Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines

James Hegarty John Brunhaver Zachary DeVito Jonathan Ragan-Kelley[†] Noy Cohen Steven Bell
Artem Vasilyev Mark Horowitz Pat Hanrahan

Stanford University [†]MIT CSAIL

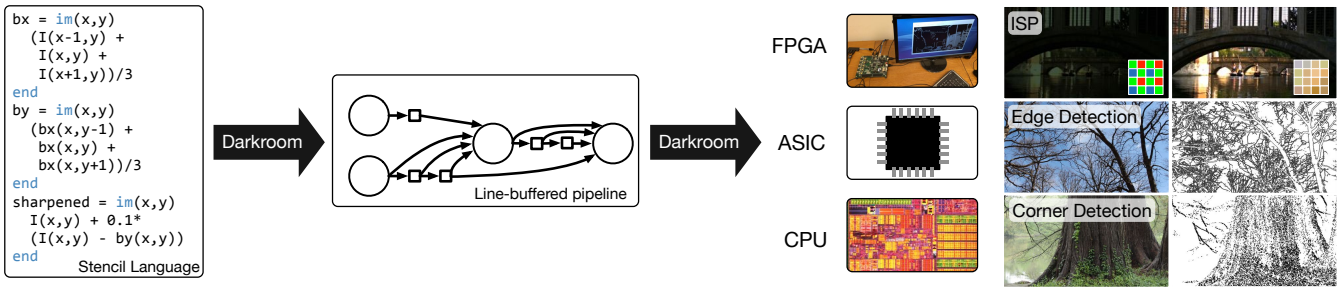


Figure 1: *Our compiler translates programs written in a high-level language for image processing into a line-buffered pipeline, modeled after optimized image signal processor hardware, which is automatically compiled to an ASIC design, or code for FPGAs and CPUs. We implement a number of example applications including a camera pipeline, edge and corner detectors, and deblurring, delivering real-time processing rates for 60 frames per second video from 480p to 16 megapixels, depending on the platform.*

Abstract

Specialized image signal processors (ISPs) exploit the structure of image processing pipelines to minimize memory bandwidth using the architectural pattern of *line-buffering*, where all intermediate data between each stage is stored in small on-chip buffers. This provides high energy efficiency, allowing long pipelines with tera-op/sec. image processing in battery-powered devices, but traditionally requires painstaking manual design in hardware. Based on this pattern, we present Darkroom, a language and compiler for image processing. The semantics of the Darkroom language allow it to compile programs directly into line-buffered pipelines, with all intermediate values in local line-buffer storage, eliminating unnecessary communication with off-chip DRAM. We formulate the problem of optimally scheduling line-buffered pipelines to minimize buffering as an integer linear program. Finally, given an optimally scheduled pipeline, Darkroom synthesizes hardware descriptions for ASIC or FPGA, or fast CPU code. We evaluate Darkroom implementations of a range of applications, including a camera pipeline, low-level feature detection algorithms, and deblurring. For many applications, we demonstrate gigapixel/sec. performance in under 0.5mm^2 of ASIC silicon at 250 mW (simulated on a 45nm foundry process), real-time 1080p/60 video processing using a fraction of the resources of a modern FPGA, and tens of megapixels/sec. of throughput on a quad-core x86 processor.

CR Categories: B.6.3 [Logic Design]: Design Aids—Automatic Synthesis; I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages; I.4.0 [Image Processing and Computer Vision]: General—Image Processing Software

Keywords: Image processing, domain-specific languages, hardware synthesis, FPGAs, video processing.

Links: DL PDF WEB

1 Introduction

The proliferation of cameras presents enormous opportunities for computational photography and computer vision. Researchers are developing ways to acquire better images, including high dynamic range imaging, motion deblurring, and burst-mode photography. Others are investigating new applications beyond photography. For example, augmented reality requires vision algorithms like optical flow for tracking, and stereo correspondence for depth extraction. However, real applications often require real-time throughput and are limited by energy efficiency and battery life.

To process a single 16 megapixel sensor image, our implementation of the camera pipeline requires approximately 16 billion operations. In modern hardware, energy is dominated by storing and loading intermediate values in off-chip DRAM, which uses over 1,000× more energy than performing an arithmetic operation [Hameed et al. 2010]. Simply sending data from mobile devices to servers for processing is not a solution, since wireless transmission uses 1,000,000× more energy than a local arithmetic operation.

Often the only option to implement these algorithms with the required performance and efficiency is to build specialized hardware. Image processing on smartphones is performed by hardware image signal processors (ISPs), implemented as deeply pipelined custom ASIC blocks. Intermediate values in the pipeline are fed directly

between stages. This pattern is often called *line-buffering*. The combination of many arithmetic operations with low memory bandwidth leads to a power efficient design. Performing image processing in specialized hardware is at least 500× more power efficient than performing the same calculations on a CPU [Hameed et al. 2010].

However, implementing new image processing algorithms in hardware is extremely challenging and expensive. In traditional hardware design languages, optimized designs must be expressed at an extremely low level, and are dramatically more complex than equivalent software. Worse, iterative development is hamstrung by slow synthesis tools: compile times of hours to days are common. Because of this complexity, designing specialized hardware, or even programming FPGAs, is out of reach to most developers. In practice, most new algorithms are only implemented on general purpose CPUs or GPUs, where they consume too much energy and deliver too little performance for real-time mobile applications.

In this paper, we present a new image processing language, Darkroom, that can be compiled into ISP-like hardware designs. Similar to Halide and other languages [Ragan-Kelley et al. 2012], Darkroom specifies image processing algorithms as functional DAGs of local image operations. However, while Halide's flexible programming model targets general-purpose CPUs and GPUs, in order to efficiently target FPGAs and ASICs, Darkroom restricts image operations to static, fixed size windows, or *stencils*. As we will show, this allows Darkroom to automatically schedule programs written in a clean, functional form into line-buffered pipelines using minimal buffering, and to compile into efficient ASIC and FPGA implementations and CPU code.

This paper makes the following contributions:

- We demonstrate the feasibility of compiling high-level image processing code directly into efficient hardware designs.
- We formalize the optimization of ISP-like line-buffered pipelines to minimize buffer size as an integer linear program. It computes optimal buffering for real pipelines in < 1 sec.
- We demonstrate back-ends that automatically compile line-buffered pipelines into structural Verilog for ASICs and FPGAs. For our camera pipeline and most tested applications, the generated ASICs are extremely energy efficient, requiring < 250 pJ/pixel (simulated on a 45nm foundry process), while a mid-range FPGA runs them at 125-145 megapixels/sec. On our applications, the generated ASICs and FPGA designs use less than 3× the optimal buffering.
- We also show how to compile efficiently to CPUs. Our results are competitive with optimized Halide, but require seconds to schedule instead of hours. We also show performance 7× faster than a clean C implementation of similar complexity.

2 Background

Camera ISPs process raw data from a camera sensor to produce appealing images for display. Most camera sensors record only one color per pixel, requiring other channels at each pixel to be estimated from its neighbors ("demosaicing"). ISPs also correct noise, optical aberrations, white balance, and perform other enhancements (Fig. 2).

Camera ISPs are typically implemented as fixed-function ASIC pipelines. Each clock cycle, the pipeline reads one pixel of input from the sensor or memory, and produces one pixel of output. ISPs are extremely deep pipelines: there are many stages, and a long delay between when a pixel enters the pipeline and when it leaves.

ISP pipelines contain two types of operation. One type only operates on single pixels, such as gamma correction. We call these *pointwise*,

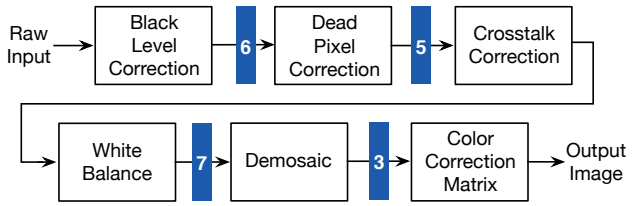


Figure 2: *Camera ISPs are pipelines of many image operations. Some stages access a single point, while others access a window around the output pixel, or* stencil *(indicated by blue boxes). No stages access arbitrary pixels from the image.*

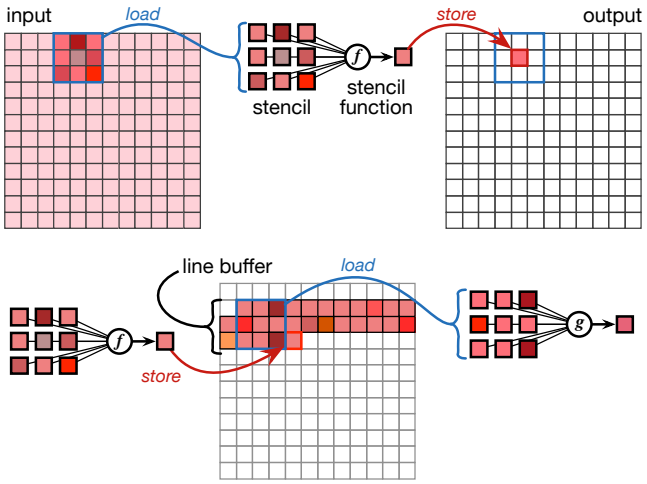


Figure 3: *Each stage of the ISP is a pure function of its stencil input and (x, y) position (top). Stencil stages communicate through a line buffer (bottom), a local SRAM that holds a sliding window of the data produced by the previous stage. The required buffer size is determined by the size of the stencil consuming it. Here, it must be large enough to store all values needed for the next iteration of* g.

because they only operate on a single point at a time. The second type needs a window of input to produce a single output pixel. For example, demosaicing needs multiple adjacent pixels to interpolate color information. We call these operations *stencils*, because they match the well-understood structure of stencil computations.

Pointwise operations do not require buffering, because the input required (a single pixel) is exactly what is produced by the prior stage. In contrast, stencil operations require multiple pixels of input from the previous stage. For example, a stencil operation could require the pixel at $(x, y-1)$ when producing the output for (x, y). However, the input stage would have produced $(x, y-1)$ W clock cycles ago, where W is the width of the sensor. To provide this data, an ISP would buffer W pixels of the input stage's results on-chip. Using standard terminology from ISP design, we call this buffer a *line-buffer*, because it buffers lines of the input image (Fig. 3).

Each operation in the ISP is a pure function of its input stencil and the (x, y) position of the pixel it is computing. This means that they can be pipelined and parallelized to an arbitrary performance target using standard techniques [Leiserson and Saxe 1991].

Camera ISPs perform an enormous amount of computation on a large amount of data. (Our camera pipeline performs roughly 1000 arithmetic operations/pixel, or 120 Gigaops/sec. for 1080p/60 video.) Combined with the fact that ISPs are used in battery-constrained mobile devices, this means that energy efficiency is crucial.

Recent work has shown that a general purpose CPU uses 500× the energy of a custom ASIC for video decoding [Hameed et al. 2010]. While the CPU can be improved, the majority of the ASIC advantage comes from using a long pipeline that performs more arithmetic per byte loaded. On a modern process, loading one byte from off-chip DRAM uses 6400× the energy of a 1 byte add; even a large cache uses 50× the energy of the add [Malladi et al. 2012; Muralimanohar and Balasubramonian 2009]. ISPs are ideally tuned to these constraints: they achieve high energy efficiency by performing a large number of operations per input loaded, and exploiting locality to minimize off-chip DRAM bandwidth. Existing commercial ISPs use less than 200mW to process HD video [Aptina].

3 Programming Model

Based on the patterns exploited in ISPs, we define the Darkroom programming language. Its programming model is similar to prior work on image processing, such as Popi, Pan, and Halide [Holzmann 1988; Elliott 2001; Ragan-Kelley et al. 2012]. Images at each stage of computation are specified as pure functions from 2D coordinates to the values at those coordinates, which we call *image functions*. Image functions are defined over all integer coordinates (x, y), though they can be explicitly *cropped* to a finite region using one of several boundary conditions.

In our notation, image functions are declared using a *lambda*-like syntax, `im(x,y)`. For example, a simple brightening operation applied to the input image `I` can be written as the function:

```
brighter = im(x,y) I(x,y) * 1.1 end
```

To implement stencil operations such as convolutions, Darkroom allows image functions to access neighboring pixels:

```
convolve = im(x,y) (1*I(x-1,y)+2*I(x,y)+1*I(x+1,y))/4 end
```

To support operations like warps, Darkroom has an explicit gather operator. Gathers allow dynamically computed indices, but must be explicitly bounded within a certain compile-time constant distance from the current (x, y) position. For example, the following performs a nearest-neighbor warp on `I`:

```
warp = im(x,y) gather(I, 4, 4, warpVX(x,y), warpVY(x,y)) end
```

`warpVX` and `warpVY` can be arbitrary expressions that will be clamped to the range $[-4, 4]$.

Compared to prior work, we make the following restrictions to fit within the line-buffered pipeline model:

1. Image functions can only be accessed (1) at an index $(x + A, y + B)$ where A,B are constants, or (2) with the explicit gather operator. Affine indices like `I(x*2,y*2)` are not allowed. This means that every stage produces and consumes pixels at the same rate, a restriction of line-buffered pipelines.
2. Image functions cannot be recursive, because this could force a serialization in how the image is computed. This makes it impossible to implement inherently serial techniques inside a pipeline.

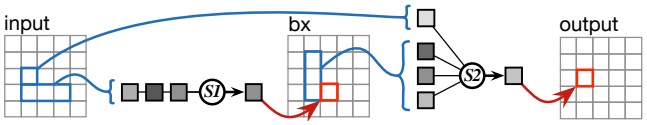
A simple pipeline in Darkroom

Let us look at a simple example program in Darkroom. The unsharp mask operation sharpens an image I by amplifying the difference between it and a blurred copy to enhance high frequencies. Implementing the 2D blur as separate 1D passes, we could write the pipeline as:

```
bx = im(x,y) (I(x-1,y) + I(x,y) + I(x+1,y))/3 end
by = im(x,y) (bx(x,y-1) + bx(x,y) + bx(x,y+1))/3 end
difference = im(x,y) I(x,y)-by(x,y) end
scaled = im(x,y) 0.1 * difference(x,y) end
sharpened = im(x,y) I(x,y) + scaled(x,y) end
```

The final three image functions—`difference`, `scaled`, and `sharpened`—are pointwise operations, so the whole pipeline can be collapsed into two stencil stages:

```
S1 = im(x,y) (I(x-1,y) + I(x,y) + I(x+1,y))/3 end
S2 = im(x,y)
       I(x,y) + 0.1*(I(x,y)-(S1(x,y-1) + S1(x,y) + S1(x,y+1))/3)
     end
```



It cannot be collapsed any further without changing the stencils of the individual computations. Notice that this is not a linear pipeline, but a general DAG of operations communicating through stencils. In this example, the final sharpened result is composed of stencils over both the horizontally-blurred intermediate, and the original input image.

4 Generating Line-buffered Pipelines

Given a high-level program written in Darkroom, we first transform it into a *line-buffered pipeline*. This pipeline processes input in time steps, one pixel at a time. During each time step, it consumes one pixel of input, and produces one pixel of output. The pipeline can contain both *combinational* nodes that perform arithmetic, and *line buffers* that store intermediate values from the previous time step.

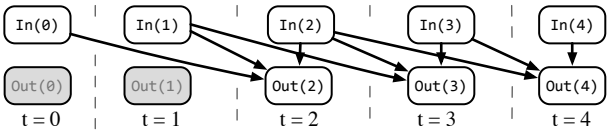
Fig. 4 (a) shows code for a simple 1-D convolution of an input `In` with a constant kernel k_i. From this code, we can create a pipeline such that at time $t = 0$, we read the value for `In(0)` and produce a value for `Out(0)`, illustrated in Fig. 4 (b). Current values such as `In(0)` can be wired directly to their consumers. Values from the past such as `In(x - 2)` are stored as entries in an N pixel shift register, known as a *line buffer*. Fig. 4 (c) shows the pipeline that results from our simple example.

This model can also handle two dimensional stencils by reducing them to one dimension, flattening the image into a continuous stream of pixels generated from concatenated lines. Given a fixed line size L, accesses $f(x+c_1, y+c_2)$ are replaced with $f'(x' + c_1 + L * c_2)$

(a) A 1-D convolution of `In` against a constant kernel k:

```
Out = im(x) k0*In(x) + k1*In(x-1) + k2*In(x-2) end
```

(b) An illustration of how this convolution would execute over time:



(c) A pipeline that implements this execution. Square nodes are pixel buffers whose output is their input from the *previous* time step. Two pixel buffers are required because `Out` accesses values of `In` two cycles in the past. This collection of pixel buffers forms a *line buffer*:

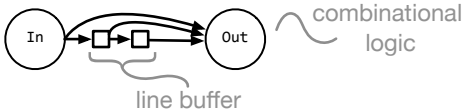
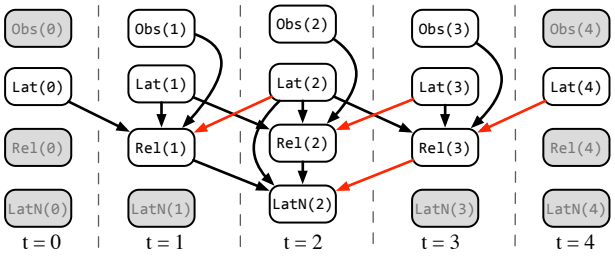


Figure 4: *Translation of a simple convolution stencil into a line-buffered pipeline.*

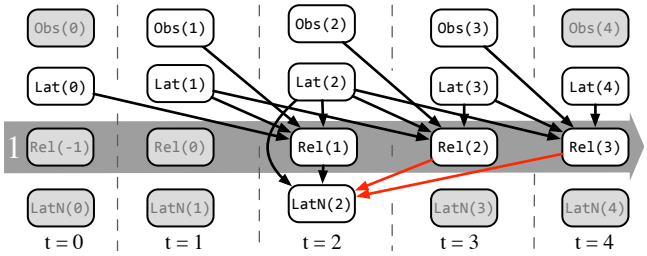
(a) Code for a 1D Richardson-Lucy deconvolution:

```
Rel = im(x) Obs(x) / (k0*Lat(x-1) + k1*Lat(x) + k2*Lat(x+1)) end
LatN = im(x) Lat(x)*(k2*Rel(x-1) + k1*Rel(x) + k0*Rel(x+1)) end
```

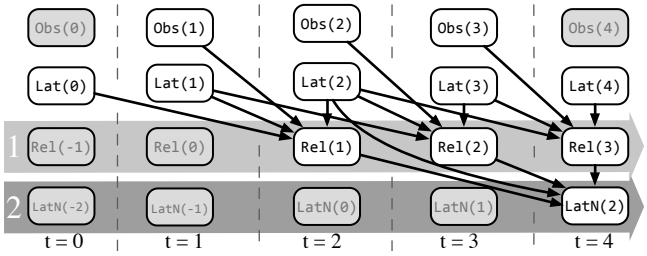
(b) Scheduling this code naively results in a non-causal pipeline—it has accesses into the future (shown in red):



(c) We can shift a value temporally to eliminate hazards. Here we have shifted `Rel` by 1:



(d) This operation can introduce hazards later in the pipeline, which can be fixed by later shifts:



(e) After eliminating hazards, we can construct a pipeline using line buffers to store previous values. Here is a correct pipeline for deconvolution:

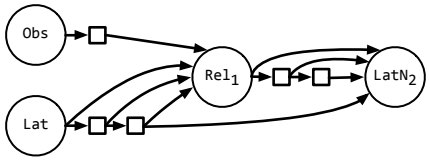


Figure 5: *We can use shifts to make a non-causal pipeline realizable.*

where $x' = x + L * y$ is the current pixel in the stream. For the remainder of the section, we will assume that the input code has already been transformed in this way.

So far, we have only handled stencils that access data from the current time step or the past. In signal processing these are referred to as *causal* filters. Fig. 5 (a) shows an example program where this is not the case. It performs a Richardson-Lucy deconvolution, taking as input the latent estimate of the deconvolved image `Lat`, the blurred input image `Obs`, and constants k_i which describe the point spread function of the blur. It calculates the relative error `Rel` of the latent estimate, and produces an improved latent estimate `LatN`.

If we naively translate this code into a pipeline as we did before, at time t, we would calculate the value of each intermediate f at position t. For instance, when $t = 0$, we would calculate `Lat(0)` and `Est(0)`. But for this example, we cannot compute `Rel(0)` since it depends on `Lat(1)`, which is not calculated until $t = 1$. This problem is a read after write hazard, illustrated in Fig. 5 (b).

We can transform non-causal pipelines like this into causal ones by shifting the time at which values are calculated relative to others. We first introduce the shift operator, and then discuss how to choose shifts that ensure causality and minimize line buffering.

4.1 Shift Operator

In our example, if we want to ensure `Rel` only relies on *current* or *previous* values of its input, we can *shift* it in time to eliminate the hazard. We define a shift operator for an integer shift s:

$$f_s(x) = f(x - s)$$

That is, at time step $t = s$, f_s will produce the value $f(0)$. We can now replace uses of a value with the equivalent shifted value. For instance, we replace `Rel` with $\texttt{Rel}_1$ and adjust the offsets:

```
Rel₁ = im(x)
         Obs(x-1) / (k0*Lat(x-2) + k1*Lat(x-1) + k2*Lat(x))
       end
```

Now all uses of `Lat` are previous values. We also need to adjust all the uses of `Rel` to be in terms of $\texttt{Rel}_1$:

```
LatN = im(x)
         Lat(x)*(k2*Rel₁(x) + k1*Rel₁(x+1) + k0*Rel₁(x+2))
       end
```

The effect of this shift is visualized in Fig. 5 (c). Note that in this case it introduced an additional hazard. We can also shift `LatN` by 2, which results in this modified program that contains no hazards:

```
Rel₁ = im(x)
         Obs(x-1) / (k0*Lat(x-2) + k1*Lat(x-1) + k2*Lat(x))
       end
LatN₂ = im(x)
          Lat(x-2)*(k2*Rel₁(x-2) + k1*Rel₁(x-1) + k0*Rel₁(x))
        end
```

Fig. 5 (d) illustrates how this pipeline will execute, and (e) shows the result of translating it into a line-buffered pipeline. As before, values accessed at the same time are piped directly to each other while values accessed in the past are implemented by inserting buffers. Calculating the original function (`LatN`) given a shifted pipeline that executes it ($\texttt{LatN}_2$) simply requires changing the indices that are calculated each cycle, e.g. $\texttt{LatN}_2\texttt{(2)}$ at $t = 0$ instead of `LatN(0)`. Similarly, evaluating shifted leaf nodes such as inputs from DRAM or the sensor simply requires shifting which address is read.

4.2 Finding optimal shifts

Despite being correct, this pipeline is not optimal: there is an unnecessary line buffer after `Obs` which would disappear if we choose to shift `Obs` by 1. To create an optimal pipeline, we must choose shifts which *both* ensure causality *and* minimize line buffer size.

The general case is complicated by the fact the program may have multiple inputs and multiple outputs (e.g., an RGB image and a separately-calculated depth map). Furthermore, individual line buffers are not always the same size. For instance, some values may be 1-byte greyscale while others might be 3-byte RGB triples. Fig. 6 shows an example of where different sized outputs can produce different scheduling decisions.

We can formulate this optimization as an integer linear programming problem. Let F be the set of image functions. For each value $p(x + d)$ evaluated in the process of evaluating $c(x)$, we generate a

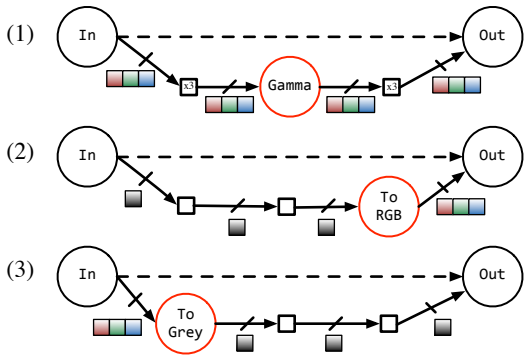


Figure 6: *Given some path between* `In` *and* `Out` *of a fixed length, the optimal placement of line buffers through the red node to match the fixed path depends on the size of the pixels entering and leaving it. If it produces the same amount of data (1), the choice does not matter, but if it produces more data (2) then buffers should be placed before it, while if it reduces data (3), they should be placed after it.*

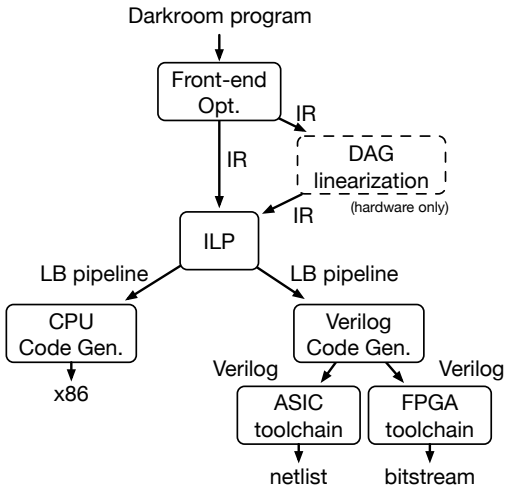


Figure 7: *The stages of the Darkroom compiler.*

use triple (c, p, d), where the consumer c and producer p are image functions, and d is an offset. Let U be the set of all uses in a program. For instance, the program:

```
Out = im(x)
        In(x - 1) + In(x) + In(x + 1)
      end
```

will result in the following values for F and U:

$$F = \{\texttt{Out}, \texttt{In}\}$$

$$U = \{(\texttt{Out}, \texttt{In}, -1), (\texttt{Out}, \texttt{In}, 0), (\texttt{Out}, \texttt{In}, 1)\}$$

Also, since the size of a pixel data type varies, we extract b_f, the pixel size of each image in bytes during typechecking.

For each image function f, we want to solve for its shift s_f such that we ensure causality and line buffer size is minimized. For each use (c, p, d) we calculate the number of delay buffers $n_{(c,p,d)}$ needed to store the value between when it is produced and when it is consumed as:

$$n_{(c,p,d)} = s_c - s_p - d$$

A negative number of delay buffers indicates a non-causal pipeline, so, to address causality, for each use we add the following constraint to the integer linear program:

$$n_{(c,p,d)} \geq 0$$

The line buffer for an image function f can be shared by all of its consumers, so for each image function $f \in F$, we calculate the size of its line buffer as the maximum number of delays needed by any consumer scaled by the pixel size, $(\max_{(c,f,d)\in U} n_{(c,f,d)}) * b_f$. The total size of the line buffers S is the sum of the line buffers for each producer:

$$S = \sum_{p \in F} (\max_{(c,p,d)\in U} n_{(c,p,d)}) * b_p$$

We use S as the objective to minimize in the integer linear program. This problem formulation is equivalent to the problem of minimizing register counts in circuit retiming literature. This problem can also be formulated as min-cost flow, which has a polynomial time solution [Leiserson and Saxe 1991].

5 Implementation

After generating an optimized line-buffered pipeline, our compiler instantiates concrete versions of the pipeline as ASIC or FPGA hardware designs, or code for CPUs (Fig. 7). The Darkroom compiler is implemented as a library in the Terra language [DeVito et al. 2013] that provides the `im` operator. When compiled, Darkroom programs are first converted into an intermediate representation (IR) that forms a DAG of high-level stencil operations. We perform standard compiler optimizations such as common sub-expression elimination and constant propagation on this IR. A program analysis is done on this IR to generate the ILP formulation of line buffer optimization, described in the previous section. We solve for the optimal shifts using an off-the-shelf ILP solver (lpsolve), and use them to construct the optimized pipeline [Berkelaar et al. 2004]. It converges to a global optimum in less than a second on all of our test applications. The optimized pipeline is then fed as input to either the hardware generator, which creates ASIC designs and FPGA code, or the software compiler, which creates CPU code.

5.1 ASIC & FPGA synthesis

ASIC and FPGA designs are traditionally developed as "structural" Verilog programs that instantiate combinational circuits, SRAMs, and the connections between them. FPGAs consist of a variety of configurable devices, including lookup tables (LUTs), SRAMs (known as block rams or BRAMs), and ALUs called "DSPs," which implement operations like multiplication, combined with a configurable interconnect network to wire them together.

Our hardware generator implements line buffers as circularly-addressed SRAMs or BRAMs. Each clock, a column of pixel data from the line buffer shifts into a 2D array of registers. These registers save the bandwidth of reading the whole stencil from the line buffer every cycle. The user's image function is implemented as combinational logic from this array of shift registers, writing into an output register (Fig. 8). We add small FIFOs to absorb stalls at the input and output of each line buffer.

Instantiating line buffers as real SRAMs presents additional difficulties beyond those present in our abstract pipeline model. First, SRAMs and BRAMs are only available in discrete sizes, each with different costs. Second, they have limited bandwidth, preventing multiple image functions reading from them simultaneously. To

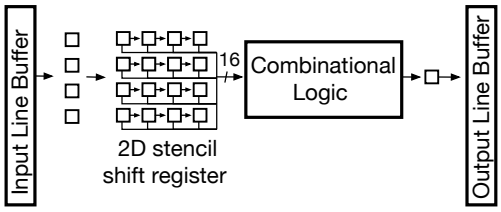
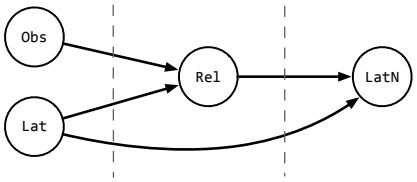
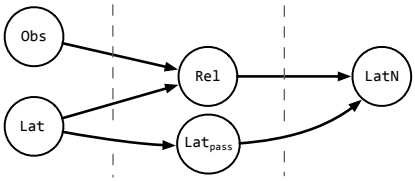


Figure 8: *Darkroom's hardware generators synthesize stages in a line-buffered pipeline using a common microarchitectural template. Columns of pixels are shifted from the input line buffer into a stencil register, processed through arbitrary computation in the data path, and pushed onto the output line buffer. According to the constraints of our programming model, the data path only has access to constants and the current contents of the stencil register.*

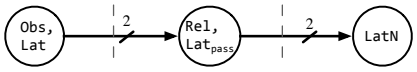
1. Group IR nodes by distance from the input:



2. Add passthrough nodes whenever an edge cross a stage:



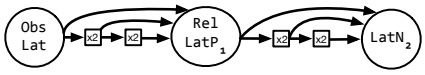
3. Merge values in each stage, producing larger pixel widths:



The transform results in the following code:

```
RelLatP = im(x)
    {ObsLat(x)[0] / (k0*ObsLat(x-1)[1] +
                     k1*ObsLat(x)[1] +
                     k2*ObsLat(x+1)[1]),
     ObsLat(x)[0]}
end
LatN = im(x)
    RelLatP(x)[1]*(k2*RelLatP(x-1)[0] +
                   k1*RelLatP(x)[0] +
                   k0*RelLatP(x+1)[0])
end
```

The code is then transformed into a pipeline:



Buffers contain merged values, so they are twice as large.

Figure 9: *Converting a generic program DAG into a linearized pipeline by merging nodes.*

simplify these issues in the first version of our hardware generator, we only support programs that are straight pipelines with one input, one output, and a single consumer of each intermediate.

Many Darkroom programs we have written contain a DAG of dependencies, where image functions have multiple inputs and multiple outputs. In order to support these programs in our hardware implementation, we first translate the Darkroom program into an equivalent Darkroom program that is a straight pipeline. This process is described in Fig. 9. While this process always produces semantically correct results, the merging of nodes in the programs can create larger line buffers than what could be achieved with a hardware implementation that supported DAG pipelines. In the future, we plan to multi-port the buffers to eliminate this restriction.

Following DAG linearization, we use Genesis2, a Verilog metaprogramming language [Shacham et al. 2012] to elaborate the topology into a SystemVerilog hardware description for synthesis. We verify functionality of the hardware description using Synopsys VCS G-2012.09, which also produces the activity factor required for ASIC power analysis. The ASIC design is synthesized and analyzed using Synopsys Design Compiler Topographical G-2012.06-SP5-1 for a 45nm cell library. The FPGA design uses Synopsys Synplify G-2012.09-SP1 to synthesize the design and Xilinx Vivado 2013.3 to place and route the design for the Zynq XC7Z045 system-on-a-chip on the Zynq 706 demo board. The FPGA performance is measured on the demo board using a custom Linux kernel module.

5.2 CPU compilation

Our CPU compiler implements the line-buffered pipeline as a multithreaded function. To enable parallelism, we tile the output image into multiple strips and compute each strip on a different core. Intermediates along strip boundaries are recomputed.

Within a thread, the code follows the line-buffered pipeline model. A simple approach is to have the thread's main loop correspond to one clock cycle of the hardware, with the individual operations scheduled in topological order to satisfy dependencies. However, the entire set of line buffers will often exceed the size of the fastest level of cache. We found that blocking the computation at the granularity of lines improved locality for this cache. The main loop calculates one line of each stencil operation with the line buffers expanded to the granularity of lines. In addition to keeping the line buffer values in the fastest level of the cache, this blocking reduces register spills in the inner loop by reducing the number of live induction variables. A stencil stage *S2* that consumes from *S1* yields the following code:

for each line y
  for each pixel x in line of *S1*
    compute $S1(x, y)$
  for each pixel x in line of *S2*
    compute $S2(x, y)$ *// loading S1 from line buffer*
  rotate line buffers

To exploit vector instructions available on modern hardware, we vectorize the computation within each line of each stage. For intermediates, we store pixels in struct-of-array form to avoid expensive gather instructions.

Line buffers are implemented using a small block of memory that we ensure stays in cache using the technique of Gummaraju and Rosenblum to simulate a scratchpad memory by restricting most memory access to this block and issuing non-temporal writes for our output images [2005]. We manage the modular arithmetic of the line buffers in the outer loop over the lines of an image so that each inner loop over pixels contains fewer instructions. For each line required of an intermediate we use one loop induction variable to track the current address in the line buffer for pixels from that line.

The compiler is implemented using Terra to generate low-level CPU code including vectors and threads, which is compiled and optimized using LLVM [Lattner and Adve 2004].

6 Results

To evaluate Darkroom, we implemented a camera pipeline (ISP), and three possible future extensions—CORNER DETECTION, EDGE DETECTION, and DEBLUR—in hardware. ISP, DEBLUR, and CORNER DETECTION are all stencil pipelines that map directly into Darkroom's programming model. EDGE DETECTION traditionally requires a long sequential iteration, which does not fit within the Darkroom model. Our implementation demonstrates that it is possible to work around some restrictions in our programming model, widening the range of applications we support at the cost of efficiency. All applications use fixed-point arithmetic for efficiency. Note also that, throughout this section, all pipelines are tested independently; on a real camera, each extension would likely be fed with the output from ISP. The input images and outputs from our test pipelines are shown in Figure 10.

ISP is a camera pipeline, including basic raw conversion operations (demosaicing, white balance, and color correction), in addition to enhancement and error correction operations (crosstalk correction, dead pixel suppression, and black level correction). Mapping ISP to Darkroom is straightforward: it is a linear pipeline of stencil operations, each of which becomes an image function. ISP is non-trivial, however, due to its size: it is 472 lines of Darkroom code, which must be scheduled, and compiled into hardware or software.

CORNER DETECTION is a classic corner detection algorithm [Harris and Stephens 1988], used as an early stage in many computer vision algorithms, and implemented as a series of local stencils.

EDGE DETECTION is a classic edge detection algorithm [Canny 1986]. It first takes a gradient of the image in x and y, classifies pixels as edges at local gradient maxima, and finally traces along these edge pixels sequentially. To implement this algorithm in Darkroom, we adapted the classic serial algorithm into a parallel equivalent, at the expense of some wasted computation and bounded information propagation. EDGE DETECTION traces along edges with a fixed pipeline of 10 3×3 stencil operations, each propagating connection information by one pixel. As a result, this implementation can only detect edges at most 10 pixels long.

DEBLUR is an implementation of the Richardson-Lucy non-blind deconvolution algorithm [Richardson 1972]. Much recent work suggests that adding deblurring capabilities to the camera pipeline could help improve photographs. DEBLUR is computationally-intense iterative algorithm, which we use as a stress test of our system. We unrolled DEBLUR to 8 iterations, which was the maximum size our hardware synthesis tools could support.

Additional applications To test the expressiveness of our language, we also implemented several additional algorithms which we evaluate on the CPU, but have not yet synthesized as hardware.

OPTICAL FLOW implements the Lucas-Kanade algorithm for dense optical flow [Lucas et al. 1981]. Optical flow serves as input to many higher-level computer vision algorithms. It is often implemented as a multi-scale algorithm, which uses an image pyramid to efficiently search a large area [Bouguet 2001]. Multi-resolution pyramid algorithms are not supported by Darkroom, so we have implemented the single-scale version that operates only at the finest resolution.

DEPTH FROM STEREO is a simple implementation of depth from stereo. First, it rectifies the left and right images based on camera calibration parameters, so that all correspondences are on the same horizontal line in the image. This resampling is accomplished using Darkroom's bounded gather, with the bound determined by the largest offset in the rectification map. Then, for each pixel in the left channel, it searches 80 horizontal pixels neighboring that point in the right channel, evaluating a 9x9 sum of absolute differences (SAD) between two. The correspondence with the lowest SAD gives the most likely depth. DEPTH FROM STEREO is a simple pipeline, but it performs an enormous amount of computation due to the large search window, testing our system's ability to cope with large image functions.

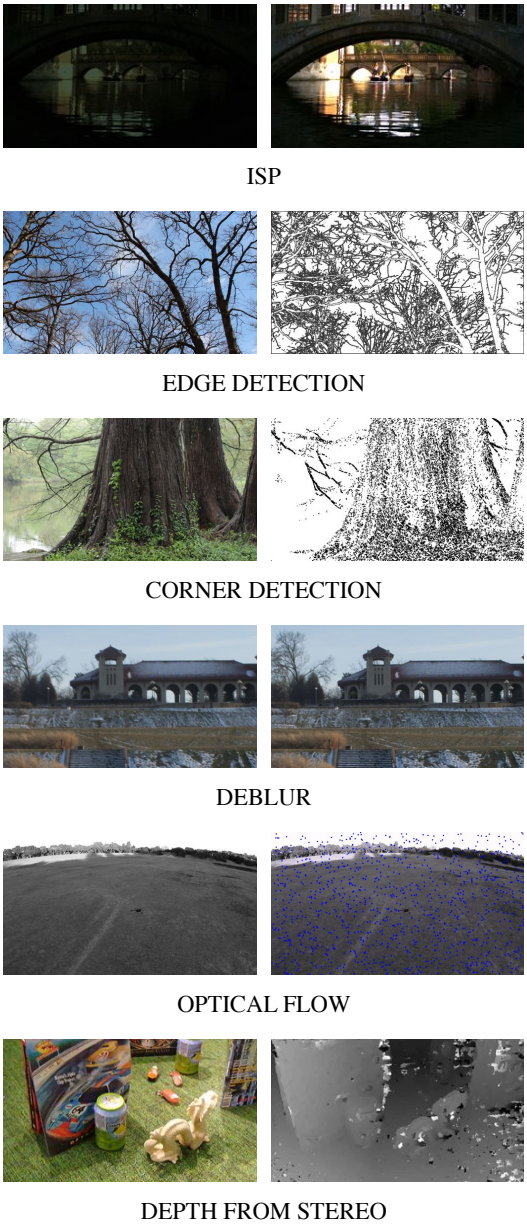


Figure 10: *To evaluate Darkroom, we implemented a camera pipeline and several extensions. Long pipelines test Darkroom's ability to analyze large programs. Acyclic pipelines test our scheduling algorithm. Multirate and serial pipelines test the overhead of programs that do not map directly into our programming model.*

6.1 Scheduling for hardware synthesis

We designed Darkroom primarily as a language for hardware synthesis. Using our system, we automatically scheduled, compiled, and synthesized ISP, EDGE DETECTION, CORNER DETECTION, and DEBLUR from relatively simple Darkroom code into real-time,

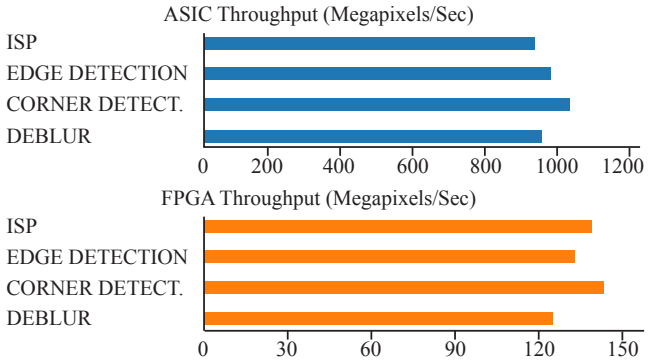


Figure 11: *Darkroom compiles our applications into efficient hardware implementations targeting a leading foundry's 45nm process (top), and a mid-range FPGA (bottom). In ASIC, a single pipeline achieves 940-1040 megapixels/sec, enough to process 16 megapixel images at 60 FPS. On the FPGA, a single-pixel pipeline achieves 125-145 megapixels/sec, enough to process 1080p/60 in real-time (124 megapixels/sec).*

	Energy (pJ/pix)		Area (mm^2)	
Pipeline	Compute	DRAM	Total	LB
ISP	224	1360	0.36	0.19
EDGE DETECTION	202	1040	0.29	0.17
CORNER DETECTION	165	1000	0.25	0.12
DEBLUR	1280	1920	2.56	1.10

Figure 12: *Darkroom compiles our applications into efficient ASIC, using a chip area of 0.3-2.6 mm^2 and energy efficiency of* $165-1280$ *pJ/pixel for compute, and* < 3.2 *nJ/pixel including the communication with DRAM. Processing 1080p/60 footage requires* < 30 *mW for most pipelines (ignoring fixed DRAM cost), similar to commercially available ISPs [Aptina]. At peak throughput, most pipelines can process 16 megapixel images at 60 frames per second using* < 210 *mW for synthesized logic.*

940-1040 megapixel/sec. implementations simulated on a leading foundry's 45nm ASIC process, and 125-145 megapixel/sec implementations on a mid-range FPGA. This throughput is sufficient to process 60 frame per second video at 16 megapixels on the ASICs, or 1920×1080 on the FPGA. At this performance, most ASIC implementations consume less than $0.36mm^2$ of die area and 210 mW of power for the synthesized hardware (Fig. 12), both of which are similar to hand-designed commercial ISP pipelines [Aptina]. Each FPGA implementation uses at most 59% of any critical resource on the FPGA, suggesting that it is feasible to prototype large imaging and vision pipelines on an FPGA platform (Fig. 13). DEBLUR uses significantly more area and energy than the other designs because it is a stress test which we unrolled to the maximum size our hardware synthesis tools will support.

ASIC

Efficient ASIC implementations are limited by both the energy and area cost of the synthesized circuit. Our ASIC implementations (Fig. 12) were synthesized using Synopsys Design Compiler. We see that the dominant area cost is memory and logic for line buffers. The computational logic and all other overhead uses at most half the total area. Most designs are able to process approximately 1 gigapixel/sec using approximately 200 mW of power for the entire synthesized pipeline. At this rate, the energy usage of each design is still dwarfed by the DRAM cost (60-86% of total energy

		Resource Utilization		
Pipeline	MPix/s	LUTs	BRAMs	DSPs
ISP	143	26%	7.5%	6.3%
EDGE DETECTION	131	7.2%	6.5%	3.9%
CORNER DETECTION	146	5.6%	4.8%	3.8%
DEBLUR	125	49%	59%	50%

Figure 13: *Darkroom compiles our applications well within the resource limits of a mid-range FPGA, while delivering enough performance from all pipelines to process 1080p/60 video in real-time. Resource utilization is reported as a percentage of the available resources on a Xilinx Zynq 7045. In practice, this platform provides enough resources to compile much larger pipelines, implementing multiple vision and image processing algorithms simultaneously in real-time.*

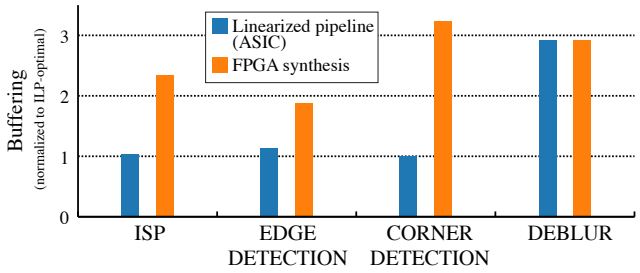


Figure 14: *Buffering efficiency of our synthesized hardware relative to the optimal line-buffered pipeline computed by* ILP *. Overhead in ASIC designs comes from our hardware backend's requirement that pipelines be linearized. FPGA designs introduce additional overhead due to the coarse granularity of BRAM allocation, combined with simplifying assumptions in our FPGA logic generator.*

in each pipeline), meaning there is room to build much deeper, more complex Darkroom pipelines without significantly changing the overall energy budget (e.g., in many of these applications, $10\times$ more pipeline energy per pixel would only double the total energy budget).

FPGA

Efficient mapping to FPGAs is limited by effective utilization of a fixed budget of heterogeneous resources. We ran our experiments on a Xilinx Zynq 7045, a mid-range FPGA platform costing $1000. On this platform, the critical resources were the look-up tables (LUTs) used to implement combinational logic, the block RAMs (BRAMs) used for on-chip local storage, and the "DSP" blocks which implement 24-bit integer ALUs (DSPs). Our scheduling algorithm optimizes the total amount of buffering throughout each pipeline, minimizing BRAM usage. LUT and DSP utilization are largely determined by the quantity and complexity of the mathematical operations in each pipeline.

As seen in Fig. 11, Darkroom achieves throughput over 125 megapixels/sec. , which allows it to process 1080p/60 video in real time. Each pipeline has been measured to produce 0.89-0.98 pixels/clock on average, indicating that they run near 100% utilization (i.e., they very rarely stall waiting for data). In each case, achieved throughput is determined by clock rate, which is limited by the physical design.

Buffering efficiency

Darkroom is built around the philosophy that efficiently using on-chip buffering is essential to minimizing expensive off-chip DRAM traffic. In practice, two effects limit the actual buffering efficiency

	Runtime (sec)		
	C++/GCC	Darkroom	Speedup
ISP	2.2	0.33	6.7×
	Halide autotuned	Darkroom	Speedup
DEBLUR (float)	0.37	0.35	1.1×

Figure 15: *On* ISP*, we compared Darkroom's performance to a reference C implementation of similar complexity which lacks vectorization, multithreading, and line buffering optimizations. Darkroom's scheduling algorithm and optimizations yield a 7× speedup over the C code. On* DEBLUR*, we compared Darkroom's performance to that delivered for the same algorithm written in and optimized by Halide. Darkroom's* ILP *finds the optimal line-buffered schedule in under 1 second for all applications, while the Halide autotuner required 8 hours to find a comparably perfoming schedule.*

	Throughput (MPix/sec)		Buffering (kB)	
Pipeline	1 core	4 cores	Optimal	Achieved
ISP	7.5	24	427	436
EDGE DETECTION	12	34	224	228
CORNER DETECTION	78	148	108	110
DEBLUR	4	14	1596	1622
OPTICAL FLOW	7.8	22	1404	1431
DEPTH FROM STEREO	0.067	0.25	108	122

Figure 16: *Darkroom programs compiled to a quad-core x86 CPU deliver throughput sufficient to process 720p/24 video on most applications, or as high as 1080p/60 for* CORNER DETECTION*.* DEPTH FROM STEREO *delivers a much lower pixel rate than other applications because it performs dramatically more arithmetic; its performance is proportional to the difference in arithmetic per-pixel. Allocated line buffer storage is near-optimal in all cases.*

of our synthesized hardware designs relative to the optimal line-buffering schedule computed by ILP :

1. Our hardware generators currently require the acyclic graphs to be flattened into linear pipelines to simplify synthesis, introducing additional buffering where earlier values are passed through intermediate stages.
2. Our FPGA hardware generator adds additional overhead due to the coarse granularity of BRAM allocation, combined with several simplifying assumptions in our logic generator.

Our ASIC backend synthesizes buffers with very little overhead, so inefficiency in our ASIC designs comes almost exclusively from linearization (1). FPGA designs can add significant additional overhead (2). In practice, for the pipelines we studied in hardware, linearization increases buffering by at most 2.9× above optimal, while FPGA overhead increases buffering up to 3.2× (Fig. 14). There are many opportunities to improve BRAM allocation in our FPGA generator, but it has not been limiting factor in existing FPGA designs.

6.2 Scheduling for general-purpose processors

We evaluate Darkroom's performance on an x86 CPU (a 4 core 3.5 GHz Intel Core i7 3770) by comparing both to existing software implementations and to the output of the Halide compiler (Fig. 15).

For ISP, we compared Darkroom to our internal reference code written as clean C. Our reference code has no multithreading, vectorization, or line buffering. Enabling these optimizations by reimplementing it in Darkroom yielded a 7× speedup, with source code of similar complexity. Of this speedup, 3.5× comes from multithreading, and 2× comes from vectorization.

We also compared Darkroom to Halide, an existing high-performance image processing language and compiler [Ragan-Kelley et al. 2012], on the DEBLUR application (Fig. 15). We performed this comparison in floating point because it resulted in better performance on our test machine. Halide's programming and scheduling models are more general than Darkroom, but as a result, automatically optimizing programs requires an expensive brute-force search process using autotuning [Ragan-Kelley et al. 2013]. By constraining the scheduling problem, Darkroom is able to automatically optimize schedules for stencil pipelines using our direct ILP optimization. As a result, we see similar performance from both Halide- and Darkroom-compiled implementations of DEBLUR, but Darkroom's schedule optimization takes under 1 second and the total compile time takes less than 2 minutes, while the Halide autotuner required 8 hours to find a comparably performing schedule.

We additionally present the absolute throughput, and corresponding buffering, of all six applications we tested (Fig. 16). ISP was benchmarked on a 7 megapixel raw image, OPTICAL FLOW on a 1080p video, DEPTH FROM STEREO on a 480p stereo video, and EDGE DETECTION, CORNER DETECTION, and DEBLUR were each benchmarked on 16 megapixel images. Throughput is approximately 30 megapixels/sec on 4 cores for each of ISP, EDGE DETECTION, DEBLUR, and OPTICAL FLOW, and as high as 148 megapixels/sec for the relatively simple CORNER DETECTION pipeline. The CPU consumes approximately 85 W of power under load. This corresponds to approximately 500-5000 nJ/pixel (for applications other than DEPTH FROM STEREO), 2-3 orders of magnitude more than the ASIC hardware plus DRAM. This is in line with existing research on specialized vs. general-purpose hardware [Hameed et al. 2010].

The brute-force DEPTH FROM STEREO algorithm delivers a much lower pixel rate than the other applications, but the algorithm is simply bound by the large amount of computation it performs per-pixel. Its performance relative to the fastest pipeline (CORNER DETECTION) is proportional to the difference in arithmetic per-pixel.

Our CPU compiler places fewer constraints on scheduling than our hardware generators. In particular, it does not require linearizing the scheduled pipeline graph. It does introduce modest overhead for implementation efficiency (e.g., enlarging buffers for better vector alignment), but achieved buffering is always within 13% of optimal, and generally less than 2%.

7 Prior Work

7.1 Image Processing Languages

A number of existing image processing languages and systems have been proposed. Languages have treated images as functions of continuous (x, y) coordinates [Holzmann 1988; Elliott 2001], and as a tree of image-wide operators [Shantzis 1994].

Halide is an image processing language that has a separate algorithm language and scheduling language [Ragan-Kelley et al. 2012]. The algorithm language describes what should be computed, and the scheduling language describes in what order to execute the operations. Different schedules can have vastly different performance: an autotuner is used to automatically find good schedules [Ragan-Kelley et al. 2013]. Darkroom is less expressive than Halide. In particular, stencil sizes must be known at compile time, but as a consequence we have shown that a deterministic scheduling algorithm can yield competitive performance to autotuned Halide schedules on CPUs, and also map to efficient custom hardware.

Stencil computations are common in physical simulations, so optimizing their performance has been extensively studied. Prior work has examined a cache oblivious approach [Frigo and Strumpen 2005], code generators [Tang et al. 2011], blocking techniques [Nguyen et al. 2010], and autotuning [Datta et al. 2008]. Most of these systems optimize a single stencil operation. Darkroom uses some of the optimizations discussed in prior work, but focuses on scheduling a large graph involving multiple stencil operations in hardware, which to our knowledge has not been be previously studied.

OpenCV is a C library that provides a number of common image processing algorithms [OpenCV]. Portions of OpenCV have been implemented on FPGAs using high-level synthesis tools [Vivado]. Due to its nature as a library of independent function calls, OpenCV isn't able to optimize memory usage between operations, which we believe is the most important optimization in image processing.

7.2 Synchronous Dataflow

In Synchronous Dataflow (SDF), the user specifies their algorithm as an acyclic data flow graph. Each node, or kernel, in the graph consumes M values from its inputs, and produces N values as output. SDF kernels are able to look at D past values. SDF graphs are normally implemented as pipelines with delays. All rates and delays are known at compile time, so SDF graphs can be statically scheduled. However, not all SDF programs can be scheduled within bounded memory, and when schedulable, the problem of minimizing buffering is NP-Complete [Lee and Messerschmitt 1987; Murthy et al. 1997]. Subsequent work reduced restrictions on the programming model, resulting in either more complex static scheduling, or dynamic scheduling [Bilsen et al. 1995; Sugerman et al. 2009].

Unlike SDF, Darkroom requires kernels to have a fixed input and output rate throughout the pipeline, which significantly simplifies scheduling. In addition, Darkroom is aware of the 2D nature of images, which makes it easier to apply optimizations like tiling.

7.3 Systolic Arrays and DSPs

Camera ISPs are similar to systolic arrays, a well-studied style of energy efficient and compute-dense architectures [Kung 1979]. Systolic arrays are grids of simple processors where each processor can communicate with its neighbors in the grid. DSPs are general-purpose processors augmented with DMAs, VLIW, vector units and special-purpose datapaths to help them perform well on certain multimedia applications [Qualcomm]. DSPs are sometimes used to implement video or image processing applications. In this paper, we have chosen to target a general-purpose x86 processor, due to their prevalence and well-tested toolchain. We believe the same locality and data-parallel optimization we make on x86 apply directly to good performance on DSPs.

8 Discussion and Future Work

We presented Darkroom, a compiler which takes a high-level definition of image processing code and maps it efficiently to ASICs, FPGAs and modern CPUs. Minimizing working set is crucial on CPU, FPGA, and ASIC, because each has a hard limit on the amount of local fast storage available: CPUs have a limited cache, FPGAs have a limited number of BRAMs, and ASICs are often limited by area. We showed that, thanks to Darkroom's carefully restricted programming model, an ILP scheduling algorithm is able to quickly schedule image processing programs into line-buffered pipelines with minimal intermediate storage. Darkroom synthesizes these optimized pipelines into efficient ASIC designs and FPGA implementations capable of real-time processing of high resolution images at video rates, and CPU code competitive with state of the art image processing compilers. Our initial ASIC and FPGA results are especially exciting, because they fit within a modest hardware budget on a 45nm process, or a small fraction of a mid-range FPGA, suggesting that there is opportunity to prototype much larger real-time image processing pipelines, given the right tools.

We have shown that Darkroom's programming model has enough generality to be useful, but we also believe that some of its restrictions could be reduced without eliminating its fast, predictable scheduling. We are interested in extending Darkroom to support image pyramids and serial operations, both of which would allow it to support operations that can propagate information further than the stencil size. This would enable applications like optical flow with larger search windows, or region labeling. In addition, there is significant interest in further investigating how to best map our programming model to existing architectures like CPUs, GPUs and DSPs, with an eye towards understanding how these architectures could be modified to support these image processing workloads with better energy efficiency.

We are excited about new areas of research Darkroom enables for the graphics and imaging community. First, extending prior work on the Frankencamera, mobile camera platforms that include FPGAs programmed by Darkroom would allow researchers to quickly experiment with new applications in real cameras, with real-time performance [Adams et al. 2010]. Second, we believe our approach has the potential to accelerate the development of commercial ISP ASICs, eventually enabling new image processing and computer vision applications on future cameras.

Acknowledgments

This work has been supported by the DOE Office of Science ASCR in the ExMatEx and ExaCT Exascale Co-Design Centers, program manager Karen Pao; DARPA Contract No. HR0011-11-C-0007; fellowships and grants from NVIDIA, Intel, and Google; and the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA). Any opinions, findings and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

ADAMS, A., TALVALA, E.-V., PARK, S. H., JACOBS, D. E., AJDIN, B., GELFAND, N., DOLSON, J., VAQUERO, D., BAEK, J., TICO, M., LENSCH, H. P. A., MATUSIK, W., PULLI, K., HOROWITZ, M., AND LEVOY, M. 2010. The Frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics 29*, 4 (July), 29:1–29:12.

APTINA. Aptina MT9P111. `http://www.aptina.com/products/soc/mt9p111/`.

BERKELAAR, M., EIKLAND, K., NOTEBAERT, P., ET AL. 2004. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*.

BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. 1995. Cyclo-static data flow. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, 3255–3258.

BOUGUET, J.-Y. 2001. Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm. Tech. rep., Intel Corporation.

CANNY, J. 1986. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6, 679–698.

DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 4.

DEVITO, Z., HEGARTY, J., AIKEN, A., HANRAHAN, P., AND VITEK, J. 2013. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 105–116.

ELLIOTT, C. 2001. Functional image synthesis. In *Proceedings of Bridges*.

FRIGO, M., AND STRUMPEN, V. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ACM, 361–366.

GUMMARAJU, J., AND ROSENBLUM, M. 2005. Stream programming on general-purpose processors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 343–354.

HAMEED, R., QADEER, W., WACHS, M., AZIZI, O., SOLOMATNIKOV, A., LEE, B. C., RICHARDSON, S., KOZYRAKIS, C., AND HOROWITZ, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ACM, 37–47.

HARRIS, C., AND STEPHENS, M. 1988. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, 147–151.

HOLZMANN, G. 1988. *Beyond Photography: The Digital Darkroom*. Prentice Hall.

KUNG, H. T. 1979. Let's design algorithms for VLSI systems. In *Proceedings of the Caltech Conference on Very Large Scale Integration*.

LATTNER, C., AND ADVE, V. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*.

LEE, E. A., AND MESSERSCHMITT, D. G. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers 100*, 1, 24–35.

LEISERSON, C. E., AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 1-6, 5–35.

LUCAS, B. D., KANADE, T., ET AL. 1981. An iterative image registration technique with an application to stereo vision. In *IJCAI*, vol. 81, 674–679.

MALLADI, K., NOTHAFT, F., PERIYATHAMBI, K., LEE, B., KOZYRAKIS, C., AND HOROWITZ, M. 2012. Towards energy-proportional datacenter memory with mobile dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 37–48.

MURALIMANOHAR, N., AND BALASUBRAMONIAN, R. 2009. Cacti 6.0: A tool to understand large caches. Tech. rep., HP Labs.

MURTHY, P., BHATTACHARYYA, S., AND LEE, E. 1997. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design 11*, 1, 41–70.

NGUYEN, A., SATISH, N., CHHUGANI, J., KIM, C., AND DUBEY, P. 2010. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *in Proc. of the 2010 ACM/IEEE Intl Conf. for High Performance Computing, Networking, Storage and Analysis, 2010*, 1–13.

OPENCV. OpenCV. http://opencv.org/.

QUALCOMM. Qualcomm hexagon SDK. https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-multimedia-optimization-hexagon-sdk.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG) 31*, 4, 32.

RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 519–530.

RICHARDSON, W. H. 1972. Bayesian-based iterative method of image restoration. *JOSA 62*, 1, 55–59.

SHACHAM, O., GALAL, S., SANKARANARAYANAN, S., WACHS, M., BRUNHAVER, J., VASSILIEV, A., HOROWITZ, M., DANOWITZ, A., QADEER, W., AND RICHARDSON, S. 2012. Avoiding game over: Bringing design to the next level. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 623–629.

SHANTZIS, M. A. 1994. A model for efficient and flexible image computing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, 147–154.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics (TOG) 28*, 1 (Feb.), 4:1–4:11.

TANG, Y., CHOWDHURY, R. A., KUSZMAUL, B. C., LUK, C.-K., AND LEISERSON, C. E. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, 117–128.

VIVADO. vivado. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.