Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

Jonathan Ragan-Kelley
MIT CSAIL

Connelly Barnes
Adobe

Andrew Adams
MIT CSAIL

Sylvain Paris
Adobe

Frédo Durand
MIT CSAIL

Saman Amarasinghe
MIT CSAIL

Abstract

Image processing pipelines combine the challenges of stencil computations and stream programs. They are composed of large graphs of different stencil stages, as well as complex reductions, and stages with global or data-dependent access patterns. Because of their complex structure, the performance difference between a naive implementation of a pipeline and an optimized one is often an order of magnitude. Efficient implementations require optimization of both parallelism and locality, but due to the nature of stencils, there is a fundamental tension between parallelism, locality, and introducing redundant recomputation of shared values.

We present a systematic model of the tradeoff space fundamental to stencil pipelines, a *schedule* representation which describes concrete points in this space for each stage in an image processing pipeline, and an optimizing compiler for the Halide image processing language that synthesizes high performance implementations from a Halide *algorithm* and a *schedule*. Combining this compiler with stochastic search over the space of schedules enables terse, composable programs to achieve state-of-the-art performance on a wide range of real image processing pipelines, and across different hardware architectures, including multicores with SIMD, and heterogeneous CPU+GPU execution. From simple Halide programs written in a few hours, we demonstrate performance up to $5\times$ faster than hand-tuned C, intrinsics, and CUDA implementations optimized by experts over weeks or months, for image processing applications beyond the reach of past automatic compilers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – compilers, optimization, code generation

Keywords domain specific language; compiler; image processing; locality; parallelism; redundant computation; optimization; GPU; vectorization

1. Introduction

Image processing pipelines are everywhere, and are essential to capturing, analyzing, mining, and rendering the rivers of visual information gathered by our countless cameras and imaging-based sensors. Applications from raw processing, to object detection and recognition, to Microsoft's Kinect, to Instagram and Photoshop, to medical imaging and neural scanning all demand extremely high performance to cope with the rapidly rising resolution and frame rate of image sensors and the increasing complexity of algorithms. At the same time, the shrinking cameras and mobile devices on which they run require extremely high efficiency to last more than a few minutes on battery power. While power-hungry radios and video CODECs can implement slow-changing standards in custom hardware, image processing pipelines are rapidly evolving and diverse, requiring high performance software implementations.

Image processing pipelines combine the challenges of stencil computation and stream programs. They are composed of large graphs of many different operations, most of which are stencil computations. These pipelines are simultaneously wide and deep: each stage exhibits data parallelism across the many pixels which it must process, and whole pipelines consist of long sequences of different operations, which individually have low arithmetic intensity (the ratio of computation performed to data read from prior stages and written to later stages). For example, an implementation of one recent algorithm, local Laplacian filters [3, 22], is a graph of 99 different stages (Fig. 1), including many different stencils and a large data-dependent resampling.

As a result of this structure, the performance difference between a naive implementation of a given pipeline and a highly optimized one is frequently an order of magnitude or more. With current tools, often the only way to approach peak performance is by hand-writing parallel, vectorized, tiled, and globally fused code in low-level C, CUDA, intrinsics, and assembly. Simple pipelines become hundreds of lines of intricately interleaved code; complex pipelines, like Adobe's Camera Raw engine, become hundreds of thousands. Tuning them requires immense effort by expert programmers, and the end result is not portable to different architectures, nor composable with other algorithms, without sacrificing much of this painstakingly earned performance. Libraries of optimized subroutines do not solve the problem, either, since many critical optimizations involve fusion for producer-consumer locality across stages.

We address this challenge by raising the level of abstraction, and decoupling the algorithm definition from its execution strategy, to improve portability and composability, while automating the search for optimized mappings of the resulting pipelines to parallel machines and complex memory hierarchies. Effective abstraction and automatic optimization enable radically simpler programs to achieve higher performance than hand-tuned expert implementations, while running across a wide range of architectures.

1.1 Image Processing Pipelines

Stencils have been well studied in scientific applications in the form of iterated stencil computations, where one or a few small stencils are applied to the same grid over many iterations [10, 16, 19]. In contrast, we are interested in other applications, in image processing

PLDI '13 June 16-21, Seattle.

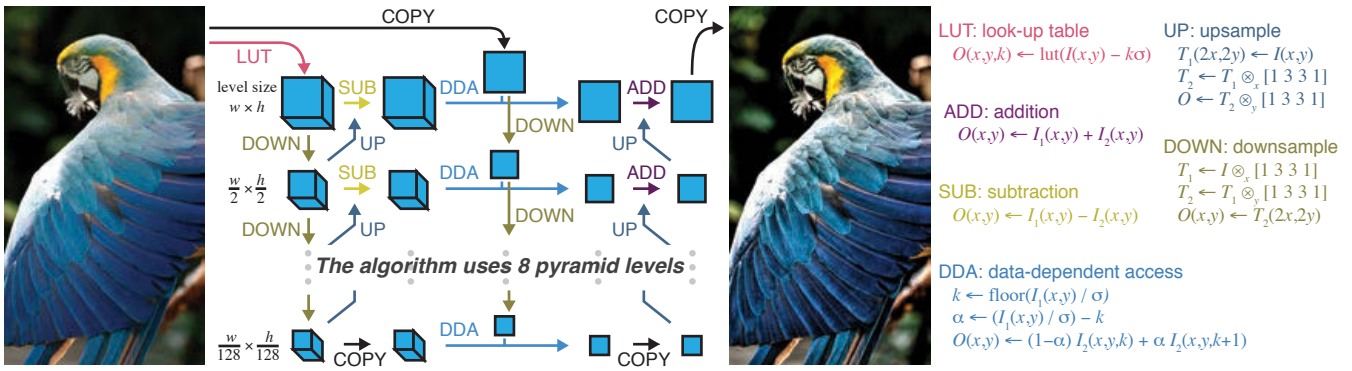

Figure 1. Imaging pipelines employ large numbers of interconnected, heterogeneous stages. Here we show the structure of the local Laplacian filter [3, 22], which is used for a variety of tasks in photographic post-production. Each box represents intermediate data, and each arrow represents one or more functions that define that data. The pipeline includes horizontal and vertical stencils, resampling, data-dependent gathers, and simple pointwise functions.

and computer graphics, where stencils are common, but often in a very different form: stencil pipelines. Stencil pipelines are graphs of different stencil computations. Iteration of the same stencil occurs, but it is the exception, not the rule; most stages apply their stencil only once before passing data to the next stage, which performs different data parallel computation over a different stencil.

Graph-structured programs have been studied in the context of streaming languages [4, 11, 29]. Static communication analysis allows stream compilers to simultaneously optimize for data parallelism and producer-consumer locality by interleaving computation and communication between kernels. However, most stream compilation research has focussed on 1D streams, where sliding window communication allows 1D stencil patterns. Image processing pipelines can be thought of as programs on 2D and 3D streams and stencils. The model of computation required by image processing is also more general than stencils, alone. While most stages are point or stencil operations over the results of prior stages, some stages gather from arbitrary data-dependent addresses, while others scatter to arbitrary addresses to compute operations like histograms.

Pipelines of simple map operations can be optimized by traditional loop fusion: merging multiple successive operations on each point into a single compound operation improves arithmetic intensity by maximizing producer-consumer locality, keeping intermediate data values in fast local memory (caches or registers) as it flows through the pipeline. But traditional loop fusion does not apply to stencil operations, where neighboring points in a consumer stage depend on overlapping regions of a producer stage. Instead, stencils require a complex tradeoff between producer-consumer locality, synchronization, and redundant computation. Because this tradeoff is made by interleaving the order of allocation, execution, and communication of each stage, we call it the pipeline's *schedule*. These tradeoffs exist in scheduling individual iterated stencil computations in scientific applications, and the complexity of the choice space is reflected by the many different tiling and scheduling strategies introduced in past work [10, 16, 19]. In image processing pipelines, this tradeoff must be made for each producer-consumer relationship between stages in the graph—often dozens or hundreds—and the ideal schedule depends on the global interaction among every stage, often requiring the composition of many different strategies.

1.2 Contributions

Halide is an open-source domain-specific language for the complex image processing pipelines found in modern computational photography and vision applications [26]. In this paper, we present the optimizing compiler for this language. We introduce:

- a systematic model of the tradeoffs between locality, parallelism, and redundant recomputation in stencil pipelines;
- a scheduling representation that spans this space of choices;
- a DSL compiler based on this representation that combines Halide programs and schedule descriptions to synthesize points anywhere in this space, using a design where the choices for *how* to execute a program are separated not just from the definition of *what* to compute, but are pulled all the way outside the black box of the compiler;
- a loop synthesizer for data parallel pipelines based on simple interval analysis, which is simpler and less expressive than polyhedral model, but more general in the class of expressions it can analyze;
- a code generator that produces high quality vector code for image processing pipelines, using machinery much simpler than the polyhedral model;
- and an autotuner that can infer high performance schedules—up to 5× faster than hand-optimized programs written by experts—for complex image processing pipelines using stochastic search.

Our scheduling representation composably models a range of tradeoffs between locality, parallelism, and avoiding redundant work. It can naturally express most prior stencil optimizations, as well as hierarchical combinations of them. Unlike prior stencil code generation systems, it does not describe just a single stencil scheduling strategy, but separately treats every producer-consumer edge in a graph of stencil and other image processing computations.

Our split representation, which separates schedules from the underlying algorithm, combined with the inside-out design of our compiler, allows our compiler to automatically search for the best schedule. The space of possible schedules is enormous, with hundreds of inter-dependent dimensions. It is too high dimensional for the polyhedral optimization or exhaustive parameter search employed by existing stencil compilers and autotuners. However, we show that it is possible to discover high quality schedules using stochastic search.

Given a schedule, our compiler automatically synthesizes high quality parallel vector code for x86 and ARM CPUs with SSE/AVX and NEON, and graphs of CUDA kernels interwoven with host management code for hybrid GPU execution. It automatically infers all internal allocations and a complete loop nest using simple but general interval analysis [18]. Directly mapping data parallel dimensions to SIMD execution, including careful treatment of strided access patterns, enables high quality vector code generation, without requiring any general-purpose loop auto-vectorization.

The end result is a system which enables terse, composable programs to achieve state-of-the-art performance on a wide range of real image processing pipelines, and across different hardware architectures, including multicores with SIMD, and heterogeneous CPU+GPU execution. From simple Halide programs written in a few hours, we demonstrate performance up to $5\times$ faster than hand-tuned C, intrinsics, and CUDA implementations written by experts over weeks or months, for image processing applications beyond the reach of past automatic compilers.

2. The Halide DSL

We use the Halide DSL to describe image processing pipelines in a simple functional style [26]. A simple C++ implementation of local Laplacian filters (Fig. 1) is described by dozens of loop nests and hundreds of lines of code. This is not practical to globally optimize with traditional loop optimization systems. The Halide version distills this into 62 lines describing just the essential dataflow and computation in the 99 stage pipeline, and all choices for how the program should be synthesized are described separately (Sec. 3).

In Halide, values that would be mutable arrays in an imperative language are instead functions from coordinates to values. It represents 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. Pipelines are specified as chains of functions. Functions may either be simple expressions in their arguments, or reductions over a bounded domain. The expressions that 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;
- 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, a separable 3×3 unnormalized box filter is expressed as a chain of two functions in x, y:

```
UniformImage in(UInt(8), 2)
Var x, y
Func blurx(x,y) =  in(x-1,y) +    in(x,y) +    in(x+1,y)
Func out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

This representation is simpler than most functional languages. It does not include higher-order functions, dynamic recursion, or additional data structures like lists. Functions simply map from integer coordinates to a scalar result. Constrained versions of more advanced features such as higher-order functions are added as syntactic sugar, but do not change the underlying representation.

This representation is sufficient to describe a wide range of image processing algorithms, and these constraints enable flexible analysis and transformation of algorithms during compilation. Critically, this representation is naturally data parallel within the domain of each function. Also, since functions are defined over an infinite domain, boundary conditions can be handled safely and efficiently by computing arbitrary *guard bands* of extra values as needed. Guard bands are a common pattern in image processing code, both for performance concerns like alignment, and for safety. Wherever specific boundary conditions matter to the meaning of an algorithm, the function may define its own.

Reduction functions. In order to express operations like histograms and general convolutions, Halide also needs a way to express iterative or recursive computations, like summation, histogram, and scan. Reductions are defined in 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. The programmer specifies 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, after recursing in lexicographic order across the reduction domain.

This pattern can describe a range of algorithms outside the scope of traditional stencil computation, but essential to image processig pipelines, in a way that bounds side effects. For example, histogram equalization combines multiple reductions and a data-dependent gather. A scattering reduction computes a histogram, a recursive scan integrates it into a CDF, and a point-wise operation remaps the input using the CDF:

```
UniformImage in(UInt(8), 2)
RDom r(0..in.width(), 0..in.height()), ri(0..255)
Var x, y, i
Func histogram(i) = 0; histogram(in(r.x, r.y))++
Func cdf(i) = 0; cdf(ri) = cdf(ri-1) + histogram(ri)
Func out(x, y) = cdf(in(x, y))
```

The iteration bounds for the reduction and scan are expressed by the programmer using explicit *reduction domains* (**RDom**s).

3. Scheduling Image Processing Pipelines

Halide's representation of image processing algorithms avoids imposing constraints on the order of execution and placement of data. Values need to be computed before they can be used, to respect the fundamental dependencies in the algorithm, but many choices remain unspecified:

- When and where should the value at each coordinate in each function be computed?
- Where should they be stored?
- How long are values cached and communicated across multiple consumers, and when are they independently recomputed by each?

These choices can not change the meaning or results of the algorithm, but they *are* essential to the performance of the resulting implementation. We call a specific set of choices for when and where values are computed the pipeline's *schedule*.

In the presence of stencil access patterns, these choices are bound by a fundamental tension between producer-consumer locality, parallelism, and redundant recomputation of shared values. To understand this tradeoff space, it is useful to look at an example.

3.1 Motivation: Scheduling a Two-Stage Pipeline

Consider the simple two-stage blur algorithm, which computes a 3×3 box filter as two 3×1 passes. The first stage, **blurx**, computes a horizontal blur of the input by averaging over a 3×1 window:

```
blurx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)
```

The second stage, **out**, computes the final isotropic blur by averaging a 1×3 window of the output from the first stage:

```
out(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)
```

A natural way to think about scheduling the pipeline is from the perspective of the output stage: how should it compute its input? There are three obvious choices for this pipeline.

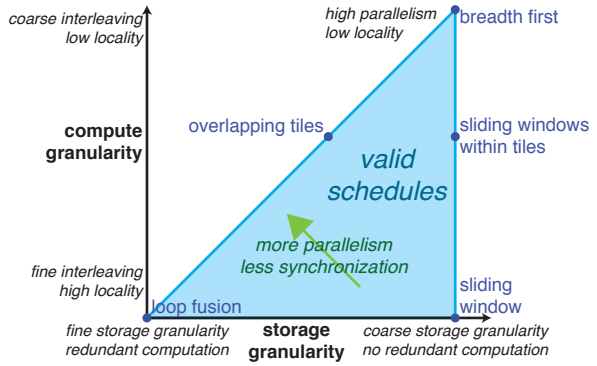

Figure 2. A natural way to visualize the space of scheduling choices is by granularity of storage (x-axis), and granularity of computation (y-axis). Breadth-first execution does coarse-grain computation into coarse-grain storage. Total fusion performs fine-grain computation into fine-grain storage (small temporary buffers). Sliding window strategies allocate enough space for the entire intermediate stage, but compute it in in fine-grain chunks as late as possible. These extremes each have their pitfalls. Breadth-first execution has poor locality, total fusion often does redundant work, and using sliding windows to avoid redundant recomputation constrains parallelism by introducing dependencies across loop iterations. The best strategies tend to be mixed, and lie somewhere in the middle of the space.

First, it could compute and store every required point in **blurx** before evaluating any points in **out**. Applied to a 6 megapixel ($3k \times 2k$) image, this is equivalent to the loop nest:

```
alloc blurx[2048][3072]
for each y in 0..2048:
  for each x in 0..3072:
    blurx[y][x] = in[y][x-1] + in[y][x] + in[y][x+1]
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    out[y][x]=blurx[y-1][x] + blurx[y][x] + blurx[y+1][x]
```

This is the most common strategy in hand-written pipelines, and what results from composing library routines together: each stage executes breadth-first across its input before passing its entire output to the next stage. There is abundant parallelism, since all the required points in each stage can be computed and stored independently, but there is little producer-consumer locality, since all the values of **blurx** must be computed and stored before the first one is used by **out**.

At the other extreme, the **out** stage could compute each point in **blurx** immediately before the point which uses it. This opens up a further choice: should points in **blurx** which are used by multiple points in **out** be stored and reused, or recomputed independently by each consumer?

Interleaving the two stages, without storing the intermediate results across uses, is equivalent to the loop nest:

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[-1..1]
    for each i in -1..1:
      blurx[i]= in[y-1+i][x-1]+in[y-1+i][x]+in[y-1+i][x+1]
    out[y][x] = blurx[0] + blurx[1] + blurx[2]
```

Each pixel can be computed independently, providing the same abundant data parallelism from the breadth-first strategy. The distance from producer to consumer is small, maximizing locality. But because shared values in **blurx** are not reused across iterations, this strategy performs redundant work. This can be seen as the result of applying classical loop fusion through a stencil dependence pattern: the body of the first loop is moved into the second loop, but its work is amplified by the size of the stencil.

Strategy	Span (iterations)	Max. reuse dist. (ops)	Work ampl.
Breadth-first	$\geq 3072 \times 2046$	$3072 \times 2048 \times 3$	$1.0\times$
Full fusion	$\geq 3072 \times 2046$	3×3	$2.0\times$
Sliding window	3072	$3072 \times (3+3)$	$1.0\times$
Tiled	$\geq 3072 \times 2046$	$34 \times 32 \times 3$	$1.0625\times$
Sliding in tiles	$\geq 3072 \times 2048/8$	$3072 \times (3+3)$	$1.25\times$

Figure 3. Different points in the choice space in Figure 2 each make different trade-offs between locality, redundant recomputation, and parallelism. Here we quantify these effects for our two-stage blur pipeline. The *span* measures the degree of parallelism available, by counting how many threads or simd lanes could be kept busy doing useful work. The *Max. reuse distance* measures locality, by counting the maximum number of operations that can occur between computing a value and reading it back. *Work amplification* measures redundant work, by comparing the number of arithmetic operations done to the breadth-first case. Each of the first three strategies represent an extreme point of the choice space, and is weak in one regard. The fastest schedules are mixed strategies, such as the tiled ones in the last two rows.

The two stages can also be interleaved while storing the values of **blurx** across uses:

```
alloc out[2046][3072]
alloc blurx[3][3072]
for each y in -1..2047:
  for each x in 0..3072:
    blurx[(y+1)%3][x]=in[y+1][x-1]+in[y+1][x]+in[y+1][x+1]
    if y < 1: continue
    out[y][x] = blurx[(y-1)%3][x]
              + blurx[ y % 3 ][x]
              + blurx[(y+1)%3][x]
```

This interleaves the computation over a sliding window, with **out** trailing **blurx** by the stencil radius (one scanline). It wastes no work, computing each point in **blurx** exactly once, and the maximum distance between a value being produced in **blurx** and consumed in **out** is proportional to the stencil height (three scanlines), not the entire image. But to achieve this, it has introduced a dependence between the loop iterations: a given iteration of **out** depends on the last three outer loop iterations of **blurx**. This only works if these loops are evaluated sequentially. Interleaving the stages while producing each value only once requires tightly synchronizing the order of computation, sacrificing parallelism.

Each of these strategies has a major pitfall: lost locality, redundant work, or limited parallelism (Fig. 3). In practice, the right choice for a given pipeline is almost always somewhere in between these extremes. For our two-stage example, a better balance can be struck by interleaving the computation of **blurx** and **out** at the level of *tiles*:

```
alloc out[2046][3072]
for each ty in 0..2048/32:
  for each tx in 0..3072/32:
    alloc blurx[-1..33][32]
    for y in -1..33:
      for x in 0..32:
        blurx[y][x] = in[ty*32+y][tx*32+x-1]
                    + in[ty*32+y][tx*32+x]
                    + in[ty*32+y][tx*32+x+1]
    for y in 0..32:
      for x in 0..32:
        out[ty*32+y][tx*32+x] = blurx[y-1][x]
                              + blurx[y  ][x]
                              + blurx[y+1][x]
```

This trades off a small amount of redundant computation on tile boundaries for much greater producer-consumer locality, while still

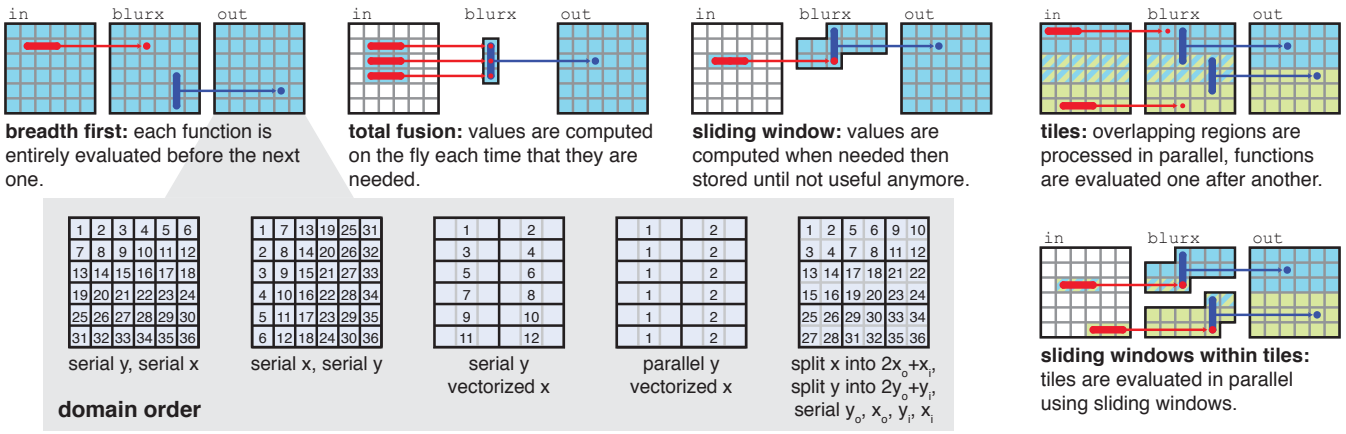

Figure 4. Even simple pipelines exhibit a rich space of scheduling choices, each expressing its own trade-off between parallelism, locality, and redundant recompute. The choice made for each stage is two-fold. The simpler choice is the *domain order*, which can express thread parallelism, vectorization, and traversal order (row-major vs column-major). Dimensions can be split into inner and outer components, which recursively expands the choice space, and can express tiling strategies. The more complex question each stage must answer is when its inputs should be computed. Choices include computing all dependencies ahead of time (breadth-first), computing values as late as possible and then discarding them (total fusion), and computing values as late as possible, but reusing previous computations (sliding window). The two categories of choice interact. For example, the fastest schedules often split the domain into tiles processed in parallel, and then compute either breadth-first or with sliding windows within each tile.

leaving parallelism unconstrained both within and across tiles. (In the iterated stencil computation literature, the redundant regions are often called "ghost zones," and this strategy is sometimes called "overlapped tiling" [17, 31].) On a modern x86, this strategy is 10× faster than the breadth-first strategy using the same amount of multithreaded and vector parallelism. This is because the lack of producer-consumer locality leaves the breadth-first version limited by bandwidth. This difference grows as the pipeline gets longer, increasing the ratio of intermediate data to inputs and outputs, and it will only grow further as the computational resources scale exponentially faster than external memory bandwidth under Moore's Law.

The very fastest strategy we found on this architecture interleaves the computation of the two stages using a sliding window over scanlines, while splitting the image into strips of independent scanlines which are treated separately:

```
alloc out[2046][3072]
for each ty in 0..2048/8:
  alloc blurx[-1..1][3072]
  for y in -2..8:
    for x in 0..3072:
        blurx[(y+1)%3][x] = in[ty*8+y+1][tx*32+x-1]
                          + in[ty*8+y+1][tx*32+x]
                          + in[ty*8+y+1][tx*32+x+1]
    if y < 0: continue
    for x in 0..3072:
      out[ty*8+y][x] = blurx[(y-1)%3][x]
                     + blurx[ y % 3][x]
                     + blurx[(y+1)%3][x]
```

Relative to the original sliding window strategy, this sacrifices two scanlines of redundant work on the overlapping tops and bottoms of independently-processed strips of **blurx** to instead reclaim fine-grained parallelism within each scanline and coarse-grained parallelism across scanline strips. The end result is 10% faster still than the tiled strategy on one benchmark machine, but 10% slower on another. The best choice between these and many other strategies varies across different target architectures. There are also global consequences to the decision made for each stage in a larger pipeline, so the ideal choice depends on the composition of stages, not just each individual stages in isolation. Real image processing pipelines often have tens to hundreds of stages, making the choice space enormous.

3.2 A Model for the Scheduling Choice Space

We introduce a model for the space of important choices in scheduling stencil pipelines, based on each stage choosing at what granularity to *compute* each of its inputs, at what granularity to *store* each for reuse, and within those grains, in what *order* its domain should be traversed (Fig. 4).

The Domain Order Our model first defines the order in which the required region of each function's domain should be traversed, which we call the *domain order*, using a traditional set of loop transformation concepts:

- Each dimension can be traversed *sequentially* or in *parallel*.
- Constant-size dimensions can be *unrolled* or *vectorized*.
- Dimensions can be *reordered* (e.g. from column- to row-major).
- Finally, dimensions can be *split* by a factor, creating two new dimensions: an outer dimension, over the old range divided by the factor, and an inner dimension, which iterates within the factor. After splitting, references to the original index become $outer \times factor + inner$.

Splitting recursively opens up further choices, and enables many common patterns like tiling when combined with other transformations. Vectorization and unrolling are modeled by first splitting a dimension by the vector width or unrolling factor, and then scheduling the new inner dimension as *vectorized* or *unrolled*. Because Halide's model of functions is data parallel by construction, dimensions can be interleaved in any order, and any dimension may be scheduled serial, parallel, or vectorized.

For reduction functions, the dimensions of the reduction domain may only be reordered or parallelized if the reduction update is associative. Free variable dimensions may be scheduled in any order, just as with pure functions.

Our model considers only axis-aligned bounding regions, not general polytopes—a practical simplification for image processing and many other applications. But this also allows the regions to

be defined and analyzed using simple interval analysis. Since our model of scheduling relies on later compiler inference to determine the bounds of evaluation and storage for each function and loop, it is essential that bounds analysis be capable of analyzing *every* expression and construct in the Halide language. Interval analysis is simpler than modern tools like polyhedral analysis, but it can effectively analyze through a wider range of expressions, which is essential for this design.

The Call Schedule In addition to the order of evaluation *within* the domain of each function, the schedule also specifies the granularity of interleaving the computation of a function with the storage and computation of each function on which it depends. We call these choices the *call schedule*. We specify a unique call schedule for each function in the entire call graph of a Halide pipeline. Each function's call schedule is defined as the points in the loop nest of its callers where it is stored and computed (Fig. 4, top). For example, the three extremes from the previous section can be viewed along these axes:

- The *breadth-first* schedule both *stores* and *computes* **blurx** at the coarsest granularity (which we call the *root* level—outside any other loops).
- The *fused* schedule both *stores* and *computes* **blurx** at the finest granularity, inside the innermost (**x**) loop of **out**. At this granularity, values are produced and consumed in the same iteration, but must be reallocated and recomputed on each iteration, independently.
- The *sliding window* schedule *stores* at the *root* granularity, while *computing* at the finest granularity. With this interleaving, values of **blurx** are computed in the same iteration as their first use, but persist across iterations. To exploit this by reusing shared values in subsequent iterations, the loops between the *storage* and *computation* levels must be strictly ordered, so that a single unique first iteration exists for each point, which can compute it for later consumers.

Together, the call schedule and domain order define an algebra for scheduling stencil pipelines on rectangular grids. Composing these choices can define an infinite range of schedules, including the vast majority of common patterns exploited by practitioners in hand-optimized image processing pipelines.

The loop transformations defined by the domain order interact with the inter-stage interleaving granularity chosen by the call schedule because the call schedule is defined by specifying the loop level at which to store or compute. A function call site may be stored or computed at any loop from the innermost dimensions of the directly calling function, to the surrounding dimensions at which it is itself scheduled to be computed, and so on through its chain of consumers. Splitting dimensions allows the call schedule to be specified at finer granularity than the intrinsic dimensionality of the calling functions, for example interleaving by blocks of scanlines instead of individual scanlines, or tiles of pixels instead of individual pixels. Since every value computed needs a logical location into which its result can be stored, *the storage granularity must be equal to, or coarser than, the computation granularity*.

Schedule Examples Revisiting the optimized examples from Sec. 3.1, the tiled schedule can be modeled as follows:

- The domain order of **out** is $split(\mathbf{y}, 32) \rightarrow \mathbf{ty}, \mathbf{y}$; $split(\mathbf{x}, 32) \rightarrow \mathbf{tx}, \mathbf{x}$; $order(\mathbf{ty}, \mathbf{tx}, \mathbf{y}, \mathbf{x})$. (This is similar to the tiled domain order shown rightmost in Fig. 4.) All dimensions may be *parallel*. **x** is *vectorized* for performance.
- The call schedule of **blurx** is set to both *store* and *compute* for each iteration of **tx** in **out**.
- The domain of **blurx** under **tx** is scheduled $order(\mathbf{y}, \mathbf{x})$, and **x** is *vectorized* for performance.

The parallel tiled sliding window schedule is modeled by making the call schedule of **blurx** *store* at **ty** in **out**, but *compute* at finer granularity, at **out**'s **y** dimension. Then:

- The domain order of **out** is $split(\mathbf{y}, 8) \rightarrow \mathbf{ty}, \mathbf{y}$; $order(\mathbf{ty}, \mathbf{y}, \mathbf{x})$. **ty** may be *parallel*, **x** is *vectorized* for performance. **y** must be *sequential*, to enable reuse.
- The domain of **blurx** under **y** only has dimension **x**, which is *vectorized* for performance.

4. Compiling Scheduled Pipelines

Our compiler combines the functions describing a Halide pipeline, with a fully-specified schedule for each function, to synthesize the machine code for a single procedure which implements the entire pipeline. The generated pipeline is exposed as a C ABI callable function which takes buffer pointers for input and output data, as well as scalar parameters. The implementation is multithreaded and vectorized according to the schedule, internally manages the allocation of all intermediate storage, and optionally includes synthesized GPU kernels which it also manages automatically.

The compiler makes no heuristic decisions about which loop transformations to apply or what will generate fast code. For all such questions we defer to the schedule. At the same time, the generated code is safe by construction. The bounds of all loops and allocations are inferred. Bounds inference generates loop bounds that ultimately depend only on the size of the output image. Bounded loops are our only means of control flow, so we can guarantee termination. All allocations are large enough to cover the regions used by the program.

Given the functions defining a Halide pipeline and a fully specified schedule as input (Fig. 5, left), our compiler proceeds through the major steps below.

4.1 Lowering and Loop Synthesis

The first step of our compiler is a lowering process that synthesizes a single, complete set of loop nests and allocations, given a Halide pipeline and a fully-specified schedule (Fig. 5, middle).

Lowering begins from the function defining the output (in this case, **out**). Given the function's domain order from the schedule, it generates a loop nest covering the required region of the output, whose body evaluates the function at a single point in that domain (Fig. 5, middle-top). The order of loops is given by the schedule, and includes additional loops for split dimensions. Loops are defined by their minimum value and their extent, and all loops implicitly stride by 1. This process rounds up the total traversed domain of dimensions which have been split to the nearest multiple of the split factor, since all loops have a single base and extent expression.

At this stage, loop bounds are left as simple symbolic expressions of the *required region* of the output function, which is resolved later. The bounds cannot have inter-dependent dimensions between the loops for a single function, so they represent a dense iteration over an axis-aligned bounding box. Each loop is labeled as being serial, parallel, unrolled, or vectorized, according to the schedule.

Lowering then proceeds recursively up the pipeline, from callers to callees (here, from **out** to **blurx**). Callees (apart from those scheduled *inline*) are scheduled to be computed at the granularity of some dimension of some caller function. This corresponds to an existing loop in the code generated so far. This site is located, and code evaluating the callee is injected at the beginning of that loop body. This code takes the form of a loop nest constructed using the domain order of the callee. The allocation for the callee is similarly injected at some containing loop level specified by the schedule. In Fig. 5, middle, **blurx** is allocated at the level of tiles ($\mathbf{out.x}_0$), while it is computed as required for each scanline within the tile

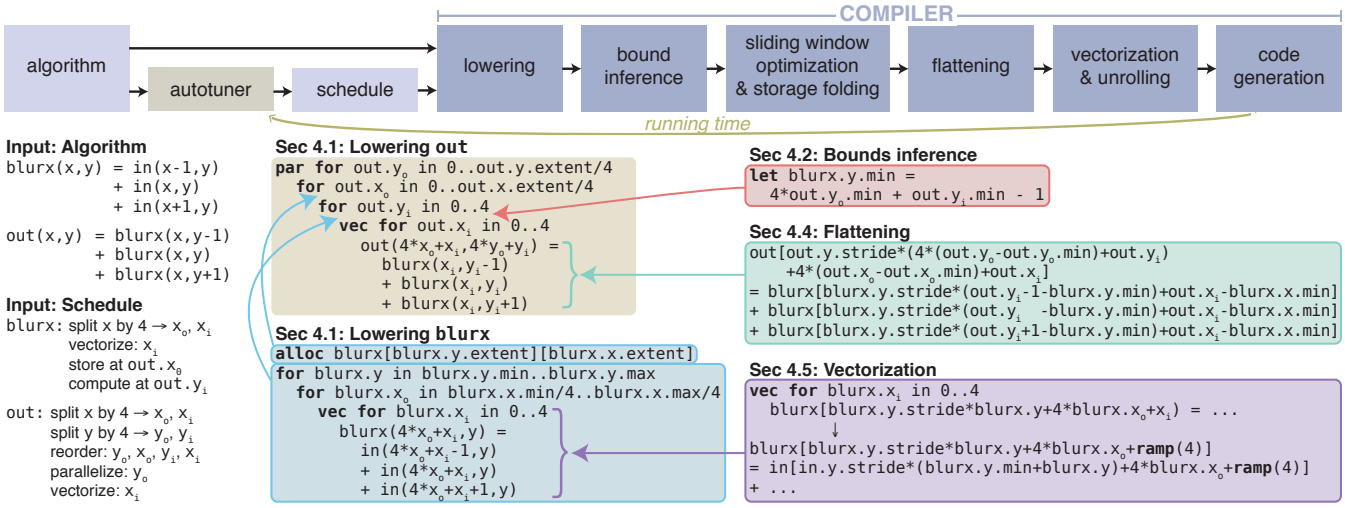

Figure 5. Our compiler is driven by an autotuner, which stochastically searches the space of valid schedules to find a high performance implementation of the given Halide program. The core of the compiler lowers a functional representation of an imaging pipeline to imperative code using a *schedule*. It does this by first constructing a loop nest producing the final stage of the pipeline (in this case **`out`**), and then recursively injecting the storage and computation of earlier stages of the pipeline at the loop levels specified by the schedule. The locations and sizes of regions computed are symbolic at this point. They are resolved by the subsequent bound inference pass, which injects interval arithmetic computations in a preamble at each loop level that set the region produced of each stage to be at least as large as the region consumed by subsequent stages. Next, sliding window optimization and storage folding remove redundant computation and excess storage where the storage granularity is above the compute granularity. A simple flattening transform converts multidimensional coordinates in the infinite domain of each function into simple one-dimensional indices relative to the base of the corresponding buffer. Vectorization and unrolling passes replace loops of constant with k scheduled as *vectorized* or *unrolled* with the corresponding k-wide vector code or k copies of the loop body. Finally, backend code generation emits machine code for the scheduled pipeline via LLVM.

(**`out.y`**$_i$). The allocation and computation for **`blurx`** are inserted at the corresponding points in the loop nest.

Reductions are lowered to a pair of loop nests: the first initializes the domain, and the second applies the reduction rule. Both allocation and loop extents are tracked as symbols of the required region of the function used by its callers. Once lowering has recursed to the end of the pipeline, all functions have been synthesized into a single set of loops.

4.2 Bounds Inference

At this stage, for allocation sizes and loop bounds the pipeline relies on symbolic bounds variables for each dimension of each function. The next stage of lowering generates and injects appropriate definitions for these variables. Like function lowering, bounds inference proceeds recursively back from the output. For each function, it symbolically evaluates the bounds of each dimension based on the bounds required of its caller and the symbolic indices at which the caller invokes it. At each step, the required bounds of each dimension are computed by interval analysis of the expressions in the caller which index that dimension, given the previously computed bounds of all downstream functions.

After bounds inference has recursed to the top of the pipeline, it walks back down to the output, injecting definitions for the bounds variables used as stand-ins during lowering. They are defined by expressions which compute concrete bounds as a preamble at each loop level (e.g., in Fig. 5, right, the minimum bound of **`blurx.y`** is computed from interval analysis of the index expressions at which it is accessed combined with the bounds of the calling function, **`out`**). In practice, hoisting dynamic bounds evaluation expressions to the outermost loop level possible makes the runtime overhead of more complex bounds expressions negligible.

Interval analysis is an unusual choice in a modern loop synthesis and code generation system. The resulting min/max bounds for each dimension are less expressive than the polyhedral model. They can only describe iteration over axis-aligned boxes, rather than arbitrary polytopes. However, it is trivial to synthesize efficient loops for any set of intervals, in contrast to the problem of scanning general polyhedra. For many domains, including image processing, this is an acceptable simplification: most functions *are* applied over rectilinear regions.

Most critically, interval analysis can analyze a more general class of expressions: it is straightforward to compute intervals through nearly any computation, from basic arithmetic, to conditional expressions, to transcendentals, and even loads from memory. As a result, this analysis can be used pervasively to infer the complete bounds of every loop and allocation in any pipeline represented in Halide. It also generalizes through constructs like symbolic tile sizes, which are beyond the scope of polyhedral analysis. For cases where interval analysis is over-conservative (e.g., when computing the bounds of a floating point number loaded from memory which the programmer knows will be between 0 and 1), Halide includes a simple *clamp* operator, which simultaneously declares and enforces a bound on an expression.

4.3 Sliding Window Optimization and Storage Folding

After bounds inference, the compiler traverses the loop nests seeking opportunities for sliding window optimizations. If a realization of a function is stored at higher loop level than its computation, with an intervening serial loop, then iterations of that loop can reuse values generated by previous iterations. Using the same interval analysis machinery as in bounds inference, we shrink the interval to be computed at each iteration by excluding the region computed by all previous iterations. It is this transformation that lets us trade off parallelism (because the intervening loop must be serial) for reuse (because we avoid recomputing values already computed by previous iterations.)

For example, in Fig. 5, **blurx** is stored for reuse within each tile of **out**, but computed as needed, for each scanline within the tile. Because scanlines (**out.y**$_\text{i}$) are traversed sequentially, intermediate values of **blurx** are computed immediately before the first scanline of **out** which needs them, but may be reused my later scanlines within the tile. For each iteration of **out.y**$_\text{i}$, the range of **blurx.y** is computed to exclude the interval covered by all prior iterations computed within the tile.

Storage folding is a second similar optimization employed at this stage of lowering. If a region is allocated outside of a serial loop but only used within it, and the subregion used by each loop iteration marches monotonically across the region allocated, we can "fold" the storage, by rewriting indices used when accessing the region by reducing them modulo the maximum extent of the region used in any given iteration. For example, in Fig. 5, each iteration of **out.y**$_\text{i}$ only needs access to the last 3 scanlines of **blurx**, so the storage of **blurx** can be reduced to just 3 scanlines, and the value **blurx(x,y+3)** will reuse the same memory address as **blurx(x,y)**, **blurx(x,y-3)**, and so on. This reduces peak memory use and working set size.

4.4 Flattening

Next, the compiler flattens multi-dimensional loads, stores, and allocations into their single-dimensional equivalent. This happens in the conventional way: a stride and a minimum offset are computed for each dimension, and the buffer index corresponding to a multidimensional site is the dot product of the site coordinates and the strides, minus the minimum. (Cf. Fig. 5, right.) By convention, we always set the stride of the innermost dimension to 1, to ensure we can perform dense vector loads and stores in that dimension. For images, this lays them out in memory in scanline order. While our model of scheduling allows extreme flexibility in the order of execution, we do not support more unusual layouts memory, such as tiled or sparse storage. (We have found that modern caching memory hierarchies largely obviate the need for tiled storage layouts, in practice.)

4.5 Vectorization and Unrolling

After flattening, vectorization and unrolling passes replace loops of constant size scheduled as vectorized or unrolled with transformed versions of their loop bodies. Unrolling replaces a loop of size n with n sequential statements performing each loop iteration in turn. That is, it completely unrolls the loop. Unrolling by lesser amounts is expressed by first splitting a dimension into two, and then unrolling the inner dimension.

Vectorization completely replaces a loop of size n with a single statement. For example, in Fig. 5 (lower right), the vector loop over **blurx.x**$_\text{i}$ is replaced by a single 4-wide vector expression. Any occurrences of the loop index (**blurx.x**$_\text{i}$) are replaced with a special value **ramp(**n**)** representing the vector $[0\ 1 ... n-1]$. A type coercion pass is then run over this to promote any scalars combined with this special value to n-wide broadcasts of the scalar expression. All of our IR nodes are meaningful for vector types: loads become gathers, stores become scatters, arithmetic becomes vector arithmetic, ternary expressions become vector selects, and so on. Later, during code generation, loads and stores of a linear expression of $k*$**ramp(**n**)**$+o$ will become dense vector loads and stores if the coefficient $k = 1$, or strided loads and stores with stride k otherwise. In contrast to many languages, Halide has no divergent control flow, so this transformation is always well-defined and straight-forward to apply. In our representation, we never split a vector into a bundle of scalars. It is always a single expression containing **ramp**s and broadcast nodes. We have found that this yields extremely efficient code *without* any sort of generalized loop auto-vectorization.

4.6 Back-end Code Generation

Finally, we perform low-level optimizations and emit machine code for the resulting pipeline. Our primary backends use LLVM for low-level code generation. We first run a standard constant-folding and dead-code elimination pass on our IR, which also performs symbolic simplification of common patterns produced by bounds inference. At this point, the representation is ready to be lowered to LLVM IR. There is mostly a one-to-one mapping between our representation and LLVM's, but two specific patterns warrant mention.

First, parallel for loops are lowered to LLVM code that first builds a closure containing state referred to in the body of a for loop. The loop body is lowered to a separate function that accepts the closure as an argument and performs one iteration of the loop. We finally generate code that enqueues the iterations of the loop onto a task queue, which a thread pool consumes at runtime.

Second, many vector patterns are difficult to express or generate poor code if passed directly to LLVM. We use peephole optimization to reroute these to architecture-specific intrinsics. For example, we perform our own analysis pass to determine alignment of vector loads and stores, and we catch common patterns such as interleaving stores, strided loads, vector averages, clamped arithmetic, fixed-point arithmetic, widening or narrowing arithmetic, etc. By mapping specific expression IR patterns to specific SIMD opcodes on each architecture, we provide a means for the programmer to make use of all relevant SIMD operations on ARM (using NEON) and x86 (using SSE and AVX).

GPU Code Generation The data parallel grids defining a Halide pipeline are a natural fit for GPU programming models. Our compiler uses the same scheduling primitives, along with a few simple conventions, to model GPU execution choices. GPU kernel launches are modeled as dimensions (loops) scheduled to be *parallel* and annotated with the GPU *block* and *thread* dimensions to which they correspond.

The limitations of GPU execution place a few constraints on how these dimensions can be scheduled. In particular, a sequence of block and thread loops must be contiguous, with no other intervening loops between the block and thread levels, since a kernel launch corresponds to a single multidimensional, tiled, parallel loop nest. Sets of kernel loops may not be nested within each other on current GPUs which do not directly implement nested data parallelism. Additionally, the extent of the thread loops must fit within the corresponding limits of the target device. Other than that, all the standard looping constructs may still be scheduled outside or within the block and grid dimensions. This corresponds to loops which internally launch GPU kernels, and loops within each thread of a GPU kernel, respectively.

Given a schedule annotated with GPU block and thread dimensions, our compiler proceeds exactly as before, synthesizing a single set of loop nests for the entire pipeline. No stage before the backend is aware of GPU execution; block and thread dimensions are treated like any other loops. The GPU backend extends the x86 backend, including its full feature set. Outside the loops over block and thread dimensions, the compiler generates the same optimized SSE code as it would in the pure CPU target. At the start of each GPU block loop nest, we carve off the sub-nest much like a parallel for loop in the CPU backend, only it is spawned on the GPU. We first build a closure over all state which flows into the GPU loops. We then generate a GPU kernel from the body of those loops. And finally, we generate the host API calls to launch that kernel at the corresponding point in the host code, passing the closure as an argument. We also generate dynamic code before and after launches to track which buffers need to be copied to or from the device. Every allocated buffer which is used on the GPU has a corresponding device memory allocation, and their contents are lazily copied only when needed.

The end result is not individual GPU kernels, but large graphs of hybrid CPU/GPU execution, described by the same scheduling model which drives the CPU backends. A small change in the schedule can transform a graph of dozens of GPU kernels and vectorized CPU loop nests, tied together by complex memory management and synchronization, into an *entirely different* graph of kernels and loops which produce the same result, expressively modeling an enormous space of possible fusion and other choices in mapping a given pipeline to a heterogeneous machine.

5. Autotuning Pipeline Schedules

We apply stochastic search to automatically find good schedules for Halide pipelines. The automatic optimizer takes a fixed algorithm and attempts to optimize the running time by searching for the most efficient schedule. The schedule search space is enormous—far too large to search exhaustively. For example in the local Laplacian filters pipeline, we estimate a lower bound of 10^{720} schedules. This is derived by labeling functions with three tilings per function and all possible store and compute granularities. The actual dimensionality of the space is likely much higher. The optimal schedule dependends on machine architecture, image dimensions, and code generation in complex ways, and exhibits global dependencies between choices due to loop fusion and caching behavior.

In this section, we describe our autotuner. Because the search problem has many local minima, we use a genetic algorithm to seek out a plausible approximate solution, inspired by the search procedure in PetaBricks [2]. We first describe the schedule search space. We show how domain-specific knowledge can be used to select reasonable starting points. Then we explain the general operations of the genetic algorithm. Finally, we show how further knowledge can be incorporated as search priors for more effective mutation rules.

Schedule Search Space Our full model of scheduling is per call but to simplify autotuning we schedule each function identically across all call sites. The domain transformations include splitting and reordering dimensions, and marking them parallel, vectorized, or unrolled, or mapping a pair of dimensions to a GPU grid launch. Variable and block size arguments are randomized and chosen from small powers of two.

Because schedules have complex global dependencies, not all schedules are valid: for example, a schedule could be computed or stored inside a dimension that does not exist in the caller's loop order. Genetic operations such as mutate and crossover may invalidate correct parent schedules. In general therefore we reject any partially completed schedules that are invalid, and continue sampling until we obtain valid schedules. We also verify the program output against a correct reference schedule, over several input images. This is just a sanity check: all valid schedules should generate correct code. Finally to prevent explosion of generated code due to complex pipelines, we limit the number of domain scheduling operations for each function.

Search Starting Point One valid starting schedule is to label all functions as computed and stored breadth first (at the outermost, *root* granularity). The tuner converges from this starting point, albeit slowly. We can often do better by seeding the initial population with reasonable schedules. For each function we find its rectangular footprint relative to the caller (via bounds inference) and inline functions with footprint one. Remaining functions are stochastically scheduled as either (1) *fully parallelized and tiled* or (2) simply parallelized over y. We define *fully parallelized and tiled* as tiled over x and y, vectorized within the tile's inner x coordinate, and parallelized over the y outer tile dimension. These choices are selected by a weighted coin that has fixed weight from zero to one depending on the individual. This allows us to often discover good starting points for functions that vectorize well, or fall back to naive parallelism when that is not the case. The dimensions x and y are chosen from adjacent dimensions at random, except when there are optional bounds annotations provided by the Halide programmer (such as the number of color channels): dimensions with small bound are not tiled.

Genetic Algorithm Search We use a fixed population size (128 individuals per generation, for all examples in this paper) and construct each new generation with population frequencies of elitism, crossover, mutated individuals, and random individuals. Elitism copies the top individuals from the previous generation. Crossover selects parents by tournament selection, followed by two-point crossover, with crossover points selected at random between functions. Random individuals are generated either by the reasonable schedules described previously, or with equal probability, by scheduling each function independently with random schedule choices. This is directly derived from the PetaBricks autotuner [2].

Schedule Mutation Rules We incorporate further prior knowledge about schedules into our mutation rules. Mutation selects a function at random and modifies its schedule with one of eight operations selected at random. Six are fairly generic: randomize constants, replace a function's schedule with one randomly generated, copy from a randomly selected function's schedule, and for the function's list of domain transformations, either add, remove, or replace with a randomly selected transformation.

Our remaining two mutations incorporate specific knowledge about imaging. These are chosen with higher probability. First, a pointwise mutation of compute or storage granularity is generally ineffective at fusing loops. Thus we have a loop fusion rule which schedules the chosen function as *fully parallelized and tiled*, followed by scheduling callees as vectorized by x, and computed and stored under the tile's inner x dimension. The callee scheduling is repeated recursively until a coin flip fails. Finally, we incorporate prior knowledge by applying a template: we replace a function's schedule with one of three common schedule patterns sampled from a text file. These are: (1) compute and store under x, and vectorize by x, (2) *fully parallelized and tiled*, and (3) parallelized over y and vectorized over x. If generating a CUDA schedule we inject a fourth pattern which simply tiles on the GPU. The x and y dimensions are determined as in the starting point.

6. Results

To evaluate our representation and compiler, we applied them to a range of image processing applications, and compared the best autotuned result found by our compiler to the best previously published expert implementation we could find. We selected this set of examples to cover a diversity of algorithms and communication patterns. It includes pipelines ranging from two to 99 stages, and including many different stencils, data-dependent access patterns, and histogram reductions. We describe each application below. Fig. 6 summarizes their properties, and Fig. 7 summarizes their performance. All evaluation was performed on a quad core Xeon W3520 x86 CPU, an NVIDIA Tesla C2070 GPU.

Blur is the simple example used in Sec. 3.1, which convolves the image with two 3×1 box kernels in two steps, a horizontal 3×1 kernel then a vertical 1×3 kernel. This is a simple example of two consecutive stencils. Our reference comparison is a hand-optimized, manually fused and multithreaded loop nest defined entirely in SSE intrinsics [26]. This version is $12\times$ faster than a simple pair of loops in C compiled by GCC 4.7. The version found by our autotuner is 10% faster, still, while being generated from a two line Halide algorithm rather than 35 lines of intrinsics.

	# functions	# stencils	graph structure
Blur	2	2	simple
Bilateral grid	7	3	moderate
Camera pipeline	32	22	complex
Local Laplacian filters	99	85	very complex
Multi-scale interpolation	49	47	complex

Figure 6. Properties of the example applications. In some cases, the number of functions exceeds the number of program lines in Fig. 7, because Halide functions are meta-programmed using higher-order functions.

Camera pipeline transforms the raw data recorded by the camera sensor into a usable image. Its demosaicking, alone, is a complex combination of 21 interleaved and inter-dependent stencils.

The reference comparison is a single carefully tiled and fused loop nest from the Frankencamera, expressed in 306 lines of C++ [1]. All producer-consumer communication is staged through scratch buffers, tiles are distributed over parallel threads using OpenMP, and the tight loops are autovectorized by GCC. The Halide algorithm is 145 lines describing 32 functions and 22 different stencils, literally translated from the pseudocode in the comments explaining the original source. It compiles to an implementation 3.4× faster than the hand-tuned original. The autotuned schedule fuses long chains of stages through complex, interleaved stencils on overlapping tiles, fully fuses other stages, vectorizes every stage, and distributes blocks of scanlines across threads.

Multi-scale interpolation uses an image pyramid to interpolate pixel data for seamless compositing. This requires dealing with data at many different resolutions. The resulting pyramids are chains of stages which locally resample over small stencils, but through which dependence propagates globally across the entire image.

The reference implementation is a carefully-structured set of loop nests which were hand-tuned by an Adobe engineer to generate a fully vectorized implementation in GCC. The Halide algorithm is substantially simpler, but compiles to an implementation 1.7× faster than the original parallel vector code. The same Halide algorithm also automatically compiles to a graph of CUDA kernels and x86 code which is 5.9× faster.

Bilateral grid is an efficient implementation of the bilateral filter, which smoothes an image while preserving its main edges [5, 21]. It first scatters the image data into a 3D grid, effectively building a windowed histogram in each column of the grid, then blurs the grid along each of is axes with three 5-point stencils. Finally, the output image is constructed by trilinear interpolation within the grid at locations determined by the input image.

The CPU reference code is a tuned but clean implementation from the original authors in 122 lines of C++. It is partially autovectorized by GCC, but is nontrivial to multithread (a naive OpenMP parallelization of major stages results in a slowdown on our benchmark CPU), so the reference is single-threaded. The Halide algorithm is 34 lines, and compiles to an implementation 4.2× faster than the original. The speedup comes from a combination of parallelism, tile-level fusion of some stages, and careful reordering of dimensions to control parallel grain size in the grid.

We also compared our implementation to a hand-tuned GPU implementation from the original authors, written in 370 lines of CUDA code. The same Halide algorithm—less than 1/10th the code—found a different schedule which was 2.3× faster than the hand-written CUDA. The Halide compiler generates similar CUDA code to the reference, but the autotuner found an unintuitive point in the schedule space which sacrificed some parallelism in the grid construction step to reduce synchronization overhead in the scattering reduction. It also uses a tiled fusion strategy which passes intermediate results through GPU scratchpad memory to improve locality through the blur steps at the expense of redundant computation. These tradeoffs were counter-intuitive to the original author, and also much harder to express in CUDA, but are easily described by our schedule representation and found by our autotuner.

x86

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Blur	11	13	1.2×	2	35	18×
Bilateral grid	36	158	4.4×	34	122	4×
Camera pipe	14	49	3.4×	123	306	2×
Interpolate	32	54	1.7×	21	152	7×
Local Laplacian	113	189	1.7×	52	262	5×

CUDA

	Halide tuned (ms)	Expert tuned (ms)	Speedup	Lines Halide	Lines expert	Factor shorter
Bilateral grid	8.1	18	2.3×	34	370	11×
Interpolate	9.1	*54**	*5.9*×	21	*152**	7×
Local Laplacian	21	*189**	*9*×	52	*262**	5×

Figure 7. Comparison of autotuned Halide program running times to hand-optimized programs created by domain experts in C, intrinsics, and CUDA. Halide programs are both faster and require fewer code lines. *(*No GPU reference available, compared to CPU reference.)*

Local Laplacian filters uses a multi-scale approach to tone map images and enhance local contrast in an edge-respecting fashion [3, 22]. It is used in the clarity, tone mapping, and other filters in Adobe Photoshop and Lightroom. The algorithm builds and manipulates several image pyramids with complex dependencies between them. The filter output is produced by a data-dependent resampling from several pyramids. With the parameters we used, the pipeline contains 99 different stages, operating at many scales, and with different computational patterns.

The reference implementation is 262 lines of C++, developed at Adobe, and carefully parallelized with OpenMP, and offloading most intensive kernels to tuned assembly routines from Intel Performance Primitives [14, 20]. It has very similar performance to a version deployed in their products, which took several months to develop, including 2-3 weeks dedicated to optimization. It is 10× faster than an algorithmically identical reference version written by the authors in pure C++, without IPP or OpenMP. The Halide version was written in one day, in 52 lines of code. It compiles to an implementation which is 1.7× faster than the highly optimized expert implementation (roughly 20× faster than the clean C++ without IPP and OpenMP). The resulting schedule is enormously complex, mixing different fusion, tiling, vectorization, and multithreading strategies throughout the expansive 99 stage graph. In C, it would correspond to hundreds of loops over more than 10,000 lines of code.

The same program compiles with a different automatically generated schedule to a hybrid CPU/GPU program with 58 unique GPU kernels, each representing a differently tiled and partially fused subset of the overall graph, and with the lookup table and several smaller levels of the pyramids scheduled as vector code on the CPU. *The generated program has more distinct CUDA kernels than the Halide algorithm describing it has lines of code*. It runs 7.5× faster than the hand-tuned Adobe implementation, and is more than 4× faster than the best parallel vector implementation on the CPU. This

	Source size (MP)	Target size (MP)	Cross-tested time (ms)	Autotuned on target (ms)	Slowdown (vs target autotuned)
Blur	0.3	30	13	11	1.2×
Bilateral grid	0.3	2	35	36	0.97×
Interpolate	0.3	2	31	32	0.97×
Blur	30	0.3	1.1	0.07	16×
Bilateral grid	2	0.3	9.6	6.7	1.4×
Interpolate	2	0.3	9.7	5.2	1.9×

Figure 8. Cross-testing of autotuned schedules across resolutions. Each program is autotuned on a source image size. The resulting schedule is tested on a target image size giving a "cross-tested time." This is compared to the result of running the autotuner directly on the target resolution. We report the ratio of the cross-tested time to the autotuned-on-target time as the "slowdown." Note that schedules generalize better from low resolutions to high resolutions. In theory the slow-down should always be at least one, but due to the stochastic nature of the search some schedules were slower when autotuned on the target.

is by far the fastest implementation of local Laplacian filters we know of.

6.1 Autotuning Performance

These examples took between 2 hours and 2 days to tune (from 10s to 100s of generations). In all cases, the tuner converged to within 15% of the final performance after less than one day tuning on a single machine. Improvements to the compiling and tuning infrastructure (for example, distributing tests across a cluster) could reduce these times significantly.

We generally found the tuned schedules to be insensitive to moderate changes in resolution or architecture, but extreme changes can cause the best schedule to change dramatically. Table 8 shows experiments in cross-testing schedules tuned at different resolutions. We observe that schedules generalize better from low resolutions to high resolutions. We also mapped the best GPU schedule for local Laplacian filter to the CPU, and found that this is 7× slower than the best CPU schedule.

6.2 Discussion

Across a range of image processing applications and target architectures, our scheduling representation is able to model, our compiler is able to generate, and our autotuner is able to discover implementation strategies which deliver state-of-the-art performance. This performance comes from careful navigation of the extremely high dimensional space of tradeoffs between locality, parallelism, and redundant recomputation in image processing pipelines. Making these tradeoffs by hand is challenging enough, as shown by the much greater complexity of hand-written implementations, but finding the ideal points is daunting when each change a programmer might want to test can require completely rewriting a complex loop nest hundreds of lines long. The performance advantage of the Halide implementations is a direct result of simply *testing many more points in the space* than a human programmer ever could manually describe at the level of explicit loops.

Nonetheless, based on our experience, brute force autotuning still poses significant challenges of robustness and usability in a real system. The tuning process is an entire extra stage added to a programmer's normal development process. Straightforward implementations are sensitive to noise in testing environment and many other factors. In Halide, while simple pipelines (like blur) can tune effectively with only trivial mutation rules, we found heuristic mutation rules are essential to converge in a reasonable amount of time when tuning more complex imaging pipelines. However these rules may be specific to the algorithm structure. For example, different template mutation rules may be necessary for 3D voxel data or unconventional color layouts. We also found that it is necessary to maintain diversity to avoid becoming trapped in local minima, which we did by using a large population (128 individuals per generation). Even so, the tuner can occasionally become trapped, requiring restarting with a new random initialization, and taking the best of several runs. In general, we feel that high dimensional stochastic autotuning has not yet developed the robust methodology or infrastructure for real-world use found elsewhere in the compiler community.

7. Prior Work

Image processing pipelines include similar structure to several problems well studied in compilers.

Split Compilers Sequoia's "mappings" and SPIRAL's loop synthesis algebra echo our separation of the model of *scheduling* from the description of the algorithm, and its lifting outside our compiler [9, 25].

Stream Programs Compiler optimization of stream programs was studied extensively in the StreamIt and Brook projects [4, 11, 29]. In this framework, sliding window communication implements stencils on 1D streams [12, 24, 30]. Stream compilers generally did not consider the introduction of redundant work as a major optimization choice, and nearly all work in stream compilation has focussed on these 1D streams. Image processing pipelines, in contrast, are effectively stream programs over 2D-4D streams.

Stencil Optimization Iterated stencil computations are important to many scientific applications, and have been studied for decades. Frigo and Strumpen proposed a cache oblivious traversal for efficient stencil computation [10]. This view of locality optimization by interleaving stencil applications in space and time inspired our model of scheduling. The Pochoir compiler automatically transforms stencil codes from serial form into a parallel cache oblivious form using similar algorithms [28].

Overlapping tiling is a strategy which divides a stencil computation into tiles, and trades off redundant computation along tile boundaries to improve locality and parallelism [16], modeled in our schedule representation as interleaving both storage and computation inside the tile loops. Other tiling strategies represent different points in the tradeoff space modeled by our representation [19]. Past compilers have automatically synthesized parallel loop nests with overlapped tiling on CPUs and GPUs using the polyhedral model [13, 16]. These compilers focussed on synthesizing high quality code given a single, user-defined set of overlapped tiling parameters. Autotuning has also been applied to iterated stencil computations, but past tuning work has focussed on exhaustive search of small parameter spaces for one or a few strategies [15].

Image Processing Languages Critically, many optimizations for iterated stencil computations are based on the assumption that the time dimension of iteration is large relative to the spatial dimension of the grid. In image processing pipelines, most individual stencils are applied only once, while images are millions of pixels in size. Image processing pipelines also include more types of computation than stencils alone, and scheduling them requires choices not only of different parameters, but of entirely different strategies, for each of many heterogeneous stages, which is infeasible with either exhaustive search or polyhedral optimization. Most prior image processing languages and systems have focused on efficient expression of individual kernels, as well as simple fusion in the absence of stencils [6, 8, 23, 27]. Recently, Cornwall et al. demonstrated fast

GPU code generation for image processing code using polyhedral optimization [7].

Earlier work on the Halide language included a weaker model of schedules, and required programmers to explicitly specify schedules by hand [26]. This is the first automatic optimizer, and therefore the first fully automatic compiler, for Halide programs. We show how they can be automatically inferred, starting from just the algorithm defining the pipeline stages, and using relatively little domain-specific knowledge beyond the ability to enumerate points in the space of schedules. Our state-of-the-art performance shows the effectiveness of our scheduling model for representing the underlying choice space. The scheduling model presented here is also richer than in past work. In particular, it separates computation frequency from storage frequency in the call schedule, enabling sliding window schedules and similar strategies which trade off parallelism for redundant work while maintaining locality.

In all, this gives a dramatic new result: automatic optimization of stencil computations, including full consideration of the parallelism, locality, and redundancy tradeoffs. Past automatic stencil optimizations have targeted individual points in the space, but have not automatically chosen *among* different strategies spanning these multi-dimensional tradeoffs, and none have automatically optimized large heterogeneous pipelines, only individual or small multi-stencils.

Acknowledgments

Eric Chan provided feedback and inspiration throughout the design of Halide, and helped compare our local Laplacian filters implementation to his in Camera Raw. Jason Ansel helped inform the design of our autotuner. This work was supported by DOE Award DE-SC0005288, NSF grant 0964004, grants from Intel and Quanta, and gifts from Cognex and Adobe.

References

[1] A. Adams, E. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy. The Frankencamera: An experimental platform for computational photography. *ACM Trans. Graph.*, 29(4), 2010.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *ACM Programming Language Design and Implementation*, 2009.

[3] M. Aubry, S. Paris, S. W. Hasinoff, J. Kautz, and F. Durand. Fast and robust pyramid-based image processing. Technical Report MIT-CSAIL-TR-2011-049, Massachusetts Institute of Technology, 2011.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[5] J. Chen, S. Paris, and F. Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph.*, 26(3), 2007.

[6] CoreImage. Apple CoreImage programming guide, 2006.

[7] J. L. T. Cornwall, L. Howes, P. H. J. Kelly, P. Parsonage, and B. Nicoletti. High-performance SIMT code generation in an active visual effects library. In *Conf. on Computing Frontiers*, 2009.

[8] C. Elliott. Functional image synthesis. In *Proceedings of Bridges*, 2001.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *ACM/IEEE conference on Supercomputing*, 2006.

[10] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, 2005.

[11] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[12] P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal – A data flow-oriented language for signal processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, 1986.

[13] J. Holewinski, L. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, 2012.

[14] IPP. Intel Integrated Performance Primitives. `http://software.intel.com/en-us/articles/intel-ipp/`.

[15] S. Kamil, C. Chan, L. Oliker, J. Shalf, , and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, 2010.

[16] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, 2007.

[17] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. In *IJPP*, 2011.

[18] R. Moore. *Interval Analysis*. 1966.

[19] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Supercomputing*, 2010.

[20] OpenMP. OpenMP. `http://openmp.org/`.

[21] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand. Bilateral filtering: Theory and applications. Foundations and Trends in Computer Graphics and Vision, 2009.

[22] S. Paris, S. W. Hasinoff, and J. Kautz. Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid. *ACM Trans. Graph.*, 30(4), 2011.

[23] PixelBender. Adobe PixelBender reference, 2010.

[24] H. Printz. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*. Ph.D. Thesis, Carnegie Mellon University, 1991.

[25] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE*, volume 93, 2005.

[26] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4), 2012.

[27] M. A. Shantzis. A model for efficient and flexible image computing. In *ACM SIGGRAPH*, 1994.

[28] Y. Tang, R. Chowdhury, B. Kuszmaul, C.-K. Luk, and C. Leiserson. The Pochoir stencil compiler. In *SPAA*, 2011.

[29] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, 2002.

[30] P.-S. Tseng. *A Parallelizing Compiler for Disributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, 1989.

[31] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling.