

# Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Jonathan Ragan-Kelley\* Andrew Adams\* Sylvain Paris†

\*MIT CSAIL †Adobe

Marc Levoy‡ Saman Amarasinghe\* Frédo Durand\*

‡Stanford University

## 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 called Halide, 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.

**CR Categories:** I.3.6 [Computer Graphics]: Methodology and Techniques—Languages

**Keywords:** Image Processing, Compilers, Performance

**Links:** DL PDF WEB CODE

## 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, optimization usually comes at the cost of programmer pain and code complexity, as computation must be reorganized to achieve memory efficiency and parallelism.

———— (a) Clean C++ : 9.94 ms per megapixel ——  

```
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

———— (b) Fast C++ (for x86) : 0.90 ms per megapixel ——  

```
void fast_blur(const Image &in, Image &blurred) {
    _m128i one_third = _mm_set1_epi16(21846);
#pragma omp parallel for
for (int yTile = 0; yTile < in.height(); yTile += 32) {
    _m128i a, b, c, sum, avg;
    _m128i tmp[(256/8)*(32+2)];
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
        _m128i *tmpPtr = tmp;
        for (int y = -1; y < 32+1; y++) {
            const uint16_t *inPtr = &(in(xTile, yTile+y));
            for (int x = 0; x < 256; x += 8) {
                a = _mm_load_si128((._m128i *) (inPtr-1));
                b = _mm_load_si128((._m128i *) (inPtr+1));
                c = _mm_load_si128((._m128i *) (inPtr));
                sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                avg = _mm_mulhi_epi16(sum, one_third);
                _mm_store_si128(tmpPtr++, avg);
                inPtr += 8;
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++) {
                _m128i *outPtr = (._m128i *) (&(blurred(xTile, yTile+y)));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(tmpPtr+(2*256)/8);
                    b = _mm_load_si128(tmpPtr+256/8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

———— (c) Halide : 0.90 ms per megapixel ——  

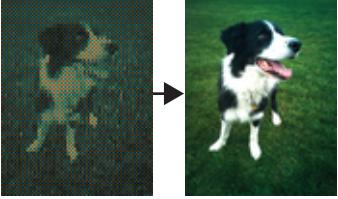
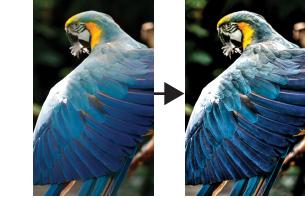
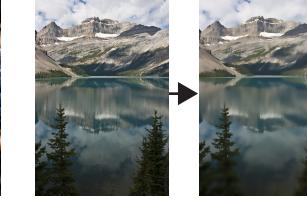
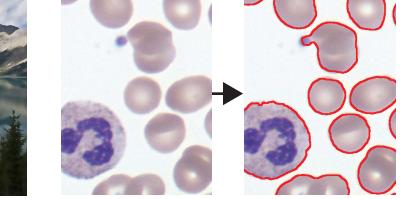
```
Func halide.blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    // The schedule
    blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    tmp.chunk(x).vectorize(x, 8);

    return blurred;
}
```

**Figure 1:** The code at the top computes a  $3 \times 3$  box filter using the composition of a  $1 \times 3$  and a  $3 \times 1$  box filter (a). Using vectorization, multithreading, tiling, and fusion, we can make this algorithm more than  $10 \times$  faster on a quad-core x86 CPU (b). 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 (c). For the full details about how this test was carried out, see the supplemental material.

			
<b>Camera Raw Pipeline</b>	<b>Local Laplacian Filter</b>	<b>Bilateral Grid</b>	<b>Snake Image Segmentation</b>
Optimized NEON ASM: Nokia N900: 463 lines 772 ms	C++, OpenMP+IPP: Quad-core x86: 262 lines 335 ms	Tuned C++: Quad-core x86: 122 lines 472ms	Vectorized MATLAB: Quad-core x86: 67 lines 3800 ms
Halide algorithm: schedule: Nokia N900: 145 lines 23 lines 741 ms	Halide algorithm: schedule: Quad-core x86: 62 lines 7 lines 158 ms	Halide algorithm: schedule: Quad-core x86: 34 lines 6 lines 80 ms	Halide algorithm: schedule: Quad-core x86: 148 lines 7 lines 55 ms
2.75x shorter 5% faster than tuned assembly	3.7x shorter 2.1x faster	3x shorter 5.9x faster	2.2x longer 70x faster
Porting to new platforms does not change the algorithm code, only the schedule			
Quad-core x86: 51 ms	CUDA GPU: 48 ms (7x)	CUDA GPU: 11 ms (42x) Hand-written CUDA: 23 ms [Chen et al. 2007]	CUDA GPU: 3 ms (1250x)

**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 and hand-written CUDA [Adams et al. 2010; Aubry et al. 2011; Paris and Durand 2009; Chen et al. 2007; 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.

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 for 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). Programmers still specify the strategy for execution, since automatic optimization remains hard, but doing so is radically simplified by this split representation.

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 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 11× 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 efficiently 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 obscuring 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 are 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.

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 and does not affect the correctness of the algorithm (e.g. Fig. 1.c), so exploring the performance of many options is fast

and easy. It can be written separately from the algorithm, by an architecture expert if necessary. 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, single program-multiple data programming model which can target both GPUs and multicore CPUs with SIMD units [Buck 2007; OpenCL 2011]. ispc provides a similar abstraction for SIMD processing on x86 CPUs [Pharr and Mark 2012]. Like C, they allow the specification of very high performance implementations for many algorithms. But because parallel work distribution, synchronization, kernel fusion, 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.

Array Building Blocks provides an embedded language for data-parallel array processing in C++ [Newburn et al. 2011]. As in our representation, whole pipelines of operations are built up and optimized globally by a compiler. It delivers impressive performance on Intel CPUs, but requires a sufficiently smart compiler to do so.

Streaming languages encode data and task parallelism in graphs of kernels. Compilers automatically schedule these graphs using tiling, fusion, and fission [Kapasi et al. 2002]. Sliding window optimizations can automatically optimize pipelines with overlapping data access in 1D streams [Gordon et al. 2002]. Our model of scheduling addresses the problem of overlapping 2D stencils, where recomputation vs. storage becomes a critical but complex choice. We assume a less heroic compiler, and focus on enabling human programmers to quickly and easily specify complex schedules.

**Programmer-controlled scheduling** A separate line of compiler research attempts to put control back in the hands of the programmer. The SPIRAL system [Püschel et al. 2005] uses a domain-specific language to specify linear signal processing operations independent of their schedule. Separate mapping functions describe how these operations should be turned into efficient code for a particular architecture. It enables high performance across a range of architectures 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.

Sequoia defines a model where a user-defined “mapping” describes how to execute tasks on a tree-like memory hierarchy [Fatahalian et al. 2006]. This parallels our model of scheduling, but focuses on hierarchical problems like blocked matrix multiply, rather than pipelines of images. Sequoia’s mappings, which are highly explicit, are also more verbose than our schedules, which are designed to infer details not specified by the programmer.

**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]. Neon embeds a similar kernel language in C# [Guenter and Nehab 2010]. All three compile kernels into optimized code for multiple architectures, including CPU SIMD instructions and GPUs, but none optimize across kernels connected by complex communication like stencils, and none support reductions or nested parallelism within kernels.

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 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 (Fig. 2).

Spreadsheets for Images extended the spreadsheet metaphor as a functional programming model for imaging operations [Levoy 1994]. Pan introduced a functional model for image processing much like our own, in which images are functions from coordinates to values [Elliott 2001]. 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 model of *scheduling* and ultimate compilation. It exists as an interpreted embedding within Haskell, and as source to source compiler to C containing basic scalar and loop optimizations [Elliott et al. 2003].

### 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 over a bounded domain. 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 describe 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 reintroduced as syntactic sugar, but they do not change the underlying representation (Sec. 4.1).

**Reduction functions.** In order to express operations like histograms and general convolutions, we need a way to express iterative or recursive computations. We call these reductions because this class of functions includes, but is not limited to, traditional reductions such as summation. Reductions are defined recursively, and consist of two parts:

- An *initial value function*, which specifies a value at each point in the *output domain*.
- A *recursive reduction function*, which redefines the value at points given by an *output coordinate expression* in terms of prior values of the function.

Unlike a pure function, the meaning of a reduction depends on the order in which the reduction function is applied. We require the programmer to specify the order by defining a *reduction domain*, bounded by minimum and maximum expressions for each dimension. The value at each point in the output domain is defined by the final value of the reduction function at that point, given recursive in lexicographic order across the reduction domain.

In the case of a histogram, the *reduction domain* is the input image, the *output domain* is the histogram bins, the *initial value* is 0,

```
UniformImage in(UInt(8), 2);
Func histogram, cdf, out;
RDom r(0, in.width(), 0, in.height()), ri(0, 255);
Var x, y, i;

histogram(in(r.x, r.y))++;
cdf(i) = 0;
cdf(ri) = cdf(ri-1) + histogram(ri);
out(x, y) = cdf(in(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 *RDom*. Like all functions in our representation, *histogram* and *cdf* are defined over an infinite domain. Entries not touched by the reduction step are zero-valued. For *cdf*, this is specified explicitly. For *histogram*, it is implicit in the `++` operator.

the *output coordinate* is the intensity of the input image, and the *reduction function* increments the value in the corresponding bin.

From the perspective of a caller, the result of the reduction is defined over an infinite domain, like any other function. At points which are never specified by an *output coordinate*, the value is the initial expression.

This relatively simple pattern can describe a range of naturally iterative algorithms in a way that bounds side effects, but still allows easy conversion to efficient implementations which need to allocate only a single value for each point in the *output domain*. Several reductions are combined to perform histogram equalization in Fig. 3.

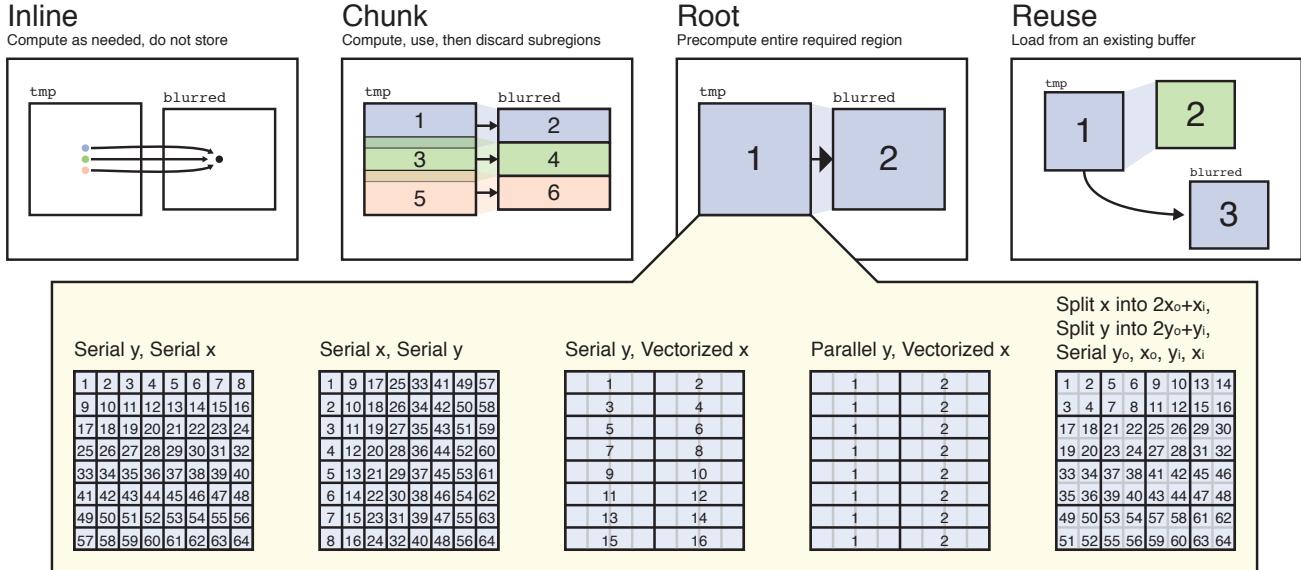
#### 3.2 The Schedule

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 often constrained by memory bandwidth and inlining passes values between functions with maximum locality, usually in registers.



**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 of its inputs. 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 (top left). This gives excellent temporal locality and requires minimal storage, but each point of **tmp** will be computed three times, once for each use of each point in **tmp**. **blurred** may compute and consume **tmp** in larger chunks. This provides some producer-consumer locality, and isolates redundant computation at the chunk boundaries (visible as overlapping transparent regions above). At the extreme, **blurred** may compute all of **tmp** before using any of it. We call this root. It computes each point of **tmp** only once, but requires storage for the entire region, and producer-consumer 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**. These choices define the interleaving of the dimensions (e.g. row- vs. column-major, bottom left), and the serial or parallel evaluation of each dimension. Dimensions may be split and their sub-dimensions further scheduled (e.g., to produce tiled traversal orders, bottom right).

**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.

**Chunk: compute, use, then discard subregions.** Alternatively, a function can be *chunked* with respect to a dimension of its caller. Each iteration of the caller over that dimension first computes all values of the callee needed *for that iteration only*. Chunking interleaves the computation of sub-regions of the caller and the callee, trading off producer-consumer locality and reduced storage footprint for potential recomputation when chunks required for different iterations of the caller overlap.

**Reuse: load from an existing buffer.** Finally, if a function is computed in *chunks* or at the *root* for one caller, another caller may *reuse* that evaluation. Reusing a chunked evaluation is only legal if it is also in scope for the new caller. Reuse is typically the best option when available.

Imaging applications exhibit a fundamental tension between total fusion down the pipeline (*inline*), which maximizes producer-consumer locality at the cost of recomputation of shared values, and breadth-first execution (*root*), which eliminates recomputation

at the cost of locality. This is often resolved by *splitting* a function's domain and *chunking* the functions upstream at a finer granularity. This achieves reuse for the inner dimensions, and producer-consumer locality for the outer ones. Choosing the granularity trades off between locality, storage footprint, and recomputation. A key purpose of our schedule representation is to span this continuum, so that the best choice may be made in any given context.

**Order of domain evaluation.** The other essential axis of control is the order of evaluation within the required region of each function, including parallelism and tiling. While evaluating a function scheduled as *root* or *chunk*, the 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. 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 rep-

resent evaluation in 2D 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  $y_o, x_o, y_i, x_i$  (illustrated in the lower right of Fig. 4). After a dimension has been split, the inner and outer components are recursively scheduled using any of the options above. Chunked call schedules, combined with split iteration dimensions, describe the common pattern of loop tiling and stripmining (as used in Fig. 1). Recursive splitting describes hierarchical tiling.

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. Tiled access patterns can be extremely important in maximizing locality and cache efficiency, and are a key effect of our schedules. The storage layout for each region, however, is not controlled by the schedule. Tiled storage layouts have mattered surprisingly little on all architectures and applications we have tried, so we do not include them. Cache lines are usually smaller than tile width, so tiled layout in main memory often has limited effect on cache behavior.

**Scheduling reductions.** The schedule for a reduction must specify a pair of loop nests: one for the initial value (over the output domain), and one for the reduction step (over the reduction domain). In the latter case, the bounds are given by the definition of the reduction, and do not need to be inferred later. Since the meaning of reductions is partially order-dependent, it is illegal for the schedule to change the order of dimensions in the update in such a way that changes the meaning. But while we semantically define reductions to follow a strict lexicographic traversal order over the reduction domain, many common reductions (such as *sum* and *histogram*) are associative, and may be executed in parallel. Scans like *cdf* are more challenging to parallelize. We do not yet address this.

### 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 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, transcendentals, 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:

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

Reductions are declared by providing two definitions for a function: one for its initial value, and one for its reduction step. The reduction step should be defined in terms of the dimensions of a reduction domain (of type `RDom`), which include expressions describing their bounds (min and extent). The left-hand-side of the reduction step may be a computed location rather than simple variables (Fig. 3).

We can initialize the bounds of a reduction domain based on the dimensions of an input image. 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`:

```
Func histogram;
RDom r(im);
histogram(im(r.x, r.y))++;
```

**Uniforms** describe the run-time parameters of an imaging pipeline. They may be scalars or entire images (in particular, an 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.

```
// 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`:

```
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", {scale, input});
```

This produces `bright.o` and `bright.h`, which together define a C callable function with the following type signature:

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

where `buffer_t` is a simple image struct defined in the same header.

**Value types.** Expressions, functions, and uniforms may have floating point, signed, or unsigned integer type of any natively-supported bit width. Domain variables are 32-bit signed integers.

## 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:

```
// 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 automatically lift any operator which combines partially applied functions 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 `x`, `g`, and `b`, the definition `f(x, y) = (x, g, b)` creates a three-dimensional function (in this case representing a color image) whose last argument selects between `x`, `g`, and `b`. It is equivalent to `f(x, y, c) = select(c==0, x, select(c==1, g, b))`.

**Inline reductions.** We provide syntax for inlining the most commonly-occurring reduction patterns: `sum`, `product`, `maximum`, and `minimum`. These simplified reduction operators implicitly use any `RDom` referenced with as the reduction domain. For example, a blurred version of some image `f` can be defined as follows:

```
Func blurry; Var x, y;
RDom r(-2, 5, -2, 5);
blurry(x, y) = sum(f(x+r.x, y+r.y));
```

## 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.

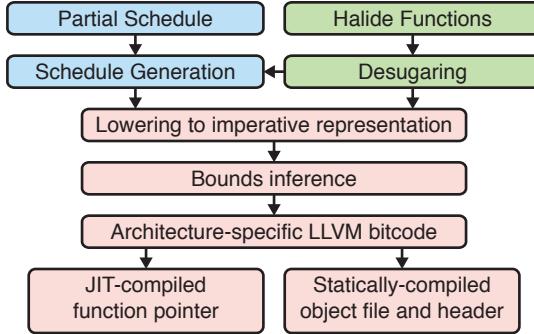
If `im` is scheduled as *root* or *chunk*, we must also specify the traversal order of the domain. By default it is traversed serially in scanline order. This can be modified using the following methods:

- `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 independently marked as parallel, serial, vectorized, or even recursively split.
- `im.tile(x, y, xi, yi, tw, th)` is a convenience method that splits `x` by a factor of `tw`, and `y` by a factor of `th`, 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, tw, th)` is a similar convenience method to `tile`. It splits `x` and `y` by `tw` and `th` 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 change the statement of the algorithm. Furthermore, each scheduling method returns a reference to the function, so calls can be chained: *e.g.*, `im.root().vectorize(x, 4).transpose(x, y).parallel(x)` directs the compiler to evaluate `im` in vectors of width 4, operating on every column in parallel, with each thread walking down its column serially.

## 5 Compiler Implementation

The Halide compiler lowers imaging pipelines into machine code for ARM, x86, and PTX. It uses the LLVM compiler infrastructure for conventional scalar optimizations, register allocation, and machine code generation [LLVM]. 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.

## 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 inlining 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 final 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  $\mathbf{f}$ :

```
for (i from a 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  $f$ . 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 *min*, *max*, and *clamp* 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 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:

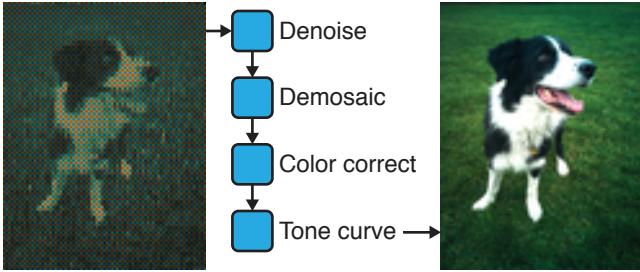
First, LLVM IR has no concept of a parallel for loop. For the CPU targets we implement these by lifting the body of the for loop into a separate function that takes as arguments a loop index and a closure containing the referenced external state. At the original site of the loop we insert code that generates a work queue containing a single task representing all instances of the loop body. A thread pool then nibbles at 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 peephole optimize 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 x86 and 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 CUDA Code Generation

When targeting CUDA, 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 CUDA execution. It tags dimensions as corresponding to the grid dimensions of CUDA's data-parallel execution model (threads and blocks, across up to 3 dimensions). Each of the resulting loop nests is mapped to a CUDA 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



**Figure 6:** The basic camera post-processing pipeline is a feed-forward pipeline in which each stage either considers only nearby neighbors (denoise and demosaic), or is point-wise (color correct and tone curve). The best schedule computes the entire pipeline in small tiles in order to exploit producer-consumer locality. This introduces redundant computation in the overlapping tile boundaries, but the reduction in memory bandwidth more than makes up for it.

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 *inline*, or by *chunking* at the CUDA block dimension. We can describe many 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, CUDA kernel launch, and synchronization. Allocations scheduled outside CUDA thread blocks are allocated in host memory, managed by the host runtime, and copied to CUDA global memory when and if they are needed by a kernel. Allocations within thread blocks are allocated in CUDA shared memory, and allocations within threads in CUDA thread-local memory.

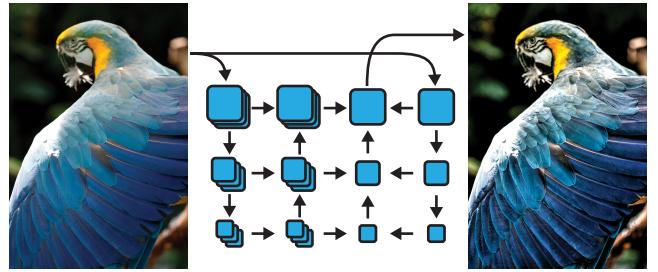
Finally, we allow associative reductions to be executed in parallel on the GPU using its native atomic operations.

## 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. Performance results are reported as the best of five runs on a 3GHz Core2 Quad x86 desktop, a 2.5GHz quad-core Core i7-2860QM x86 laptop, 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. (All application code and schedules are included in supplemental material.)

### 6.1 Camera Pipeline

We implement a simple camera pipeline that converts raw data from an image sensor into color images (Fig. 6). The pipeline performs four tasks: hot-pixel suppression, demosaicing, color correction, and a tone curve that applies gamma correction and contrast. 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.



**Figure 7:** The local Laplacian filter enhances local contrast using Gaussian and Laplacian image pyramids. The pipeline mixes images at different resolutions with a complex network of dependencies. While we show three pyramid levels here, for our four megapixel test image we used eight.

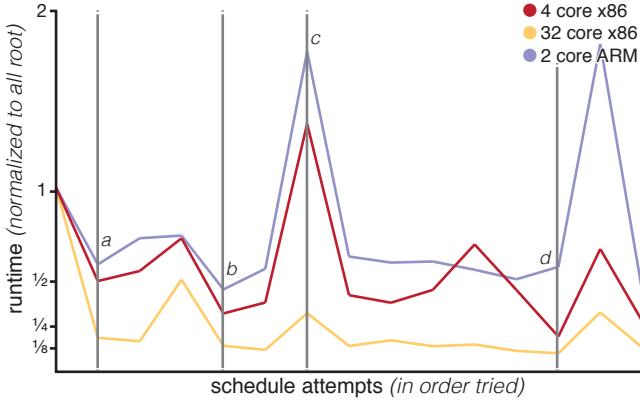
The tightly bounded stencil communication down the pipeline makes fusion of stages to save bandwidth and storage a critical optimization for this application. In the Frankencamera implementation, the entire pipeline is computed on small tiles to take advantage of producer-consumer locality and minimize memory footprint. Within each tile, the evaluation of each stage is vectorized. These strategies render the algorithm illegible. 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. 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 multiple strategies for scheduling it efficiently on several different machines (Fig. 8). The statement of the algorithm did not change during the exploration 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 laptop, processing a 4 megapixel image takes 158 ms. On the same processor the hand-optimized version used by Aubry et al. takes 335 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 third implementation, in ispc [Pharr and Mark 2012], us-

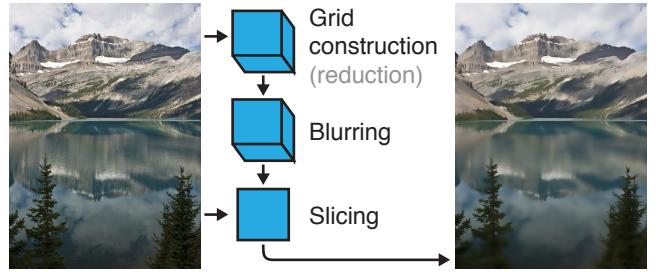


**Figure 8:** We found effective schedules for the local Laplacian filter by manually testing and refining a small, hand-tuned schedule, across a range of multicore CPUs. Some major steps are highlighted. To begin, all functions were scheduled as `root` and computed serially. (a) Then, each stage was parallelized over its outermost dimension. (b) Computing the Laplacian pyramid levels `inline` improves locality, at the cost of redundant computation. (d) But excessive inlining is dangerous: the high spike in runtimes results from additionally inlining every other Gaussian pyramid level. (d) The best performance on the x86 processors required additionally inlining only the bottom-most Gaussian pyramid level, and vectorizing across `x`. The ARM performs slightly better with a similar schedule, but no vectorization. The entire optimization process took only a couple of hours. (The full sequence of schedules from this graph, and their performance, are shown at the end of this application’s source code in supplemental material.)

ing OpenMP to distribute the work across multiple cores, used 288 lines of code. It is longer than in Halide due to explicit boundary handling, memory management, and C-style kernel syntax. The ispc implementation takes 327 ms to process the 4-megapixel image. The Halide implementation is faster due to fusion down the pipeline. The ispc implementation can be manually fused by rewriting it, but this would further lengthen and complicate the code.

A schedule equivalent to naive parallel C, with all major stages scheduled as `root` but evaluated in parallel over the outer dimensions, performs much less redundant computation than the fastest schedule, but takes 296 ms 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 278 ms on the x86 laptop—75% longer than the best x86 schedule.

This algorithm maps well to the GPU, where processing the same four-megapixel image takes only 49 ms. 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, similar to the x86 and ARM schedules. Each stage is split into  $32 \times 32$  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.



**Figure 9:** 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.

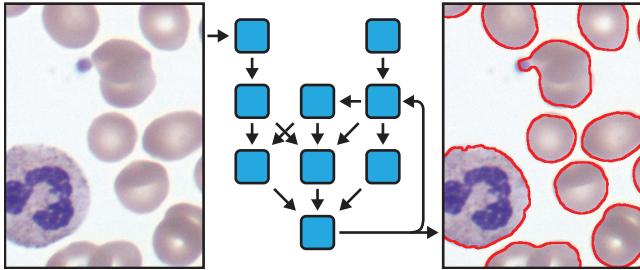
### 6.3 The Bilateral Grid

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. 9). 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 for commonly used filter sizes the time taken by the blur is insignificant. Using this schedule on our quad-core x86 desktop, 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 parallelism, and some is due to generating superior scalar code. We use 34 lines of code to describe the algorithm, and 6 for its schedule, compared to 122 lines in the C++ reference.

We first tried running the same algorithm on the GPU using a schedule which performs the reduction over each tile of the input image on a single CUDA block, with each thread responsible for one input pixel. Halide detected the parallel reduction, and automatically inserted atomic floating point adds to memory. The runtime was 40 ms—only 2× faster than our optimized CPU code, due to atomic contention. The latest hand-written GPU implementation by Chen et al. [2007] expresses the same algorithm and a similar schedule in 370 lines of CUDA C++, and takes 24 ms on the same GPU.

With the rapid schedule exploration enabled by Halide, we quickly found a better schedule that trades off some parallelism to reduce atomics 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 image. When we rewrite the hand-tuned CUDA implementation to match the schedule found with Halide, it takes 8 ms. The 3 ms improvement over Halide comes from the use of texture units for the slicing stage. Halide does not currently use texture hardware. In general, hand-tuned CUDA can surpass the performance Halide achieves when there is a significant win from clever use of specific CUDA features not expressible in our schedule, but exploring different optimization strategies is much harder



**Figure 10:** Adaptive contours segment objects from the background. Level-set approaches are useful to cope with smooth objects and when the number of elements is unknown. The algorithm iterates a series of differential operators and nonlinear functions to progressively refine the selection. The final result is a set of curves that tightly delineate the objects of interest (in red on the right).

than in Halide. Compared to the original CUDA bilateral grid, the schedule found with Halide saved 13 ms, while the clever use of texture units saved 3 ms.

With the final GPU schedule, the same 34-line Halide algorithm runs over 40× faster than the more verbose reference C++ implementation on the CPU, and twice as fast as the reference CUDA implementation using 1/10th the code.

#### 6.4 Image Segmentation using Level Sets

Active contour selection (a.k.a. *snake* [Kass et al. 1988]) is a method for segmenting objects from a background (Fig.10). 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. 10), 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 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 array-wise notation that MATLAB executes most efficiently.

On a  $1600 \times 1200$  test image, our Halide implementation takes 55 ms per iteration of the optimization loop on our quad-core x86 desktop, 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 have found on these applications 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 machines. On a single machine, it enables rapid performance space exploration. The algorithm specification also 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. Heterogeneous architectures are an important potential target. Our existing implementation already generates mixed CPU & GPU code, with the schedule managing the orchestration. On PCs with discrete GPUs, data movement costs tend to preclude fine-grained collaboration, but on more integrated SoCs being able to quickly explore a wide range of schedules combining multiple execution resources is appealing.

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 unified into a similar programming model, but doing so remains an open challenge.

### 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 efficiently uses the memory hierarchy large—often an order of magnitude. And speed matters for image processing. People expect image processing that is interactive, that runs on their cell phone or camera. An order of magnitude in speed is often the difference between an algorithm being used in practice, and not being used 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 into implementations with state-of-the-art performance across a diversity of devices. We have done so without a heroic compiler. Rather, we have found that the most practical design provides programmer control over both algorithm and schedule, while inferring and mechanizing as many low-level details as possible to make this high-level control manageable. This is in contrast to most compiler research, but it is what made it feasible to achieve near peak performance on these real applications with a simple and predictable system.

However, we think future languages *should* exploit compiler automation. A domain-specific representation of scheduling, like the one we have demonstrated, is essential to automatically inferring similar optimizations. Even the prototype we have described infers many details in common cases. The ultimate solution must allow a smooth trade off between inference when it is sufficient, and sparse programmer control when it is necessary.

**Acknowledgments** This work was partially funded by the Quanta T-Party, NSF grants 0964004, 0964218, and 0832997, DOE award DE-SC0005288, and gifts from Cognex and Adobe.

## References

- ADAMS, A., TALVALA, E.-V., PARK, S. H., JACOBS, D. E., AJDIN, B., GELFAND, N., DOLSON, J., VAQUERO, D., BAEK, J., TICO, M., LENSCHE, H. P. A., MATUSIK, W., PULLI, K., HOROWITZ, M., AND LEVOY, M. 2010. The Frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics* 29, 4 (July), 29:1–29:12.
- AUBRY, M., PARIS, S., HASINOFF, S. W., KAUTZ, J., AND DURAND, F. 2011. Fast and robust pyramid-based image processing. Tech. Rep. MIT-CSAIL-TR-2011-049, Massachusetts Institute of Technology.
- BUCK, I. 2007. GPU computing: Programming a massively parallel processor. In *CGO ’07: Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society, 17.
- CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics* 26, 3 (July), 103:1–103:9.
- COREIMAGE. Apple CoreImage programming guide. <http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/CoreImaging>.
- ELLIOTT, C., FINNE, S., AND DE MOOR, O. 2003. Compiling embedded languages. *Journal of Functional Programming* 13, 2. Updated version of paper by the same name that appeared in SAIG ’00 proceedings.
- ELLIOTT, C. 2001. Functional image synthesis. In *Proceedings of Bridges*.
- FATAHALIAN, K., HORN, D. R., KNIGHT, T. J., LEEM, L., HOUSTON, M., PARK, J. Y., EREZ, M., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. 2006. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, SC ’06.
- FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20.
- GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LEGER, C., LAMB, A. A., WONG, J., HOFFMAN, H., MAZE, D. Z., AND AMARASINGHE, S. 2002. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
- GUENTER, B., AND NEHAB, D. 2010. The neon image processing language. Tech. Rep. MSR-TR-2010-175, Microsoft Research.
- IPP. Intel Integrated Performance Primitives. <http://software.intel.com/en-us/articles/intel-ipp/>.
- KAPASI, U. J., MATTSON, P., DALLY, W. J., OWENS, J. D., AND TOWLES, B. 2002. Stream scheduling. Concurrent VLSI Architecture Tech Report 122, Stanford University, March.
- KASS, M., WITKIN, A., AND TERZOPOULOS, D. 1988. Snakes: Active contour models. *International Journal of Computer Vision* 1, 4.
- LEVOY, M. 1994. Spreadsheets for images. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, 139–146.
- LI, C., XU, C., GUI, C., AND FOX, M. D. 2010. Distance regularized level set evolution and its application to image segmentation. *IEEE Transactions on Image Processing* 19, 12 (December), 3243–3254.
- LLVM. The LLVM compiler infrastructure. <http://llvm.org>.
- MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Graphics Hardware 2002*, 57–68.
- NEWBURN, C. J., SO, B., LIU, Z., MCCOOL, M., GHULOUM, A., TOIT, S. D., WANG, Z. G., DU, Z. H., CHEN, Y., WU, G., GUO, P., LIU, Z., AND ZHANG, D. 2011. Intel’s array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, CGO ’11, 224–235.
- OPENCL, 2011. The OpenCL specification, version 1.2. <http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>.
- OPENMP. OpenMP. <http://openmp.org/>.
- PARIS, S., AND DURAND, F. 2009. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision* 81, 1, 24–52.
- PARIS, S., KORNPROBST, P., TUMBLIN, J., AND DURAND, F. 2009. Bilateral filtering: Theory and applications. Foundations and Trends in Computer Graphics and Vision.
- PARIS, S., HASINOFF, S. W., AND KAUTZ, J. 2011. Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid. *ACM Transactions on Graphics* 30, 4.
- PHARR, M., AND MARK, W. R. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing (InPar)*.
- PIXELBENDER. Adobe PixelBender reference. [http://www.adobe.com/content/dam/Adobe/en/devnet/pixelbender/pdfs/pixelbender\\_reference.pdf](http://www.adobe.com/content/dam/Adobe/en/devnet/pixelbender/pdfs/pixelbender_reference.pdf).
- PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GACIC, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”* 93, 2, 232–275.
- SHANTZIS, M. A. 1994. A model for efficient and flexible image computing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, SIGGRAPH ’94, 147–154.