in digital image processing systems. In signal processing context we can visualize unsharp mask as a filter that amplifies the high-frequency components of an input signal. The algorithm is expressed as a composition of four pipelined stages. The first two stages perform a Gaussian blur of the original input image along the x and the y directions subsequently.

These stages are written using the PolyMage’s Stencil construct. This construct takes as input the point around which the stencil should be applied, a global weight for stencil output, and a stencil kernel with its weights specified using a list of lists. The third stage of unsharp mask computes a weighted sum of the resultant blur and outputs a sharpened image. The final stage chooses pixels, to be written to the output, between the original and the sharpened image, which is done in PolyMage using the Select construct. Sharpened pixels are chosen, upon the difference between the original and its blur image crossing a threshold value.

3. MAPPING TO PIPELINES ON FPGAS

This section describes our approach and primary contribution in detail. We describe our technique to map image processing pipelines to parallel implementations on the FPGA. We also show how we add this support into PolyMage’s DSL compiler. A brief overview of our framework is given in Figure 2.

3.1 Parallelism and Reuse

Image processing pipelines exhibit several forms of parallelism that can be extracted on FPGAs. They can be classified into four categories. Most importantly, there exists data parallelism due to the application of the same operator on all the pixels of an image. This data parallelism can be exploited in a fine-grained manner or a coarse-grained manner. We discuss this choice a little later. Secondly, there exists parallelism that can be obtained by pipelining through the different stages or filters of the pipeline. We choose to extract this inter-stage pipelined parallelism in a fine-grained manner, which is clearly the right granularity for FPGAs. Thirdly, a filter that works on color images has to be applied to each of the channels of an image (three in case of RGB). This can be done in a completely parallel fashion if the filter computation at a point
3.2 Dependences

User code written in PolyMage is translated into a directed acyclic graph (DAG) wherein each node corresponds to a stage of the pipeline and the edges represent the dependences between the stages. Figure 1 shows the DAG associated with the Unsharp Mask program. The stage `img` is an idempotent operation which provides the input image to the subsequent consumer stages such as `blurx`, `sharpen` and `mask`. We associate with each edge of the DAG, a set of integral vectors similar to the traditional dependence vectors used in compiler analysis, but with a different notion. For example, the distance vector set associated with the edge (`blurx`, `blurx`) is `{ (0,0,0) | -2 ≤ i ≤ 2 }` indicating that the computation of `blurx(c,x,y)` depends on the values `{ `blurx(c,x,y+i)` | -2 ≤ i ≤ 2 }`. Similarly, the distance vector set annotating the edge (`blurx`, `sharpen`) contains only one vector, `(0,0,0)`, as `sharpen` is a point-wise computation.

3.3 Structuring the Pipeline in Hardware

The pipeline compiler uses the DAG representation to topologically sort all the stages. Each stage of the pipeline is translated into an equivalent C++ code using a k-way nested loop where k is the number of dimensions of the domain on which the corresponding stage acts. A stream or a queue is associated with every edge of the DAG. The stage at the source of an edge enqueues its output on to the associated stream, from which the destination stage dequeues during its computation. The dequeue and enqueue operations associated with a stream s are similar to s.read() and s.write() method calls. Such a streaming approach has two key advantages. Firstly, a stage need not wait for the completion of its predecessor stages. It can start computing an array location as soon as all the required inputs are available in its input streams. It thus follows the data flow model of computation. The second advantage is that the size of the intermediate buffer linking a producer and a consumer stage will be a function of the stencil structure as opposed to the total image size. For example, if the consumer stage is a point operation, the size of the intermediate buffer would be just a constant. Overall, when the entire C++ code generated by the hardware compiler is fed to a high-level synthesis compiler with appropriate

```cpp
# Param
R = Parameter(Int, "R")
C = Parameter(Int, "C")
thresh = Parameter(Float, "thresh")
weight = Parameter(Float, "weight")

# Vars
x = Variable(Int, "x")
y = Variable(Int, "y")
c = Variable(Int, "c")

# Input Image
img = Image(Float, "input", [3, R+4, C+4])

# Intervals
cr = Interval(Int, 0, 2, 1)
xrow = Interval(Int, 2, C+1, 1)
ycol = Interval(Int, 2, R+1, 1)
yrow = Interval(Int, 2, R+1, 1)

# Pipeline
blurx = Function([c, x, y], [cr, xrow, xcol], Float, "blurx")
blurx.defn = [ Stencil(img(c, x, y), 1.0/16, [[1, 4, 6, 4, 1]]) ]
blury = Function([c, x, y], [cr, yrow, ycol], Float, "blury")
blury.defn = [ Stencil(blurx(c, x, y), 1.0/16, [[1, 4, 6, 4, 1]]) ]
sharpen = Function([c, x, y], [cr, yrow, ycol], Float, "sharpen")
sharpen.defn = [ img(c, x, y) * (1 + weight) \ + blurx(c, x, y) * (-weight) ]
masked = Function([c, x, y], [cr, yrow, ycol], Float, "mask")
masked.defn = [ Select(Condition(absv, '<', thresh), img(c, x, y), sharpen(c, x, y)) ]

Listing 1: PolyMage DSL code for Unsharp Mask
```
Non-linear Pipeline Structure.

As mentioned earlier, our framework follows a dataflow model of computation, wherein a stage can perform a computation as soon as the required input data is available. It even supports non-linear pipelines, as opposed to Darkroom [15] where the pipeline is completely linearized. So, given a DAG such as in Figure 3b, the stages $s_1$ and $s_2$ can perform their computations independent of the stages $s_3$ and $s_4$, and vice-versa. Between the stages $s_1$ and $s_3$ ($s_3$ and $s_4$), there will be pipelined parallelism. However, for the DAG in Figure 3a, although the stages $s_2$ and $s_3$ can execute in parallel, we introduce a false dependency between them to reduce the write load on the stage $s_1$. However, depending on the available memory bandwidth, such false dependences can be avoided. Our compiler framework allows for such design space exploration with ease.

3.4 Line Buffer Allocation

Each stage maintains a line buffer corresponding to each incoming input stream. Hence, if a stage has $k$ input streams, it maintains $k$ line buffers. The size of a line buffer depends on the stencil structure and the structure of the DAG. We compute the minimum size required for each line buffer in two steps.

In the first step, we use the distance vector set associated with an edge to estimate the line buffer size. Let $d = d_1 - d_1$, where $d_i$ and $d_f$ are the lexicographically smallest and largest distance vectors respectively. Assuming a two dimensional case, for the sake of simplicity, let $d = (d_1, d_2)$. Then, the line buffer size is determined as $w d_1 + d_2 + 1$, where $w$ is the image width. For example, in the case of blurx, the size of the line buffer is five as its distance vector set is $\{0, 0, 1\}$. Similarly in the case of blury, the size of the line buffer is $4w + 1$, where $w$ is the image width, as the distance vector set is $\{(0, 0, 0)\}$.

In the second step, to find the exact buffer estimates, we increase the line buffer size between two stages by computing the longest path between them in the DAG wherein each edge has been annotated with the line buffer size estimates from the first step. Figure 4a shows the DAG representation of the Unsharp Mask program where the edges are annotated with the corresponding line buffer size estimates. These estimates define a lower bound on the buffer requirements, since for each dependence, we considered the producer and consumer stage in isolation from the rest of the DAG. For example in Figure 4a, the sharpen stage cannot start its computation until it gets data from the blurx stage and the relayed img data from the blury stage. The blury stage starts producing data only once its line buffer is full with the output of blurx; img data is however relayed to the sharpen stage without any delay. Hence, we need a minimum buffer size of five associated with the edge going from blurx to sharpen, to store img data until the first output of blury is available. In the second step, to find the exact buffer estimates, we increase the line buffer size between two stages by computing the longest path between them in the DAG wherein each edge has been annotated with the line buffer size estimates from the first step. Figure 4b shows the DAG for the Unsharp Mask program with revised line buffer size requirements.

A stage reads data from an input stream and moves it to the corresponding line buffer. Once all line buffers are full,
3.4.1 Discussion on Line Buffer Allocation and Scheduling

Consider the DAG in Figure 6 wherein the label $l_{ij}$ on an edge $(s_i, s_j)$ indicates the minimum data required by the destination stage $s_j$ from the corresponding source stage $s_i$. If the throughput of each stage is one pixel per clock cycle, then the stage $s_5$ can start its computation from clock cycle $w + 1$.

Meanwhile, the pixels generated by stage $s_2$ have to be accumulated on its output line buffer, and hence its size has to be at least $w$. We can avoid this extra line buffer usage by scheduling stage $s_1$ to start its computation in clock cycle $w - 1$.

This observation can be generalized to obtain a polynomial time algorithm to compute the line buffer sizes on all the DAG edges such that the sum total of memory allocated to all of them is minimized. This is accomplished by delaying the clock cycle at which each stage of the DAG starts computing by as much as possible without affecting the critical path. Let $S$ be the set of source vertices (stages) whose in-degree is zero, and $S'$ be the set of sink vertices whose out-degree is zero. Let $e_0$ be the earliest clock cycle by which a stage $s$ can start execution without violating dependences, i.e., by clock cycle $e_s$ the entire required input data is available to stage $s$. Then $e_s = 0$ for all $s \in S$. For every other stage $s$ in the DAG, $e_s = \max_{p \in \text{Pred}(s)} (e_p + l_{ps})$ where $\text{Pred}(s)$ consists of all the stages whose output is consumed by stage $s$. We can compute all $e_s$ values by making a forward pass on the stages of the DAG in the topologically sorted order. Then we make a backward pass on the DAG in reverse topological order to determine $\epsilon_0$, which gives the last clock cycle to which the computation in stage $s$ can be delayed without affecting the critical path leading to any of the sink nodes.

For every $s \in S'$, $\epsilon_s = e_s$. For every other stage $s$ in the DAG, $\epsilon_s = \min_{p \in \text{Succ}(s)} (\epsilon_{sp} - l_{sp})$ where $\text{Succ}(s)$ consists of all stages which consume the output generated by stage $s$.

From the above discussion, we notice that the initial computational cycle of any stage can be shifted to any value in the range $[\epsilon_s, \epsilon_s']$. If a stage $s$ is in the critical path of any output sink stage, then $\epsilon_s = \epsilon_s'$ in that case. If we schedule the initial computation of each stage $s$ in cycle $\epsilon_s$, then the required line buffer size associated with a DAG edge $(q, r)$ is $\epsilon_r - \epsilon_q$. In Figure 6, $\epsilon_s = w + 1$ and $\epsilon_{s2} = w$, and hence the required size of the line buffer on the associated edge is one. Finally, if two source stages consume the same input image and their initial cycle shifts are different, then we either need more memory bandwidth to read the same input again or store it in an on-chip memory resource such as a line buffer for reuse. In the latter case, where we use a line buffer to store the input image for reuse, the savings obtained in other parts of the DAG pipeline structure gets consumed here. For example, in Figure 6, if both $s_1$ and $s_3$ are reading the same input image, we need to add a line buffer of size $w - 2$ before the stage $s_1$, in which the streamed input image pixels can be stored. This offsets the line buffer savings we obtained by shifting the initial computation cycle of stage $s_2$ to cycle $w$.
beats and forward them to accelerators. Usually, only a sub-
set of pixels inside the data beat are required in the over-
lap region. Thus, the unpack units must extract only those
pixels that belong to the iteration domain of the accelerators
and contribute to the computation, rest are discarded. After
processing, the output of the accelerators is packed into data
beats by pack units and gathered by a collector in the same or-
der as provided by the distributor. In effect, the data rate at
the input and output will ideally match the combined pro-
cessing rate of the accelerators. If an accelerator’s through-
put is \( q \) pixels per clock cycle and there are \( p \) such accelerator
replicas, then we pack \( pq \) pixels per data beat to sustain the
maximal pixel throughput under the constraints. For ideal
load balancing, the input is ordered in a way such that it al-
ternates between the pixels required by each accelerator, i.e.,
in the first data beat, pixels from the first image stripe are
sent; in the second one, pixels from the second image slice,
and so on.

3.5 Coarse-Grained Data Parallelism with Fusion

The approach described so far to generate FPGA imple-
mentations can only process one pixel per clock cycle and
does not make use of available data parallelism. For ex-
ample, each pixel output from the blurx stage can be computed
in parallel from the input image instead of in a streaming/se-
quential fashion. High-speed serial transceiver technology
or PCI-ex DMA provides FPGAs with communication inter-
faces capable of operating at high data rates and deliver mul-
tiple pixels per clock cycle as an aggregated bit vector, which
we will refer to as a data beat in the rest of this section.

In order to further improve overall pixel throughput, we
split the image vertically into partitions, where each vertical
partition is fed into its corresponding streaming accelerator.
Thus, pixels from different partitions are processed in paral-
lel. The performance gain is thus limited only by either the
external off-chip memory bandwidth or available on-chip re-
sources.

Next, we avoid communication between accelerator units
while processing boundary pixels using a realization of an
existing technique for general-purpose parallelization called
overlapped tiling [20]. This is shown in Figure 7. The vertical
slices of the image being processed overlap at their bound-
daries, i.e., it is not a disjoint partitioning. Such an overlap
is necessary to avoid communication when convolutions or
stencil type filters are involved. The width of the overlap at
the boundaries depends on the size of the convolution or the
width of the stencil applied. The idea of performing re-
dundant computation at the boundaries to avoid communication
or cache misses has been used in several code generation ap-
proaches for CPUs and GPUs including [17, 39, 32, 23].

The architecture of the parallel implementation can be bro-
den into multiple stages as shown in Figure 8. The first stage
of the implementation is a distributor that assigns incoming
data beats to buffer queues. From the buffer queues, data
is consumed by unpack units, which extract pixels from data
beats and forward them to accelerators. Usually, only a sub-

Figure 7: Partitioning computation in an overlapped manner.
Image credit: This image is a derivative of a still from the Big
Buck Bunny movie (C) Copyright 2008 by Blender Founda-
tion [4] used under the CC BY 3.0 license [7].

Figure 8: Architecture for parallel computation of image
slices

Upsampling and downsampling operations are handled
without any special additional techniques. If a scaled access
with a scale factor of \( k \) or \( 1/k \) (\( k \) being an integer) is observed,
instead of performing the calculation every iteration, it is per-
formed every \( k \) iterations (or \( k \) operations are performed in 1
iteration when the scale factor is \( 1/k \)).

4. OPTIMIZATIONS FOR HLS TARGET

In this section, we describe additional steps necessary to
use a HLS software as a target.

4.1 Vivado Synthesis Directives

A central element of Vivado HLS is the synthesis direc-
tives, which allow us to specify how an input design is to be
parallelized and optimized. These directives can be placed
directly in the code as pragmas or written in a script file that
is applied during synthesis. The few major directives that we
use are for pipelining the innermost loops, array partitioning
of the buffers, and specifying depth of FIFOs that intercon-
nect the streaming pipeline stages. The other important di-
rective we use is the DATAFLOW directive which specifies
to the Vivado HLS compiler that the loops and functions can
be processed as soon as their input is made available, and to
not execute them sequentially. Our DSL compiler automati-
cally annotates code with the appropriate directives as part
of the code generation process.
4.2 Using Arbitrary Data Types

Software programmers often tend to use standard primitive data types like int, float for variables like loop counters and pixel data. For image processing applications, we rarely need the entire width of these types. Using more bits than required results in excessive resource utilization and can also degrade the maximum achievable clock frequency. Hence, we use arbitrary precision data types provided by Vivado HLS to ensure that our design always uses the optimal representation for variables — this reduces resource consumption and helps achieve shorter critical paths.

4.3 Transforming Kernels

A kernel in image processing is a matrix that on convolution with an image can achieve various effects like blurring, edge-detection, etc. This convolution in effect is a sum of products of the similarly positioned elements in the two matrices. In many cases, the floating point kernel coefficients can be given an integral representation and all arithmetic can be carried out in the integral domain reducing the number of required DSP units on an FPGA. The integral representation can be translated back to floating point representation in the last stage of computation.

If user-provided mask coefficients have fractional parts, our framework automatically transforms them into integers. This transformation is only applied if each coefficient can be scaled to a natural number when multiplied by a normalization factor $N$, where $N$ is the inverse of the smallest coefficient (ensuring that the smallest coefficient is scaled to one). If the scaled coefficients are not natural numbers, they are rounded to the nearest natural number. Only if the rounding error is less than a certain threshold for all scaled coefficients, this optimization is applied; else the kernel transform is skipped to avoid inaccurate results.

4.4 Restructured HLS Code

We now show how our code generator restructures the code making it amenable for the HLS compiler to generate an efficient hardware design in Verilog/VHDL. We will also point out as to why writing such code by hand is non-trivial and tedious. We use the Unsharp Mask program from the Listing 1 as the running example here.

A standard way of implementing filters in hardware is through the use of a line buffer or a window buffer depending on the requirement. Assuming the input image is serialized in a row major order, pixel by pixel, the Gaussian blur along the horizontal direction just needs a sliding window buffer, as only the pixels equal to the size of the stencil are needed to be kept in memory at any point in time to perform the computation. The corresponding window buffer declaration with the variable name blury_blurx_Wbuffer can be obtained in the generated source code from the Listing 2. We use a line buffer when applying Gaussian blur along the vertical direction, as that requires us to store four lines of the input image. Ideally this needs a line buffer that can store pixels equal to $4 \times \text{image\_width} + 1$. For ease of understanding, we declare a 2-d array for the line buffer with variable name and dimensions as blury_input_Lbuffer $[5] \times \text{MAX\_C}$ (refer Listing 2).

4.5 Discussion

Listing 2 shows the auto-generated C++ code from the PolyMage Unsharp Mask program source. This is only a partial listing due to space constraints. The code for every pipeline function is skipped to avoid inaccurate results.

```cpp
void pipeline_unsharp_mask(int_t R, int_t C, int_t thresh, int_t weight, hls::stream<int_t> &input, hls::stream<int_t> &mask) {
    /* Stage 1 - Gaussian Blur Vertical */
    const int_t _ct0 = 3;
    const int_t _ct1 = (R+4);
    const int_t _ct2 = (C+4);
    hls::stream<data_t> blurx_out_stream("blurx_out_strm");
    hls::stream<data_t> input_sharpen_stream("input_sharpen_strm");
    data_t blury_input_LBuffer[5][MAX_C];
    const int_t blurx_input_Coeff[5] = {1,4,6,4,1};
    const int_t blurx_input_N = 0.0625;
    for (int_t i=0;i<5;i++) {
        data_t input_in_val = input.read();
        data_t out_val = 0;
        for (int_t i=0;i<5;i++) {
            if (i>0) blurx_input_LBuffer[i-1][ci2]=vwin_val;
            out_val += blurx_input_LBuffer[i][ci2]*vwin_val;
        }
        out_val*=blurx_input_N;
        if (ci5>=4) blury_out_stream.write(out_val);
    }
    /* Stage 2 - Gaussian Blur Horizontal */
    const int_t _ct3=3;
    const int_t _ct4=R;
    const int_t _ct5=(C+4);
    hls::stream<data_t> blury_out_stream("blury_out_strm");
    data_t blury_blurx_WBuffer[5];
    const int_t blury_blurx_Coeff[5] = {1,4,6,4,1};
    const int_t blurx_blurx_N = 0.0625;
    for (int_t i=0;i<5;i++) {
        data_t input_in_val = blury_out_stream.read();
        data_t out_val = 0;
        for (int_t i=0;i<5;i++) {
            if (i>0) blurx_blurx_WBuffer[i-1]=vwin_val;
            blury_blurx_WBuffer[i]*=vwin_val;
            out_val+=blurx_blurx_WBuffer[i];
        }
        out_val*=blurx_blurx_N;
        if (ci5>=4) blury_out_stream.write(out_val);
    }
    /* Stage 3 - Sharpening */
    ...
    /* Stage 4 - Masking */
    ...
}
```

Listing 2: Auto-generated restructured HLS code for Unsharp Mask
stage consists of a set of data structure declarations and a nested loop that does the actual computation. In this implementation, we see that each pipeline stage, either consumes the input image or the outputs of the previous stages. Before providing this code to the HLS tool to generate a hardware implementation, we annotate the code with certain directives: two of the most important directives are `dataflow` and `pipeline`. The dataflow directive ensures that each stage is scheduled as soon as its required inputs are available, which gives us the desired parallel processing of different stages in a pipelined manner. The `pipeline` directive is used to further parallelize each stage internally — this exploits the operator-level parallelism mentioned in Section 3.1.

An important point is to ensure that for each stream, the number of pixels written are exactly the same as number of pixels read. For this the iteration domain for each stage is matched to the unified output domains of all its input stages. Also, it is necessary to ensure that the set of pixels written by the producer stage into a stream belong to the same location as expected by the consumer stage reading that stream. This becomes a non-trivial task, since these pixels are not array-indexed, but a flowing stream of data and there has to exist a lock-step synchronization between the producer and consumer stages to ensure correctness of the generated HLS code. For the example in Listing 2, we see that the iteration domain of stage 2 is $[3, R, C+4]$ which is the output filtered domain of its input stage (Stage 1). Furthermore, when Stage 1 replicates the input stream for Stage 3, it is bounded by if conditionals to match the image domain that Stage 3 processes.

In this implementation, once all pipeline buffers are filled, a throughput of one data output per clock cycle can be achieved, which is optimal by manual design. Furthermore, if we treat this pipeline as a single accelerator unit of the parallel architecture in Figure 8, we can achieve a throughput of multiple data output per clock cycle. We will see in the next section that the upper bound on the number of PEs will be decided by the amount of memory bandwidth available, i.e., we run out of memory bandwidth before we hit the limit for any type of on-chip resource.

5. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our approach, we consider several standard image processing algorithms. Although these algorithms are well-known, they can vary in their implementations. Hence, we provide below a brief overview of the benchmarks used for evaluation.

5.1 Benchmarks

We briefly describe each of the benchmarks evaluated below. Versions of these benchmarks in C (both naive and optimized by PolyMage) are publicly available [29]. The number of stages in the DAGs representing the respective pipelines of these benchmarks are provided in Table 1 along with the number of lines of code required to express them in PolyMage DSL.

Unsharp Mask (USM) algorithm is actually used to sharpen an image. This is done by computing a weighted sum of the image and its blur at pixels where both differ significantly in intensity.

Harris Corner Detection (HC), first introduced by Harris and Stephens, consists of a complex image processing pipeline comprising local and point operators. The algorithm uses the Sobel operator to compute image gradients, Gaussian blur for smoothing, and computes a corner response map. This map is later used to pick corner points using a threshold.

Optical Flow (OF) is the distribution of apparent velocities of objects in an image. Our implementation uses the Horn-Schunck method [19] to iteratively solve for the estimates of optical flow from one image to another, assuming that it is smooth over the entire image. This benchmark involves a total of 15 stages in the pipeline, out of which the last five stages are run for four iterations.

Downsampling and Upsampling (DUS): The original image is sampled down to half the original resolution along both the dimensions. The smaller image thus obtained is interpolated to get back an approximation of the original.

5.2 Experimental Setup

The FPGA results were obtained using the Xilinx Vivado Design Suite version 2015.2 for a Xilinx Virtex-7 690T FPGA. The code for the algorithm specification was synthesized using Vivado HLS, and Post Place and Route (PPnR) characteristics were obtained using the Vivado Design Suite 2015.2. For the evaluation, we used 24-bit floating point data types, where 12 bits are used to store the fractional part and the remaining 12 bits for the integral part. Images used are color images (3-channel RGB) of size 1920 x 1080, except for Harris corner detection, which takes a grayscale image as input. The throughput and latency results reported in this section are obtained by post place and route simulations using the Xilinx Vivado tool.

5.3 Performance Analysis

Table 2 shows resource usage of the generated designs on a Virtex-7 FPGA. Results with one and four PEs are shown to demonstrate scalability with more PEs. In Table 3, the reported designs have processing elements that are 3-way parallel in addition — to compute on all three channels of an image concurrently. On the other hand, designs of Table 2 are for grayscale images or a single channel. Note that several image processing algorithms (like HC) operate on grayscale input. The results also show the amount of external I/O bandwidth needed. Assuming a 16-lane PCI-express 3.0 interface, the peak theoretical bandwidth available is 15.76 GB/s (16 * 985 MB/s). If we use a sustainable bandwidth estimate of 9.9 GB/s (measured using NVIDIA CUDA’s bandwidth test sample), we observe that the amount of bandwidth that can be sustained in each direction is four times the bandwidth required for the listed designs (in Table 2), i.e., 16 PEs can be accommodated for each of the designs reported. Similarly, for the designs in Table 3, five PEs can be accommodated. We also note that these designs utilize a small fraction of the resources of a modern FPGA such as Virtex-7 690T (cf. Table 2 last column). Hence, the entire off-chip band-

### Table 1: Benchmarks: number of stages in pipeline and lines of code in PolyMage DSL

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Stages</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC</td>
<td>11</td>
<td>43</td>
</tr>
<tr>
<td>USM</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>OF</td>
<td>15</td>
<td>75</td>
</tr>
<tr>
<td>DUS</td>
<td>4</td>
<td>50</td>
</tr>
</tbody>
</table>
Table 2: Post place and route results comparing resource requirements and throughput achieved when using a single PE and when using 4 PEs, where each PE processes a single channel.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clock period (nanoseconds)</th>
<th>Latency (milliseconds)</th>
<th>Slices</th>
<th>LUTs</th>
<th>BRAM slices</th>
<th>DSP slices</th>
<th>Throughput (MPixels/s)</th>
<th>Bandwidth (MB/s)</th>
<th>Virtex-7 690T slices used</th>
</tr>
</thead>
<tbody>
<tr>
<td>USM</td>
<td>2.990 3.643</td>
<td>18.6 5.85</td>
<td>352</td>
<td>2,500</td>
<td>751</td>
<td>16</td>
<td>115</td>
<td>357</td>
<td>345 1071 2.30%</td>
</tr>
<tr>
<td>HC</td>
<td>2.880 3.429</td>
<td>5.95 1.83</td>
<td>1,161</td>
<td>5,422</td>
<td>1,902</td>
<td>22</td>
<td>56</td>
<td>21</td>
<td>60 351 1133 5.00%</td>
</tr>
<tr>
<td>OF</td>
<td>3.208 3.827</td>
<td>19.9 6.09</td>
<td>1,923</td>
<td>8,386</td>
<td>3,928</td>
<td>46</td>
<td>60</td>
<td>28</td>
<td>112 108 345 7.74%</td>
</tr>
<tr>
<td>DUS</td>
<td>3.047 3.446</td>
<td>19.0 5.39</td>
<td>401</td>
<td>2,332</td>
<td>811</td>
<td>12</td>
<td>20</td>
<td>0</td>
<td>0 115 386 2.17%</td>
</tr>
</tbody>
</table>

Table 3: Post place and route results comparing resource requirements and throughput achieved when using a single PE, and when using 2 PEs, where each PE processes 3 channels in parallel; throughput is in MegaPixels/sec.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clock period (ns)</th>
<th>Latency (ms)</th>
<th>Slices</th>
<th>LUTs</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>USM</td>
<td>3.127</td>
<td>3.182</td>
<td>2.48</td>
<td>3.32</td>
<td>1,279</td>
<td>2,431</td>
<td>2,794</td>
<td>6,342</td>
</tr>
<tr>
<td>OF</td>
<td>3.335</td>
<td>3.27</td>
<td>6.48</td>
<td>3.41</td>
<td>6,063</td>
<td>13,023</td>
<td>12,128</td>
<td>28,022</td>
</tr>
<tr>
<td>DUS</td>
<td>2.855</td>
<td>3.126</td>
<td>5.92</td>
<td>3.36</td>
<td>1,277</td>
<td>2,572</td>
<td>2,681</td>
<td>6,177</td>
</tr>
</tbody>
</table>

Table 4: Comparison with Xilinx’s OpenCV library for Vivado HLS.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Xilinx OpenCV</th>
<th>PolyMage-FPGA (1 PE)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC</td>
<td>8.67</td>
<td>2,180</td>
<td>1.45×</td>
</tr>
<tr>
<td>USM</td>
<td>10.23</td>
<td>6,936</td>
<td>1.59×</td>
</tr>
</tbody>
</table>

Table 5: Estimated performance (execution time) on a Xilinx Virtex-7 690T FPGA (with PCIex 3.0 connectivity) and comparison with an optimized and tuned reference implementation on a 16-core (2-socket) Intel Xeon E5-2680 (SandyBridge).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PolyMage-CPU-naive</th>
<th>PolyMage-CPU-opt</th>
<th>PolyMage-FPGA</th>
<th>Estimated speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC</td>
<td>35.25 ms</td>
<td>15.39 ms</td>
<td>0.53 ms</td>
<td>2.85×</td>
</tr>
<tr>
<td>USM</td>
<td>36.92 ms</td>
<td>19.12 ms</td>
<td>1.62 ms</td>
<td>1.06×</td>
</tr>
<tr>
<td>OF</td>
<td>303.6 ms</td>
<td>38.96 ms</td>
<td>2.50 ms</td>
<td>15.60×</td>
</tr>
<tr>
<td>DUS</td>
<td>16.07 ms</td>
<td>16.23 ms</td>
<td>1.40 ms</td>
<td>1.05×</td>
</tr>
</tbody>
</table>

Figure 9: Throughput obtained with various FPGA designs (in million pixels/s).
width is saturated before all on-chip FPGA resources can be utilized, i.e., the designs are bandwidth-bound.

5.4 Comparison with Xilinx OpenCV Library

We also compare with Xilinx’s OpenCV library for Vivado HLS. The Xilinx OpenCV library is expected to have manually optimized and tuned FPGA implementations of commonly used image processing functions, and it thus serves as an optimized reference implementation for FPGAs. We were able to perform this comparison for Harris Corner Detection and Unsharp Mask – for the former, the library includes a routine, while for the latter, we had to write it by composing simpler routines from the library. Table 4 shows the comparison of the designs; we use the single PE version of PolyMage-FPGA to provide a level comparison with respect to chip resources used. We observe that designs generated by us even outperform such pre-optimized library routines; these routines are also known to optimize across stencils kernels in the pipeline. Without access to the sources and the underlying implementation details, we are unable to comment on the reasons for the better performance.

5.5 Comparison with Multicore CPUs

In order to place the performance of these FPGA designs in relation to general-purpose multicores, we compare with highly optimized and tuned codes generated by PolyMage for general-purpose multicore systems. This comparison is shown in Table 5. PolyMage generated code for the CPU was compiled using Intel C/C++ compiler 15.0 with flags “-O3 -xHost”, and executed on a dual socket 16-core Intel Xeon E5-2680 2.70 GHz (Intel SandyBridge-based) server. Note that the performance reported is the execution time per image in a steady-state mode where input and output are continuously being streamed in and out respectively. A trend that we observe is that the acceleration with FPGAs is higher for benchmarks with deeper pipelines — this is as expected. Significant acceleration is obtained for Harris corner and Optical flow benchmarks.

5.6 Summary

Figure 9 compares throughput obtained by various FPGA implementations expressed in millions of pixels processed per second. The results demonstrate the effectiveness of our approach. Although a direct experimental comparison with the approaches of Darkroom [15] or HIPAcc [16] is not possible, both of these approaches do not extract coarse-grained data parallelism that we do through multiple PEs; hence, we expect the performance obtained by Darkroom and HIPAcc to be of the order of our single PE case. A comparative discussion is provided in Section 6.

6. RELATED WORK

Halide [32], Darkroom [15], PolyMage [23], Forma [33], and HIPAcc [22, 16] are recent DSL compilation efforts aimed to improve productivity and simultaneously deliver high performance. While the Halide DSL uses a scheduling language that allows a programmer to obtain high performance code in a semi-automatic manner, Forma and PolyMage attempt to arrive at schedules completely automatically although using very different transformation strategies. None of these approaches consider targeting FPGAs. HIPAcc [16] focuses on programmability and the ability to target accelerators (GPUs and Xeon Phi), and provides relatively little detail on transformations that enhance reuse and parallelism. Although our approach provides a mapping and code generation strategy for FPGAs for PolyMage, the developed techniques are equally applicable to the other image processing DSLs. FPGA backends have been developed for Darkroom [15] and HIPAcc that we will compare with below.

Darkroom [15] is a recent effort on compiling image processing pipelines to FPGAs. The Darkroom compiler directly maps programs written in the Darkroom language into line-buffered pipelines with all intermediate data stored on chip. Although this approach extracts maximal reuse, it does not exploit degrees of data parallelism, either in a fine-grained or coarse-grained manner. Whenever a larger amount of external memory bandwidth is available via PCI-express or another interconnect, exploiting such additional parallelism will also lead to a greater utilization of available chip resources (slices and block RAM). For the experimental results reported by Hegarty et al. [15], although the designs therein were memory bandwidth bound, had much higher bandwidth been available, not all of it would have been utilized. In addition, Darkroom currently only supports compositions of pointwise and stencil type filters. In addition to these, our approach is able to deal with multiscale computations as well. The FPGA backend support for HIPAcc [34] targets a C-based HLS like ours, but has the same limitations that we mentioned for Darkroom above.

There have been several HLS approaches that are able to apply high-level compiler transformations including loop transformations such as loop fusion, tiling, unrolling, and pipelining to aid HLS. The key optimization that drives the domain-specific approach presented in this paper is that of overlapped tiling of multiple fused stages by performing redundant computation at the boundaries. The idea of performing redundant computation at the boundaries to avoid communication or cache misses, or to aid local buffering has been used in several automatic code generation approaches including [17, 39, 32, 23] for CPUs or GPUs. Such an optimization cannot be realized as a composition of loop fusion and other traditional loop transformations performed by existing FPGA compiler approaches like ROCCC [13], StreamC [12], SA-C [24] and many others [35, 9, 36, 5, 1, 31]. The redundant computation introduced at the boundaries is necessary to break the dependence between two adjacent tiles, achieving both coarse-grained parallelization and locality after fusion. In addition, memory for the tiles need to be re-allocated for overlapped tiling – it is tedious to analyze or perform such a transformation starting from C code or any other languages where memory / memory allocation has not been abstracted away from the programmer. PolyMage input leaves memory allocation completely to the compiler – one of our motivations behind using a DSL. Without the overlapped tiling, the fused loops for a sequence of stencils would be sequential, and coarse-grained data parallelism cannot be exploited.

There are more generic compiler-based approaches to address problems associated with design space exploration and mapping to FPGAs or accelerators with similar resource constraints [9, 35, 11, 36, 5, 1, 31]. Alias et al. [1] address generation of off-chip communication code to maximize reuse on the FPGA and hide communication costs by pipelining. Their approach primarily deals with precise determination of communication sets (to maximize on-chip reuse), once a loop tiling and other polyhedral transformations have been
applied for a coarse-grained mapping. Pouchet et al. [31] developed a design space exploration approach by leveraging polyhedral loop transformations – effectively studying the impact of loop transformations on factors such as communication volume and on-chip memory utilization, and thus reuse. In contrast to all of these works that are more generic, our approach is domain-specific and customized not just to the computation patterns in context but to the particular parallelization strategy employed. For example, a standard loop tiling does not involve redundant computation while our mapping approach does exploit it to provide a better mapping for parallelism, locality, and simplified on-chip scratchpad access. A domain-specific approach thus simplifies the mapping choice and code generation, as is usually the case with other efforts on high-performance DSL compilation.

NI LabView [10] and MATLAB HDL coder [21] are frameworks that provide a high-level design language with an interactive environment to target FPGAs, besides other architectures. Several OpenCL-based and CUDA-based HLS frameworks [26, 28] exist to compile explicitly parallel languages to VHDL/Verilog. Although all of these HLS frameworks aim to improve programmability, we found C-based HLS tools that provided dataflow style primitives to be the right abstraction for our DSL code generator to target. Although C is used as input by such HLS tools, we have been able to easily realize dataflow style designs (as opposed to Von Neumann style architectures) by using the right constructs and directives (in this case provided by Xilinx Vivado) through our DSL approach.

There has been work on compiling high-level streaming languages or streaming applications to streaming hardware designs on FPGAs [18, 14]. The space of optimization and design choices is different from that involved in image processing since streams are one-dimensional while image processing pipelines involve parallelism and reuse along typically two dimensions.

7. CONCLUSIONS

We described a domain-specific high-level synthesis (HLS) approach to accelerate image processing pipelines using FPGAs. Our approach is implemented in the compiler for the PolyMage domain-specific language and uses existing vendor HLS as its target. Our mapping technique exploits all available reuse and maximal parallelism under the constraints of available off-chip memory bandwidth and chip resources. When compared to other state-of-the-art approaches to map image processing pipelines to FPGAs, our approach (a) leads to designs that deliver significantly higher throughput, and (b) supports a greater variety of image processing filters than the state-of-the-art. Furthermore, our designs perform even faster (by a factor of $1.5 \times$ on average) than corresponding pre-optimized FPGA implementations from vendor-supplied libraries.

8. REFERENCES

[7] Creative Commons Attribution 3.0 license (CC BY 3.0). https://creativecommons.org/licenses/by/3.0/.
[8] Creative Commons Attribution-ShareAlike 3.0 license (CC BY-SA 3.0). https://creativecommons.org/licenses/by-sa/3.0/.


