Automatically Scheduling Halide Image Processing Pipelines

Ravi Teja Mullapudi* Andrew Adams‡ Dillon Sharlet‡ Jonathan Ragan-Kelley† Kayvon Fatahalian*

*Carnegie Mellon University ‡Google †Stanford University

Abstract

The Halide image processing language has proven to be an effective system for authoring high-performance image processing code. Halide programmers need only provide a high-level strategy for mapping an image processing pipeline to a parallel machine (a *schedule*), and the Halide compiler carries out the mechanical task of generating platform-specific code that implements the schedule. Unfortunately, designing high-performance schedules for complex image processing pipelines requires substantial knowledge of modern hardware architecture and code-optimization techniques. In this paper we provide an algorithm for automatically generating high-performance schedules for Halide programs. Our solution extends the function bounds analysis already present in the Halide compiler to automatically perform locality and parallelism-enhancing global program transformations typical of those employed by expert Halide developers. The algorithm does not require costly (and often impractical) auto-tuning, and, in seconds, generates schedules for a broad set of image processing benchmarks that are performance-competitive with, and often better than, schedules manually authored by expert Halide developers on server and mobile CPUs, as well as GPUs.

Keywords: image processing, optimizing compilers, Halide

Concepts: •**Computing methodologies** → *Graphics systems and interfaces;*

1 Introduction

Image processing pipelines are essential components of a wide range of applications spanning computer graphics, computer vision, computational photography, medical imaging, and basic science. Trends such as the increasing sophistication of modern pipelines, growing resolution of image sensors, and deployment of image processing applications on resource-constrained devices has created an acute need for highly efficient image processing pipeline implementations.

In recent years, the Halide image processing language [Ragan-Kelley et al. 2012; Ragan-Kelley et al. 2013] has proven to be an effective system for authoring high-performance image processing code, and it is now used to synthesize production code used in datacenters and on hundreds of millions of smartphones. The key benefit of Halide is that it provides abstractions that enable programmers to rapidly explore the space of code optimizations most relevant to image processing workloads. Programmers need only provide a compact, functional description of an image processing *algorithm* and a separate, high-level description of how to globally optimize the algorithm's execution on a machine (called a *schedule*). The Halide compiler then handles the tedious, mechanical task of generating platform-specific code that implements the schedule (e.g., spawning threads, managing buffers, generating SIMD instructions).

Although Halide provides high-level abstractions for expressing schedules, *designing* schedules that perform well on modern hardware is hard; it requires expertise in modern optimization techniques and hardware architectures. For example, around 70 software engineers at Google currently write image processing algorithms in Halide, but they rely on a much smaller cadre of Halide scheduling experts to produce the most efficient implementations. Further, production image processing pipelines are long and complex, and are difficult to schedule even for the best Halide programmers. Arriving at a good schedule remains a laborious, iterative process of schedule tweaking and performance measurement. Also, in large production pipelines, software engineering considerations (e.g., modularity, code reuse) may preclude experts from having the global program knowledge needed to create optimal schedules.

In this paper we address this problem by providing an algorithm for automatically generating high-performance schedules for Halide programs. Our approach is to leverage the function bounds analysis already present in the Halide compiler to automatically perform locality enhancing global program transformations similar to those employed by expert Halide developers. The algorithm does not require costly (and often impractical) auto-tuning, and, in seconds, generates schedules that are competitive with, and sometimes better than, the best manually-created schedules for server and mobile CPUs, as well as GPUs. Because it is built using Halide's interval-based bounds-analysis system, our analyses apply to a broader set of image processing workloads than recent prior work. We demonstrate this advantage by presenting automatic scheduling results for a large set of complex image processing and image analysis benchmarks.

2 Prior Work

There have been a number of recent efforts to automatically generate efficient image processing pipelines from high-level programs.

Ragan-Kelley et al. [2013] employed auto-tuning guided by genetic search to automatically generate Halide schedules that were performance competitive with hand-tuned implementations. However, the search was guided by fragile, sometimes benchmark-specific, heuristics (that do not work with the modern formulation of the Halide language), and required a day or more to find high-quality schedules. A more general Halide auto-tuner was later implemented within the OpenTuner framework [Ansel et al. 2014]. This system was able to find efficient schedules for simpler pipelines (e.g., bilateral filtering) in about an hour, but fails to converge to good solutions on more complex pipelines. OpenTuner output for more complex pipelines such as RAW camera processing, pyramid blending, and multi-scale interpolation is five to ten times slower than hand-tuned implementations [Mullapudi et al. 2015].

While auto-tuning may seem like an attractive strategy for optimizing Halide programs, its use in a production setting is problematic for several reasons. First, the large size of production image processing pipelines presents convergence problems for auto-tuning systems—the choice space is too large and complex for brute force

SIGGRAPH '16 Technical Paper, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925952

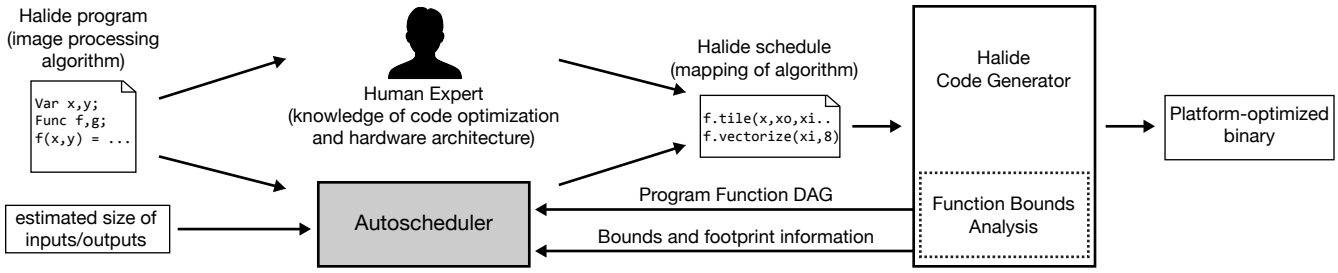

Figure 1: *Our system automatically generates schedules for Halide programs, a task currently performed by expert Halide programmers.*

search. Second, programs can take minutes to compile and must be deployed to target devices to benchmark (e.g., a cell phone or tablet) since build machines have different performance characteristics than deployment targets. This build-deploy-benchmark cycle can make tuning strategies that rely on measuring the real performance of a large number of program variants infeasibly slow. While programmers must also test the performance of their code on real machines, experts make judicious choices of which strategies to test, and learn quickly from each measurement, allowing them to converge to good solutions from a smaller number of benchmarking samples.

Other recent efforts have achieved high performance by limiting the space of image processing programs considered. For example, Darkroom [Hegarty et al. 2014] limits pipelines to contain only fixed-size stencil operations and no resampling, and adopts a line-buffered scheduling strategy that is ideal for the performance concerns of FPGA architectures. These constraints allow pipeline scheduling to be formulated as an integer linear programming problem that can be solved in seconds for storage-optimal schedules. This strategy can be combined with image tiling optimizations to achieve good performance on multi-core CPUs. However, Darkroom's analysis does not extend to data-dependent image processing operations, reductions, or resampling and rate changes. Building upon Darkroom, Rigel employs the synchronous dataflow model to further extend pipeline scheduling capabilities to multi-resolution operations and dynamic filtering [Hegarty et al. 2016], but in doing so it sacrifices fully automatic scheduling.

PolyMage [Mullapudi et al. 2015] extends polyhedral analysis techniques to schedule image processing pipelines implemented in a Halide-like dataflow language. Although the space of possible schedules for these programs is large, PolyMage demonstrates that good solutions lie in a subspace of schedules that consider only pipeline stage fusion and overlapped tiling of the output image. While PolyMage uses polyhedral analysis to generate efficient loop nests once stages have been fused, it relies on auto-tuning over a range of tile sizes to make stage-fusion decisions. Data-dependent operations (histograms, lookup tables), non-affine programs, and computations that feature significant input data reuse (deep neural networks, matrix multiplication) fall outside the scope of PolyMage's polyhedral overlapped tiling analysis (which is limited to stencils and up/down-sampling). We adopt PolyMage's grouping-then-tiling approach, but use interval analysis, rather than polyhedral techniques, to do so, yielding completely different, and more practical, compiler internals. Our solution does not require autotuning, features wider application scope, and emits human-understandable Halide schedules.

3 Representing and Scheduling Programs

The top half of Figure 1 illustrates the current workflow of developing an efficient Halide program. A programmer first authors a functional description of an image processing application. Next, a programmer (potentially a different one, with code optimization expertise) develops a schedule for the application. Finally the Halide compiler uses the program definition and schedule to generate efficient multi-threaded, vectorized code for a target machine. While the Halide compiler is responsible for the mechanical details of code generation, it is the schedule writer that is responsible for using global knowledge of the Halide application to perform the most critical program transformations.

In this section we summarize common global program restructuring decisions made by Halide developers when authoring efficient schedules. We assume familiarity with the Halide system, and refer the reader to [Ragan-Kelley et al. 2012; Ragan-Kelley et al. 2013] for a comprehensive description of the language and its features.

A Halide program is a DAG of computation, where each node in the DAG corresponds to a function defined on an n-D domain. For example, the top of Figure 2 shows a simple Halide program which performs a two-pass image blur, as well as its corresponding DAG. The program contains two functions (`blurx` and `out`), each defined on a 2-D domain parameterized by variables `x` and `y`. Edges in the DAG correspond to data dependencies between functions. For example, there is an edge from `blurx` to `out` because each value of `out` is the sum of three values produced by `blurx`.

A Halide algorithm only specifies what computations are needed to evaluate the output function `out` at points in its domain. It *does not* specify the order in which different points in the domain are computed, or the order of intermediate computations necessary to produce these points. It is the job of a Halide schedule to specify an efficient execution order for all points in the output domain.

3.1 Scheduling for Producer-Consumer Locality

One plausible execution order for the blur pipeline is given by Schedule 1 in Figure 2. When applied to a 6 megapixel ($3k \times 2k$) output image, the implementation computes all required elements of `blurx`, stores them in a large buffer, then uses this buffer to compute all elements of `out`. This solution is simple, trivially parallelizable, but suffers from poor producer-consumer locality. All outputs of `blurx` are stored to memory before they are subsequently loaded in the computation of `blur`. The implementation will be memory-bandwidth bound, and perform poorly on most modern processors.

Schedules 2 and 3 in Figure 2 provide alternative implementations that *place* the computation of elements at `blurx` at different levels in the loop nest that computes `out` (see shaded regions). The different placement decisions reduce the *reuse distance* (the time between accesses to the same piece of data) for intermediate values computed by the program. Placing the computation of `blurx` at the innermost loop of `out` (Schedule 2) maximizes producer-consumer locality: three elements of `blurx` are produced, then immediately consumed (likely out of the processor's register file) to compute one value of `out`. However, the cost of achieving high locality is redundant computation: each element of `blurx` is now computed three times over the course of the entire computation. Schedule 3 achieves a better locality vs. redundant-compute balance by interleaving com-

Halide Program (image blur)

```
Image in(UInt(8), 2);
Var x, y;
Func blurx, out;
blurx(x,y) = (in(x-1,y)    + in(x,y)    + in(x+1,y)) / 3;
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3;
```

Pipeline Graph

Schedule 1: compute blurx at "top level"
(minimal work, minimal locality)

```
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]) / 3
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]) / 3
```

Schedule 2: compute blurx inside x loop of out
(maximal redundant work, maximal locality)

```
alloc out[2046][3072]
for each y in 1..2047:
  for each x in 0..3072:
    alloc blurx[3]
    for each yi in -1..1:
       blurx[yi+1] = (in[y+yi][x-1]+in[y+yi][x]+in[y+yi][x+1]) / 3
    out[y][x] = (blurx[0] + blurx[1] + blurx[2]) / 3
```

Schedule 3: compute 2D tile of blurx inside 2D tile of out

```
alloc out[2046][302]
for each ty in 0..2048/32:
  for each tx in 0..3072/32:
    alloc blurx[34][32]
    for each y in -1..33:
      for each x in 0..33:
      blurx[y+1][x] = (in[ty*32+y][tx*32+xi-1] +
                       in[ty*32+y][tx*32+xi]   +
                       in[ty*32+y][tx*32+xi+1]) / 3
    for each y in 0..32:
      for each x in 0..32:
        out[ty*32+y][tx*32+x] = (blurx[y-1][x] + blurx[y][x] +
                                 blurx[y+1][x]) / 3
```

Figure 2: *Three loop orderings for a Halide* 3×3 *blur program each offer a different balance of producer-consumer locality, redundant computation (of* `blurx` *values), and available parallelism.*

putation of `blurx` and `out` at the level of 32×32 tiles. Because the amount of intermediate data is small, it can remain in the processor's cache. Also, redundant computation of elements of `blurx` is limited to the border regions of tiles. This strategy of *overlapped tiling* (which can be carried out in any number of dimensions) is a common optimization in image processing pipelines. Programs that exploit overlapped tiling are typically parallelized across tiles, so tile sizes are chosen to be large enough to amortize the overhead of each parallel task, but small enough to expose enough parallel work and ensure intermediate data remains in a cache.

Even in this simple example, global loop reorderings have substantial impact on the performance of the program. Real-world image processing pipelines may contain hundreds of functions, and scheduling them efficiently requires negotiating the trade-off between locality, parallelism, and redundant computation by multi-dimensional tiling of function loop nests and deciding where to place producers within the loop nests of consumers.

3.2 Scheduling for Input Reuse

In addition to increasing locality by moving the computation of values closer in time to their consumption (placing code for producer functions inside the loop nest of their consumer), it is also possible to increase the locality of multiple reads of the same value

Halide Program (dense matrix-matrix multiplication)

```
Image A( Float(32), 2);
Image B( Float(32), 2);
Var x, y;
RDom r(0, A.width());
Func C;
C(x,y) = sum(A(x,r.x) * B(r.x,y));
```

Schedule 1: no tiling, minimal locality

```
alloc out[1024][1024]
for each y in 0..1024:
  for each x in 0..1024:
    for each rx in 0...1024:
      out[y][x] += A[y][rx] * B[rx][x]
```

Schedule 2: tiled into 32x32 tiles for input locality

```
alloc out[1024][1024]
for each ty in 0..1024/32:
  for each tx in 0..1024/32:
    for each y in 0..32:
      for each x in 0...32:
         out[ty*32+y][tx*32+x] = 0
    for each trx in 0..1024/32:
      for each y in 0..32:
        for each x in 0..32:
          for each rx in 0..32:
             out[ty*32+y][tx*32+x] += A[ty*32+y][trx*32+rx] *
                                      B[trx*32+rx][tx*32+x]
```

Figure 3: *Tiling the output and reduction loop dimensions of matrix-multiplication (Schedule 2) improves locality of access to the inputs* `A` *and* `B`*, significantly improving program performance.*

by reordering the consumer's loop nest. This can be the preferred approach when the locality of data reuse is more important than producer-consumer locality.

The canonical example of such a workload is dense matrix-matrix multiplication, illustrated in Figure 3. When this program is executed on inputs `A` and `B` of size $N \times N$, computing `C` will access each element of `A` and `B` a total of N times. Note that in Schedule 1, an entire row or column of the input functions is accessed between accesses to the same input value (the reuse distance is proportional to the size of the input matrices). For large input matrices, input values will no longer be resident in the processor's cache, yielding low performance. A well-known optimization, shown in Schedule 2, is to tile the program's loop dimensions to reduce the problem to a sequence of smaller matrix multiplications whose inputs do remain cache resident. Loop tiling to maximize input locality is a key optimization in workloads featuring dense linear algebra or convolutional neural networks.

3.3 Function Bounds Analysis

Halide's scheduling primitives (`compute_at`, `reorder`, and `tile`) enable programmers to specify a wide range of locality-increasing program transformations such as the ones described above. In order to implement these strategies, the compiler must be able to determine the appropriate loop bounds and intermediate buffer sizes. For example, in Figure 2, Schedule 1, three values of `blurx` from the interval `(x,y-1..y+1)` must be computed and stored prior to computing the value of `out` at domain point `(x,y)`.

The Halide compiler infers symbolic function bounds via interval analysis on expressions used to define functions. For example, in the case of Figure 2, given the bounds `(xmin..xmax,ymin..ymax)` for the function `out`, Halide computes the following *symbolic bounds* expressions for the required values from `blurx` and `in`:

```
blurx:   (xmin..xmax,     ymin-1..ymax+1)
in:      (xmin-1..xmax+1, ymin-1..ymax+1)
```

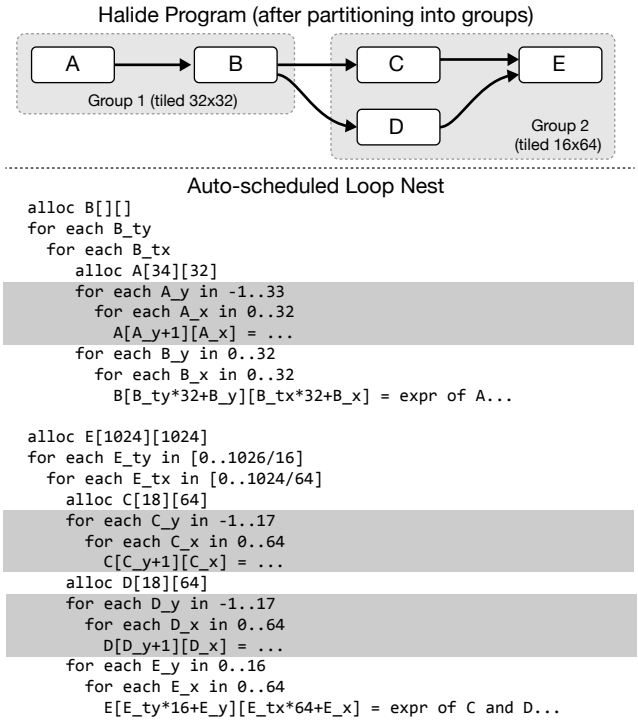

Figure 4: *The auto-scheduling algorithm partitions a Halide program into groups of functions and determines an efficient (potentially different) tiling for each group. Communication between groups is performed through in-memory buffers.*

If the bounds on the output domain are made *concrete*, i.e., assigned values such as `(5..10,10..20)`, then the required bounds for upstream functions can be concretely determined as well:

```
blurx:   (5..10, 9..21)
in:      (4..11, 9..21)
```

Starting from the output function, bounds inference propagates up the function dependency chain, ascribing bounds to all functions in the program DAG. When the Halide compiler cannot infer tight bounds for a function (e.g., due to data-dependent access by a consumer), the programmer can explicitly provide bounds to assist the compiler in generating efficient code (e.g., the programmer may have static knowledge that all accesses to a lookup table will be in the range `0..8`). Our automatic scheduling algorithm relies heavily on the results of bounds analysis to make decisions about how and when to employ the program restructuring optimizations described in this section.

4 Algorithm

In this section we describe an algorithm for automatically generating efficient schedules for Halide programs. The main idea is to partition the functions in a large Halide program into groups (subprograms), and independently perform producer-consumer locality and input reuse locality transformations on these groups.

As an overview of the method, Figure 4 illustrates a Halide program DAG containing five functions, organized into two groups. While Halide allows many possible computation orderings for a group, to make exploration of the optimization space tractable, our system follows Mullapudi et al. [2015] and only considers a narrower space schedules that tile the loop nest corresponding to the group's output function. The computation of all other functions in the group is placed within this tiled loop (a single placement decision is made for all producers). As we demonstrate in Section 5, we are able to find high-performance schedules in this subspace.

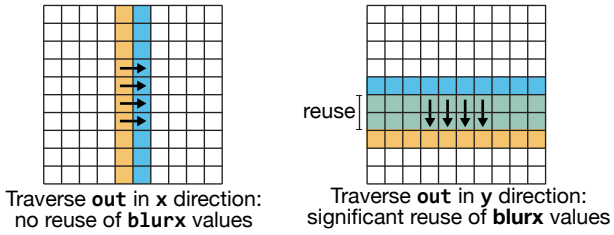

Figure 5: *Domain iteration order impacts the locality of data access. The shaded regions indicate values of* `blurx` *required to compute two sequential columns (left) or rows (right) of* `out` *(note region overlap on the right). The auto-scheduler uses estimates of data reuse from different domain orderings to generate candidate tile sizes and to reorder loops.*

The bottom of Figure 4 shows one possible schedule that results from applying these rules to the illustrated groups. (Loop bounds are written assuming functions are simple vertical stencils with support 3.) Notice that all functions are computed within the loop nest of the group's output function (gray regions), and that producer-consumer relationships in a group are managed through small intermediate buffers (`A`,`C`,`D`) that will likely remain resident in cache. Dependencies between groups are implemented via communication (via buffer `B`) through main memory.

Our auto-scheduler can analyze all programs expressible in the current version of the Halide language, with one exception: input programs are constrained to use only one update rule per function. Thus far we have not found this restriction to be too limiting, as our system is able to process a superset of the programs studied in all prior Halide publications.

In addition to an unmodified Halide program, the auto-scheduler requires as input an estimate of concrete bounds for the domain of the program's output function (e.g., the size of the outputs) as well as concrete bounds for any inputs that cannot be directly inferred from the output bounds. This extra requirement is reasonable for image processing workloads: it is common to know the approximate size of images a program will process as well as the size of inputs such as lookup tables whose bounds cannot be inferred due to data-dependent access. Also, when manually scheduling programs, programmers typically must estimate such sizes when making optimization decisions. The supplied concrete bounds need only be estimates: if the size of inputs supplied at runtime differ widely from the statically estimated bounds, generated schedules may be less efficient, but the compiled program will still be functionally correct.

4.1 Function Preprocessing

The first step in our analysis is to precompute properties of each function in the input program that will be useful when making scheduling decisions. For each function f we:

Estimate arithmetic cost. We estimate the arithmetic cost of computing one value of the function, given values for all required inputs. For example, the function `out` consists of three additions. We assign unit cost to most arithmetic operations and higher costs to more complex operations such as division and transcendentals.

Compute concrete bounds. We use Halide's bounds analysis to compute concrete bounds for all functions. This is possible since the programmer has supplied concrete bounds for the program's output function, as well as for all functions whose bounds cannot be inferred from the output.

Compute per-direction input reuse. To make scheduling decisions, it is helpful to know, for each function, which domain-iteration order yields the highest amount of input data reuse. For example, consider computing elements of `out` in row-major order as is the case in Figure 5-left. Since computing each output only requires inputs from the same column of values of `blurx`, each element of `out` requires a new set of inputs. There is no input data reuse, as evidenced by the non-overlapping yellow and blue bars in the figure. In contrast, if `out` was computed in column-major order, then successful computations would reuse two or three required inputs from `blurx` (Figure 5-right).

We estimate average input reuse of values from producer function g due to domain-traversal along a particular axis of consumer function f by computing input bounds on g for two adjacent strips of f. We compute the intersection of these two input regions as an estimate of input reuse. For example, to compute reuse due to domain-traversal in the `x` direction:

```
mid = c_xmin + (c_xmin + c_xmax) / 2
b1  = bounds(f, g, mid..mid+1,   c_ymin..c_ymax, ...)
b2  = bounds(f, g, mid+1..mid+2, c_ymin..c_ymax, ...)
b_reuse = intersect(b1, b2)
```

The function `bounds` above returns a concrete bound on elements of g needed to compute elements within the specified bound on f. The preprocessing step computes reuse bounds for all f's ancestors, for all domain directions. It is sufficient to sample reuse along a single strip of the domain since Halide's loops iterate over rectangular domains (the overlap estimate will apply throughout the domain).

4.2 Function Grouping and Tiling

Following preprocessing, the algorithm begins the process of function grouping. Grouping seeks to identify points in the program where it is beneficial to restructure computation ordering to improve producer-consumer locality. This requires determining if and how to tile consumer loop nests and where to place the computation of producer functions within these loop nests.

The algorithm makes these decisions using a greedy, agglomerative process that iteratively assigns functions in the Halide program into groups. The scheduling of each group is performed independently, and different ordering/tiling decisions are made per group. All groups containing multiple functions are constrained to have only a single output function (all dependency edges leaving the group are from the same function), and all other functions in the group will be computed within the loop nest of the output function. The iterative grouping process is similar to that employed by Mullapudi et al. [2015]. However, while their work makes grouping decisions given a predetermined loop tiling structure, our solution jointly makes grouping and loop tiling decisions.

Function grouping begins by placing each function in the program in its own group. Each of these singleton groups is initially tiled to maximize input data reuse (if reuse opportunities exist). Then, each iteration attempts to increase producer-consumer locality by merging two existing groups. The process terminates when the auto-scheduler estimates there is no performance benefit to merging any of the remaining groups. Each iteration of the function grouping process involves the following steps:

- Enumerate all remaining group merging opportunities.
- For each merging opportunity, estimate the performance benefit of merging the two groups. The act of evaluating the benefit of a merge requires determining a tiled loop-nest structure for the potential merged group.

```
1  def tile_singleton_group(g):
2    if is_pure(g):
3      return SINGLE_ELEMENT_TILE
4
5    cbounds        = concrete_bounds(output_func(g))
6    tiles          = generate_tile_sizes(output_func(g))
7    best_tile_size = SINGLE_ELEMENT_TILE
8    best_benefit   = 0
9
10   for each tile_size in tiles:
11     tile_cbounds   = tile_bounds(output_func(g),
12                                  tile_size)
13     total_tiles    = num_tiles(cbounds, tile_size)
14     footprint = group_tile_footprint(g, tile_cbounds)
15
16     # tile size too big
17     if (num_tiles < PARALLELISM_THRESHOLD ||
18         footprint > CACHE_SIZE):
19         continue
20
21     no_tile_loads = total_load_ops(g, cbounds)
22     tile_loads    = num_tiles * footprint
23     tile_benefit  = no_tile_loads - tile_loads
24
25     if (tile_benefit > best_tile_benefit):
26       best_tile_benefit = tile_benefit
27       best_tile_size = tile_size
28
29   return best_tile_size
```

Listing 1: `tile_singleton_group` *selects a loop tiling that minimizes the estimated number of cache misses when accessing input data.*

- Select the merge that yields the greatest performance benefit (provided at least one merge that provides benefit does exist) and merge the two groups.

We now describe the process of determing an initial tiling for all functions, and key steps in each merging iteration in more detail.

4.2.1 Initialization: Tiling for Input Data Reuse

Before grouping functions to improve the program's producer-consumer locality, the auto-scheduler attempts to find loop tilings that improve locality of input data accesses by each non-pure function. (Pure functions—functions that do not contain Halide update definitions or reduction domains, such as small stencils—exhibit high input locality without tiling.) Pseudocode for this decision process is given in Listing 1. The analysis adopts a simple model of a single-level memory hierarchy with a cache size given by the parameter `CACHE_SIZE`.

The method makes the simplifying assumption that when a non-pure function is not tiled, all its input data accesses are cache misses (counted in `total_load_ops`), and that when tiling the function all input data needed to compute one tile of output (`group_tile_footprint`) fits in cache (they are loaded exactly once per tile). The algorithm then seeks to find a tile size that minimizes the number of loads required to compute the final output. To accelerate the search over tile sizes, tile size selection is biased toward sizes that are elongated in the directions of most input reuse. (We defer description of the auto-scheduler's tile size selection process to Section 4.2.3.) The auto-scheduler also rejects tile sizes that result in too few tiles to adequately parallelize computation across all the target machine's cores.

```
1  # number of loads needed to compute region of output
2  def group_loads(g, cbounds):
3    total_tiles = num_tiles(cbounds, g.tile_size)
4    tile_cbounds = tile_bounds(group, g.tile_size)
5    return total_tiles * group_tile_loads(g, tile_cbounds)
6
7  # determine whether to merge groups prod and cons
8  def evaluate_group_merge(prod, cons):
9    merged = prod + cons  # potential merged group
10   prod_out     = output_func(prod)
11   merged_out   = output_func(merged)
12   prod_cbounds = concrete_bounds(prod_out)
13   cons_cbounds = concrete_bounds(merged_out)
14
15   prod_cost = arith_cost(prod, prod_cbounds) +
16               LOAD_COST * group_loads(prod, prod_cbounds)
17   cons_cost = arith_cost(cons, cons_cbounds) +
18               LOAD_COST * group_loads(cons, cons_cbounds)
19   no_merge_cost = prod_cost + cons_cost
20
21   tiles = generate_tile_sizes(merged_out)
22   best_benefit = -1
23   best_tile_size = SINGLE_ELEMENT_TILE
24
25   for each tile_size in tiles:
26     tile_cbounds   = tile_bounds(merged_out, tile_size)
27     total_tiles    = num_tiles(cons_cbounds, tile_size)
28     footprint = group_tile_footprint(merged, tile_cbounds)
29
30     if (num_tiles < PARALLELISM_THRESHOLD ||
31         footprint > CACHE_SIZE):
32         continue
33
34     tile_cost = arith_cost(merged, tile_cbounds) +
35         LOAD_COST * group_tile_loads(merged, tile_cbounds)
36     merge_cost    = num_tiles * tile_cost
37     merge_benefit = no_merge_cost - merge_cost
38
39     if (merge_benefit > best_benefit):
40       best_benefit   = merge_benefit
41       best_tile_size = tile_size
42
43     return (best_benefit, best_tile_size)
```

Listing 2: `evaluate_group_merge` *estimates the performance benefit of merging producer group* `prod` *into consumer group* `cons`*. It selects a consumer loop tiling that balances increased producer-consumer locality with redundant computation.*

4.2.2 Enumerating Merging Opportunities

The purpose of grouping functions is to increase producer-consumer locality between functions in the group. Therefore, two groups g_1 and g_2 are candidates for merging if the output of group g_1 is consumed by a function in g_2.

Since computing a function within two different loop nests would require duplicating its evaluation, grouping only attempts to merge groups where the output of g_1 is consumed by functions in *exactly one* group. For example, in Figure 4, function pairs `A-B`, `C-E`, and `D-E` are potential merging opportunities, but `B-C` and `B-D` are not. This condition ensures that all groups containing multiple functions have exactly one output function.

4.2.3 Evaluating Potential Merges

A merge is likely to benefit overall program performance if the performance benefit of reducing memory traffic outweighs the cost of introducing additional redundant computation. This trade-off is evaluated by the function `evaluate_group_merge` in Listing 2, which accepts as input two function groups (a producer group and a consumer group that depends on the output of the producer). `evaluate_group_merge` determines the best schedule for the merged group (a loop tiling of the output function and loop nest placement of all other functions) and returns an estimate of the performance benefit of performing the merge.

Generating tile sizes. Different tilings of the output loop nest yield different locality-redundant-compute trade-offs, so it is important for the auto-scheduler to establish the best tile size when estimating the benefit of a merge. It is cost-prohibitive for the auto-scheduler to evaluate the cost of all possible tilings (high-dimensional domains, such as the 5-D domains present in convolutional neural network layers, quickly explode the space of possible tile sizes), so we bias the set of tile dimensions using knowledge of the group output function's directional input reuse (Section 4.1). Specifically, we constrain tile sizes to be n-D hypercubes or n-D volumes elongated in the dimensions of greatest output function reuse. Redundant computation introduced by tiling is reduced when tiles have large extents in these dimensions. We further require tile sizes to have a minimum extent in the output functions innermost dimension (it's innermost loop). This minimum is set to a small multiple of the target machine vector width to avoid tile sizes that produce loop nests that cannot be efficiently vectorized. Tile sizes that are too large: they either result in a group's intermediate storage overflowing the cache, or yield too few tiles to enable parallel execution on the target machine, are immediately discarded. Note that evaluating the function `group_tile_footprint` (line 27) requires the compiler to determine the size of required temporary buffer allocations by scheduling the merged group's loop nest according to the candidate tile size.

Comparing costs. For each candidate tile size, the auto-scheduler estimates and compares the cost of performing a group merge (computing input group functions within the tile loop of the group's output function) with the cost of not merging the groups. The cost of not merging is given by the arithmetic cost of the producer and consumer groups (`arith_cost`, Listing 2, lines 14,16), plus the cost incurred by these groups to loads their inputs from memory (`group_loads`). The total number of loads (over the entire program's execution) is given by the concrete bounds of their output function (lines 11-12) and the chosen tiling of the groups. Loads from main memory are assumed to incur a cost specified by the auto-scheduler parameter `LOAD_COST`.

Assuming that all merged group intermediates are stored in cache resident buffers (footprint check in line 30), the cost of producing a tile of merged output (line 35) is the arithmetic cost of the operation, plus the size of the inputs needed to compute a single tile of the merged group (`group_tile_loads`). The total cost of the merged group is obtained by multiplying this estimate by the total number of tiles. `evaluate_group_merge` returns the estimated performance benefit of the best tiling found (if any benefit due to merging the groups exists).

4.3 Function Inlining

In addition to grouping functions (placing producers in the loop nests of consumers), it can be advantageous to inline producer functions into their consumers (a manually written Halide schedule requests this optimization using the `inline` directive). Although inlining is conceptually similar to choosing a tile size of one element when merging two groups, inlining in Halide results in more efficient generated code since intermediates are communicated through registers rather than via loads and stores to cached intermediate buffers.

Although described after grouping in this paper for the purposes of exposition, the compiler performs function inlining optimizations immediately after precomputation, and prior to grouping. Function inlining decisions are made using the same greedy, iterative approach used to make grouping decisions, with the following modifications:

- Since inlining duplicates the producer function's expression into the consumer, inlining can be applied to functions with multiple consumers. For simplicity, the auto-scheduler makes inlining decisions on a per-function basis. Either a function is inlined into all of its consumers (essentially removing the function from the program DAG), or to none of its consumers (it remains in the DAG and is subsequently considered for grouping as described in Section 4.2).
- When evaluating the performance benefit of inlining, the auto-scheduler must consider the cost of inlining a function into *all consumers* (not just a single consumer as was the case during group merging decisions).
- The auto-scheduler does not use bounds analysis to estimate the arithmetic cost of the results of an inlining transformation. Instead it substitutes the producer function expression into the consumer function and reevaluates the arithmetic cost of the new expression. This is a more accurate measure of cost because bounds analysis can overestimate the actual number of values required by a consumer.

4.4 Final Schedule Generation

After inlining and merging function groups the auto-scheduler is left with a list of groups, each with a specified loop tiling. The final step of scheduling is to perform final optimizations and to generate a complete Halide schedule for each of these groups.

The first step is to reorder each group output function's loops in order of maximal input locality. This is a general optimization that helps to reduce the reuse distance of accesses to group inputs. To maintain spatial locality of data access and the ability to synthesize efficient vectorized code, loop reordering is constrained to never move the innermost dimension of the output function's loop nest out of its starting position. Next, the auto-scheduler unrolls the innermost loop if it contains only a small number of iterations. It then vectorizes this loop. Finally, the auto-scheduler parallelizes as many outermost dimensions of the loop nest as necessary to obtain sufficient multi-core parallelism for the target machine.

After these operations, each group has complete, optimized schedule. The Halide compiler then generates machine code for this schedule using its standard compilation process.

5 Evaluation

We evaluated our auto-scheduling algorithm on the 14 Halide pipelines listed in Table 1. These benchmarks span a range of computational photography, image processing, and computer vision workloads. Eight of the benchmarks are drawn from public literature [Ragan-Kelley et al. 2012; Ragan-Kelley et al. 2013; Ragan-Kelley et al. 2015] and the Halide open source community. We also added six new Halide benchmarks:

- Three computational photography pipelines (LENSBLUR, NLMEANS, and MAXFILTER) written and manually-scheduled by professional Halide developers.
- Dense matrix-matrix multiplication (MATMUL).
- Two deep neural network (DNN) workloads: CONV (a single convolutional layer) and VGG (a full implementation of forward pass of the VGG-16 object detection network).

BLUR 2 functions 6400×4800 compile time: 1 ms	The simple two-pass 3×3 image blur described in Figure 2.
UNSHARP 9 functions 7×7 filter $1536 \times 2560 \times 3$ compile time: 20 ms	Enhances local contrast by smoothing an image with a small support gaussian and subtracting it from the original to isolate the high-frequency content, which is then combined with the original image.
HARRIS 13 functions $1530 \times 2554 \times 3$ compile time: 21 ms	Implementation of the popular harris corner detection algorithm [Harris and Stephens 1988] which combines multiple stencils and point-wise operations.
CAMERA 30 functions $2560 \times 1920 \times 3$ compile time: 650 ms	The Frankencamera pipeline for processing raw data from an image sensor into a color image [Adams et al. 2010]. The pipeline performs hot-pixel suppression, demosaicing, color correction, gamma correction, and contrast.
NLMEANS 13 functions $192 \times 320 \times 3$ compile time: 49 ms	Fast non-local means image denoising using the method of Darbon et al. [2008]. Computes a 7x7 image blur with weights determined by 7×7 patch similarity.
MAXFILTER 9 functions filter radius 26 $1536 \times 2560 \times 3$ compile time: 15 ms	Computes the maximum-brightness pixel within a circular region around each target pixel. Uses a precomputed table of differently-sized vertical max filters to reduce complexity from $O(\text{radius}^2)$ per output pixel to $O(\text{radius})$.
INTERP 52 functions 10 pyramid levels $1536 \times 2560 \times 3$ compile time: 2.6 sec	Interpolation of image pixel values using an image pyramid for seamless compositing, based on the newest healing brush in Photoshop. Pyramid construction deals with image data at multiple resolutions and creates chains of stages with complex dependencies.
LOCAL_LAP 103 functions 8 pyramid levels $1536 \times 2560 \times 3$ compile time: 3.9 sec	A local Laplacian filter: an edge-aware, multi-scale approach for enhancing local contrast [Paris et al. 2011]. The pipeline builds multiple image pyramids with complex dependencies and performs data-dependent sampling.
LENSBLUR 74 functions $992 \times 1024 \times 3$ compile time: 55 sec	Given a rectified stereo pair of images, produces a synthetic shallow-depth-of-field image. It first solves for depth by constructing and filtering a cost volume [Rhemann et al. 2011] using a convolution pyramid [Farbman et al. 2011], then renders the synthetically defocused image by randomly sampling the source image over a virtual aperture.
BILATERAL 8 functions 1536×2560 compile time: 26 ms	Fast bilateral filter using the bilateral grid [Chen et al. 2007]. Constructs the grid using a histogram reduction, followed by stencil and sampling operations.
HIST_EQ 7 functions $1536 \times 2560 \times 3$ compile time: 2 ms	Normalizes the histogram of an image's luminance channel and performs back projection using the normalized histogram. Uses reductions to compute the histogram and point-wise operations to perform color-scale conversion and back projection.
CONV 4 functions $128 \times 128 \times 64 \times 4$ compile time: 9 ms	Typical convolutional layer in a deep neural network [Krizhevsky et al. 2012] (DNN). The layer evaluates a large filter bank at each spatial location of the input feature map followed by a rectified linear unit. Convolutional layers dominate the cost of evaluating and training DNNs.
VGG 64 functions $224 \times 224 \times 3 \times 4$ compile time: 6.9 sec	Evaluation of the VGG-16 object detection network [Simonyan and Zisserman 2014]. The network has 22 layers. 9 of the network's 13 convolutional layers operate on filter banks and input feature maps of different sizes.
MATMUL 2 functions 2048×2048 compile time: 1 ms	Dense matrix-matrix multiplication written as a straightforward reduction, as in Figure 3.

Table 1: *Fourteen Halide benchmarks spanning a range of computational photography, image processing, and computer vision workloads were used to evaluate the auto-scheduler. The number of functions per benchmark, the size of program inputs (concrete bounds), and auto-scheduler compilation times are given.*

The number of Halide functions in each of these pipelines, the size of pipeline inputs (given as concrete bounds to the auto-scheduler), and the compile time for all benchmarks is given in Table 1. In nearly all cases, the auto-scheduler generates a schedule within a few seconds (LENSBLUR's 55 second compile time is the only exception).

5.1 Server CPU Performance

We first analyzed the performance of auto-scheduled pipelines on a server-class, Intel Xeon E5-2620 v3 CPU (six 2.4 Ghz Haswell cores). For this machine, we set the auto-scheduler's parameters as follows:

- `VECTOR_WIDTH`: 16, twice the native vector width of AVX vector instructions for 32-bit data.
- `PARALLELISM_THRESHOLD`: 12, a small multiple of core count.
- `CACHE_SIZE`: 256 KB, the per-core L2 cache size.
- `LOAD_COST`: 10, a rough estimate of the relative cost of DRAM load vs. compute on modern multi-core machines.

We compare the performance of the auto-scheduler's output against that of several alternative scheduling approaches:

Baseline. An automatically generated schedule that employs the code analyses described in Section 4, but never groups functions or reorders loops: it only inlines functions (when inlining introduces no redundant computation), parallelizes each outermost loop, and vectorizes each innermost loop. This baseline is representative of the most complex schedule many new Halide programmers are able to write, or the first schedule a skilled Halide programmer designs when starting to optimize.

Manual. Hand-optimized schedules created by expert Halide developers. Hand-optimized CPU schedules for these pipelines match or significantly outperform the best manually-optimized C or assembly implementations generally available and, in some cases, proprietary commercial implementations (e.g., Photoshop's local Laplacian filters).

Auto-tune. Since prior work showed that stochastic auto-tuning systems struggled to converge quickly (or at all) on complex pipelines [Mullapudi et al. 2015], we implemented a simple, brute-force auto-tuning system that searched the low-dimensional space of auto-scheduler parameters, rather than the space of Halide schedules. The auto-tuner uses the auto-scheduler to generate schedules for each point in the parameter space, runs the resulting programs to measure real performance on the target machine, and picks the best performing schedule. The auto-tuner searched over 1152 total parameter configurations:

- `PARALLELISM_THRESHOLD={6,12,18,24}`
- `LOAD_COST={5,10,15,...,80}`
- `VECTOR_WIDTH={4,8,16}`
- `CACHE_SIZE={16,32,64,...,512}`.

This parameter sweep can take hours to days for our more complex benchmarks.

PolyMage. We approximate the scheduling behavior of PolyMage [Mullapudi et al. 2015] by restricting the auto-scheduler to tile only two spatial dimensions and to consider only a single tile size. Also following PolyMage, we then auto-tune over seven tile sizes in each spatial dimension and three `LOAD_COST` factors (10,20,40), then pick the best performing schedule. Functions with dependence patterns that cannot be analyzed by PolyMage are scheduled using the baseline scheduler.

Figure 6, which plots the performance of pipelines generated by each scheduling approach (relative to the throughput of the best schedule for each benchmark), shows that the auto-scheduled pipelines (orange bars) consistently deliver performance competitive with expert-tuned schedules (yellow bars). Absolute running times for all the bechmarks are shown in Table 2. Auto-scheduled performance is within 12% of, or better than, that of the manual schedules for eight of the 14 benchmarks. Auto-scheduled performance is never

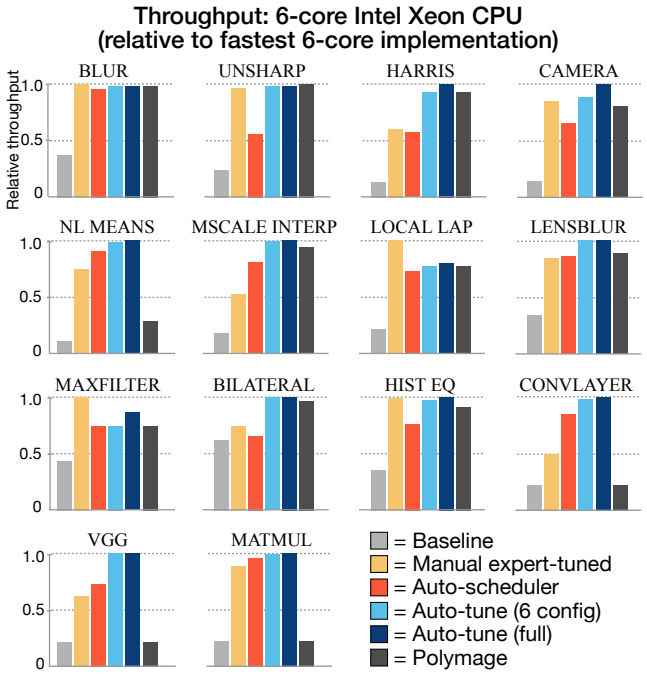

Figure 6: *Performance (in throughput: 1/sec) of auto-scheduled Halide programs relative to baseline, expert manually-optimized, auto-tuned, and PolyMage-like schedules. For each benchmark, performance is normalized to the fastest implementation.*

slower than a factor of two (UNSHARP). The reference schedule uses a much smaller tiling granularity hence fitting in L1 and achieving better locality, this performance gap is completely bridged when the auto-scheduler uses the L1 cache size to generate a schedule. Comparison of the resulting schedules indicates that the manually authored schedule for CAMERA benefits from loop unrolling decisions that simplify conditional logic in inner loops of the pipeline (it may be possible for the auto-scheduler to consider hoisting conditionals in the future) and that in the case of BILATERAL, the expert programmer chose to fuse the grid computation with sampling operation that uses the grid whereas the auto-scheduler did not.

In addition to comparing auto-scheduled output against other Halide schedules, we also compared the performance of auto-scheduled VGG pipeline to that of Caffe [Jia et al. 2014], a widely used DNN framework. (We configure Caffe to use the Intel Math Kernel library 11.2.4.) The auto-scheduled implementation of VGG outperforms Caffe/MKL by 1.5× on a 12-core, two-socket server with a Intel Xeon E5-2620 v3 CPU in each socket.

Comparison to auto-tuning. Although the auto-scheduler generates schedules significantly faster than the auto-tuning alternatives (seconds as opposed to minutes or hours) and avoids the need to run code on a target machine, the auto-tuning experiments provide a high watermark that provides insight into the quality of resulting schedules and the importance of various auto-scheduler parameters. Although auto-tuning based on actual performance measurement does find better schedules in many situations (light-blue bar, Figure 6), the auto-scheduler's generated code always remains within a factor of two for all benchmarks, and is within 25% of the best auto-tuned schedule in nine of 14. Analysis of auto-tuning results indicates that the auto-scheduler's output is largely invariant to `PARALLELISM_THRESHOLD` (provided sufficient parallelism exists), but can be sensitive to choice of `CACHE_SIZE` (some benchmarks are best tuned for L1, others L2) and `LOAD_COST`. We

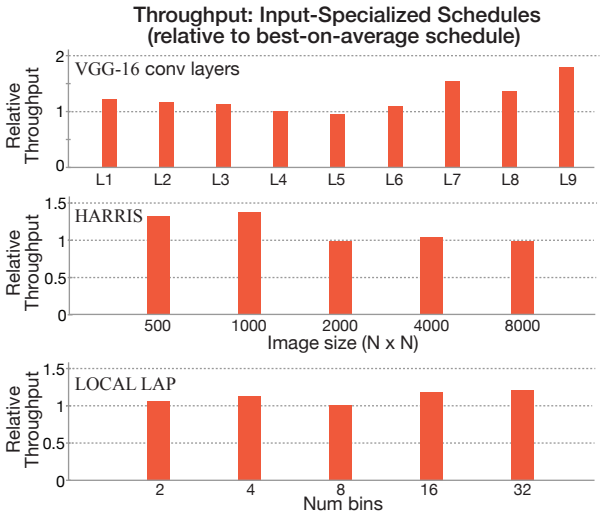

Figure 7: *Specializing schedules to specific problem sizes of individual VGG-16 network layers yields up to 1.8× benefit (compared to the schedule that performs best on average across all layers). Gains from image-size specialization (on* HARRIS*) and bin count specialization (*LOCAL_LAP*) are modest.*

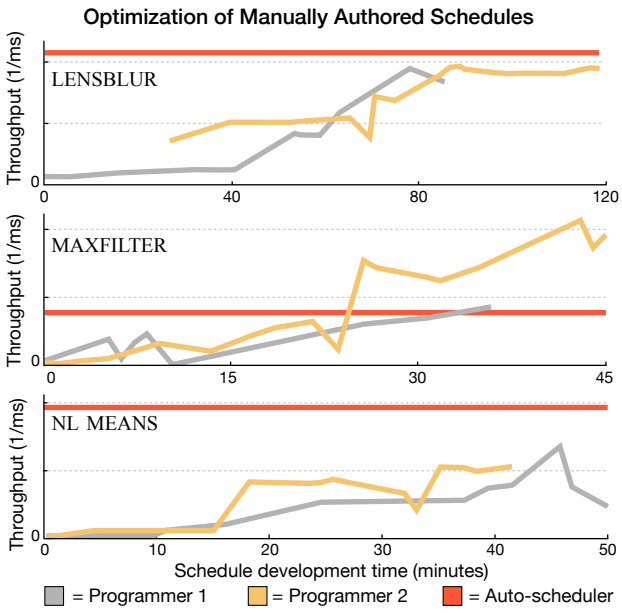

Figure 8: *Two professional Halide developers were tasked with developing schedules for new programs. In two of three cases, even after nearly an hour of work, the manually-authored schedules perform worse than auto-scheduled results (generated in seconds).*

find that a reduced auto-tuning search that varies only `CACHE_SIZE` (32,128,256 KB) and `LOAD_COST` (10,20) completes in under ten minutes for our benchmarks, but yields results comparable to the full brute search taking hours to days (dark blue bar in Figure 6). When auto-tuning to achieve greater performance is acceptable, this simple six-configuration auto-tuning scheme may serve as a practical and effective alternative to prior work utilizing advanced stochastic search techniques to explore the full space of Halide schedules.

5.2 Specializing Schedules to Problem Size

Automatic scheduling presents the opportunity to aggressively specialize schedules to specific dataset sizes. For example, nine of the 13 convolutional layers in the VGG-16 network operate on datasets of different size (the first convolutional layer's input is $224 \times 224 \times 3$, while the last layer's input is $14 \times 14 \times 512$). To understand the value of specializing schedules to different input configurations, we compared the performance of schedules generated specifically for each layer's input size, with the performance of a single fixed schedule. (The single fixed schedule was chosen to be the schedule that performed best on average when run on all layers.) Figure 7-top shows that several layers do benefit from schedule specialization. The maximum performance benefit of a specialized schedule was 1.8×. We observe the auto-scheduler makes different global optimization decisions (exploiting reuse of layer weights or input feature maps) based on problem size.

We further explored the sensitivity of schedule performance to problem size by performing similar experiments that varied image size in HARRIS (the single, fixed schedule was created for 2000×2000 images) and the number of bins in LOCAL_LAP (fixed scheduled assumes 8 bins). In both cases, modest benefits (but no more than 40% on HARRIS, 20% on LOCAL_LAP) are observed by providing the auto-scheduler accurate estimates of the problem size used at runtime. In general we find that while it is possible to gain additional performance by auto-scheduling for various input sizes, for many image processing pipelines the performance of an auto-scheduled pipeline generated for a reasonable estimate of average problem dimensions is robust across a range of sizes.

5.3 Comparison with Manual Scheduling Effort

The previous subsections demonstrated that the auto-scheduler produces schedules yielding performance on par with *the best known* manually-optimized schedules. To better understand how long it would take for an expert Halide developer to match the performance of the auto-scheduler when starting from scratch, we recruited two professional Halide developers to "race" the auto-scheduler. (These developers are authors on the paper.)

The experts selected three benchmarks (LENSBLUR, NLMEANS, and MAXFILTER) they had never scheduled before, and implemented the original Halide algorithm for these programs. For each benchmark, each expert programmer independently developed a schedule in a single programming session. The programmer stopped optimizing after converging on a solution they considered their best. While developing the schedules the developers documented their progress by measuring the performance of their current schedule at various points of time in each session. We then compared the auto-scheduled code's performance to that of the manually authored schedules.

Results of the comparison, are shown in Figure 8. The X-axis in each of the graphs indicates development time (in minutes) for the manually developed schedules. The Y-axis shows the performance of the benchmark (measured as pixel throughput, so higher is better). The horizontal line corresponds to the performance of the schedule generated by the auto-scheduler (produced in seconds). The yellow and gray lines each correspond to the progress of a programmer. The races were conducted using four cores of an Intel E5-2690 Xeon CPU owned by the developers, not the 6-core Xeon CPU used in our prior results.

On both the LENSBLUR and NLMEANS pipelines, the auto-scheduler outperforms the experts (by nearly a factor of two on NLMEANS). The experts outperform the auto-scheduler on MAXFILTER. (One of the experts found a solution nearly three times faster on their machine, but the performance difference between this schedule and the auto-scheduler's result is narrower on the 6-core machine used to

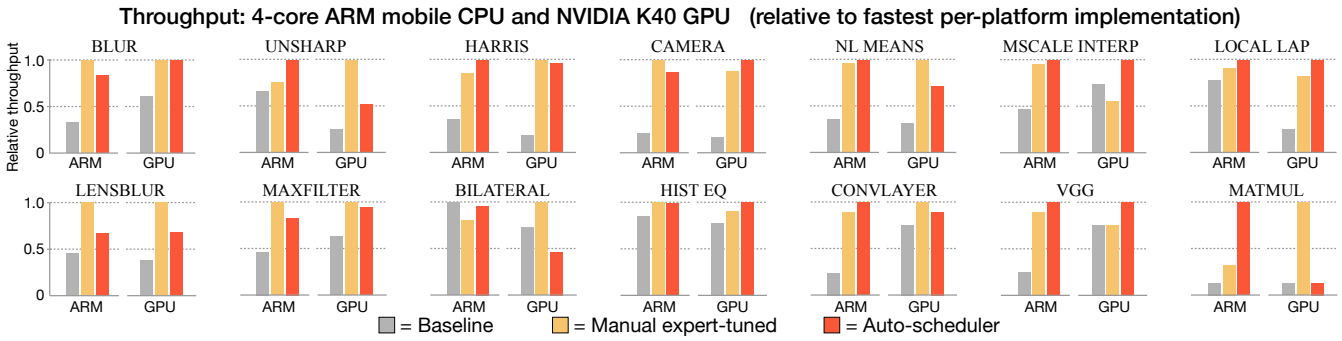

Figure 9: *Performance (in throughput: 1/sec) of auto-scheduled Halide programs relative to baseline and expert manually-optimized schedules for quad-core ARM mobile CPU and K40 GPU platforms.*

generate Figure 6). The expert schedule reduces memory footprint by fusing a function with high input data reuse with its consumer, and then choosing a tile size that simultaneously achieved both input reuse and producer-consumer locality. The auto-scheduler does not perform this fusion since the cost model deems exploiting reuse on both the functions individually to be the better choice.

As shown in Figure 8, arriving at a good schedule requires significant optimization effort, even for experts. Even in the case of MAXFILTER, where the experts devise schedules that outperform the auto-scheduler, they only reach this point after 25 minutes of optimization. In the other examples, the experts optimized for nearly an hour or two without matching the performance of the auto-scheduler.

5.4 Portability to Different Architectures

We also evaluated the ability of the auto-scheduler to target the quad-core ARM Cortex-A57 CPU (1.9 GHz, shared 2MB L2 cache) in the NVIDIA Tegra X1 and an NVIDIA K40 discrete GPU.

ARM CPU performance. To generate pipelines for the X1's ARM Cortex CPU, we made no changes to the auto-scheduler other than adjusting its parameters to match the processor as follows:

- `VECTOR_WIDTH`: 8, twice the native vector width of NEON vector instructions for 32-bit data.
- `PARALLELISM_THRESHOLD`: 8, a small multiple of core count.
- `CACHE_SIZE`: 512 KB, the per-core fraction of the L2 cache.
- `LOAD_COST`: 10, the same value as used for Xeon CPUs.

Figure 9 shows that the auto-scheduled pipelines for ARM execute faster than manually tuned schedules for all benchmarks but CAMERA and BLUR. (Manual schedules are the same schedules used in the prior Xeon CPU experiments.) Although we do not provide per-benchmark results in this paper, the benefits of auto-tuning schedules for ARM follows similar trends as those reported for Xeon.

GPU Performance. Figure 9 also provides results of using the auto-scheduler to generate schedules for a NVIDIA K40 GPU. We configure the auto-scheduler to target the GPU by setting the `PARALLELISM_THRESHOLD` to 128, `VECTOR_WIDTH` to 32, and `CACHE_SIZE` to 48 KB. Additionally, we add two new parameters `TARGET_THREADS_PER_BLOCK` and `MAX_THREADS_PER_BLOCK` whose values are set to 128 and 2048 respectively. These parameters enable the auto-scheduler to avoid tiling configurations that generate too few or too many threads per GPU thread block. The inlining, tiling, and grouping processes are otherwise similar to the CPU case. Groups resulting from merging are mapped to CUDA kernels by designating the outer tile loops as GPU block grid dimensions and the inner tile loops as GPU thread block dimensions. All intermediate buffers within a group are allocated in GPU shared memory.

We compared the performance of the auto-scheduled pipelines against that of a set of manually created GPU schedules. (We manually authored a new set schedules for the K40 GPU.) Auto-scheduled performance is nearly as good as, or better than, manually scheduled pipelines on eight of the 14 benchmarks. The faster manual schedules for UNSHARP and MATMUL reduce the total number of loads performed by unrolling inner loops in cases where multiple loop iterations reload the same values. This unrolling serves as a another form of tiling for locality, and we believe that a future auto-scheduler could be modified to perform such optimizations by treating registers as an additional level of the memory hierarchy.

6 Discussion

In this paper we have demonstrated the ability to automatically schedule Halide programs, obtaining results competitive with those obtained by the world's best Halide developers. We are excited that users of Halide may no longer require expert code-optimization knowledge in order to obtain the benefits of highly optimized code. We believe that by generating compiled solutions in seconds, avoiding the need for auto-tuning, and operating entirely within the Halide compiler, our solution is an attractive for use in production environments. We also believe our system will be valuable to expert Halide developers. In the process of evaluating our generated schedules professional developers remarked that the auto-scheduler was making decisions they had not thought of before.

Of course, this work constitutes only an initial attempt at efficiently scheduling Halide programs. Extending our approach to hierarchical levels of tiling and reasoning about the effects of data layout on vector code generation are all areas of immediate future system improvement. We predict these improvements would allow auto-scheduling to meet or surpass manually authored schedules in most of our benchmarks.

The design philosophy of Halide has always involved mixed responsibilities between the compiler and a human programmer to achieve efficient code. While our auto-scheduling algorithm makes it possible for the system to take on the full responsibility of schedule generation, a developer (particularly an expert) may still wish to intervene if necessary. We are interested in exploring interfaces for the auto-scheduler to accept partially written schedules by experts, and then fill in the missing details.

7 Acknowledgements

This research was supported by the National Science Foundation (IIS-1253530, IIS-1539069), the Intel Corporation, a Google Faculty Research Award, DARPA agreement FA8750-14-2-0009, the Stanford Pervasive Parallelism Lab (supported by Oracle, AMD, Intel, and NVIDIA), and by equipment donations from NVIDIA.

Benchmark	Xeon CPU 1 core		Xeon CPU 6 cores		Xeon CPU 12 cores		GPU K40		ARM Cortex A57 4 cores	
	Manual	Auto	Manual	Auto	Manual	Auto	Manual	Auto	Manual	Auto
BLUR	28.6	26.4	7.6	7.9	7.5	8.5	2.0	2.0	18.1	21.5
UNSHARP	29.2	37.3	6.6	11.4	6.8	7.4	0.9	1.8	57.8	43.8
HARRIS	43.3	36.5	9.7	10.3	5.9	6.7	1.1	1.1	51.1	44.0
CAMERA	52.0	41.3	7.8	10.1	6.1	6.0	2.3	2.0	26.8	31.0
NLMEANS	16.1	10.3	5.2	4.3	3.0	2.31	1.4	2.0	51.4	49.7
INTERP	58.2	70.5	28.3	18.2	46.5	15.6	25.6	14.2	114.6	110.0
LOCAL_LAP	331.4	361.2	77.8	106.7	60.6	120.1	11.5	9.6	734.3	670.3
LENSBLUR	1027.1	818.8	232.8	227.6	201.0	224.0	31.9	46.5	1260.2	1867.1
MAXFILTER	359.7	593.1	146.2	198.4	193.2	179.6	42.7	45.0	598.5	716.3
BILATERAL	66.0	70.8	14.1	15.9	10.5	11.4	2.0	4.3	173.5	145.1
HIST_EQ	22.3	28.1	8.8	11.4	4.8	7.0	5.5	5.0	85.6	86.2
CONV	259.9	212.6	82.0	48.3	44.9	33.7	35.8	39.7	395.3	356.6
VGG	31833.3	23111.4	7539.5	6449.4	5128.7	2945.8	4986.9	3756.8	43007.8	38501.8
MATMUL	2100.8	1363.5	317.6	293.6	214.8	193.5	20.3	152.4	5810.0	1891.5

Table 2: *Execution time (in ms) for the auto-scheduled (auto) and manually created (manual) schedules for all benchmarks. Auto-scheduling for the Xeon CPU was performed with the settings given in Section 5.1 (targeting a 6-core CPU). The 1- and 12-core (dual socket Xeon) results are provided to illustrate the multi-core scaling of this schedule.*

References

ADAMS, A., TALVALA, E.-V., 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., AND LEVOY, M. 2010. The frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics 29*, 4 (July), 29:1–29:12.

ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ACM, 303–316.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics 26*, 3 (July), 103:1–103:9.

DARBON, J., CUNHA, A., CHAN, T. F., OSHER, S., AND JENSEN, G. J. 2008. Fast nonlocal filtering applied to electron cryomicroscopy. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, IEEE, 1331–1334.

FARBMAN, Z., FATTAL, R., AND LISCHINSKI, D. 2011. Convolution pyramids. *ACM Transactions on Graphics 30*, 6 (Dec.), 175:1–175:8.

HARRIS, C., AND STEPHENS, M. 1988. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, 147–151.

HEGARTY, J., BRUNHAVER, J., DEVITO, Z., RAGAN-KELLEY, J., COHEN, N., BELL, S., VASILYEV, A., HOROWITZ, M., AND HANRAHAN, P. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics 33*, 4 (July), 144:1–144:11.

HEGARTY, J., DALY, R., DEVITO, Z., RAGAN-KELLEY, J., HOROWITZ, M., AND HANRAHAN, P. 2016. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics 36*, 4 (July).

JIA, Y., SHELHAMER, E., DONAHUE, J., KARAYEV, S., LONG, J., GIRSHICK, R., GUADARRAMA, S., AND DARRELL, T. 2014. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

MULLAPUDI, R. T., VASISTA, V., AND BONDHUGULA, U. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 429–443.

PARIS, S., HASINOFF, S. W., AND KAUTZ, J. 2011. Local Laplacian filters: edge-aware image processing with a Laplacian pyramid. *ACM Transactions on Graphics 30*, 4 (July), 68:1–68:12.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics 31*, 4 (July), 32:1–32:12.

RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. 2013. 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*, 519–530.

RAGAN-KELLEY, J., ADAMS, A., AND SHARLET, D. 2015. An introduction to Halide. In *ACM SIGGRAPH 2015 Courses*, ACM, 3:1–3:160.

RHEMANN, C., HOSNI, A., BLEYER, M., ROTHER, C., AND GELAUTZ, M. 2011. Fast cost-volume filtering for visual correspondence and beyond. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, 3017–3024.

SIMONYAN, K., AND ZISSERMAN, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.