DOI:10.1145/3150211

Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing

By Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand

Abstract

Writing high-performance code on modern machines requires not just locally optimizing inner loops, but globally reorganizing computations to exploit parallelism and locality—doing things such as tiling and blocking whole pipelines to fit in cache. This is especially true for image processing pipelines, where individual stages do much too little work to amortize the cost of loading and storing results to and from off-chip memory. As a result, the performance difference between a naïve implementation of a pipeline and one globally optimized for parallelism and locality is often an order of magnitude. However, using existing programming tools, writing high-performance image processing code requires sacrificing simplicity, portability, and modularity. We argue that this is because traditional programming models conflate the computations defining the *algorithm* with decisions about intermediate storage and the order of computation, which we call the *schedule*.

We propose a new programming language for image processing pipelines, called Halide, that separates the algorithm from its schedule. Programmers can change the schedule to express many possible organizations of a single algorithm. The Halide compiler then synthesizes a globally combined loop nest for an entire algorithm, given a schedule. Halide models a space of schedules which is expressive enough to describe organizations that match or outperform state-of-the-art hand-written implementations of many computational photography and computer vision algorithms. Its model is simple enough to do so often in only a few lines of code, and small changes generate efficient implementations for x86, ARM, Graphics Processors (GPUs), and specialized image processors, all from a single algorithm.

Halide has been public and open source for over four years, during which it has been used by hundreds of programmers to deploy code to tens of thousands of servers and hundreds of millions of phones, processing billions of images every day.

1. INTRODUCTION

Computational photography and computer vision algorithms require highly efficient implementations to be used in practice, from power-constrained mobile devices to data centers processing billions of images. This is not a simple matter of programming in a low-level language such as C: even in C, the performance difference between naïve and highly optimized image processing code for the same algorithm is often an order of magnitude. Unfortunately, optimization usually comes at a large cost in programmer time and code complexity, as computation must be globally reorganized to efficiently exploit the memory system (locality, e.g., in caches) and many execution units (parallelism, e.g., across threads and vector lanes).

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 performance and efficiency therefore come not just from optimizing the inner loops, but also from global program transformations that exploit producer-consumer locality down the pipeline. For example, computing a first stage on the entire image before processing the second stage causes cache misses when storing and loading the intermediate results; instead, an optimized pipeline might transform the organization of computation with tiling and fusion to compute both stages at the granularity of smaller image tiles that fit in cache.

Image processing exhibits a rich space of possible organizations of computation. The best choice of organization is architecture-specific. Implementations optimized for an x86 multicore and for a modern GPU often bear little resemblance to each other. There is also a tension between parallelism, locality, and storing versus recomputing intermediate values, which can make the ideal organization subtle and unpredictable.

Halide enables simpler programming of high-performance code by separating the intrinsic algorithm of an image processing pipeline from the decisions about how to run efficiently on a particular machine. Programmers may still specify the strategy for execution, since automatic optimization remains hard, but doing so is radically simplified by this split representation, which allows them to concisely express many optimization strategies without obfuscating

The original version of this work appeared in two papers, entitled "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines" published in *ACM Transactions on Graphics 31* 4, (July 2012), ACM, and "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines" published in the *Proceedings of PLDI*, June 2013, ACM.

the code or accidentally modifying the algorithm itself.

This separation of concerns is important, and the ideal code organization nontrivial, even for a problem as simple as a 3×3 box filter implemented as separate horizontal and vertical passes (see Figure 1). We might write this in C++ as a sequence of two loop nests (see Figure 1.a). An efficient implementation on a modern CPU requires Single Instruction Multiple Data (SIMD) vectorization and multithreading. Once we start to exploit parallelism, however, 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, for example by tiling and fusing the loops. Tiles must be carefully sized for alignment, and efficient fusion requires subtleties such as redundantly computing values on the overlapping boundaries of intermediate tiles. The resulting implementation is 22 times faster on a quad-core CPU, but together these optimizations have fused two simple, independent steps into a single intertwined, architecture-specific mess (see Figure 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*.

Functional programming provides a natural basis for separating the *what* from the *when* and *where*. Divorced from explicit storage, images are no longer arrays filled 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 (see Figure 1.c). This functional representation also omits boundaries, and the order and extent of iteration, 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:

1. the order in which points in a function are evaluated, including tiling, the exploitation of parallelism, and mapping onto SIMD execution units;
2. the granularity with which the evaluation of points in one function are interleaved with evaluating points in the functions which call it;
3. the memory locations into which the values of a function are stored, including registers, scratchpad memories, and regions of main memory;
4. whether a value is recomputed, or from where it is loaded, at each place a function is used.

The key challenge in doing this is defining a representation of schedules which is both simple and expressive. Halide's model (Section 3) decomposes the organization of a pipeline into four major choices for each function, corresponding to the points above, each described as a composition of simple primitives.

Figure 1. The C++ code at the top (a) computes a 3×3 box filter using the composition of a 1×3 and a 3×1 box filter. Using vectorization, multithreading, tiling, and fusion, we can make this algorithm more than 20 times faster on a quad-core x86 CPU (b) However, in doing so we have lost simplicity and portability. Halide (c) separates the algorithm description from its schedule, describing the same optimizations, generating very similar machine code, and achieving the same performance without making these sacrifices. (Benchmarked on an Intel Core i7–4790, from the BLUR app in the Halide repository.[14])

(a) Clean C++: 6.5ms per megapixel

```
void blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 Image<uint16_t> bh(in.width(), in.height();

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

(b) Fast C++ (for x86) : 0.30ms per megapixel

```
void fast_blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 __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 bh[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *bhPtr = bh;
   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_loadu_si128((__m128i*)(inPtr - 1));
       b = _mm_loadu_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(bhPtr++, avg);
     inPtr += 8;
   }}
   bhPtr = bh;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(bv(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_load_si128(bhPtr + (256 * 2) / 8);
       b = _mm_load_si128(bhPtr + 256 / 8);
       c = _mm_load_si128(bhPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

(c) Halide : 0.29ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

Halide can most flexibly schedule operations which are data parallel with statically analyzable access patterns (such as stencils), but also supports the bounded iterative algorithms and irregular access patterns that occur in image processing and general array computation. It imposes a few restrictions on the range of expressible schedules, but is sufficient to concisely express implementations of many image processing algorithms, with state-of-the-art performance on architectures ranging from mobile and server CPUs, to GPUs, to specialized image processors.

Once the programmer has specified an algorithm and a schedule, the Halide compiler combines them into an efficient implementation. Optimizing the execution strategy 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. Figure 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, and we have also shown that good schedules can often be found automatically.[17, 23]

In the rest of this article we will briefly introduce Halide's language for algorithms (Section 2), discuss its model of schedules and the organizational choices they represent (Section 3), touch on the design of the Halide compiler (Section 4), demonstrate results on several real computational photography algorithms, (Section 5), and conclude by discussing connections with the wealth of related work (Section 6), and our perspective after five years of development and widespread use (Section 7).

2. THE HALIDE ALGORITHM LANGUAGE

Halide describes image processing pipelines in a simple functional style. A straightforward C++ implementation of an algorithm such as local Laplacian filters is described by dozens of loop nests and hundreds of lines of code.[2] This is not practical to globally optimize with traditional loop optimization systems.[10] 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 in a separate *schedule* (Section 3).

Halide represents images as pure functions defined over an infinite integer domain, where the value of a function at a point represents the value of the image at the corresponding coordinate. Halide functions can have arbitrary dimensionality (not just two), and may be tuple-valued (they can store a "`struct`" of values at each point, not just a single number). Pipelines are specified as a directed acyclic graph of functions. The expressions that define functions are side-effect free, and are much like those in any simple functional language, including arithmetic and logical operations, if-then-else expressions, loads from memory buffers, and 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:

```
Func bh, bv; Var x, y;
ImageParam in(UInt(8), 2);

bh(x, y) = in(x-1, y) + in(x, y) + in(x+1, y);
bv(x, y) = bh(x, y-1) + bh(x, y) + bh(x, y+1);
```

This representation is simpler than most functional languages. It does not include higher-order functions, dynamic recursion, or additional data structures such as lists. Functions simply map from integer coordinates to a scalar or tuple-valued result.

Halide is embedded in C++. It uses simple type and operator overloading (not template metaprogramming) to lazily construct programs, rather than eagerly executing expressions as they are written. This "staged" nature makes the Halide front-end easily extensible. Many advanced constructs are expressible by using C++ as a meta-programming layer for Halide. For example, you can simulate higher-order functions by writing a C++ function that takes and returns Halide functions. This provides a powerful tool for structuring code and it does not change the underlying representation of a pipeline.

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 in two ways. For intermediate pipeline stages, an implementation can compute arbitrary *guard bands* of extra values. For input images, or stages for which specific boundary conditions matter to the meaning of an algorithm, the function may define its own. The compiler will partition the resulting loops so that the boundary conditions have minimal impact on performance.

Update definitions

Functions are typically defined by simple expressions in their arguments, but may additionally have a sequence of bounded *updates* to accommodate reductions (e.g., large-support convolution), scatters (e.g., histograms), and recursive scans (e.g., Infinite Impulse Response (IIR) filters). Sequential iteration within an update can be introduced with an `RDom` ("reduction domain," a multidimensional iteration domain). The value of the function at each point in the output domain is defined by the final value after all updates are applied. The key constraint relative to arbitrary loops is that the bound of an `RDom` cannot depend on the values computed inside its updates. This guarantees that all iteration bounds are decidable (and the language is therefore not Turing complete).

This pattern can describe a range of algorithms outside the scope of traditional stencil computation but essential to image processing pipelines, in a way that encapsulates side effects. To the rest of the pipeline, a function with updates still acts as stateless pure function that can be evaluated over an arbitrary domain. 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 Cumulative Distribution Function (CDF), and a simple point-wise operation remaps the input using the CDF:

```
Func histogram, cdf, out; Var x, y, i;
ImageParam in(UInt(8), 2);


RDom r(0, in.width(),
       0, in.height())'
histogram(i) = 0; // initial value
histogram(in(r.x, r.y)) += 1; // update
```

```
RDom ri(0, 255);
cdf(i) = 0; // initial value
cdf(ri) = cdf(ri-1) + histogram(ri); // update

out(x, y) = cdf(in(x, y));
```

3. SCHEDULING IMAGE PROCESSING PIPELINES

A complete Halide algorithm is a DAG of functions over regular grids. Actually evaluating an algorithm requires iterating over and computing all of the required points in each function. But in what order should we compute these points? And where should we store and load the results of intermediate stages to communicate them between stages? Unlike a looping program in a traditional language, the Halide *algorithm* does not specify these choices. Instead, they're specified by a separate *schedule*.

For each stage, we think about these choices in four parts:

1. In what order should we iterate over the points in that stage?
2. In what order should we lay out those points in memory to store their results?
3. At what granularity should we interleave the computation of that stage with the computation of the downstream stages which consume its results?
4. At what granularity should we store blocks of the function for reuse across iterations of its consumers?

These choices affect performance through locality (e.g., cache behavior), exposed parallelism, and the need for recomputation (at grain boundaries). Together, they specify a set of nested loops (invisible to the programmer) that our compiler uses to generate executable code. Scheduling instructions are concise and can be written independently of the definition of the algorithms, as shown in Figure 1(c).

3.1 Scheduling within a function

The order of evaluation of the points within a function is defined by a family of common transformations applied to a default (sequential, row-major) loop nest over the grid of points in the function's domain. Loop dimensions can be split, merged, and reordered. Because the regions computed are simple intervals (axis-aligned bounding boxes within the grid), the result is always a perfect loop nest. In addition, the resulting loops can be unrolled, or mapped to parallel threads, SIMD vector lanes, and GPU kernel dimensions.

The dimensions of the storage layout for a function's results can similarly be reordered, allowing common transformations such as column- versus row-major layouts. Storage can also be *folded* across a dimension into a circular buffer of a fixed size.

The dimensions of an RDom in an update step may also be reordered, parallelized, etc., but only if the compiler can prove that different update iterations across those dimensions do not interact. Recent work has added support for splitting associative reductions to create parallel reduction trees.[25]

3.2 Scheduling across functions

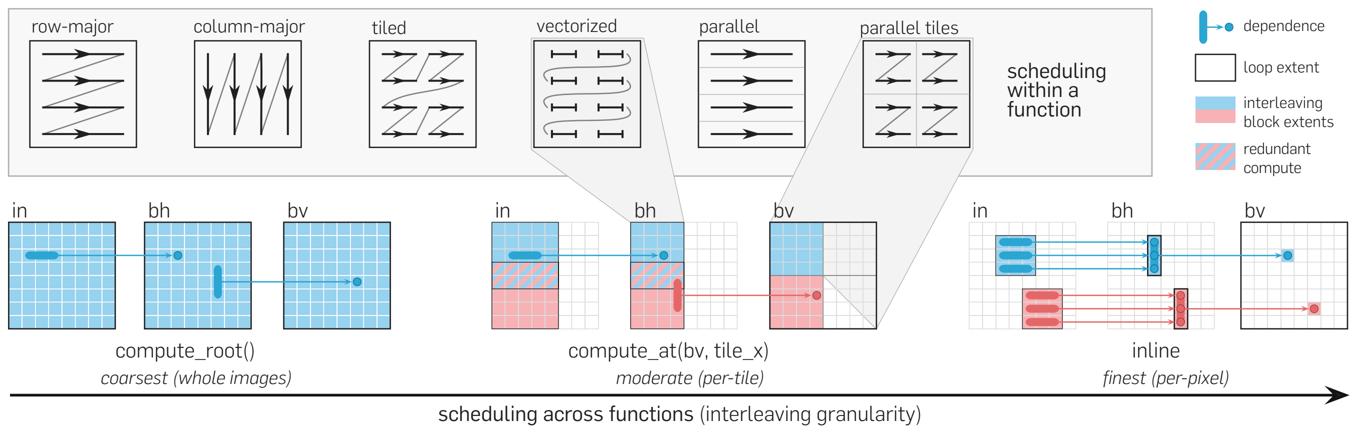
The more unique part of Halide's model of schedules is how it transforms computation and storage *across* functions.

Consider the simple two-stage blur algorithm. When scheduling across these two stages, we call the first stage, bh, the "producer," and the second stage, bv, its "consumer." So far, the scheduled order of computation *within* each function (discussed above) defines a perfect loop nest for each function. For example, the default schedule for the output stage bv gives a simple row-major loop nest equivalent to:

```
for bv.y in bv.min.y to bv.max.y:
  for bv.x in bv.min.x to bv.max.x:
    compute and store bv(bv.x, bv.y)
```

The schedule *across* functions specifies at what level in this

Figure 2. A Halide schedule specifies the order of computation *within* each stage in an image processing algorithm (top, discussed in Sec. 3.1), and the granularity of interleaving *across* producer and consumer stages (bottom, discussed in Sec. 3.2). In the two-stage blur algorithm, coarse-grained interleaving (bottom-left) computes whole stages at a time, sacrificing producer-consumer locality. Finer-grained interleaving improves locality but introduces redundant computation where the stencils for the grains overlap (denoted by hatched regions in the bottom-middle). In the extreme case of per-pixel interleaving (bottom-right), values are immediately consumed as soon as they are computed (maximizing locality), but all values of the producer stages are redundantly recomputed in each place they are reused (here, 3 times in bh and 9 times transitively in in). Whereas this shows a single shared order and granularity choice made applied throughout a simple pipeline, all of these choices can be made separately for each function in a whole algorithm.



loop nest around the *consumer* stage the program should evaluate the required part of the *producer* stage (here, bh). This determines the granularity with which the computation of the different functions is interleaved.

For example, Halide's default schedule for producers is *inline*, which means that if we leave the default schedule for bh, it is evaluated directly everywhere it is called:

```
for bv.y in bv.min.y to bv.max.y:
  for bv.x in bv.min.x to bv.max.x:
    bv(bv.x, bv.y) = // bh(x,y-1)+bh(x,y)+bh(x,y+1) =
        (in(bv.x-1,bv.y-1)+in(bv.x,bv.y-1)+in(bv.x+1,bv.y-1))
       +(in(bv.x-1,bv.y  )+in(bv.x,bv.y  )+in(bv.x+1,bv.y  ))
       +(in(bv.x-1,bv.y+1)+in(bv.x,bv.y+1)+in(bv.x+1,bv.y+1))
```

This maximizes locality between the producer and consumer, since each value of bh is immediately used as soon as it is computed, so it does not need to be stored and loaded in far-away memory. However, because the stencils of neighboring pixels overlap, every pixel in bh is computed three times instead of once: when the outer loop moves to the next row down, each pixel of bv recomputes two of the same pixels of bh that were just computed by the pixel right above it.

The other obvious choice we could make is to not interleave these two stages at all. Instead, we compute all of bh before computing any of bv. We express this in Halide by saying that bh should be computed at the very outermost, or "*root*" level, outside all loops over its consumer. This is done by calling bh.compute_root(), generating the complete equivalent program:

```
// compute bh over slightly enlarged window,
// to fulfill all uses in bv
for bh.y in bv.min.y-1 to bv.max.y+1:
  for bh.x in bv.min.x to bv.max.x:
    bh(bh.x, bh.y) = in(bh.x-1, bh.y)
                   + in(bh.x  , bh.y)
                   + in(bh.x+1, bh.y)

// compute bv using bh
for bv.y in bv.min.y to bv.max.y:
  for bv.x in bv.min.x to bv.max.x:
    bv(bv.x, bv.y) = bh(bv.x, bv.y-1)
                   + bh(bv.x, bv.y)
                   + bh(bv.x, bv.y+1)
```

This only computes each pixel in each stage exactly once, wasting no work, but it destroys any producer-consumer locality between the two stages: an entire image of intermediate results has to be computed between where values are produced in the first stage and where they are consumed in the second. This schedule is equivalent to the clean C++ as shown in Figure 1(a), which suffers from the same problem.

Looking at these two examples, we see a tension between locality and (excess, or redundant) computation. This tension is fundamental to computations with stencil dependencies, since stencils overlap between iterations and prevent simple loop fusion without duplicating work. However, this tension can be balanced along a continuum between these extremes.

For example, we can change the schedule within bv to iterate in a tiled order by splitting its two dimensions by a chosen tile size, and then reordering the four resulting dimensions. We can do this by saying:

```
bv.split(x, tx, xi, tile_size)
  .split(y, ty, yi, tile_size)
  .reorder(xi, yi, tx, ty)
```

which can be abbreviated simply as:

```
bv.tile(x, y, tx, ty, xi, yi, tile_size, tile_size)
```

This generates the loop nest:

```
// simplified to ignore non-evenly divisible tile sizes
for bv.ty in bv.min.y/tile_size to bv.max.y/tile_size:
  for bv.tx in bv.min.x/tile_size to bv.max.x/tile_size:
    for bv.yi in 0 to tile_size-1:
      for bv.xi in 0 to tile_size-1:
        compute bv(bv.tx*tile_size + bv.xi,
                   bv.ty*tile_size + bv.yi)
```

Now, by computing bh at the granularity of *tiles* of bv—at the bv.tx loop level—we get a less fine-grained interleaving than individual pixels, but one more fine-grained than the whole image. In Halide's scheduling language, we specify this by saying bh.compute_at(bv, tx). This is the key scheduling choice in the optimized implementations as shown in Figure 1(b) and (c). This generates the loop nest:

```
for bv.ty in bv.min.y/tile_size to bv.max.y/tile_size:
  for bv.tx in bv.min.x/tile_size to bv.max.x/tile_size:
    // compute tile of bh
    for bh.y in (bv.ty*tile_size-1)to ((bv.ty+1)*tile_size):
      for bh.x in (bv.tx*tile_size) to ((bv.tx+1)*tile_
      size-1):
        bh(bh.x, bh.y)= in(bh.x-1, bh.y)
                      + in(bh.x  , bh.y)
                      + in(bh.x+1, bh.y)

    // compute tile of bv
    for bv.yi in 0 to tile_size-1:
      for bv.xi in 0 to tile_size-1:
        bv.y = bv.ty*tile_size + bv.y
        bv.x = bv.tx*tile_size + bv.x
        bv(bv.x, bv.y) = bh(bv.x, bv.y-1)
                       + bh(bv.x, bv.y  )
                       + bh(bv.x, bv.y+1)
```

This organization recomputes only one row on the top and bottom of each tile, rather than every row, while improving locality to the level of tiles rather than whole images. By increasing or decreasing the size of these tiles, we can easily trade off between locality and computation. Common choices would be sized to fit tiles in a given level of cache. The ideal choice depends on the relative cost of recomputing a pixel of the producer function versus storing and loading it from each level of the memory hierarchy.

It is important to realize that this is not just a constant-factor tweak in the resulting code: if the tile size is a constant, we have *asymptotically* improved the locality relative to the naïve interleaving, by reducing the reuse distance for intermediate values from $O(n)$ to $O(1)$.

We can also choose to compute bh at any of the other loop levels in bv: the schedule of computation across stages in Halide is specified, for each producer function, as any single level in the scheduled loop nest of the downstream pipeline.

The fourth and final core part of the schedule similarly specifies the granularity of storage across functions. The granularity of storage determines at what loop level in the evaluation of the consuming pipeline the results of a producer function should be stored for reuse. On architectures such as GPUs, different levels are also mapped to different parts of the specialized memory hierarchy. Using this and other scheduling primitives, Halide can express additional locality-optimizing patterns of computation, such as line-buffering or software pipelining, but these are beyond the scope of this paper. More complete discussion can be found in prior publications.[21, 23]

In the DAG of functions that makes up a program, we have local ordering choices for each node, and producer-consumer granularity choices for each edge. In the real applications presented in Section 5, the optimized schedules make different but inter-dependent choices for each of dozens of stages and producer-consumer dependencies.

4. THE COMPILER

The Halide compiler takes a graph of Halide functions and *lowers* it to imperative code for a variety of CPU, GPU, and Digital Signal Processor (DSP) architectures. This process is directed by the schedule. It then converts this imperative code to LLVM's intermediate representation, and generates machine code from there.

In contrast to a traditional compiler, lowering makes very few heuristic decisions. Instead, it is designed to be highly deterministic and controllable. Any time there is a choice to be made that might affect performance we defer to the schedule.

Lowering has two essential responsibilities. The first is synthesizing the loop nest specified by the schedule, including any vectorization, unrolling, multi-core parallelism, prefetching, memoization of stages, and offloading work to GPU or DSP accelerators. The second responsibility of lowering is inferring the regions of each function that must be computed to satisfy all producer-consumer dependencies.

Lowering additionally implements optional passes for tracing, profiling, and similar instrumentation, and passes that apply opportunistic optimizations. These include partitioning loops in order to ameliorate the effect of boundary conditions, tightening loop bounds when it can be proved that the loop body does nothing for some of its domain, and deriving runtime conditions under which the computation of stages can be skipped entirely.

For a full description of the compiler see our original PLDI paper,[21] and the source code itself.[14]

5. RESULTS & EVALUATION

Halide has been open source since it was released by MIT in 2012, Halide source repository.[14] The largest user of and contributor to Halide is Google. The largest Halide pipeline we are aware of is Google's HDR+[15] which produces every photograph taken on their flagship Pixel phones. Halide pipelines are also run at scale within Google Photos, YouTube, and elsewhere. Outside of Google, Halide code is shipping in products from Instagram, Facebook, Adobe, and others.

In the original Halide publications,[17, 22, 23] we evaluated the Halide representation and compiler by implementing a variety of modern image processing pipelines in Halide and comparing the result to the best previously-published expert implementations we could find. The pipelines exhibit a wide variety of data types, access patterns, and pipeline lengths. Here, we highlight four representative pipelines that approximately span this space (Figure 3). These pipelines also approximately correspond to the major algorithms in Google's HDR+ pipeline, albeit at a more modest scale. Our more recent work and open source code also includes competitive matrix-matrix multiply (GEMM) and convolutional neural network implementations in very little code.[14, 17] We generally find that Halide code is many times faster than equivalently simple reference code, many times more concise than equivalently fast code, or some tradeoff between those two extremes.

Benchmark applications

Bilateral grid is an efficient implementation of the bilateral filter, which smoothes an image while preserving its main edges.[7] 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 its axes with three

Figure 3. Summary of the code size and running time Halide implementations compared to handwritten reference implementations of four representative computational photography algorithms. The time measurements are reported in milliseconds to process the representative image (4–6 megapixels), except for the fast Fourier transform. The FFT experiment is for the tiled real-to-complex transform, and reports time in nanoseconds per 16 × 16 tile.

Bilateral grid	Local Laplacian filters	Fast Fourier transform	Camera pipeline
Reference C++: 122 lines	**C++, OpenMP+iIPP:** 262 lines	**FFTW:** *thousands*	**Optimized assembly**: 463 lines
Quad core x86: 150ms	Quad core x86: 210ms	Quad core x86: 384ns	ARM core: 39ms
CUDA C++: 370 lines		Quad core ARM: 5960ns	
GTX 980: 2.7ms			
Halide algorithm: 34 lines	**Halide algorithm:** 62 lines	**Halide algorithm:** 350 lines	**Halide algorithm:** 170 lines
schedule: 6 lines	**schedule:** 11 lines	**schedule:** 30 lines	**schedule:** 50 lines
Quad core x86: 14ms	Quad core x86: 92ms	Quad core x86: 250ns	ARM core: 41ms
GPU schedule: 6 lines	**GPU schedule:** 9 lines	Quad core ARM: 1250ns	**DSP schedule:** 70 lines
GTX 980: 2.3ms	GTX 980: 23ms		Hexagon 680: 15ms

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 naïve 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 11 times faster than the original. The speedup comes from a combination of parallelism, tilelevel fusion of some stages, and careful reordering of dimensions to control parallel grain size in the grid.

We also compared the Halide version to a hand-tuned GPU implementation from the original authors, written in 370 lines of CUDA code. The same Halide algorithm, paired with a different schedule, was 17% faster than the hand-written CUDA, but about $\frac{1}{10}$th the total code size. The Halide compiler generates similar GPU code to the reference, but with Halide we quickly found a new point in the schedule space. It sacrifices some parallelism in the grid construction step to reduce synchronization overhead, and 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 much harder to express in CUDA, but are easily described by our schedule representation.

Local Laplacian filters uses a multi-scale approach to tone map images and enhance local contrast in an edge-respecting fashion.[2] It is used in the clarity, tone mapping, and other filters in Adobe Photoshop and Lightroom. The algorithm builds and manipulates several image pyramids. The filter output is produced by a data-dependent resampling from the processed pyramids. With the parameters we used, the pipeline contains 99 different stages, operating at many scales, and with several different computational patterns.

The reference implementation is 262 lines of C++, developed at Adobe. It is carefully parallelized with OpenMP, and offloads most intensive kernels to tuned assembly routines from Intel Performance Primitives (IPP). 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 times 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 2.3 times faster than the highly optimized expert implementation (at least 20 times faster than the clean C++ without IPP and OpenMP). The resulting schedule is complex, mixing different fusion, tiling, vectorization, and multithreading strategies throughout the 99 stage graph. In C, it would correspond to hundreds of loops over thousands of lines of code, but in Halide it is just 11 lines.

The same program compiles with a different schedule to a hybrid CPU/GPU program with 23 unique GPU kernels, each representing a different subset of the overall graph. It runs 9.1 times faster than the hand-tuned Adobe implementation, and is four times faster than the best parallel and vectorized implementation on the CPU.

Fast Fourier transform implements small 2D FFTs, based on an algorithm adapted for GPUs.[13] Small 2D tiled FFTs are used in many image processing workloads. The strategy in this implementation is to perform Cooley-Tukey FFTs on columns, which is convenient for vectorizing along the rows. The data is transposed repeatedly such that all of the FFTs are performed on columns; after the FFT is complete, the data is transposed back to the original orientation. Real FFTs are implemented by computing pairs of columns of FFTs simultaneously.

We compare the performance of our FFT implementation to FFTW.[11] For 16×16, 32×32, and 48×48 complex-to-complex FFTs, the Halide FFT is about 1.3 times faster than FFTW. For similarly sized real FFTs, Halide is between 1.5 times and 2.1 times faster than FFTW on x86, and up to 4.6 times faster than FFTW on ARM. For larger FFTs, the Halide FFT performance begins to drop relative to FFTW, becoming comparable to FFTW at 64×64 and dropping further from there, as we have not implemented a vectorization strategy more suitable for these larger FFTs.

In addition to the raw performance, the Halide FFT has a number of other advantages over FFT libraries. Because the FFT is expressed in pure Halide, the operations being performed in the Fourier domain can be fused into the FFT itself, improving locality. The implementation is also more easily modified, for example to compute fixed point or zero-padded FFTs. Because the FFT is expressed at a high level, with the optimizations expressed in the schedule, it can deliver state-of-the-art performance across a variety of platforms.

Camera pipeline transforms the raw data recorded by a camera sensor into a photograph. It performs four tasks: hot pixel suppression, demosaicking, color correction, and global tone mapping (i.e., gamma correction and contrast enhancement). These steps contain a variety of operations including stencils, color space transformations, and table lookups. The demosaicking alone is a combination of 21 inter-dependent stencils.

The reference comparison is a single carefully tiled and fused loop nest from the Frankencamera, expressed in 463 lines of C++ with ARM NEON intrinsics and inline assembly.[1] All producer-consumer communication is staged through scratch buffers. The Halide implementation is 170 lines describing 32 functions and 22 different stencils, literally translated from the pseudocode in the comments explaining the original source. With 70 lines of combined schedule code, the Halide algorithm can be compiled to target 32- and 64-bit x86 and ARM, is trivially parallelized across CPU cores, and can also be compiled to run on a Qualcomm Hexagon DSP.

We benchmarked this workload on a Google Pixel smartphone, which contains a Hexagon 680 DSP with HVX (Hexagon Vector Extensions), a standard feature of the popular Snapdragon 820 SoC. The Halide CPU code performs almost identically to the ARM reference implementations on a single core. Using all four CPU cores, the Halide implementation is 2.7 times faster than the single-core reference implementation. (The CPU cores on this processor are asymmetric, with a theoretical linear speedup of 3.5 times across all four.) The Halide Hexagon implementation performs similar to the four core CPU implementation, which is in line with the theoretical throughput of two HVX processing clusters. However, while we expect that to be the case, it delivers that throughput using up to 10 times less power.

Porting the camera pipeline to Hexagon required modifying the schedule to use a sliding window on the rows of the image, instead of tiling. Hexagon has extremely large vectors, and also benefits heavily from loop pipelining, which means that efficient 2D tile sizes are too large to fit in cache. Halide allowed us to easily rewrite the camera pipeline from a tiled schedule to a sliding window schedule, without changing the algorithm description. In contrast, rewriting the handwritten ARM implementation to use a sliding window would be a serious undertaking, as the tiling logic is inextricably linked to the algorithm description, all the way into the indexing arithmetic in the assembly inner loops.

Scheduling new algorithms

To better understand how long it takes an expert Halide developer to schedule an algorithm when starting from scratch, we recruited two professional Halide developers (authors on this paper) into a scheduling "race." They selected three new, nontrivial applications they had never scheduled before (LENSBLUR, NLMEANS, and MAXFILTER) and implemented the original Halide algorithm for these programs. For each benchmark, each programmer independently developed a schedule in a single programming session. The programmer stopped optimizing after converging on a solution they considered their reasonable best. While developing the schedules the developers documented their progress by measuring the performance of their current schedule at various points in the session.

In each case, the developers started out with relatively simple baseline schedules, but which generally still included at least SIMD vectorization and multicore parallelism. Nonetheless, the speedup from this baseline to their best optimized performance was always many times.

Results of the race are as shown in Figure 4. The X-axis in each of the graphs indicates development time (in minutes) for the schedules. The Y-axis shows the performance of the benchmark (measured as pixel throughput, so higher is better). The yellow and gray lines each correspond to one of the programmers.

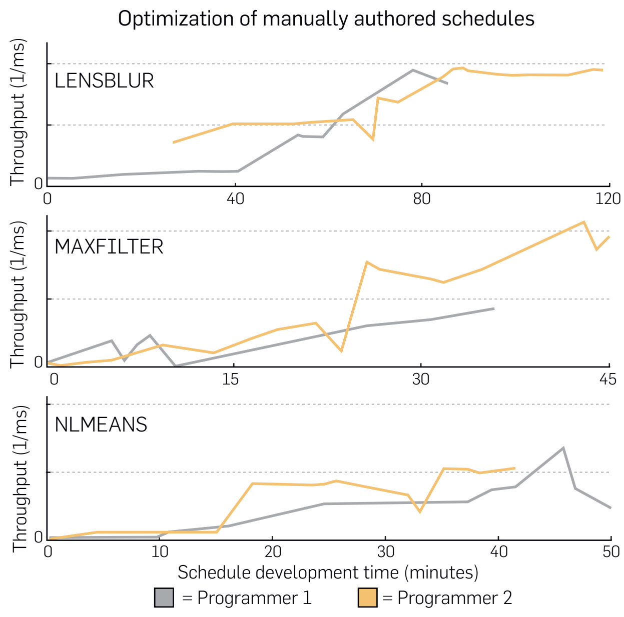
Arriving at a good schedule requires significant optimization effort, even for experts. For these applications, it took on the order of an hour of iteration and experimentation for the developers to feel their optimization had reasonably converged. The resulting schedules, however, are only a few lines of code, and dramatically simpler than equivalent code in explicit C++ loops.

Automatically determining good schedules

To further accelerate development of high-performance code, and to make Halide accessible to programmers without optimization experience, we have also worked on ways to *automatically* optimize Halide programs.

In the original version of this paper we demonstrated a prototype *autotuner* which automatically searched the (exponentially large) space of schedules for a program using stochastic search with a genetic algorithm.[23] For each generated schedule, the autotuner benchmarked its actual runtime by compiling and running a complete test program given by the user. In a few hours to days of tuning, it was able to generate schedules competitive with hand-tuned results

Figure 4. Two professional Halide developers were tasked with developing schedules for three new programs. The graphs plot the runtime of their schedules, over time as they developed them. The developers converged to what they considered "reasonable" performance in on the order of an hour per program.



for several applications, including some presented above.

To converge in reasonable time on real programs, we found that higher-level *heuristics* are critical to guide the search, because of the high cost of compiling and running each new test. Drawing on this experience, we have since found that using a simple cost model in place of recompiling and benchmarking each configuration, and restricting the search to greedily consider variants of common interleaving patterns, gave a much faster and more scalable automatic scheduling method.[17] The cost model is based directly on the tradeoff we observe between parallelism, locality, and redundant work. The tool is also easier to use since it directly analyzes the algorithm, rather than requiring a runnable benchmark program and data set.

6. RELATED WORK

Graphics & image processing languages

Domain-specific languages for image processing go back at least as far as Bell Labs' Pico and POPI.[16] Most prior image processing languages have focused on efficient expression of individual kernels. Pan introduced a functional model for image processing much like our own, in which images are functions from coordinates to values.[8] Recently, others have demonstrated automatic optimization of image processing code in a subset of the Halide's algorithm model using the polyhedral model.[18]

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.[4] 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.

Loop transformation

Most compiler optimizations for numerical programs are based on loop analysis and transformation, including auto-vectorization, loop interchange, fusion, and tiling.[3] The polyhedral model is a powerful tool for modeling and transforming looping imperative programs.[10] Halide's model considers only axis-aligned bounding regions, not general polytopes—a practical simplification for image processing and many other applications. Interval analysis is simpler than modern polyhedral analysis, but it can effectively analyze through a wide range of expressions, and it is trivial to generate high quality loop nests from intervals. This simple generality is essential for Halide's design in which all bounds are inferred from context. Most traditional loop optimizations also do not consider *recomputation* of values, but in image processing this can be a large performance win and is central to the choices we consider during optimization.

Parallel languages

Many data-parallel languages have been proposed. Chapel and its predecessors focus on computations over regions of regular grids, much like Halide, with explicit primitives to control distribution of grid regions across memory and processors.[6] Popular today, CUDA and OpenCL expose an imperative, single program-multiple data programming model which can target both GPUs and multicore CPUs with SIMD units.[5, 19] Like C, these languages allow the specification of very high performance implementations for many algorithms, but because their semantics closely model the underlying machine, they also deeply conflate algorithms with optimization.

Streaming languages encode data and task parallelism in graphs of kernels, which compilers automatically schedule using tiling, fusion, fission, and 1D stencil optimization.[12] Halide's model of scheduling addresses the problem of multidimensional stencils with parallelism, where recomputation versus storage becomes a critical but complex choice.

A separate line of research creates explicit languages for choices of how problems are mapped into physical execution, much like Halide's decoupling of schedules from algorithms. SPIRAL uses a domain-specific language to specify linear signal processing operations at the level of mathematical operators,[20] and a separate algebra of rewrite rules describes ways these operations can be turned into efficient code for a particular architecture. Sequoia defines a model where a user-defined "mapping file" describes how to execute tasks on a tree-like memory hierarchy.[9] The CHiLL framework exposes polyhedral program transformations and code generation choices via a dedicated scripting language.[24]

7. CONCLUSION AND PERSPECTIVE

Our initial hope in designing Halide was to allow expert-level manual control over the organization of computation in image processing code, which was at once far more controllable and powerful than traditional compiler flags and auto-vectorization, and far more productive than hand coding in low-level languages. After more than 4 years in production, we found this works even better than we first hoped. Across a range of applications and target architectures, we find that Halide's scheduling representation is able to model, and its compiler is able to generate, implementations which deliver state-of-the-art performance from surprisingly little code.

This performance comes from careful navigation of the high dimensional space of tradeoffs between locality, parallelism, and redundant recomputation in image processing pipelines. Expressing these tradeoffs in traditional languages is challenging enough, as shown by the much greater complexity of handwritten implementations, but finding the ideal balance is daunting when each change a programmer might want to try 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 *making it easy to test many more points in the space* than a human programmer could manually describe using explicit loops. We have found this design to generalize well from its initial CPU and GPU targets to new, more specialized (and notoriously hard-to-program) image processing accelerators and DSPs. We think it helps point the way towards the "OpenGL" of future specialized image processing hardware.

Still, while Halide's model of schedules is powerful and productive in the hands of experts, we have found it challenging for novices and those unfamiliar with high performance programming to master. Even for experts, optimized schedules grow difficult to maintain for algorithms beyond moderately sized blocks, and there's no complete answer for what should be done if those blocks are meant to be reused in different places or get moved around a larger pipeline (as with a library of reusable components). Fundamentally, schedules describe how to interleave computation across composition boundaries. Decoupling them from the fundamental algorithm gives simpler, more modular algorithm code, but modularizing scheduling choices without sacrificing performance remains an open problem. This led to our attempts to automate the process of scheduling in Halide,[17, 23] which have shown promise but remain ongoing work. An automatic scheduling system could both remove the burden from novice programmers, and schedule globally across composition boundaries even in large applications.

Finally, we have also found that, while the constraints and style of the Halide language were motivated by examples in modern image processing, the programming model is significantly more general: it expresses arbitrary bounded computations over multidimensional regular grids, and it has been applied to algorithms in linear algebra, scientific simulation, and machine learning. Really, Halide should be thought of as specific to the "data structure" of regular grids or multidimensional arrays, not to the "domain" of image processing.

Acknowledgments

As an open source project, Halide has received contributions from many people. Most notably, Zalman Stern, Steven Johnson, and Patricia Suriana are full-time developers on the project at Google and are responsible for a large amount of the current code. The Hexagon backend was developed with Pranav Bhandarkar, Ron Lieberman, Dan Palermo, and Anshuman Dasgupta at the Qualcomm Innovation Center. Eric Chan provided feedback and inspiration throughout the original design of Halide. This work was supported by DOE Award DE-SC0005288, NSF grant 0964004, grants from Intel and Quanta, and gifts from Cognex and Adobe. C

References

1. Adams, A., Talvala, E., Park, S.H., Jacobs, D.E., Ajdin, B., Gelfand, N., Dolson, J., Vaquero, D., Baek, J., Tico, M., Lensch, H.P.A., Matusik, W., Pulli, K., Horowitz, M., Levoy, M. The Frankencamera: An experimental platform for computational photography. *ACM Trans. Graph. 29*, 4 (2010), 29:1–29:12.
2. Aubry, M., Paris, S., Hasinoff, S.W., Kautz, J., Durand, F. Fast local Laplacian filters: Theory and applications. *ACM Trans. Graph. 33*, 5 (2014), 167.
3. Bacon, D.F., Graham, S.L., Sharp, O.J. Compiler transformations for high-performance computing. *ACM Comput Surv. 26*, 4 (Dec. 1994).
4. Blythe, D. The Direct3D 10 system. *ACM Trans. Graph. 25*, (2006), 724–734.
5. Buck, I. GPU computing: Programming a massively parallel processor. In *Proceedings of the International Symposium on Code Generation and Optimization* (Tessellations Publishing, Phoenix, Arizona, 2007).
6. Chamberlain, B., Callahan, D., Zima, H. Parallel programmability and the Chapel language. *Int J High Perform Comput Appl. 21*, (2007), 291–312.
7. Chen, J., Paris, S., Durand, F. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph. 26*, 3 (2007), 103:1–103:9.
8. Elliott, C. Functional image synthesis. In *Proceedings of Bridges 2001, Mathematical Connections in Art, Music, and Science* (IEEE Computer Society, Washington, DC, USA, 2001).
9. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P. Sequoia: Programming the memory hierarchy. In *ACM/IEEE conference on Supercomputing* (ACM, New York, NY, 2006).
10. Feautrier, P. Dataflow analysis of array and scalar references. *Int J Parallel Program. 20*, 1 (1991), 23–53.
11. Frigo, M., Johnson, S.G. The design and implementation of FFTW3. *Proc IEEE 93*, 2 (2005).
12. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., Meli, A.S., Leger, C., Lamb, A.A., Wong, J., Hoffman, H., Maze, D.Z., Amarasinghe, S. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (ACM, New York, NY, 2002).
13. Govindaraju, N., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE (Washington, DC, January 2008).
14. Halide source repository. http://github.com/halide/Halide.
15. Hasinoff, S.W., Sharlet, D., Geiss, R., Adams, A., Barron, J.T., Kainz, F., Chen, J., Levoy, M. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Trans. Graph. 35*, 6 (2016).
16. Holzmann, G. *Beyond Photography: The Digital Darkroom*. Prentice Hall, Englewood Cliffs, NJ, 1988.
17. Mullapudi, R.T., Adams, A., Sharlet, D., Ragan-Kelley, J., Fatahalian, K. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph. 35*, 4 (2016).
18. Mullapudi, R.T., Vasista, V., Bondhugula, U. PolyMage: Automatic optimization for image processing pipelines. In *ACM SIGPLAN Notices* (ACM, New York, NY, 2015), volume 50, 429–443.
19. The OpenCL specification, version 1.2. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2011.
20. 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., Rizzolo, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232–275.
21. Ragan-Kelley, J. Decoupling algorithms from the organization of computation for high performance image processing. PhD thesis, Massachusetts Institute of Technology (2014).
22. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*, 4 (2012).
23. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, New York, NY, 2013).
24. Rudy, G., Khan, M.M., Hall, M., Chen, C., Chame, J. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing* LCPC'10, (Springer-Verlag, Berlin, Heidelberg, 2011), 136–150.
25. Suriana, P., Adams, A., Kamil, S. Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (ACM, New York, NY, 2017).

Andrew Adams, and Dillon Sharlet ([abadams, dsharlet]@google.com), Google.
Saman Amarasinghe, and Frédo Durand ([saman, fredo]@csail.mit.edu), MIT CSAIL.
Connelly Barnes (connelly@cs.virginia.edu), University of Virginia.
Marc Levoy (levoy@google.com), Stanford University & Google.
Sylvain Paris (sparis@adobe.com), Adobe.
Jonathan Ragan-Kelley (jrk@eecs.berkeley.edu), UC Berkeley.