Abstract

Using existing programming tools, writing high-performance image processing code requires sacrificing readability, portability, and modularity. We argue that this is a consequence of conflating what computations define the algorithm, with decisions about storage and the order of computation. We refer to these latter two concerns as the schedule, including choices of tiling, fusion, recomputation vs. storage, vectorization, and parallelism.

We propose a representation for feed-forward imaging pipelines that separates the algorithm from its schedule, enabling high-performance without sacrificing code clarity. This decoupling simplifies the algorithm specification: images and intermediate buffers become functions over an infinite integer domain, with no explicit storage or boundary conditions. Imaging pipelines are compositions of functions. Programmers separately specify scheduling strategies for the various functions composing the algorithm, which allows them to efficiently explore different optimizations without changing the algorithmic code.

We demonstrate the power of this representation by expressing a range of recent image processing applications in an embedded domain-specific language, and compiling them for ARM, x86, and GPUs. Our compiler targets SIMD units, multiple cores, and complex memory hierarchies. We demonstrate that it can handle algorithms such as a camera raw pipeline, the bilateral grid, fast local Laplacian filtering, and image segmentation. The algorithms expressed in our language are both shorter and faster than state-of-the-art implementations.

Keywords: Image Processing, Compilers, Performance

1 Introduction

Computational photography algorithms require highly efficient implementations to be used in practice, especially on power-constrained mobile devices. This is not a simple matter of programming in a low-level language like C. The performance difference between naive C and highly optimized C is often an order of magnitude. Unfortunately, this usually comes at the cost of programmer pain and code complexity, as computation must be reorganized to achieve memory efficiency and parallelism.

For image processing, the global organization of execution and storage is critical. Image processing pipelines are both wide and deep: they consist of many data-parallel stages that benefit hugely from parallel execution across pixels, but stages are often memory bandwidth limited—they do little work per load and store. Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. The best choice of transformations is architecture-specific; implementations optimized for an x86 multicore and a modern GPU often bear little resemblance to each other.

In this paper we enable simpler high-performance code by separating the intrinsic algorithm from the decisions about how to run efficiently on a particular machine (Fig. 2).

Figure 1: The code at the top computes a 3x3 box filter in using the composition of a 1x3 box filter and a 3x1 box filter. Using vectorization, multithreading, tiling, and fusion, we can make this algorithm more than 10× faster on a quad-core x86 CPU (middle). However in doing so we’ve lost readability and portability. Our compiler separates the algorithm description from its schedule, achieving the same performance without making the same sacrifices (bottom). For the full details about how this test was carried out, see the supplemental material.

---

1 Introduction

Computational photography algorithms require highly efficient implementations to be used in practice, especially on power-constrained mobile devices. This is not a simple matter of programming in a low-level language like C. The performance difference between naive C and highly optimized C is often an order of magnitude. Unfortunately, this usually comes at the cost of programmer pain and code complexity, as computation must be reorganized to achieve memory efficiency and parallelism.

For image processing, the global organization of execution and storage is critical. Image processing pipelines are both wide and deep: they consist of many data-parallel stages that benefit hugely from parallel execution across pixels, but stages are often memory bandwidth limited—they do little work per load and store. Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. The best choice of transformations is architecture-specific; implementations optimized for an x86 multicore and a modern GPU often bear little resemblance to each other.

In this paper we enable simpler high-performance code by separating the intrinsic algorithm from the decisions about how to run efficiently on a particular machine (Fig. 2).

Abstract

Using existing programming tools, writing high-performance image processing code requires sacrificing readability, portability, and modularity. We argue that this is a consequence of conflating what computations define the algorithm, with decisions about storage and the order of computation. We refer to these latter two concerns as the schedule, including choices of tiling, fusion, recomputation vs. storage, vectorization, and parallelism.

We propose a representation for feed-forward imaging pipelines that separates the algorithm from its schedule, enabling high-performance without sacrificing code clarity. This decoupling simplifies the algorithm specification: images and intermediate buffers become functions over an infinite integer domain, with no explicit storage or boundary conditions. Imaging pipelines are compositions of functions. Programmers separately specify scheduling strategies for the various functions composing the algorithm, which allows them to efficiently explore different optimizations without changing the algorithmic code.

We demonstrate the power of this representation by expressing a range of recent image processing applications in an embedded domain-specific language, and compiling them for ARM, x86, and GPUs. Our compiler targets SIMD units, multiple cores, and complex memory hierarchies. We demonstrate that it can handle algorithms such as a camera raw pipeline, the bilateral grid, fast local Laplacian filtering, and image segmentation. The algorithms expressed in our language are both shorter and faster than state-of-the-art implementations.

Keywords: Image Processing, Compilers, Performance

1 Introduction

Computational photography algorithms require highly efficient implementations to be used in practice, especially on power-constrained mobile devices. This is not a simple matter of programming in a low-level language like C. The performance difference between naive C and highly optimized C is often an order of magnitude. Unfortunately, this usually comes at the cost of programmer pain and code complexity, as computation must be reorganized to achieve memory efficiency and parallelism.

For image processing, the global organization of execution and storage is critical. Image processing pipelines are both wide and deep: they consist of many data-parallel stages that benefit hugely from parallel execution across pixels, but stages are often memory bandwidth limited—they do little work per load and store. Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. The best choice of transformations is architecture-specific; implementations optimized for an x86 multicore and a modern GPU often bear little resemblance to each other.

In this paper we enable simpler high-performance code by separating the intrinsic algorithm from the decisions about how to run efficiently on a particular machine (Fig. 2).
To understand the challenge of efficient image processing, consider a $3 \times 3$ box filter implemented as separate horizontal and vertical passes. We might write this in C++ as a sequence of two loop nests (Fig. 1.a). An efficient implementation on a modern CPU requires SIMD vectorization and multithreading. But once we start to exploit parallelism, the algorithm becomes bottlenecked on memory bandwidth. Computing the entire horizontal pass before the vertical pass destroys producer-consumer locality—horizontally blurred intermediate values are computed long before they are consumed by the vertical blur pass—doubling the storage and memory bandwidth required. Exploiting locality requires interleaving the two stages, by tiling and fusing the loops. Tiles must be carefully sized for alignment, and efficient fusion requires subtleties like redundantly computing values on the overlapping boundaries of intermediate tiles. The resulting implementation is over $10\times$ faster on a quad-core CPU, but together, these optimizations have fused two simple, independent steps into a single intertwined, non-portable mess (Fig. 1.b).

We believe the right answer is to separate the intrinsic algorithm—what is computed—from the concerns of efficient mapping to machine execution—decisions about storage and the ordering of computation. We call these choices of how to map an algorithm onto resources in space and time the schedule.

Image processing exhibits a rich space of schedules. Pipelines tend to be deep and heterogeneous (in contrast to signal processing or array-based scientific code). Efficient implementations must trade off between storing intermediate values, or recomputing them when needed. However, intentionally introducing recomputation is seldom considered by traditional compilers. In our approach, the programmer specifies an algorithm and its schedule separately. This makes it easy to explore various optimization strategies without obfuscating the code or accidentally modifying the algorithm itself.

Functional languages provide a natural model for separating the what from the when and where. Divorced from explicit storage, images are no longer arrays populated by procedures, but instead pure functions that define the value at each point in terms of arithmetic, reductions, and the application of other functions. A functional representation also allows us to omit boundary conditions, making images functions over an infinite integer domain.

### Figure 2: We compare algorithms in our prototype language, Halide, to state of the art implementations of four image processing applications, ranging from MATLAB code to highly optimized NEON vector assembly [Adams et al. 2010; Aubry et al. 2011; Paris and Durand 2009; Li et al. 2010]. Halide code is compact, modular, portable, and delivers high performance across multiple platforms. All speedups are expressed relative to the reference implementation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Platform</th>
<th>Lines</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera Raw Pipeline</td>
<td>Halide:</td>
<td>145</td>
<td>2.75x shorter</td>
<td></td>
</tr>
<tr>
<td>Local Laplacian Filter</td>
<td>23</td>
<td>2.1x faster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bilateral Grid</td>
<td>62</td>
<td>5% faster than tuned assembly</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Snake Image Segmentation</td>
<td>34</td>
<td>3x shorter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Porting to new platforms does not change the algorithm code, only the schedule.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Lines</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quad-core x86</td>
<td>51</td>
<td>3 ms (1267x)</td>
</tr>
<tr>
<td>CUDA GPU</td>
<td>48 (13x)</td>
<td></td>
</tr>
<tr>
<td>CUDA GPU</td>
<td>11 (42x)</td>
<td></td>
</tr>
<tr>
<td>CUDA GPU</td>
<td>3 (1267x)</td>
<td></td>
</tr>
</tbody>
</table>

In this representation, the algorithm only defines the value of each function at each point, and the schedule specifies:

- The order in which points in the domain of a function are evaluated, including the exploitation of parallelism, and mapping onto SIMD execution units.
- The order in which points in the domain of one function are evaluated relative to points in the domain of another function.
- The memory location into which the evaluation of a function is stored, including registers, scratchpad memories, and regions of main memory.
- Whether a value is recomputed, or from where it is loaded, at each point a function is used.

Once the programmer has specified an algorithm and a schedule, our compiler combines them into an efficient implementation. Optimizing execution for a given architecture requires modifying the schedule, but not the algorithm. The representation of the schedule is compact (e.g. Fig. 1.c), so exploring the performance of many options is fast and easy. We can most flexibly schedule operations which are data parallel, with statically analyzable access patterns, but also support the reductions and bounded irregular access patterns that occur in image processing.

In addition to this model of scheduling (Sec. 3), we present:

- A prototype embedded language called Halide, for functional algorithm and schedule specification (Sec. 4).
- A compiler which translates functional algorithms and optimized schedules into efficient machine code for x86 and ARM, including SSE and NEON SIMD instructions, and CUDA GPUs, including synchronization and placement of data throughout the specialized memory hierarchy (Sec. 5).
- A range of applications implemented in our language, composed of common image processing operations such as convolutions, histograms, image pyramids, and complex stencils. Using different schedules, we compile them into optimized programs for x86 and ARM CPUs, and a CUDA GPU (Sec. 6). For these applications, the Halide code is compact, and performance is state of the art (Fig. 2).
2 Prior Work

Loop transformation Most compiler optimizations for numerical programs are based on loop analysis and transformation, including auto-vectorization, loop interchange, fusion, and tiling. The polyhedral model is a powerful tool for transforming imperative programs [Feautrier 1991], but traditional loop optimizations do not consider recomputation of values: each point in each loop is computed only once. In image processing, recomputing some values—rather than storing, synchronizing around, and reloading them—can be a large performance win (Sec. 6.2), and is central to the choices we consider during optimization.

Data-parallel languages Many data-parallel languages have been proposed. Particularly relevant in graphics, CUDA and OpenCL expose an imperative data-parallel programming model which can target both GPUs and multicore CPUs with SIMD units [Buck 2007; OpenCL]. Like C, they allow the specification of very high performance implementations for many algorithms, but because parallel work distribution, synchronization, and memory are all explicitly managed by the programmer, complex algorithms are often not composable in these languages, and the optimizations required are often specific to an architecture, so code must be rewritten for different platforms.

Intel’s Array Building Blocks provides an embedded language for data-parallel array processing in C++ [ArBB]. Like in our representation, whole pipelines of operations are built up and optimized globally by a compiler. It delivers impressive performance for many algorithms on Intel CPUs. However, the inherently imperative structure—in particular the explicit specification of storage locations—fundamentally affords less flexibility in scheduling a given pipeline. Trading off recomputation vs. storage is challenging in this representation, and is not considered by the compiler.

Image processing languages Shantzis described a framework and runtime model for image processing systems based on graphs of operations which process tiles of data [Shantzis 1994]. This is the inspiration for many scalable and extensible image processing systems, including our own. Apple’s CoreImage and Adobe’s PixelBender include kernel languages for specifying individual point-wise operations on images [CoreImage; PixelBender]. Kernels compile into optimized code for multiple architectures, including GPUs. Neither optimizes across graphs of kernels, which often contain complex communication like stencils, and neither supports reductions or nested parallelism within kernels.

The SPIRAL system [Püschel et al. 2005] uses a domain-specific language, SPL, for specifying linear signal processing functions independent of their schedule. Complementary mapping functions describe how these operations should be turned into efficient code for a particular architecture, similarly to our schedule specifications. It enables high performance across a range of architectures for linear filtering pipelines, by making deep use of mathematical identities on linear filters. Computational photography algorithms often do not fit within a strict linear filtering model. Our work can be seen as an attempt to generalize this approach to a broader class of programs.

Elsewhere in graphics, the real-time graphics pipeline has been a hugely successful abstraction precisely because the schedule is separated from the specification of the shaders. This allows GPUs and drivers to efficiently execute a wide range of programs with little programmer control over parallelism and memory management. This separation of concerns is extremely effective, but it is specific to the design of a single pipeline. That pipeline also exhibits different characteristics than image processing pipelines, where reductions and stencil communication are common, and kernel fusion is essential for efficiency. Embedded DSLs have also been used to specify the shaders themselves, directly inside the host C++ program that configures the pipeline [McCool et al. 2002].

MATLAB is also extremely successful as a language for image processing. Its high level syntax enables terse expression of many algorithms, and its widely-used library of built-in functionality shows that the ability to compose modular library functions is invaluable for programmer productivity. However, simply bundling fast implementations of individual kernels is not sufficient for fast execution on modern machines, where optimization across stages in a pipeline is essential for efficient use of parallelism and memory.

Pan introduced a functional model for image processing much like our own [Elliott 2001]. In Pan, images are functions from coordinates to values. Modest differences exist (Pan’s images are functions over a continuous coordinate domain, while in ours the domain is discrete), but Pan is a close sibling of our intrinsic algorithm representation. However, it has no corollary to our complementary model of scheduling and ultimate compilation. It exists only as a direct embedding within Haskell, and is not compiled for high performance execution.

3 Representing Algorithms and Schedules

We propose a functional representation for image processing pipelines that separates the intrinsic algorithm from the schedule with which it will be executed. In this section we describe the representation for each of these components, and how they combine to create a fully-specified program.

3.1 The Intrinsic Algorithm

Our algorithm representation is functional. Values that would be mutable arrays in an imperative language are instead functions from coordinates to values. We represent images as pure functions defined over an infinite integer domain, where the value of a function at a point represents the color of the corresponding pixel. Imaging pipelines are specified as chains of functions. Functions may either be simple expressions in their arguments, or reductions. The expressions which define functions are side-effect free, and are much like those in any simple functional language, including:

- Arithmetic and logical operations;
- Loads from external images;
- If-then-else expressions (semantically equivalent to the ternary operator in C);
- References to named values (which may be function arguments, or expressions defined by a functional let construct);
- Calls to other functions, including external C ABI functions.

For example, our separable $3 \times 3$ box filter in Figure 1 is expressed as a chain of two functions in $x, y$. The first horizontally blurs the input; the second vertically blurs the output of the first.

This representation is simpler than most functional languages. We omit higher-order functions, dynamic recursion, and richer data structures such as tuples and lists. Functions simply map from integer coordinates to a scalar result. This representation is sufficient to represent a wide range of image processing algorithms, and these constraints enable extremely flexible analysis and transformation of algorithms during compilation. Constrained versions of more advanced features, such as higher-order functions and tuples, are
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RVar rx(0, in.width()), ry(0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(rx, xy))+;
cdf(i) = 0;
cdf(ri) = cdf(ri-i) + histogram(ri);
out(x, y) = cdf(input(x, y));

Figure 3: Histogram equalization uses a reduction to compute a histogram, a scan to integrate it into a cdf, and a point-wise operation to remap the input using the cdf. The iteration domains for the reduction and scan are expressed by the programmer using RVar. Like all functions in our representation, histogram and cdf are defined over an infinite domain. At entries not touched by the reduction step they are zero-valued. For cdf this is specified explicitly. For histogram this is implicit in the ++ operator.

reintroduced as syntactic sugar, but they do not change the underlying representation (Sec. 4.1).

- An initial expression which specifies a value at each point in the output domain in terms of the function arguments.
- A list of reduction variables, bounded by minimum and maximum expressions.
- A reduction expression, which redefines the value of the function at a computed output coordinate as a function of the reduction variables and recursive references to the same function.

The value at a given point in the output domain is defined by the last reduction expression that touched that output coordinate, given a lexicographic traversal of all values of the reduction variables. Any point which was not touched by a reduction expression has the value of the initial expression.

Reduction expressions are usually recursive. For example, histogram in Figure 3 defines a new value in terms of the old value at the same point, while cdf defines a new value in terms of the value to the left. While we semantically define a strict lexicographic traversal order over the reduction variables, many common reductions (such as histogram) are associative, and may be executed in parallel given appropriate atomics. Scans like cdf are inherently more challenging to parallelize. We do not yet address this.

3.2 The Schedule

Our formulation of imaging pipelines as chains of functions intentionally omits choices of when and where these functions should be computed. The programmer separately specifies this using a schedule. A schedule describes not only the order of evaluation of points within the producer and consumer, but also what is stored and what is recomputed. The schedule further describes mapping onto parallel execution resources such as threads, SIMD units, and GPU blocks. It is constrained only by the fundamental dependence between points in different functions (values must be computed before they are used).

Schedules are demand-driven: for each pipeline stage, they specify how the inputs should be evaluated, starting from the output of the full pipeline. Formally, when a callee function such as tmp in Fig.1(c) is invoked in a caller such as blurred, we need to decide how to schedule it with respect to the caller.

We currently allow four types of caller-callee relationships (Fig. 4). Some of them lead to additional choices, including traversal order and subdivision of the domain, with possibly recursive scheduling decisions for the sub-regions.

Inline: Compute as needed, do not store  In the simplest case, the callee is evaluated directly at the single point requested by the caller, like a function call in a traditional language. Its value at that point is computed from the expression which defines it, and passed directly into the calling expression. Reductions may not be inlined because they are not defined by a single expression; they require evaluation over the entire reduction domain before they can return a value. Inlining performs redundant computation whenever a single point is referred to in multiple places. However, even when it introduces significant amounts of recomputation, inlining can be the most efficient option. This is because image processing code is very often constrained by memory bandwidth and inlining passes values between functions without touching memory.

Root: Precompute entire required region  At the other extreme, we can compute the value of the callee for the entire subdomain needed by the caller before evaluating any points in the caller. In our blur example, this means evaluating and storing all of the horizontal pass (tmp) before beginning the vertical pass (blurred). We call this call schedule root. Every point is computed exactly once, but storage and locality may be lost: the intermediate buffer required may be large, and points in the callee are unlikely to still be in a cache when they are finally used. This schedule is equivalent to the most common structure seen in naive C or MATLAB image processing code: each stage of the algorithm is evaluated in its entirety, and then stored as a whole image in memory.

While evaluating the callee, there are further choices in the traversal of the required subdomain. A root schedule must specify, for each dimension of the subdomain, whether it is traversed:

- sequentially,
- in parallel,
- unrolled by a constant factor,
- or vectorized by a constant factor.

The schedule also specifies the relative traversal order of the dimensions (e.g. row- vs. column-major).

The schedule does not specify the bounds in each dimension. The bounds of the domain required of each stage are inferred during compilation (Sec. 5.2). Ultimately, these become expressions in the size of the requested output image. Leaving bounds specification to the compiler makes the algorithm and schedule simpler and more flexible. Explicit bounds are only required for indexing expressions not analyzable by the compiler, such as the result of a reduction. In these cases, we require the algorithm to explicitly clamp the problematic index.

The schedule may also split a dimension into inner and outer components, which can then be treated separately. For example, to represent evaluation in tiles, we can split the x into outer and inner dimensions x_o and x_i, and similarly split y into y_o and y_i, which can then be traversed in the order y_o, x_o, y_i, x_i (as illustrated in the lower right of Fig. 4). After a dimension has been split, the inner and outer component must still be scheduled using any of the options discussed above.

4
Figure 4: We model scheduling an imaging pipeline as the set of choices that must be made for each stage about how to evaluate each input. Here we consider blurred’s dependence on tmp from the example in Fig. 1. blurred may inline tmp, computing values on demand and not storing anything for later reuse. This provides excellent temporal locality, but each point of tmp will be computed three times. blurred may also compute and consume tmp in larger chunks. This provides some temporal locality, and performs redundant computation at the chunk boundaries. blurred may simply compute all of tmp before using any of it. We call this root. It computes each point of tmp only once, but temporal locality is poor—each value is unlikely to still be in cache when it is needed. Finally, if some other consumer (in green on the right) had already evaluated all of tmp as root, blurred could simply reuse that data. If blurred evaluates tmp as root or chunked, then there are further choices to make about the order in which to compute the given region of tmp. The choices we implement are shown at the bottom.

Splitting a dimension expands its bounds to be a multiple of the extent of the inner dimension. Vectorizing or unrolling a dimension similarly rounds its extent up to the nearest multiple of the factor used. Such bounds expansion is always legal given our representation of images as functions over infinite domains.

These choices amount to specifying a complete loop nest which traverses the required region of the output domain. The schedule for a reduction must specify a pair of loop nests: one for its initialization (over the output domain), and one for its update (over the reduction domain). In the latter case, the bounds are given by the definition of the reduction, and are do not need to be inferred later.

3.3 The Fully Specified Program

Lowering an intrinsic algorithm with a specific schedule produces a fully specified imperative program, with a defined order of operations and placement of data. The resulting program is made up of ordered imperative statements, including:

- Stores of expression values to array locations;
- Sequential and parallel for loops, which define a range of variable values over which a contained statement should be executed;
- Producer-consumer edges, which define an array to be allocated (its size given by a potentially dynamic expression), a block of statements which may write to it, and a block of statements which may read from it, after which it may be freed.

This is a general imperative program representation, but we don’t need to analyze or transform programs in this form. Most challenging optimization has already been performed in the lowering from

3.3 The Fully Specified Program

Lowering an intrinsic algorithm with a specific schedule produces a fully specified imperative program, with a defined order of operations and placement of data. The resulting program is made up of ordered imperative statements, including:

- Stores of expression values to array locations;
- Sequential and parallel for loops, which define a range of variable values over which a contained statement should be executed;
- Producer-consumer edges, which define an array to be allocated (its size given by a potentially dynamic expression), a block of statements which may write to it, and a block of statements which may read from it, after which it may be freed.

This is a general imperative program representation, but we don’t need to analyze or transform programs in this form. Most challenging optimization has already been performed in the lowering from
intrinsic algorithm to imperative program. And because the compiler generates all imperative allocation and execution constructs, it has a deep knowledge of their semantics and constraints, which can be very challenging to infer from arbitrary imperative input. Our lowered imperative program may still contain symbolic bounds which need to be resolved. A final bounds inference pass infers concrete bounds based on dependence between the bounds of different loop variables in the program (Sec. 5.2).

4 The Language

We construct imaging pipelines in this representation using a prototype language embedded in C++, which we call Halide. A chain of Halide functions can be JIT compiled and used immediately, or it can be compiled to an object file and header to be used by some other program (which need not link against Halide).

Expressions. The basic expressions are constants, domain variables, and calls to Halide functions. From these, we use C++ operator overloading to build arithmetic operations, comparisons, and logical operations. Conditional expressions, type-casting, transcendental, external functions, etc. are described using calls to provided intrinsics. For example, the expression `select(x > 0, sqrt(cast<float>(x)), f(x+1))` returns either the square root of `x`, or the application of some Halide function `f` to `x+1`, depending on the sign of `x`. Finally, `debug` expressions evaluate to their first argument, and print the remainder of their arguments at evaluation-time. They are useful for inspecting values in flight.

Functions are defined in a functional programming style. The following code constructs a Halide function over a two dimensional domain that evaluates to the product of its arguments:

```cpp
Func f;
Var x, y;
f(x, y) = x * y;
```

Reductions are produced by defining a function twice: once for its initial value, and once for its reduction step. The reduction step should be in terms of reduction variables (of type `RVar`), which include expressions describing their bounds. The span of all reduction variables referenced defines the reduction domain. The left-hand-side of the update definition may be a computed location rather than simple variables (Fig. 3).

In many cases we can infer bounds of reduction variables based on their use. We can also infer reasonable initial values in common cases: if a reduction is a sum, the initial value defaults to zero; if it is a product, it defaults to one. The following code takes advantage of both of these features to compute a histogram over the image `im`:

```cpp
Func histogram;
RVar x, y;
histogram(im(x, y))++;
```

Uniforms describe the run-time parameters of an imaging pipeline. They may be scalars or entire images (in particular the input image). When using Halide as a JIT compiler, uniforms can be bound by assigning to them. Statically-compiled Halide functions will expose all referenced uniforms as top-level function arguments. The following C++ code builds a Halide function that brightens its input using a uniform parameter.

```cpp
// A floating point parameter
Uniform<float> scale;
// A two-dimensional floating-point image
UniformImage input(Float(32), 2);
Var x, y:
Func bright;
bright(x, y) = input(x, y) * scale;
```

We can JIT compile and use our function immediately by calling `realize`, or we can statically compile it using `compileToFile`. For example, we can apply the above brighten function immediately:

```cpp
Image<float> im = load("input.png");
input = im;
scale = 2.0f;
Image<float> output = bright.realize(im.width(), im.height());
```

Alternatively, we can statically compile with `bright.compileToFile("bright")`. This produces `bright.o` and `bright.h`, which together define a C callable function with the following type signature:

```cpp
void bright(float scale, buffer_t *in, buffer_t *out);
```

where `buffer_t` is a bare-bones image struct defined in the same header.

4.1 Syntactic Sugar

While the constructs above are sufficient to express any Halide algorithm, functional languages typically provide other features that are useful in this context. We provide restricted forms of several of these via syntactic sugar.

Higher-order functions. While Halide functions may only have integer arguments, the code that builds a pipeline may include C++ functions that take and return Halide functions. These are effectively compile-time higher-order functions, and they let us write generic operations on images. For example, consider the following operator which shrinks an image by subsampling:

```cpp
// Return a new Halide function that subsamples f
Func subsample(Func f) {
  Func g; Var x, y;
g(x, y) = f((2*x, 2*y));
  return g;
}
```

C++ functions that deal in Halide expressions are also a convenient way to write generic code. As the host language, C++ can be used as a metaprogramming layer to more conveniently construct Halide pipelines containing repetitive substructures.

Partial application. When performing trivial point-wise operations on entire images, it is often clearer to omit pixel indices. For example if we wish to define `f` as equal to `a` plus a subsampling of `b`, then `f = a + subsample(b)` is clearer than `f(x, y) = a(x, y) + subsample(b)(x, y)`. We therefore support such partial application of Halide functions. Any operator which combines partially applied functions is automatically lifted to point-wise operation over the omitted arguments.

Tuples. We overload the C++ comma operator to allow for tuples of expressions. A tuple generates an anonymous function that maps from an index to that element of the tuple. The tuple is then treated as a partial application of this function. For example, given expressions `r`, `g`, and `b`, the definition `f(x, y) = (r, g, b)` creates a three-dimensional function (in this case representing a color image) whose last argument selects between `r`, `g`, and `b`. It is equivalent to

```cpp
f(x, y, c) = select(c==0, r, select(c==1, g, b));
```
4.2 Specifying a Schedule

Once the description of an algorithm is complete, the programmer specifies a desired partial schedule for each function. The compiler fills in any remaining choices using simple heuristics, and tabulates the scheduling decisions for each call site. The function representing the output is scheduled as root. Other functions are scheduled as inline by default. This behavior can be modified by calling one of the two following methods:

- `im.root()` schedules the first use of `im` as root, and schedules all other uses to reuse that instance.
- `im.chunk(x)` schedules `im` as chunked over `x`, which must be some dimension of the caller of `im`. A similar reuse heuristic applies; for each unique `x`, only one use is scheduled as chunk, and the others reuse that instance.

If `im` is scheduled as root or chunk, we must also specify the traversal order of the domain. By default it is traversed serially in scanline order. This can be modified using the following methods:

- `im.transpose(x, y)` moves iteration over `x` outside of `y` in the traversal order (i.e., this switches from row-major to column-major traversal).
- `im.parallel(y)` indicates that each row of `im` should be computed in parallel across `y`.
- `im.vectorized(x, k)` indicates that `x` should be split into vectors of size `k`, and each vector should be executed using SIMD.
- `im.unroll(x, k)` indicates that the evaluation of `im` should be unrolled across the dimension `x` by a factor of `k`.
- `im.split(x, xo, xi, k)` subdivides the dimension `x` into outer and inner dimensions `xo` and `xi`, where `xi` ranges from zero to `k`, `xo`, and `xi` can then be marked as parallel, serial, vectorized, or even recursively split.
- `im.tile(x, y, xi, yi, k)` is a convenience method that splits `x` by a factor of `k`, and `y` by a factor of `l`, then transposes the inner dimension of `y` with the outer dimension of `x` to effect traversal over tiles.
- `im.gpu(bx, by, tx, ty)` maps execution to the CUDA model, by marking `bx` and `by` as corresponding to block indices, and `tx` and `ty` as corresponding to thread indices within each block.
- `im.gpuTile(x, y, k)` is a similar convenience method to `tile`. It splits `x` and `y` by `k` and `l` respectively, and then maps the resulting four dimensions to CUDA's notion of blocks and threads.

Schedules that would require substantial transformation of code written in C can be specified tersely, and in a way that does not touch the statement of the algorithm. Furthermore, each scheduling method returns a reference to the function, so calls can be chained:

```c
Func blurry; Var x, y;
BVar i(-2, 5), j(-2, 5);
blurry(x, y) = sum(f(x+i, y+j));
```

5 Compiler Implementation

The Halide compiler lowers imaging pipelines into machine code for ARM, x86, and PTX. It is built on top of the LLVM compiler infrastructure [LLVM], which it uses for conventional scalar optimizations, register allocation, and machine code generation. While LLVM provides some degree of platform neutrality, the final stages of lowering must be architecture-specific to produce high-performance machine code. Compilation proceeds as shown in Fig. 5.

![Figure 5: The programmer writes a pipeline of Halide functions and partially specifies their schedules. The compiler then removes syntactic sugar (such as tuples), generates a complete schedule, and uses it to lower the pipeline into an imperative representation. Bounds inference is then performed to inject expressions that compute the bounds of each loop and the size of each intermediate buffer. The representation is then further lowered to LLVM IR, and handed off to LLVM to compile to machine code.](https://example.com/figure5.png)

5.1 Lowering

After the programmer has created an imaging pipeline and specified its schedule, the first role of the compiler is to transform the functional representation of the algorithm into an imperative one using the schedule. The schedule is tracked as a table mapping from each call site to its call schedule. For root and chunked schedules, it also contains an ordered list of dimensions to traverse, and how they should be traversed (serial, parallel, vectorized, unrolled) or split.

The compiler works iteratively from the end of the pipeline upwards, considering each function after all of its uses. This requires that the pipeline be acyclic. It first initializes a seed by generating the imperative code that realizes the output function over its domain. It then proceeds up the pipeline, either infilling function bodies, or injecting loop nests that allocate storage and evaluate each function into that storage.

The structure of each loop nest, and the location it is injected, are precisely specified by the schedule: a function scheduled as root has realization code injected at the top of the code generated so far; functions scheduled as chunked over some variable have realization code injected at the top of the body of the corresponding loop; inline functions have their uses directly replaced with their function bodies, and functions that reuse other realizations are skipped over for now. Reductions are lowered into a sequential pair of loop nests: one for the initialization, and one for the reduction step.

The ultimate goal of lowering is to replace calls to functions with loads from their realizations. We defer this until after bounds inference.
5.2 Bounds Inference

The compiler then determines the bounds of the domain over which each use of each function must be evaluated. These bounds are typically not statically known at compile time; they will almost certainly depend on the sizes of the input and output images. The compiler is responsible for injecting the appropriate code to compute these bounds. Working through the list of functions, the compiler considers all uses of each function, and derives expressions that give the minimum and maximum possible argument values. This is done using symbolic interval arithmetic. For example, consider the following pseudocode that uses $\epsilon$:

\[
\text{for } (i \text{ from } a \text{ to } b) \ g[i] = f(i+1) + f(i*2)
\]

Working from the inside out it is easy to deduce that $f$ must be evaluated over the range $[\min (a + 1, a * 2), \max (b + 1, b * 2)]$, and so expressions that compute these are injected just before the realization of $\epsilon$. Reductions must also consider the bounds of the expressions that determine the location of updates.

This analysis can fail in one of two ways. First, interval arithmetic can be over-conservative. If $x \in [0, a]$, then interval arithmetic computes the bounds of $x(a - x)$ as $[0, a^2]$, instead of the actual bounds $[0, a^2/4]$. We have yet to encounter a case like this in practice; in image processing, dependence between functions is typically either affine or data-dependent.

Second, the compiler may not be able to determine any bound for some values, e.g. a value returned by an external function. These cases often correspond to code that would be unsafe if implemented in equivalent C. Unbounded expressions used as indices cause the compiler to throw an error.

In either case, the programmer can assist the compiler using \textit{min} and \textit{max} expressions to simultaneously declare and enforce the bounds of any troubling expression.

Now that expressions giving the bounds of each function have been computed, we replace references to functions with loads from or stores to their realizations, and perform a conventional constant-folding and simplification pass. The imperative representation is then translated directly to LLVM IR with a few architecture-specific modifications.

5.3 CPU Code Generation

Generating machine code from our imperative representation is largely left to LLVM, with two caveats:

First, LLVM IR has no concept of a parallel for loop. For the CPU targets we implement these by lifting the body of the for loop into a separate function that takes as arguments a loop index and a closure containing the referenced external state. At the original site of the loop we insert code that generates a work queue containing a single task representing all instances of the loop body. A thread pool then examines this task until it is complete. If a worker thread encounters a nested parallel for loop this is pushed onto the same task queue, with the thread that encountered it responsible for managing the corresponding task.

Second, while LLVM has native vector types, it does not reliably generate good vector code in many cases on both ARM (targeting the NEON SIMD unit) and x86 (using SSE). In these cases we perform peephole optimization patterns in our representation, replacing them with calls to architecture-specific intrinsics. For example, while it is possible to perform efficient strided vector loads on both ARM for small strides, naive use of LLVM compiles them as general gathers. We can leverage more information than is available to LLVM to generate better code.

5.4 GPU Code Generation

When targeting the GPU, the compiler still generates functions with the same calling interface: a host function which takes scalar and buffer arguments. We compile the Halide algorithm into a heterogeneous program which manages both host and device execution.

The schedule describes how portions of the algorithm should be mapped to GPU execution. It tags dimensions as corresponding to the grid dimensions of the GPU’s data-parallel execution model (threads and blocks, across up to 3 dimensions). Each of the resulting loop nests is mapped to a GPU kernel, launched over a grid large enough to contain the number of threads and blocks active at the widest point in that loop nest. Operations scheduled outside the kernel loop nests execute on the host CPU, using the same scheduling primitives and generating the same highly optimized x86/SSE code as when targeting the host CPU alone.

Fusion is achieved by scheduling functions \textit{inline}, or by \textit{chunking} at the GPU block dimension. We can describe a wide space of kernel fusion choices for complex pipelines simply by changing the schedule.

The host side of the generated code is responsible for managing most data allocation and movement, GPU kernel launch, and synchronization. Allocations scheduled outside GPU thread blocks are allocated in host memory, managed by the host runtime, and copied to GPU global memory when and if they are needed by a kernel. Allocations within thread blocks are allocated in GPU shared memory, and allocations within threads in GPU thread-local memory.

Finally, we allow associative reductions to be executed in parallel on the GPU using its native atomic operations.

6 Applications and Evaluation

We present four image processing applications that test different aspects of our approach. For each we compare both our performance and our implementation complexity to existing optimized solutions. The results are summarized in Fig. 2. The Halide source for each application can be found in the supplemental materials.

All performance results are reported as the best of five runs on a 3GHz quad-core x86 desktop, a Nokia N900 mobile phone with a 600MHz ARM OMAP3 CPU, a dual core ARM OMAP4 development board (equivalent to an iPad 2), and an NVIDIA Tesla C2070 GPU (equivalent to a mid-range consumer GPU). In all cases, the algorithm code does not change between targets.

6.1 Camera Pipeline

We implement a simple camera pipeline that converts raw data from the image sensor into color images (Fig. 6). This pipeline performs four tasks: hot-pixel suppression, demosaicking, color correction, and a tone curve that applies gamma correction and contrast enhancement. This reproduces the software pipeline from the Frankencamera [Adams et al. 2010], which was written in a heavily optimized mixture of vector intrinsics and raw ARM assembly targeted at the OMAP3 processor in the Nokia N900. Our code is shorter and simpler, while also slightly faster and portable to other platforms.

The tightly bounded stencil communication down the pipeline makes fusion of stages to save bandwidth and storage a critical optimization for this application. In the Frankencamera implementation, the entire pipeline is computed on small tiles to take advantage of producer-consumer locality and minimize memory footprint. Within each tile, the evaluation of each stage is vectorized. These strategies render the algorithm illegible (see the supplemental
material). Portability is sacrificed completely: an entirely separate, slower C version of the pipeline has to be included in the Frankencamera source in order to be able to run the pipeline on a desktop processor.

We can express the same optimizations used in the Frankencamera assembly, separately from the algorithm: the output is tiled, and each stage is computed in chunks within those tiles, and then vectorized. This requires one line of scheduling choices per pipeline stage. With these transformations, our implementation takes 741 ms to process a 5 megapixel raw image on a Nokia N900 running the Frankencamera code, while the Frankencamera implementation takes 772 ms. We specify the algorithm in 145 lines of code, and the schedule in 23 (see supplemental material). The Frankencamera code uses 463 lines to specify both. Our implementation is also portable, whereas the Frankencamera assembly is entirely platform specific. The same Halide code compiles to multithreaded x86 SSE code, which takes 51 ms on our quad-core desktop.

6.2 Local Laplacian Filters

One of the most important tasks in producing compelling photographic images is adjusting local contrast. Paris et al. [2011] introduced local Laplacian filters for this purpose. The technique was then modified and accelerated by Aubry et al. [2011] (Fig. 7). This algorithm exhibits a high degree of data parallelism, which the original authors took advantage of to produce an optimized implementation using a combination of Intel Performance Primitives [IPP] and OpenMP [OpenMP].

We implemented this algorithm in Halide, and explored several strategies for scheduling it efficiently on several different machines. The statement of the algorithm did not change during the exploration of the space of plausible schedules. We found that on several x86 platforms, the best performance came from a complex schedule involving inlining certain stages, and vectorizing and parallelizing the rest. Using this schedule on our quad-core desktop, processing a 4 megapixel image takes 293 ms. On the same processor the hand-optimized version used by Aubry et al. takes 627 ms. The reference implementation requires 262 lines of C++, while in Halide the same algorithm is 62 lines. The schedule is specified using seven lines of code.

A schedule equivalent to naive C, with all major stages scheduled root, performs much less redundant computation than the fastest schedule, but takes 1.5 seconds because it sacrifices producer-consumer locality and is limited by memory bandwidth. The best schedule on a dual core ARM OMAP4 processor is slightly different. While the same stages should be inlined, vectorization is not worth the extra instructions, as the algorithm is bandwidth-bound rather than compute-bound. On the ARM processor the algorithm takes 5.5 seconds with vectorization and 4.2 seconds without. Naive evaluation takes 9.7 seconds. The best schedule for the ARM takes 427ms on the x86—50% slower than the best x86 schedule. (A range of schedule choices from our exploration, along with their performance on several architectures, are shown at the end of this application’s source code in our supplemental material.)

This algorithm maps well to the GPU, where processing the same four-megapixel image takes only 49 milliseconds. The best schedule evaluates most stages as root, but fully fuses (inlines) all of the Laplacian pyramid levels wherever they are used, trading increased computation for reduced bandwidth and storage. Each stage is split into 32x32 tiles that each map to a single CUDA block. The same algorithm statement then compiles to 83 total invocations of 25 distinct CUDA kernels, combined with host CPU code that precomputes lookup tables, manages device memory and data movement, and synchronizes the long chain of kernel invocations. Writing such code by hand is a daunting prospect, and would not allow for the rapid performance-space exploration that Halide provides.

6.3 The Bilateral Grid

The bilateral filter [Paris et al. 2009] is used to decompose images into local and global details. It is efficiently computed with the bilateral grid algorithm [Chen et al. 2007; Paris and Durand 2009]. This pipeline combines three different types of operation (Fig. 8). First, the grid is constructed with a reduction, in which a weighted histogram is computed over each tile of the input. These weighted histograms become columns of the grid, which is then blurred with a small-footprint filter. Finally, the grid is sampled using trilinear interpolation at irregular data-dependent locations to produce the output image.

We implemented this algorithm in Halide and found that the best schedule for the CPU simply parallelizes each stage across an appropriate axis. The only stage regular enough to benefit from vectorization is the small-footprint blur, but the commonly used filter sizes the time taken by the blur is insignificant. Using this schedule on our quad-core x86 CPU, we compute a bilateral filter of a four megapixel input using typical filter parameters (spatial standard deviation of 8 pixels, range standard deviation of 0.1), in 80 ms. In comparison, the moderately-optimized C++ version provided by Paris and Durand [2009] takes 472 ms using a single thread on the same machine. Our single-threaded runtime is 254 ms; some of our speedup is due to parallelizing, and some is due to generating supe-
The bilateral filter smoothes detail without losing strong edges. It is useful for a variety of photographic applications including tone-mapping and local contrast enhancement. The bilateral grid computes a fast bilateral filter by scattering the input image onto a coarse three-dimensional grid using a reduction. This grid is blurred, and then sampled to produce the smoothed output.

The speed was unimpressive—only 2× faster than our optimized CPU code, due to the high degree of contention. We modified the schedule to use one thread per tile of the input, with each thread walking serially over the reduction domain. This one-line change in schedule gives us a runtime of 11 ms for the same four megapixel image. Halide still conservatively injects an atomic add operation, but with no contention this is in fact 10% faster than non-atomic floating point read-modify-writes to the same memory location, as it offloads the addition to a separate unit. The same 34-line Halide algorithm now runs over 40× faster than the more verbose reference C++ implementation.

Active contour selection (a.k.a. *snake* [Kass et al. 1988]) is a method for segmenting objects from a background (Fig. 9). It is well suited for medical applications. We implemented the algorithm proposed by Li et al. [2010]. The algorithm is iterative, and can be interpreted as a gradient-descent optimization of a 2D function. Each update of this function is composed of three terms (Fig. 9), each of them being a combination of differential quantities computed with small $3 \times 1$ and $1 \times 3$ stencils, and point-wise nonlinear operations, such as normalizing the gradients.

We factored this algorithm into three feed-forward pipelines. Two simple pipelines create images that are invariant to the optimization loop, and one primary pipeline performs a single iteration of the optimization loop. While Halide can represent bounded iteration over the outer loop using a reduction, it is more naturally expressed in the imperative host language. We construct and chain together these pipelines at runtime using Halide as a just-in-time compiler in order to perform a fair evaluation against the reference implementation from Li et al., which is written in MATLAB. MATLAB is notoriously slow when misused, but this code expresses all operations in the data-parallel array-wise notation that MATLAB executes most efficiently.

On a 1600 × 1200 test image, our Halide implementation takes 55 ms per iteration of the optimization loop on our quad-core x86, whereas the MATLAB implementation takes 3.8 seconds. Our schedule is expressed in a single line: we parallelize and vectorize the output of each iteration, while leaving every other function to be inlined by default. The bulk of the speedup comes not from vectorizing or parallelizing: without them, our implementation still takes just 202 ms per iteration. The biggest difference is that we have completely fused the operations that make up one iteration. MATLAB expresses algorithms as sequences of many simple array-wise operations, and is heavily limited by memory bandwidth. It is equivalent to scheduling every operation as *root*, which is a poor choice for algorithms like this one.

The fully-fused form of this algorithm is also ideal for the GPU, where it takes 3 ms per iteration.

**6.5 Discussion and Future Work**

The performance gains we’ve found in this section demonstrate the feasibility and power of separating algorithms from their schedules. Changing the schedule enables a single algorithm definition to achieve high performance on a diversity of targets. On a single machine, it enables rapid performance space exploration. Furthermore, the algorithm specification becomes considerably more concise once scheduling concerns are separated.

While the set of scheduling choices we enumerate proved sufficient for these applications, there are other interesting options that our representation could incorporate, such as sliding window schedules in which multiple evaluations are interleaved to reduce storage, or dynamic schedules in which functions are computed lazily and then cached for reuse. We are also exploring autotuning and heuristic optimization enabled by our ability to enumerate the space of legal schedules. We further believe we can continue to clarify the algorithm specification with more aggressive inference.

Some image processing algorithms include constructs beyond the capabilities of our current representation, such as non-image data structures like lists and graphs, and optimization algorithms that use iteration-until-convergence. We believe that these and other patterns can also be decoupled from their schedules, but this remains future work.

**7 Conclusion**

Image processing pipelines are simultaneously deep and wide; they contain many simple stages that operate on large amounts of data. This makes the gap between naive schedules and highly parallel execution that makes efficient use of the memory hierarchy large—often an order of magnitude. And speed matters for image process-
ing. People expect image processing that is interactive, that runs on their cell phone or their camera. An order of magnitude in speed is often the difference between an algorithm being used in practice, or not at all.

With existing tools, closing this gap requires ninja programming skills; imaging pipelines must be painstakingly globally transformed to simultaneously maximize parallelism and memory-efficiency. The resulting code is often impossible to modify, reuse, or port efficiently to other processors. In this paper we have demonstrated that it is possible to earn this order of magnitude with less programmer pain, by separately specifying the algorithm and its schedule—the decisions about ordering of computation and storage that are critical for performance but irrelevant to correctness. Decoupling the algorithm from its schedule has allowed us to compile simple expressions of complex image processing pipelines to implementations with state-of-the-art performance across a diversity of devices.

References


OPENMP. OpenMP. http://openmp.org/.


