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, 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.

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. 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,
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 \textit{lambda}-like syntax, \(\text{im}(x, y)\). For example, a simple brightening operation applied to the input image \(i\) can be written as the function:

\[
\text{brighter} = \text{im}(x, y) \cdot 1.1
\]

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

\[
\text{convolve} = \text{im}(x, y) \cdot (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/4
\]

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 \(I:\)

\[
\text{warp} = \text{im}(x, y) \cdot \text{gather}(I, 1, 4, 4, \text{warpVX}(x, y), \text{warpVY}(x, y))
\]

\text{warpVX} and \text{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, \((2)\) with the explicit gather operator. Apline indices like \((sx2, sy2)\) 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:

\[
\begin{align*}
\text{bx} &= \text{im}(x, y) \cdot (\text{in}(x-1, y) + \text{in}(x, y) + \text{in}(x+1, y))/3 \\
\text{by} &= \text{im}(x, y) \cdot (\text{bx}(x, y-1) + \text{bx}(x, y) + \text{bx}(x, y+1))/3 \\
\text{difference} &= \text{im}(x, y) \cdot \text{bx}(x, y) \\
\text{scaled} &= \text{im}(x, y) \cdot \text{difference}(x, y) \\
\text{sharpened} &= \text{im}(x, y) + \text{scaled}(x, y)
\end{align*}
\]

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

\[
\begin{align*}
S1 &= \text{im}(x, y) \cdot (I(x-1, y) + I(x, y) + I(x+1, y))/3 \\
S2 &= \text{im}(x, y) \cdot (I(x, y) + 0.1 \cdot (I(x, y) - (S1(x, y-1) + S1(x, y) + S1(x, y+1))/3)
\end{align*}
\]

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 \textit{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 \textit{combinational} nodes that perform arithmetic, and \textit{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 \(I\) with a constant kernel \(k\). From this code, we can create a pipeline such that at time \(t = 1\), we read the value for \(In(8)\) and produce a value for \(Out(8)\), illustrated in Fig. 4 (b). Current values such as \(In(8)\) 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 \textit{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 \(f(x + c1, y + c2)\) are replaced with \(f(x' + c1 + L \cdot c2)\).

\[
\begin{align*}
\text{(a) A 1-D convolution of In against a constant kernel k:} \\
\text{Out} &= \text{in}(x) \cdot k0 \cdot \text{In}(x) + k1 \cdot \text{In}(x-1) + k2 \cdot \text{In}(x-2)
\end{align*}
\]

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

\[
\begin{align*}
\text{(b) A pipeline that implements this execution. Square nodes are} \\
\text{pixel buffers whose output is their input from the previous time step. Two} \\
\text{pixel buffers are required because our accesses values of In} \\
\text{in two cycles in the past. This collection of pixel buffers forms a line} \\
\text{buffer:}
\end{align*}
\]

Figure 4: Translation of a simple convolution stencil into a line-buffered pipeline.
(a) Code for a 1D Richardson-Lucy deconvolution:
\[
\begin{align*}
\text{Rel} &= \text{im}(s) \times \text{Obs}(s) / (k_0 + \text{Lat}(s-1) + k_1 + \text{Lat}(s)) \quad \text{end} \\
\text{LatN} &= \text{im}(s) \times (k_2 + \text{Rel}(s-1) + k_3 + \text{Rel}(s)) \quad \text{end}
\end{align*}
\]
(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 \text{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 \( x' = x + L \times y \) 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 \( \text{Lat} \), the blurred input image \( \text{Obs} \), and constants \( k_i \) which describe the point spread function of the blur. It calculates the relative error \( \text{Rel} \) of the latent estimate, and produces an improved latent estimate \( \text{LatN} \).

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 \( \text{Lat}(0) \) and \( \text{Est}(0) \). But for this example, we cannot compute \( \text{Rel}(0) \) since it depends on \( \text{Lat}(1) \), 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 \( \text{Rel} \) only relies on \text{current or previous} values of its input, we can \text{shift} \( f \) 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 \( \text{Rel} \) with \( \text{Rel}_1 \) and adjust the offsets:

\[
\begin{align*}
\text{Rel}_1 &= \text{im}(s) \times \text{Obs}(s-1) / (k_0 + \text{Lat}(s-2) + k_1 + \text{Lat}(s-1) + k_2 + \text{Lat}(s)) \\
\text{LatN} &= \text{im}(s) \times (k_2 + \text{Rel}_1(s-1) + k_3 + \text{Rel}_1(s)) \quad \text{end}
\end{align*}
\]

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

\[
\begin{align*}
\text{Rel}_1 &= \text{im}(x) \times \text{Obs}(x-1) / (k_0 + \text{Lat}(x-2) + k_1 + \text{Lat}(x-1) + k_2 + \text{Lat}(x)) \\
\text{LatN} &= \text{im}(x) \times (k_2 + \text{Rel}_1(x-1) + k_3 + \text{Rel}_1(x+1)) \quad \text{end}
\end{align*}
\]

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 \( \text{LatN} \) by 2, which results in this modified program that contains no hazards:

\[
\begin{align*}
\text{Rel}_1 &= \text{im}(x) \times \text{Obs}(x-1) / (k_0 + \text{Lat}(x-2) + k_1 + \text{Lat}(x-1) + k_2 + \text{Lat}(x)) \\
\text{LatN}_2 &= \text{im}(x) \times (k_2 + \text{Rel}_1(x-2) + k_3 + \text{Rel}_1(x-1) + k_0 + \text{Rel}_1(x+1)) \quad \text{end}
\end{align*}
\]

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 \( \text{Lat}(n) \) given a shifted pipeline that executes it \( \text{LatN}_2 \) simply requires changing the indices that are calculated each cycle, e.g., \( \text{LatN}_2(2) \) at \( t = 0 \) instead of \( \text{LatN}(0) \). 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 \( \text{Obs} \) which would disappear if we choose to shift \( \text{Obs} \) 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 \( e(x) \), we generate a
We 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:

\[
\text{out} = \text{in}(x) + \text{in}(x - 1) + \text{in}(x) - \text{in}(x + 1)
\]

will result in the following values for \(F\) and \(U\):

\[
F = \{\text{Out}, \text{In}\}
\]

\[
U = \{(\text{Out, In, } -1), (\text{Out, In, 0}), (\text{Out, In, 1})\}
\]

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 \(\Rightarrow\) 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 (lp_solve), 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
The code is then transformed into a pipeline:

```
Bu ff er s contain merged values, so they are twice as large.

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

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 meta-programming 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 multi-threaded 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 $S_2$ that consumes from $S_1$ yields the following code:

```
for each line $y$
  for each pixel $x$ in line of $S_1$
    compute $S_1(x, y)$
  for each pixel $x$ in line of $S_2$
    compute $S_2(x, y)$ // loading $S_1$ 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 \(10 \times 3 \times 3\) 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.

DEEP 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. DEEP 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.

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).

Figure 12: Darkroom compiles our applications into efficient ASIC, using a chip area of 0.3-2.6 mm² 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.

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, 10x 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...
We evaluated 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 $7 \times$ speedup, with source code of similar complexity. Of this speedup, $3.5 \times$ comes from multithreading, and $2 \times$ 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.

### 6.2 Scheduling for general-purpose processors

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. The inefficiency in our ASIC designs comes almost exclusively from linearizing (1). FPGA designs can add significant additional overhead (2). In practice, for the pipelines we studied in hardware, linearization increases buffering by at most $2.9 \times$ above optimal, while FPGA overhead increases buffering up to $3.2 \times$ (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.

### 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.
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 multimedia applications [Qualcomm]. DSPs are sometimes used to 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


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


