Decoupling Algorithms from the Organization of Computation for High Performance Image Processing

by

Jonathan Ragan-Kelley

B.S., Stanford University (2004)
S.M., Massachusetts Institute of Technology (2007)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Jonathan Ragan-Kelley. Some Rights Reserved.

This work is licensed under the *Creative Commons Attribution-NonCommercial-NoDerivs 3.0* United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/us/

The author here by grants to MIT the permission to reproduce and to distribute publicly paper and electronic copies of the thesis document in whole or in part in any medium now known or hereafter created.

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2014

Certified by . . . .
Frédo Durand
Professor
Thesis Supervisor

Certified by . . . .
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

DECOUPLING ALGORITHMS *from the* ORGANIZATION *of* COMPUTATION *for* HIGH PERFORMANCE IMAGE PROCESSING

The design and implementation of the Halide language and compiler

Jonathan Ragan-Kelley

June 2014

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

to the

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

ABSTRACT

Future graphics and imaging applications—from self-driving cards, to 4D light field cameras, to pervasive sensing—demand orders of magnitude more computation than we currently have. This thesis argues that the efficiency and performance of an application are determined not only by the algorithm and the hardware architecture on which it runs, but critically also by the organization of computations and data on that architecture. Real graphics and imaging applications appear embarrassingly parallel, but have complex dependencies, and are limited by locality (the distance over which data has to move, e.g., from nearby caches or far away main memory) and synchronization. Increasingly, the cost of communication—both within a chip and over a network—dominates computation and power consumption, and limits the gains realized from shrinking transistors. Driven by these trends, writing high-performance image processing code is challenging because it requires global reorganization of computations and data, not simply the local optimization of an inner loop.

Traditional programming languages make it difficult for clear and composable code to express optimized organizations because they conflate the intrinsic algorithms being defined with their organization. To address the challenge of productively building efficient, high-performance programs, this thesis presents the Halide language and compiler for image processing. Halide explicitly separates what computations define an algorithm from the choices of execution structure which determine parallelism, locality, memory footprint, and synchronization. For image processing algorithms with the same complexity—even the exact same set of arithmetic operations and data—executing on the same hardware, the order and granularity of execution and placement of data can easily change performance by an order of magnitude because of locality and parallelism. I will show that, for data-parallel pipelines common in graphics, imaging, and other data-intensive applications, the organization of computations and data for a given algorithm is constrained by a fundamental tension between parallelism, locality, and redundant computation of shared values. I will present a systematic model of "schedules" which explicitly trade off these pressures by globally reorganizing the computations and data for an entire pipeline, and an optimizing compiler that synthesizes high performance implementations from a Halide algorithm and a schedule. The end result is much simpler programs, delivering performance often many times faster than the best prior hand-tuned C, assembly, and CUDA implementations, while scaling across radically different architectures, from ARM mobile processors to massively parallel GPUs.

ACKNOWLEDGMENTS

Halide represents the ongoing work of numerous people. Above all others, Andrew Adams is at least as responsible for Halide as I am. Andrew was my day to day collaborator from the inception of the project through its first two years, coauthored both the SIGGRAPH [74] and PLDI [75] papers on which I drew heavily in writing this thesis, wrote most of the tutorial material included as appendices to this thesis, and at this point is responsible for significantly more of the code in Halide's implementation than I. Now at Google, Andrew leads ongoing Halide development, where Zalman Stern is a also a prolific contributor. Other coauthors on those papers were also instrumental to the work presented in this thesis: Sylvain Paris, Connelly Barnes, and Marc Levoy, as well as Frédo Durand and Saman Amarasinghe. Connelly Barnes built the prototype autotuner used for all autotuned results presented here, using the Petabricks autotuner as a starting point [8]. Jason Ansel and I built the still-in-progress production version using his OpenTuner framework [9]. Sylvain Paris implemented the Snake pipeline in Halide, as well as the hand-optimized local Laplacian pipeline to which I compare. Jim McCann implemented both the C++ and Halide versions of the multi-scale interpolation pipeline. Eric Chan of Adobe wrote and optimized the version of local Laplacian filters in Camera Raw, and helped me prototype and test the Halide-based replacement for comparison. The energy and scaling data in Chapter 2 was compiled and distilled by John Brunhaver, Mark Horowitz, and Bill Dally. The Darkroom language and hardware compiler, referenced as related ongoing work in my conclusion, are led by James Hegarty and John Brunhaver.

I have been fortunate to spend most of the past decade working with and learning from Frédo Durand and Saman Amarasighne. Since I first considered coming to MIT and worried about being the only systems researcher in a graphics group far more connected to computer vision than compilers or computer architecture, Frédo has constantly impressed me with the ease, speed, clarity, depth of insight, and enthusiasm with which he attacks any problem that catches my interest, no matter how far afield it seems from his better-known areas of interest in applications and algorithms. And, since he first served on my SM committee, Saman has been the perfect complement. Thanks to their patience, wisdom, and inspiring example, I leave MIT an immeasurably better researcher, presenter, writer, and teacher than I even imagined possible when I arrived. In addition to Frédo and Saman, Charles Leiserson invested significant time helping me better present and distill the ideas in this thesis, and gave valuable feedback on multiple drafts as the third member of my committee, without which I would not have been able to produce a document of the quality you see here.

CONTENTS

INTRODUCTION

1

Image processing underlies, and is a dominant cost, in many of the most important applications of computation to the physical world. Image processing and computational photography algorithms require highly efficient implementations to be used in practice, especially on power-constrained mobile devices. Algorithms and computational hardware constantly evolve and improve, but as they do it becomes more and more difficult for researchers and developers to extract the required performance and efficiency from image processing code. This is not a simple matter of programming in a low-level language like C. The performance difference between naive C and highly optimized C is often an order of magnitude. Efficient implementations on modern hardware require complex global transformations of the computation and data structures, far beyond the inner loops. Unfortunately, this optimization usually comes at the cost of programmer pain and code complexity, as computation must be reorganized to efficiently exploit the memory hierarchy and parallel execution hardware.

Image processing pipelines[1] combine the challenges of stencil computation and stream programs. They are composed of large graphs of many different operations, most of which are stencil computations. They also contain non-stencil stages, including complex reductions, and stages with global or data-dependent access patterns. For example, an implementation of one recent algorithm, local Laplacian filters [67, 10], is a graph of approximately 100 different stages, including many different stencils and a large data-dependent resampling.

1. Throughout this thesis, and in much of the literature surrounding Halide, I refer to "image processing pipelines." By this I mean the largely feed-forward pixel processing common in low-level image processing for computational photography and computer vision. We do not mean only strict single-producer, single-consumer data flow, but rather general graphs of computation over arrays of pixels. In Halide, we even allow a limited form of *recursion* (cycles) within these graphs (discussed further in Chapter 3).

These pipelines are simultaneously wide and deep: each stage exhibits massive data parallelism across the many pixels it processes, and whole pipelines consist of long sequences of different operations, which individually have low arithmetic intensity (the ratio of computation performed to data read from prior stages and written to later stages). Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. The performance difference between a naive implementation of a pipeline and an optimized one is often an order of magnitude; unfortunately, the optimized code is also an order of magnitude more complex. Efficient implementations require optimization of both parallelism and locality, but due to the nature of stencils, there is a fundamental tension between parallelism, locality, and introducing redundant recomputation of shared values. These tradeoffs must be explored to find the ideal balance. The best choice of transformations is architecture-specific: implementations optimized for an x86 multicore and for a modern GPU often bear little resemblance to each other. Programmers are thus forced to choose between high performance and simple, modular, and portable code. Worse, production-quality code requires a proliferation of specialized versions of each core algorithm, each targeted to a different architecture or fused into a different pipeline[2]. Optimizing image processing pipelines under these competing pressures is challenging, time-consuming, and expensive.

2. For example, there are at least four different optimized implementations of the bilateral filter in the codebase of Adobe Photoshop [69].

I argue that the root of this challenge is that traditional programming languages conflate the definition of image processing algorithms, and their composition into larger pipelines, with the way their computations and data are organized on the underlying machine. This makes it hard to write algorithms, compose them into larger pipelines and applications, organize them for efficient execution on a given machine, or reorganize them to execute efficiently on different architectures.

I argue that the right way to program image processing pipelines is to decouple the definition of the algorithm from its organization on the underlying machine. Based on this philosophy, this thesis presents Halide, a new language which explicitly separates the definition of image processing algorithms from the concerns of their organization, and a compiler which synthesizes efficient code implementing an algorithm given an organization on a particular machine. The separation of concerns is a common goal in systems and programming languages, but Halide is unusual in promoting choices of organization to an orthogonal, first-class part of the language, directly controllable by the programmer, independently of their algorithm.

Halide enables much simpler programs to deliver performance often many times faster than the best prior hand-tuned C, assembly, and CUDA implementations of image processing pipelines. These programs are *portable* across radically different architectures, from ARM mobile processors to massively parallel GPUs, by making changes in the organization, where traditionally optimized implementations are highly specific to a single target architecture. They are also *modular and composable*, where traditional implementations have to fuse many operations into a monolithic whole for performance. And this is being proven in production use. We released Halide in 2012 and continue to develop it in collaboration with a growing open source community.[3] Over the past year and a half, dozens of engineers have written tens of thousands of lines of Halide code, shipping in millions of devices, including Google Glass and the latest Nexus phones, and running on tens of thousands of data center cores.

3. http://halide.io

1.1 THE STATE OF THE ART

To understand the challenge of efficient image processing, consider a 3×3 box filter implemented as separate horizontal and vertical passes. The first stage, *bh*, computes a horizontal blur of the input by averaging over a 3×1 window:

$$bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3$$

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

$$bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3$$

This example is much simpler than real image processing pipelines—in particular, it is much shorter, where real pipelines are often tens or hundreds of stages deep—but it is useful to make concrete the essential challenges. We might write this in C++ as a sequence of two loop nests:

```
void blur(const Image &in, Image &bv) {
 Image 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;
}
```

On a 3.5ghz quad core Intel Core i7-3770, this organization takes 47 ms/megapixel to process a 30 megapixel image.[4]

4. Compiled with Halide—a GCC 4.9-compiled version of this exact code is actually slower.

An efficient implementation on a modern CPU requires SIMD vectorization and multithreading. However, once we start to exploit parallelism, the algorithm becomes bottlenecked on memory bandwidth. Simply parallelizing and vectorizing this version improves performance by 7.5×, relative to a theoretical combined parallel speedup on this machine of 32×. 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.

Improving locality requires interleaving the two stages. For example, we can tile and fuse the loops. Tiles must be carefully sized for alignment, and efficient fusion requires subtleties like redundantly computing values on the overlapping boundaries of intermediate tiles. The resulting implementation is 2× faster still, but together these optimizations have fused two simple, independent steps into a single intertwined, non-portable mess:

```
void fast_blur(const Image &in, Image &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+(2*256)/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);
}}}}}
```

I argue that the complexity in optimizing even this simple two-stage blur algorithm is all about reorganizing its computations and intermediate data. To understand what I mean, let's take a more systematic look the ways this example can be organized.

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

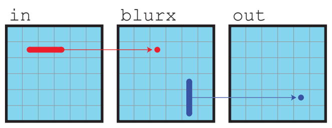


Figure 1.1: **breadth first** - each function is entirely evaluated before the next one.

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

```
**allocate** $bh[2048][3072]$
**for all** $y = 0$ **to** $2048$
    **for all** $x = 0$ **to** $3072$
        $bh[x, y] = in[x-1, y] + in[x, y] + in[x+1, y]$
**allocate** $bv[2046][3072]$
**for all** $y = 1$ **to** $2047$
    **for all** $x = 0$ **to** $3072$
        $bv[x, y] = bh[x, y-1] + bh[x, y] + bh[x, y+1]$
```

This is the most common strategy in hand-written pipelines, and what results from composing library routines together: each stage executes breadth-first across its input before passing its entire output to the next stage. There is abundant parallelism available, since all the required points in each stage can be computed and stored independently of one another, but there is little producer-consumer locality, since all the values of *bh* must be computed and stored before the first one is used by *bv*. As before, without parallelism, the organization takes 47 ms/megapixel on a modern desktop x86; fully parallelized and vectorized, it takes 6 ms/megapixel.

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

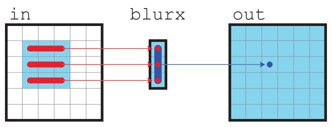


Figure 1.2: **total fusion** - values are computed on the fly each time that they are needed.

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

```
**allocate** $bv[2046][3072]$
**for all** $y = 1$ **to** $2047$
    **for all** $x = 0$ **to** $3072$
        **allocate** $bh[-1..1]$
        for all $i = -1$ to $1$
            $bh[i] = in[x-1, y-1+i] + in[x, y-1+i] + in[x+1, y-1+i]$
        $bv[x, y] = bh[0] + bh[1] + bh[2]$
```

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

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

```
allocate $bv[2046][3072]$
allocate $bh[3][3072]$
for all $y = -1$ to $2047$
    for all $x = 0$ to $3072$
        $bh[x, (y+1) \bmod 3] = in[x-1, y+1]$
                                $+ in[x, y+1]$
                                $+ in[x+1, y+1]$

        if $y < 1$: continue
        $bv[x, y] = bh[x, (y-1) \bmod 3]$
                    $+ bh[x, y \bmod 3]$
                    $+ bh[x, (y+1) \bmod 3]$
```

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

Each of these strategies has a major pitfall: lost locality, redundant work, or limited parallelism (Figure 1.4). In practice, the right choice for a given pipeline is almost always somewhere in between these extremes. For our two-stage example,

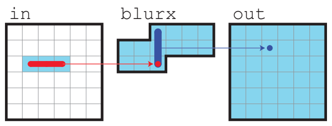


Figure 1.3: **sliding window** - intermediate values are computed immediately before their first use, and freed immediately after their last use. For each new intermediate value produced, one is freed and three are consumed to produce one output value. The computation of new intermediate and output values is tightly coupled in a sliding window over the image.

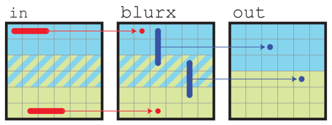


Figure 1.5: **tiles** - overlapping regions are processed in parallel, functions are evaluated one after another.

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

Figure 1.4: Different points in the choice space in Figures 1.1-1.6 each make different trade-offs between locality, redundant recomputation, and parallelism. Here we quantify these effects for our two-stage blur pipeline. The *span* measures the constraints on parallel execution, by counting the sequential critical path assuming infinite parallel processors. The *Max. reuse distance* measures locality, by counting the maximum number of operations that can occur between computing a value and reading it back. *Work amplification* measures redundant work, by comparing the number of arithmetic operations done to the breadth-first case. Each of the first three strategies represent an extreme point of the choice space, and is weak in one regard. The fastest schedules are mixed strategies, such as the tiled ones in the last two rows.

a better balance can be struck by interleaving the computation of *bh* and *bv* at the level of *tiles*:

```
allocate $bv[2046][3072]$
for all $ty = 0$ to $\frac{2048}{32}$
    for all $tx = 0$ to $\frac{3072}{32}$
        allocate $bh[-1..33][32]$
        for $y = -1$ to $33$
            for $x = 0$ to $32$
                $bh[x, y] = in[tx \times 32 + x - 1, ty \times 32 + y]$
                          $+in[tx \times 32 + x, ty \times 32 + y]$
                          $+in[tx \times 32 + x + 1, ty \times 32 + y]$
        for $y = 0$ to $32$
            for $x = 0$ to $32$
                $bv[tx \times 32 + x, ty \times 32 + y] = bh[x, y - 1]$
                                                $+bh[x, y]$
                                                $+bh[x, y + 1]$
```

This trades off a small amount of redundant computation on tile boundaries for much greater producer-consumer locality, while still leaving parallelism unconstrained both within and across tiles. (In the iterated stencil computation literature, the redundant regions are often called "ghost zones," and this strategy is sometimes called "overlapped tiling" [58, 91].) On a modern x86, this strategy is almost exactly 2×

faster than the breadth-first strategy (3 ms/megapixel), using the same amount of multithreaded and vector parallelism. This is because the lack of producer-consumer locality leaves the breadth-first version limited by bandwidth; the tiled version uses half the memory bandwidth of the breadth-first version on this two-stage pipeline. This difference grows as the pipeline gets longer, increasing the ratio of intermediate data to inputs and outputs, and it will only grow further as the computational resources scale exponentially faster than external memory bandwidth under Moore's Law (Cf. Chapter 2).

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

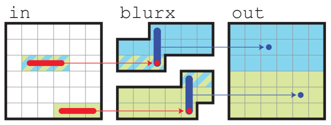


Figure 1.6: **sliding window within tiles** - tiles are evaluated in parallel, using sliding windows internally.

```
**allocate** $bv[2046][3072]$
**for all** $ty = 0$ **to** $\frac{2048}{8}$
    **allocate** $bh[-1..1][3072]$
    **for** $y = -2$ **to** $8$
        **for** $x = 0$ **to** $3072$
            $bh[x, (y+1) \bmod 3] = in[tx \times 32 + x - 1, ty \times 8 + y + 1]$
                                   $+in[tx \times 32 + x, ty \times 8 + y + 1]$
                                   $+in[tx \times 32 + x + 1, ty \times 8 + y + 1]$
        if $y < 0$: continue
        **for** $x = 0$ **to** $3072$
            $bv[x, ty \times 8 + y] = bh[x, (y-1) \bmod 3]$
                               $+bh[x, y \bmod 3]$
                               $+bh[x, (y+1) \bmod 3]$
```

Relative to the original sliding window strategy, this sacrifices two scanlines of redundant work on the overlapping tops and bottoms of independently-processed strips of *bh* to instead reclaim fine-grained parallelism within each scanline and coarse-grained parallelism across scanline strips. The end result is 10% faster still than the tiled strategy on one benchmark machine, but 10% slower on another. The best choice between these and many other strategies varies across different target architectures. The ideal balance depends on the computational characteristics of the stages, and the architecture of the target machine.

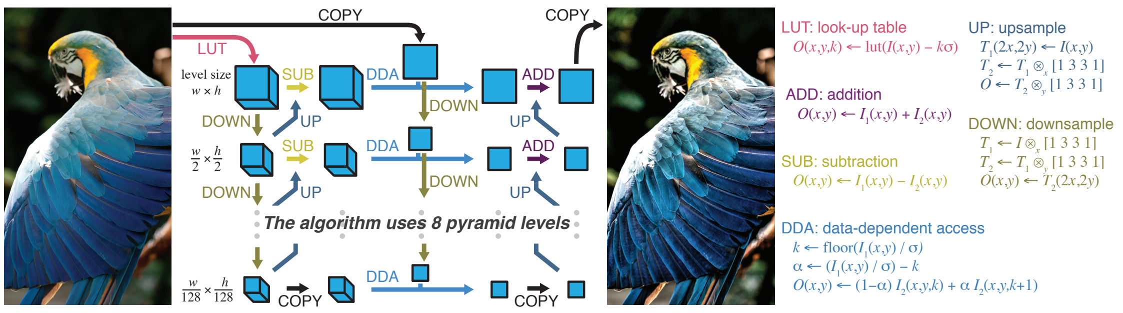

COPY
COPY
LUT
level size $w \times h$
SUB
DDA
ADD
DOWN
UP
DOWN
UP
$\frac{w}{2} \times \frac{h}{2}$
SUB
DDA
ADD
DOWN
UP
DOWN
UP
The algorithm uses 8 pyramid levels
$\frac{w}{128} \times \frac{h}{128}$
COPY
DDA
COPY
LUT: look-up table
$O(x,y,k) \leftarrow \mathrm{lut}(I(x,y) - k\sigma)$
ADD: addition
$O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$
SUB: subtraction
$O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$
UP: upsample
$T_1(2x,2y) \leftarrow I(x,y)$
$T_2 \leftarrow T_1 \otimes_x [1\ 3\ 3\ 1]$
$O \leftarrow T_2 \otimes_y [1\ 3\ 3\ 1]$
DOWN: downsample
$T_1 \leftarrow I \otimes_x [1\ 3\ 3\ 1]$
$T_2 \leftarrow T_1 \otimes_y [1\ 3\ 3\ 1]$
$O(x,y) \leftarrow T_2(2x,2y)$
DDA: data-dependent access
$k \leftarrow \mathrm{floor}(I_1(x,y) / \sigma)$
$\alpha \leftarrow (I_1(x,y) / \sigma) - k$
$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

Figure 1.7: Local Laplacian filters.

Real pipelines are both wide and deep.

The blur algorithm explored so far is highly simplified for clarity; with only two stages, it is *wide* (each stage operates over many pixels), but not *deep*. Real image processing pipelines are *both* wide *and* deep: they have tens to hundreds of stages, connected in a large graph of dependencies. Consider the fast local Laplacian filters algorithm [10]. The algorithm can be decomposed into roughly 100 stages connected in a complex graph (Figure 1.7). A hand-optimized version used in Photoshop is thousands of lines long, and represents only a single step in the much larger Camera Raw pipeline. Even a clean, reference C++ version is over 300 lines of code. As a result of this scale, the overall space of choices for organizing the interaction among all the stages in real algorithms like this is enormous. The version in the Camera Raw pipeline delivers 10× the performance of the reference implementation. This performance difference comes primarily from *reorganizing* the computation, including manually multithreading and hand-coding for SSE. In this case, the production version represents about three months of implementation and optimization effort, but in that time the developer could explore only a few different organizations.

There are global consequences to the decisions made for each stage in a larger pipeline, so the ideal choice of organization depends on the composition of stages, not just each individual stage in isolation. The most critical choices in the organization, both for locality and for the granularity and coherence of parallelism, relate to the *interaction* between stages. This is also why libraries of optimized code cannot deliver efficient performance when building real image processing pipelines: individually optimized subroutines *do not compose* into an optimized whole, since they cannot reorganize computation for locality or parallel execution *across* function boundaries.

1.2 THE HALIDE SOLUTION

I believe the right way to program image processing pipelines is to separate the *intrinsic algorithm*—*what* is computed—from the concerns of efficiently organizing it for machine execution—decisions about *storage* and the *ordering* of computation. This is the core design philosophy of the Halide language. Concretely, in the blur example, the intrinsic algorithm specifies the arithmetic definition of *bh* and *bv* at each pixel; all the choices we explored in the previous section do not change this algorithm, only the order in which these pixels are computed and where their intermediate values are stored. I call these choices of how to map an algorithm onto resources in space and time a *schedule*. In the Halide approach, the programmer specifies an algorithm and its schedule separately. This makes it easy to explore various optimization strategies without obfuscating the code or accidentally modifying the algorithm itself.

Halide makes this space of schedule choices a first-class part of the language, directly expressible in code. For example, the complete optimized blur algorithm and organization is specified by the code:

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xo, yo, 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, xo, yo, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(yo);
 bh.compute_at(bv, xo).vectorize(x, 8);

 return bv;
}
```

The schedule precisely defines a global loop nest over the required regions of all functions in the pipeline, which is synthesized by the compiler.[5] This generates nearly identical machine code, and identical performance, to the hand-optimized C++.

In practice, this design has proven surprisingly powerful. For example, where Adobe's implementation of local Laplacian filters took three months and thousands of lines of hand-tuned, architecture-specific code to implement and optimize, we built and optimized a Halide equivalent in less than two days and 60 lines of code which ran twice as fast on the same machine. The added performance did not come from any compiler trick; rather, because of the ease with which we could explore different organizations, within a few hours we found organizations which the original developer did not have time to try in three months. Then, with a few more changes to the schedule, we generated a GPU version several times faster, still, where the original implementation would have required months more to rewrite in CUDA or OpenCL. Since then, Halide is now in regular use by dozens of engineers at several companies. Most visible is Google, where several dozen developers in the mobile imaging and photo sharing groups have shipped over ten thousand lines of Halide code on Glass, Android phones, and data centers.

5. Halide's schedules parameterize the space of *semiperfect loop nests*, a concept I define below.

A language of image processing algorithms

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

Concretely, Halide models image processing algorithms as a rooted directed acyclic graph (DAG) of image functions. Most image functions are pure functions of their arguments (coordinates in their domain), other functions, input images, and scalar parameters. Edges in the graph correspond to caller-callee dependencies that pass pixel data between image functions. The root of the graph is the output of the pipeline.

For example, Halide would model the two-stage box filter algorithm with a simple two-function pipeline, exactly as we sketched it before:

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

That is, as a graph of $2D$ functions, *bh* is defined at any point x, y as the sum of three points in the input, while the output, *bv*, is defined at any point x, y as the sum of three points in bh.

Functions may call other functions at arbitrary, dynamically computed coordinates—accesses need not be constant, affine, or otherwise constrained[6]. Functions may also be iteratively re-defined at potentially dynamic locations in their domain via a series of recursive *update* definitions, which are layered on top of the initial definition. This allows efficient expression of operations not easily modeled as statically-unrolled chains of pure gather stages—things like histograms, reductions, and scans—but this model of computation is still intentionally restricted. In particular, recursion is only allowed within a single function, using update definitions, and the recursion is bounded to a fixed depth before it begins by an explicit *reduction domain*. As a result, Halide's language of algorithms is not Turing-complete, but is amenable to extensive analysis and transformation.

6. However, they are easier to analyze when they are.

A language of schedules

In the Halide representation, the algorithm only defines the value of each function at any point; a schedule specifies specifies the organization of computation over all points in all functions. Halide's schedules specify the organization of computation as a loop nest over the required regions of all functions in an algorithm. They specify the storage and communication of intermediate data as an allocation granularity within this loop nest for each function's results, and the linearized mapping of each function's domain into addresses within its allocation. Halide represents schedules as a set of four choices for each function in the algorithm:

- The **domain order**, or order in which points in the domain of the function are evaluated, including row- vs. column-major orders, tiling, and the exploitation of parallelism and mapping onto SIMD execution units.
- The **storage order**, or layout of the buffer into which the eval-

uation of a function is stored.

- The **computation granularity** at which points in the domain of one function are evaluated relative to points in the domain of the others that depend on it, which interleaves their execution.
- The **storage granularity** of intermediate storage for function results, which dictates whether a value is recomputed, or from where in the memory hierarchy it is loaded, at each point a function is used.

For example, the naive row-major, breadth-first organization of the simple blur algorithm is specified by the following choices:

```
bh:                                   bv:
    domain order:                         domain order:
        $y \to x$                             $y \to x$
        $x$ : parallel                        $x$ : parallel
        $y$ : parallel                        $y$ : parallel
    storage order: $y \to x$              storage order: $y \to x$
    compute at: root granularity          compute at: root granularity
    store at: root granularity            store at: root granularity
```

which describes the organization:

```
allocate bh
parallel for bh.y
    parallel for bh.x
        compute $bh(bh.x, bh.y)$ using $in$
allocate bv
parallel for bv.y
    parallel for bv.x
        compute $bv(bv.x, bv.y)$ using $bh$
```

Once these order and granularity choices are fixed, the size of the regions computed for each function, and the dependence order between producers and consumers, are dictated by and inferred from the algorithm[7]. Required regions are modeled conservatively as simple multidimensional intervals (axis-aligned bounding boxes of the dimensionality of the function). With region definitions symbolically expanded, the simple blur organization becomes:

7. In the blur examples, the fact that *bh* must be computed before *bv*, and the minimum required regions of *bh* needed by *bv* at any given granularity of interleaving, are implied by the definition of the algorithm $in \to bh \to bv$. These are not exposed as choices, and are not specified by the schedule.

```
// the required region of bv, given as min and max in x and y,
// is a parameter to the pipeline
```

```
$bh.y.min = bv.y.min - 1$
$bh.y.max = bv.y.max + 1$
$bh.y.extent = bh.y.max - bh.y.min$
$bh.x.min = bv.x.min$
$bh.x.max = bv.x.max$
$bh.x.extent = bh.x.max - bh.x.min$
**allocate** $bh[bh.y.extent][bh.x.extent]$
**parallel for** $bh.y = bh.y.min$ **to** $bh.y.max$
    **parallel for** $bh.x = bh.x.min$ **to** $bh.x.max$
        compute $bh(bh.x, bh.y)$ using $in([bh.x-1, bh.x+1], bh.y)$
**allocate** $bv[bv.y.extent][bv.x.extent]$
**parallel for** $bv.y = bv.y.min$ **to** $bv.y.max$
    **parallel for** $bv.x = bv.x.min$ **to** $bv.x.max$
        compute $bv(bv.x, bv.y)$ using $bh(bv.x, [bv.y-1, bv.y+1])$
```

Halide schedules describe semiperfect loop nests.

Conceptually, each function is computed by a *perfectly-nested loop*[41] which scans points in the required region of its domain. Because required regions are axis-aligned bounding boxes, the bounds of different dimensions of a function's domain cannot be interdependent; the bounds of the whole domain must be fixed before entering its loop nest. However, the overall organization is a *sequence* of loop nests which computes the required regions of all functions. In general, the granularity at which a function is evaluated—its *computation granularity*—is specified as a level in the loop nest evaluating the domains of its callers. As a result, the loop nest for each function may be inserted at any level of the loop nests of its caller functions, creating an imperfect loop nest over all regions of all functions. The end result is that Halide's schedules are restricted to a subset of all possible orders of execution over the required regions of each function which we call *semiperfect loop nests*.

Formally, I define a *semiperfect loop nest* as a sequence of statements where:

- The collection of loops and statements which compute the domain of any single function form a perfectly-nested loop with respect to each other[8]. That is, each block is either a sequence of non-loop statements with a single entry and single exit, or one perfectly-nested loop containing exactly one block as its body.

8. Ignoring statements and loops which compute other functions.

- The loop nests for each function are interleaved at any granularity and in any order which respects producer-consumer dependence between functions.
- The allocation of the buffer into which a function's results are stored are at a granularity which encloses where the function's loop nest is interleaved.

A tiled and interleaved organization of the simple blur algorithm, like that used in the hand-optimized C++ implementation shown earlier, is specified by the schedule:

```
bh:
    domain order:
        $y \to x$
    compute at: bv.xo
    store at: bv.xo
```

```
bv:
    domain order:
        split $y \to y_o, y_i$ by 8
        split $x \to x_o, x_i$ by 8
        $y_o \to x_o \to y_i \to x_i$
        $y_o$ : parallel
```

The domain orders separately describe a local perfect-loop nest over the required region of each function:

```
for $bh.y$
    for $bh.x$
        compute $bh(bh.x, bh.y)$ using $in$
```

and

```
parallel for $bv.y_o$
    for $bv.x_o$
        for $bv.y_i$
            for $bv.x_i$
                compute $bv(bv.x_o \times 8 + bv.x_i, bv.y_o \times 8 + bv.y_i)$ using $bh$
```

Then, the *compute* and *storage granularities* chosen for *bh* define the levels in the loop nest of *bv* at which the computation and storage of the first function should interleaved, giving the semiperfect loop nest which computes the entire pipeline:

```
allocate $bv$
parallel for $bv.y_o$
    for $bv.x_o$
        allocate $bh$
        for $bh.y$
            for $bh.x$
                compute $bh(bh.x, bh.y)$ using $in$
```

```
for $bv.y_i$
    for $bv.x_i$
        compute $bv(bv.x_o \times 8 + bv.x_i, bv.y_o \times 8 + bv.y_i)$ using
        $bh$
```

Halide's schedules only model a subset of all possible organizations.

Halide's schedules only model part of the space of all possible organizations of computation and storage for pipelines expressible as Halide algorithms. In particular, they only express some possible interleavings of the computations in an algorithm. Computations can only be interleaved as allowed by semiperfect loop nests, expressible through the space of meaningful domain order and computation granularity choices for each function in an algorithm. Computation cannot be interleaved at finer granularity than the evaluation of a function at a point in its domain. Finally, the required region of each function is tracked conservatively as a multidimensional interval, which may include points not actually required to compute the output of the pipeline.

These constraints restrict the space of possible organizations representable in Halide, but they provide a model which is *simple*, from which it is easy to generate fast code, whose meaning and performance is reasonably easy to understand, which can express most commonly-used patterns in optimized image processing code, and which naturally maps to GPU and SIMD multicore programming models.

Code generation from scheduled algorithms

Once the programmer has specified an algorithm and a schedule, our compiler combines them into an efficient implementation. Optimizing execution for a given architecture requires modifying the schedule, but not the algorithm. The representation of the schedule is compact and does not affect the correctness of the algorithm, so exploring the performance of many options is fast and easy. The schedule can be written separately from the algorithm, by an architecture expert if necessary. Halide can most flexibly schedule operations which are data parallel, with statically analyzable access patterns, while still allowing the reductions, scans, and bounded irregular access patterns that occur in image processing.

1.3 DISSERTATION OVERVIEW

The rest of this thesis motivates, explains, and evaluates the design and implementation of the Halide language as follows:

- Chapter 2 provides broader context for the problem of organizing computation for efficiency in image processing pipelines.
- Chapter 3 describes Halide's language for functional algorithm specification.
- Chapter 4 explores the problem of organizing computation in image processing pipelines.
- Chapter 5 describes Halide's model for the organization of computation, and its embodiment in the language of *schedules*.
- Chapter 6 demonstrates the application of Halide schedules to describe and explore the space of optimized organizations in real image processing pipelines, and discusses methods for finding efficient schedules for Halide algorithms.
- Chapter 7 presents an algebraic view of Halide's schedule space.
- Chapter 8 explains how the Halide compiler translates functional algorithms and optimized schedules into efficient machine code for x86 and ARM, including SSE and NEON SIMD instructions, and CUDA and OpenCL GPUs, including synchronization and placement of data throughout the specialized memory hierarchy.
- Chapter 9 evaluates Halide implementations of a range of applications composed of common image processing operations such as convolutions, histograms, image pyramids, and complex stencils. Using different schedules, we compile them into optimized programs for x86 and ARM CPUs, and a CUDA GPU. For these applications, the Halide code is compact, and performance is state of the art. This chapter additionally evaluates an autotuner for automatically scheduling programs using stochastic search, and discusses several cases of the adoption and deployment of Halide in the real world.
- Chapter 10 puts Halide in the context of related work in programming languages, compilers, and image processing.
- Chapter 11 discusses and analyzes our experience with Halide, and suggests directions for future work.

- A deeper, example-driven introduction to using the Halide language is provided as an appendix.

2 THE OPPORTUNITY AND CHALLENGE OF IMAGE PROCESSING

Image processing matters for much more than just photography: from self-driving cars, to high throughput gene sequencing, to neural scanning and connectomics, today most complex sensing uses imaging under the hood [79, 86, 5]. Image sensors offer a simple, high resolution building block for digitizing the physical world, including both spatial and spectral information, while image processing and computer vision let us analyze and comprehend the captured data in numerous ways. As a result, imaging is used for everything from localization and tracking to human-computer interaction to vital sign monitoring, traffic analysis, mapping, bar code scanning, and industrial defect detection [46, 80, 59, 55, 78].

The resulting explosion of sensors offers an enormous opportunity to build graphics and imaging applications orders of magnitude richer than any we have today; however, doing so will require exponentially more computation. For example, replacing video with real-time 4D light fields, imaging the connectome of a complete human brain, and building machines which pervasively understand the visual world all require orders of magnitude more computational power than we have today. Orders of magnitude more energy efficiency will let us leverage cheap, high data rate cameras for everything from human-computer interaction to search, and move powerful image processing and analysis into our glasses and clothing where it will transform how we see, think, remember, and are entertained. Always-on cameras will passively monitor our vital signs and health every moment of our lives. Even existing applications—from Instagram and Photoshop, to object detection and recognition, to Microsoft Kinect, to personal genome sequencing—all demand extremely high performance to cope with the rapidly rising resolution, frame rate, and sheer number of image sensors, as well as the increasing complexity of imaging and vision algorithms. At the same time, the cameras, mobile devices, and data centers on which these applications run require extremely high efficiency to stay within thermal limits, or to last more than a few minutes on battery power.

Unlike other domains of low-power, high-throughput computation, the answer cannot simply be traditional hardware specialization: while radio basebands and video CODECs implement slow-changing standards which can be built into custom hardware, image processing algorithms are rapidly evolving and diverse, requiring high performance *software* implementations. Neither can we simply ship pixels to the cloud for processing: it takes a cellular radio four orders of magnitude more energy to transmit each pixel than a sensor spends to capture it (Figure 2.2).

Future imaging systems need large capacity for software-controlled image processing, as close to the image sensor as possible. The good news is that the transistor scaling trends we have come to call "Moore's Law" are alive and well: we can expect more than an order of magnitude increase in density over the next decade (Figure 2.1); the challenge will be in keeping tens of billions of transistors busy.

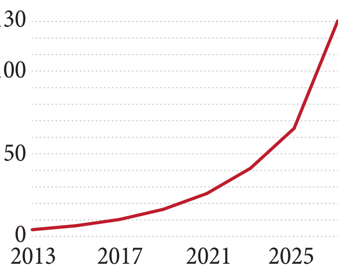


Figure 2.1: Projected ASIC logic density (millions of 4-input NAND gates per mm^2) (Source: ITRS Report, 2013 [1]).

Performance requires complex tradeoffs between parallelism, locality, and the total amount of computation.

One of the major challenges is exposing more and more parallelism for future hardware to exploit [81]. Superficially, this seems easy: imaging algorithms are enormously data-parallel, which should make them easy to scale on highly parallel hardware. The real challenge is that their millions of data-parallel computations are not independent—they need to be able to communicate with each other, and to share data over time through memory. Real imaging and vision algorithms have complex dependencies, and are limited by locality (the distance over which data has to move, e.g., from nearby caches or far away main memory) and synchronization. The same hardware trends which have pushed us from uniprocessors to lots of parallel cores have made communication and data movement—

ALU op	1.0 pJ–4.0 pJ	1×
Move 10mm across chip (LVDS)	2.6 pJ–10 pJ	2.5×
Read from SRAM	5 pJ–20 pJ	5×
Move 10mm across chip (CMOS)	26 pJ–44 pJ	25×
Send off chip	200 pJ–800 pJ	200×
Send to DRAM	200 pJ–800 pJ	200×
Read from image sensor (4x8b)	3.2 nJ–4 nJ	4,000×
Send over LTE	50 uJ–600 uJ	50,000,000×

Figure 2.2: Energy cost of different operations on 32-bit values in a leading 45nm foundry process . Communication and storage are significantly more expensive than computation, and their cost is proportional to the distance data is moved.

both within a chip and over a network—dominate the cost of computation, and limit the gains realized from shrinking transistors. Today, relative to the energy cost of doing some arithmetic operation on a piece of data, loading or storing that data in a small local SRAM like a cache can be several times more expensive; moving the result 10 millimeters across the chip is an order of magnitude more expensive; and moving it to or from off-chip RAM is three to four orders of magnitude more expensive than computing the value in the first place (Figure 2.2). This disparity is only growing over time.

Because of the inversion in the cost of communication and computation, it can often be most efficient to make surprising tradeoffs, like redundantly recomputing values used in multiple places instead of storing and reloading them from memory. In this way, there is a tension between locality and the total amount of computation performed: it is possible to improve locality by redundantly recomputing values. Further, we find that this fundamental tension exists between all three factors we want to minimize for performance and efficiency:

- order constraints, which limit parallelism;
- data movement, which reduces locality;
- and the total amount of work performed.

We can sacrifice parallelism to tightly couple producers and consumers; we can save and reload values from far away to avoid redundant work; or we can recompute values near independent consumers to avoid moving them long distances. The only thing we can not do in the presence of nontrivial dependencies among operations is simultaneously minimize all three.

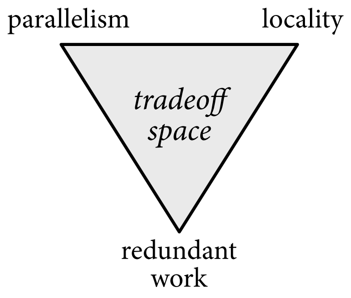


Figure 2.3: Parallelism, locality, and redundant work are often in tension, and must be traded off with each other to maximize the overall efficiency of an algorithm on a particular architecture.

Optimization is challenging because these are not discrete tradeoffs. The best strategies balance them all in different ways in different parts and at different granularities of an overall system. The ideal balance depends on the interaction between the individual algorithms, the hardware architecture onto which they are being mapped, and the larger pipeline into which they are composed. The best tradeoffs for a given pipeline on a given architecture are rarely obvious, and finding them often requires extensive experimentation.

Tradeoffs between parallelism, locality, and redundant computation are determined by the organization of computation.

I argue that, driven by these tradeoffs, the efficiency and performance of an application are determined not just by the algorithm and the hardware architecture on which it runs, but critically also by the organization of the computations and data on that hardware. For algorithms with the same complexity—even the exact same set of arithmetic operations and data—executing on the same hardware, the order and granularity of execution and placement of data can easily change performance by an order of magnitude because of locality and parallelism.

Consider a simple loop nest which computes a 2D grid of values. Because of locality, switching from column-major to row-major execution order can change performance by an order of magnitude:

```
for $x = 0$ to $width$ do                 for $y = 0$ to $height$ do
    for $y = 0$ to $height$ do                for $x = 0$ to $width$ do
        $f[x, y] = \ldots$                        $f[x, y] = \ldots$
    end for                                   end for
end for                          $\Rightarrow$    end for
```

I argue that these are, in fact, the *same algorithm*, reorganized to compute its values and access memory in a different order.

This simple reorganization is well known, and something many compilers will do today. In this thesis, I will model and exploit a much wider range of organizations. Especially in multi-stage algorithms like image processing pipelines, the difference in efficiency between different organizations comes not from the optimization of individual stages in isolation, but from the global interleaving of computations and data. For example, computing each stage completely before the next—even with the optimal inner loop, spread over thousands of threads on a GPU—destroys producer-consumer locality, repeatedly pushing intermediate data to and from main memory.

Exploring the space of potential optimizations is challenging because making different tradeoffs requires globally reorganizing the computations and data throughout a pipeline. In a traditional language, this means rewriting all of the code. Even once you find an efficient organization, you can rarely reuse it in a new context: because locality requires interleaving operations and data across stages, libraries of even the best optimized subroutines do not compose into efficient pipelines; and the best organization of a given pipeline varies enormously across common architectures, from mobile multicores to server GPUs. This challenge is combinatorial: the complexity scales with the combination of individual algorithms, pipelines, and target architectures. As a result, efficient image processing code is hard to write, modify, port across architectures, or compose into new pipelines. Worst of all, the sacrifices programmers must make to the gods of performance and efficiency—in the form of painfully complex, manual reorganization of computations and data, for each system and each architecture—will only grow more dear as the trends driven by "Moore's Law" scaling demand ever more extreme transformations for parallelism and locality just to keep each new generation of hardware busy.

Halide defines a new way of programming to address these challenges.

To address this, we propose a new programming language with a unique programming model. Halide decouples choices of organization from the definition and composition of algorithms. Halide defines image processing algorithms as graphs of pure functions from pixel coordinates to values. This defines the by which to compute each pixel value, but excludes all choices of the order of computation of pixels within each function, and their storage and communication through memory. Halide then describes the organization of the algorithm separately from their arithmetic definition. It elevates organization to a first-class feature in the programming model, using an explicit co-language of *schedules*. Schedules are defined by annotating functions used in a pipeline with a handful of basic primitives; complex organizations are described by *composing* multiple primitive annotations in different ways.

The explicit separation of algorithm definition from organization dramatically simplifies the definition and composition of image processing pipelines. The ability to describe numerous complex organizations of a single algorithm by composing a handful of scheduling primitives makes exploring the enormous space of organizations to find efficient implementations on different architectures dramatically faster and easier than it is in traditional languages used for high performance image processing software. The end result are simpler programs, which can match or exceed the performance of state-of-the-art hand-tuned implementations, while being portable and scalable across a wide range of architectures with different balances of parallelism, locality, and recomputation vs. storage cost.

3 REPRESENTING IMAGE PROCESSING ALGORITHMS

The Halide language is best thought of in two parts: a language of *algorithms*, and a complementary language of *schedules*. This chapter describes Halide's representation of image processing algorithms. To simplify analysis, and to maintain the flexibility to apply many different schedules to a given algorithm, we define a new domain-specific representation for image processing algorithms. The language of algorithms is explicitly designed to omit specification of the order of evaluation and the allocation and layout of intermediate storage. This gives both the programmer and the compiler extreme flexibility in scheduling when and where these functions should be computed and stored. This representation also dramatically simplifies common image processing algorithms relative to alternatives like C and CUDA, and simplifies the composition of large algorithms from many parts.

Halide represents image processing operations in a simple functional form. In a traditional imperative language, "images" are represented as mutable arrays, passed between subroutines which explicitly iterate over pixels. In Halide, images are instead functions from coordinates to values; pixels are defined by *how they are computed*, not by *where they are stored*. We represent images as pure functions defined over an infinite integer domain, where the value of a function at a point represents the color of the corresponding pixel. Imaging pipelines are specified as chains or graphs of functions. Functions may either be simple expressions in their arguments, or a sequence of iterative updates applied over a bounded domain. The expressions which define functions are side-effect free, and are much like those in any simple functional language, including:

- arithmetic and logical operations;
- if-then-else expressions;
- references to named values, which may be the free variables in a function's definition, Uniform values, or expressions defined by a functional *let* construct;
- calls to other functions, including loads from external images or scalar calls to external C ABI functions.

For example, we can define a two-dimensional function f:
$f(x, y) = x + y$
f is defined in terms of the free variables, x and y, mapping its two-dimensional domain. These variables, bound on the left-hand side of the definition, may be used in the expression on the right-hand side to define the value at the corresponding coordinate. Their range is not specified, and the function may be evaluated anywhere in its infinite domain. Points in the function are free to be evaluated, cached, duplicated, or thrown away and recomputed without affecting their meaning.

Image processing algorithms are built by composing functions together into graphs, using first-order function application. For example, we can compute an (unnormalized) 3×3 box filter of the input in with a simple two-stage pipeline:

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

Each function is defined over its own infinite domain, and the algorithm does not specify the regions to be computed. Rather, the compiler automatically analyzes the dependence between functions (using a bounds analysis, discussed in Chapter 8); given the output region a program wants to compute, all intermediate bounds are inferred. This is important for two reasons:

1. It leaves the compiler free to compute *more* of a given function than is required, which is often useful to improve alignment or simplify control flow, and can be controlled by the schedule.
2. It dramatically simplifies the algorithm code, eliminating most complex indexing and explicit boundary handling.

Language primitives

The core primitives of the language are:

- *Func*, a pure function, defining image values over some domain.
- *Var*, a free variable in the domain of a function.
- *IterDom*, a multi-dimension iteration domain, effectively an ordered list of bounded variables.[^In the current implementation, what I here call an *IterDom* is instead named *RDom*.]
- *Expr*, an expression defining the value of a function in terms of the free variables which make up its domain, as well as constants, iteration domains, parameters, and the application of other functions.
- *Image*, an immutable reference to an external memory buffer, visible to the algorithm as a function which may be applied only over the finite domain given by the image's dimensions.
- *Param*, a runtime variable parameter, providing a scalar argument to the algorithm.

All expressions are unambiguously typed as floating point, or signed or unsigned integer values, of a specific bit width, or *Tuples* of these atomic types. In the current implementation, only the most commonly supported widths are allowed[1]. Boolean values, as returned by logical expressions, are of type *UInt*(1). The types of all intermediate expressions are inferred from their operands using type promotion rules similar to C, and may be controlled by explicit arithmetic *cast* operations .[2]

1. 16, 32, & 64-bit floating point
8, 16, 32, & 64-bit signed and unsigned integer

2. The exact type promotion rules are more conservative than C about implicitly widening data types, since this often impairs vectorized code quality.

Expressions may only describe these simple scalar numeric types, or static tuples thereof; there are no pointers, references, or more complex data structures like lists or trees. The core Halide model allows only simple first-order functions, without recursion: functions simply map from integer coordinates to a scalar or statically-sized tuple result. Higher-order functions, general recursion, and dynamically-sized tuples are not allowed.

This representation is simpler than most functional languages, but it is sufficient to describe a wide range of image processing algorithms. These constraints enable extremely flexible analysis and transformation of algorithms during compilation. In this representation, all applications of a callee are statically analyzable, and they are always free to be evaluated before any applications of a caller (properties we will exploit to efficiently and flexibly schedule the execution of generated code). Constrained versions of more advanced features such as higher-order functions are added as syntactic sugar, but they do not change the underlying representation.

Iterative functions

In order to express operations like histograms, convolution by dynamically-sized kernels, summations, and scans [12], Halide also needs to express iterative or recursive computations. To do so, we introduce two additional primitives to the language:

- Beyond their initial pure definition, functions may have one or more recursive *update definitions*, which redefine the value at points given by an *output coordinate expression* in terms of prior values of the function. Their initial pure definition is treated as the *initial value function*, which specifies a value at each point in the *output domain* as a pure function of its input coordinates. All updates must be applied before the function may be used by any other function. The value of the function is defined at any point as the result of any updates which affect that point, applied in the order of their definition, to the initial value.
- Beyond pure variables, updates may be defined in terms of an *iteration domain* of one or more dimensions. An update defined with an iteration domain is iteratively evaluated and reapplied for every point in the domain. The programmer specifies a minimum value expression and an extent expression for each dimension of the iteration domain, as well as the relative order of the dimensions. The bounds of the iteration domain must be defined prior to the function in which it is used, and cannot depend on the function's value.

For example, we can compute the sum of all values over an entire image:

```
**IterDom** $r(0, in.width(), 0, in.height())$
$s() = 0$
$s() += in(r.x, r.y)$
```

Update definitions may also include pure variables, as well as iteration domain variables. For example, we can convolve an image by a variable-sized box:

```
**Param** $w$ // dynamic size parameter
**Var** $x, y$
```

```
**IterDom** $r(-w, 2\times w, -w, 2\times w)$
// iteration over $w \times w$ kernel in a single function:
$f(x, y) = 0$
$f(x, y) += in(x + r.x, y + r.y)$
```

Together, these two extensions provide an imperative escape hatch in Halide's otherwise purely-functional model of algorithms. They let Halide describe a range of algorithms outside the scope of traditional stencil computation, but essential to image processing pipelines, in a way that still isolates the order constraints of imperative execution and bounds side effects.[3] From the perspective of a caller, the result of the reduction is defined over an infinite domain, like any other function. At points which are never specified by an output coordinate, the value is the initial expression. The pure variables used in each update can still be evaluated in any order, as with pure functions; if an *update definition* is associative, even the iterative dimensions can be reordered or evaluated in parallel without changing the function's meaning.

3. The effects of these imperative features are isolated and bounded by the combination of the fact that functions may never be updated after they are used, and the fact that iteration domains must be bounded independently of and prior to the evaluation of the function in which they are used.

In order to isolate the effect of updates, and provide scheduling flexibility for their computation without changing their meaning, there is a subtle constraint on the definition of update steps: pure variables in the left hand side of an update definition must appear unadorned (as bare variables, not compound expressions) in the same arguments of the recursive calls to the function on the right hand side of the update definition. For example:

```
**Func** $f$; **Var** $x$; **Expr** $e$
$f(x, e) = x + f(x, e + x)$ // ok
$f(x, e) = x + f(x + 1, e + x)$ // illegal
$f(x, e) = x + f(e, x)$ // illegal
```

Scope

The specific, constrained scope of this programming model gives enormous flexibility in scheduling. Several constraints are particularly important to note:

1. Halide only models first-order functions over regular grids, up to five dimensions in the current implementation. The programming model trivially generalizes to higher-dimensional functions, but the current implementation's assumption of

dense, regular storage and iteration domains does not practically scale to high dimensionality.

2. Halide focuses on feed-forward pipelines. It can express scatters and iterative mutations, but these have bounded depth and range at the time they are invoked. This means the Halide programming model is *not Turing-complete*, because it would need infinite-sized pipelines to express arbitrary complexity computations.[4]
3. Halide omits most explicit conditional control flow, allowing only conditional selection among multiple, potentially eagerly evaluated expressions. This maps well to the functional model of image processing, and makes vectorized code generation simple and flexible, but it can cause unnecessary evaluation of expressions which are never used.
4. To work well, the compiler needs to be able to infer accurate dependence patterns for how functions access each other. For most patterns, our analyses are general and precise. However, in some cases, the programmer might need to explicitly clamp an index expression to a reasonable range to avoid allocating or computing unnecessary values.

4. Intuitively, iteration domains guarantee that all iterations have bounded extent independent of the execution of the expressions they iterate. As a result, all Halide algorithms are guaranteed to halt, and cannot be Turing-complete.

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

Differences from simple stencils

As constrained as it is, our model of algorithms more general than the well-studied domain of iterated stencil computations in several ways which are essential to expressing many real image processing pipelines.

Where stencil computations access their inputs through a single, static, shift-invariant set of taps (the stencil), Halide functions can also gather values from dynamically computed or data-dependent coordinates in other functions.

Through *update definitions*, Halide pipelines can include iteration more general than the time axis of an iterated stencil computation. In particular, a function's *update* steps may update each point in the domain differently. This allows the formulation, for example, of *spatially* iterative algorithms like scans, as opposed to iterating only in the temporal dimension, uniformly over all spatial points, as in iterated stencil computations. Further, the coordinates to be updated in a given iteration may be computed or data-dependent, allowing general scatters. The bounded *iteration domain* tightly constrains these more general computational patterns within the overall pipeline.

A simple histogram equalization algorithm combines multiple iterative functions, expressing a histogram reduction and a sum scan, with a data-dependent gather. All of these operations are beyond the scope of simple stencil computations:

```
ImageParam in(UInt(8), 2)
IterDom r(0, in.width(), 0, in.height()), ri(0.255)
Var x, y, i; Func histogram, cdf, out
histogram(i) = 0
histogram(in(r.x, r.y))+ = 1
cdf(i) = 0
cdf(ri) = cdf(ri − 1) + histogram(ri)
out(x, y) = cdf(in(x, y))
```

In detail, the histogram populated by a scattering update definition which iterates over all pixels in the input, a recursive sum scan integrates the histogram into a cumulative distribution function, and a pure function remaps the input using a data-dependent gather from the CDF. The iteration bounds for the histogram reduction and the sum scan are expressed by the programmer using explicit iteration domains (*IterDoms*): the first (*r*) over the domain of the input image, the second (*ri*) over the domain of all histogram buckets.

Finally, where iterated stencil computations recursively apply one or a small number of stencils numerous times, image processing pipelines are large graphs of *different* functions—different stencils, as well as more general gathers, scatters, and iterative functions—each of which is only applied once. Halide's pure functions express the common case of a single stencil iteration, while the call graph describes the entire pipeline. Iterated stencils can still be expressed as iterative update functions, but this is the exception, not the rule, in image processing.

Embedding & Metaprogramming

The current implementation of Halide is an embedded DSL in C++. It is a simple type-based embedding.[5] Halide code is C++ code written using the `Func`, `Var`, and other types to construct a Halide algorithm.[6] Function and expression definitions are embedded in C++ syntax using simple operator overloading on the corresponding types. For example, `Expr::operator*(int)` is overloaded such that `e*2` evaluates not to a concrete value, but to another `Expr` symbolically representing the operation $e \times 2$ in Halide IR.

5. There is also a Python embedding with a similar implementation.

6. The core C++ types for defining a Halide program are: `Func`, `Var`, `Expr`, `Image`, `Param`, `ImageParam`, and `IterDom` (called `RDom` in existing literature).

In practice, Halide is best thought of as a *staged language*[82] in C++. The *run-time* of this C++ code is really *elaboration-time* of the Halide algorithm definition. The complete data structure representing a fully-elaborated Halide pipeline is then passed, along with a corresponding schedule, to Halide's compilation logic to emit machine code, either into memory for just-in-time execution, or to an object file on disk for linking into a separate program.

As a consequence of this staged nature, C++ allows powerful *metaprogramming* of Halide programs.[7] C++ logic which evaluates to concrete numeric values become compile-time constant values in the Halide program. For example, `e*sqrtf(2.0f)` evaluates to the expression $e \times 1.4142$ at elaboration time; the resulting Halide algorithm will never execute a square root.

7. This should not be confused with so-called "template metaprogramming[4]," which is not used anywhere in Halide's implementation.

More generally, while control flow is highly constrained in Halide, arbitrary C++ control flow can execute during elaboration of a Halide program to simplify or parameterize its construction. For example, we could write a C++ function which programmatically constructs a chain of simple box filters of parameterized length, applied to an input function and returning an output function:

```
Func boxchain(Func in, int length) {
  Func cur, prev = in;
  Var x, y;
  for (int i = 0; i < length; i++) {
    cur(x,y) = 0.5f*prev(x-1,y) + 0.5f*in(x+1,y);
    prev = cur;
  }
  return cur;
}
```

This is a regular C++ function, operating on objects of Halide types, not itself a Halide algorithm. The function can be applied to different input functions, with different values for the length parameter. At any given invocation, it will elaborate a statically unrolled chain chain of Halide functions, of the specified length, beginning with the given input function. The resulting Halide algorithm is a flat sequence of functions, with no control flow; the C++ `for` loop, subroutine interface, and `length` argument are never part of the compiled or executed Halide pipeline.

PERFORMANCE TRADEOFFS AND THE ORGANIZATION OF COMPUTATION

4

My thesis argues that the performance of image processing pipelines is limited by fundamental tradeoffs between *parallelism*, *locality*, and *redundant computation*; that these tradeoffs are determined by the way computations and data are *organized*; and that we can more effectively program and optimize image processing pipelines by *explicitly modeling* the space of possible organizations as part of the program, and *decoupling* the description of organization from the definition of the underlying algorithm. This chapter will define a model of the problem of organizing computation in image processing pipelines. I will show how the characteristics we wish to optimize—parallelism, locality, and the amount of work—emerge from choices of the organization of computation, and how they are in tension with each other. Chapter 5 will build on this to define a compact but expressive *language* for the space of organizations, and demonstrate how we can apply it to describe optimized implementations by balancing the tradeoffs fundamental to this domain.

To understand these ideas, first, it is useful to define exactly what I mean by "algorithm" and "organization" in the context of feed-forward image processing pipelines.

4.1 ALGORITHMS

The *intrinsic algorithm* defines each value as an arithmetic expression and its dependencies on other values. Formally, I model the algorithm as a directed acyclic graph (DAG) of arithmetic expressions. For example, a simple dot product corresponds to a DAG which multiplies and accumulates the terms of two vectors (Figure 4.1). A three pixel box filter over a five pixel input, with three output pixels, is represented by the DAG in Figure 4.2, where each node in the *in* row corresponds to an input pixel, and each node in the *outt* row computes the average of its three inputs by evaluating the expression:

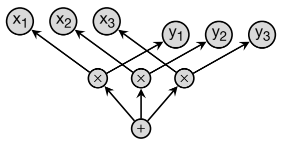


Figure 4.1: A $3D$ dot product as a task graph.

$$out_i = (in_i + in_{i+1} + in_{i+2}) \times \frac{1}{3}$$

noindent This DAG representation models the class of "straight-line" programs, sufficient to describe many numerical algorithms, but it cannot express the Turing-complete generality of "branching" programs [76]. Conditional and data-dependent evaluation are allowed through *conditional expressions*, but the topology of the program's dependence graph is fixed. For example, we can constrain the range of an output value by specifying each as:

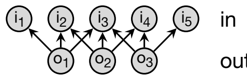


Figure 4.2: Blur algorithm as a task graph.

$$\begin{aligned} out'_i = \textbf{let } v = (in'_i + in'_{i+1} + in'_{i+2}) \times \tfrac{1}{3} \textbf{ in}\\ \textbf{if } v < 0 \textbf{ then } 0\\ \textbf{else if } v > 1 \textbf{ then } 1\\ \textbf{else } v \end{aligned}$$

The value of out_i depends conditionally on the result of the box filter (v), but the control flow and dependence of the algorithm's DAG is unaffected. You can think of this as *predicated execution*. The structure of the DAG for three pixels of out' computed from five pixels of in' (Figure 4.3) is *unchanged* from the non-conditional version; only the expressions at the nodes, and the values they actually compute, have changed. In this sense, *data values* may be conditional on inputs, but their control and connectivity remain fixed (Figure 4.3). There may still be heterogeneous structure, but it must be encoded into the topology of the DAG, itself. This representation can also encode a *trace*, or *unrolling*, of any general branching program, but a single graph can only represent a single control flow path.

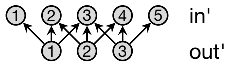


Figure 4.3: Algorithms have fixed structure, whether or not defined using conditional expressions.

Halide's constraints

This model makes concrete some of the features and constraints of the Halide algorithm language introduced in the previous chapter. In particular, Halide's view of an algorithm as a composition of functions over integer domains corresponds to grouping the nodes of a DAG into discrete layers, each corresponding to a single function. All nodes in a given layer are then defined by the same expression (Figure 4.4). These nodes are also embedded in an integer domain, and the expressions may be defined in terms of their coordinates in this domain. Because of this, I will usually express indices as arguments using function notation, rather than subscripts. For example, the three pixel box filter can be represented by the graph:

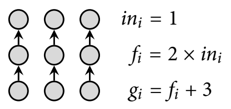
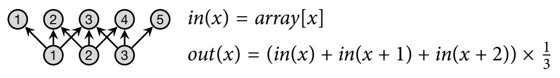
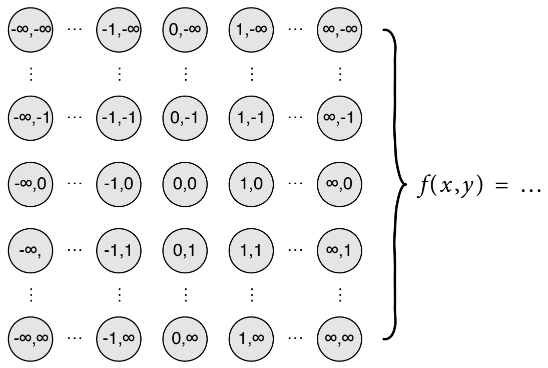


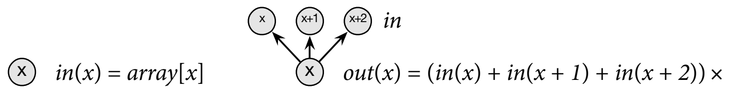
Figure 4.4: Nodes are grouped into functions.



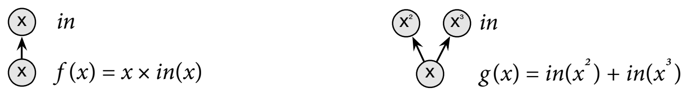
These domains generalize to n dimensions, and extend infinitely in all directions:



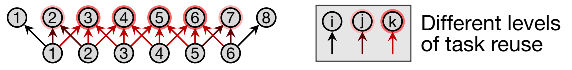
The dependence structure of the arithmetic expressions defining each function are *shift-invariant* with respect to the domain. As a result, a program can be represented as a set of local DAGs for each function:



It is important to note, however, that the resulting image operations are *not* necessarily shift-invariant. Even the indices of the upstream functions consumed may be shift-dependent, but the structure of the DAG for a single output pixel remains the same:



These shift-invariant functions may be evaluated anywhere, unrolling into the complete DAG for a given output range, which also reveals reuse of indices in one function by multiple indices in its caller:



This DAG representation, as presented, also ignores the restricted form of recursion allowed by Halide's iterative update functions[1]. For the purposes of this analysis, I will assume that the graph has been unrolled across the full output domain to expose all reuse of shared values, and to remove any recursion.

1. Iterative update functions are beyond the scope of *straight-line programs*, and introduce restricted *branching* functionality.

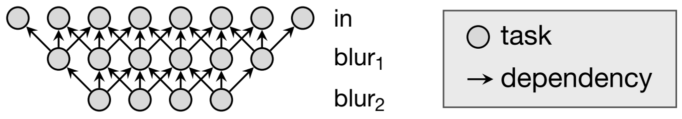
This definition of the intrinsic algorithm omits choices of the *order* of execution and placement of data. I instead consider these as complementary choices, defining how the algorithm's computation is *organized*.

4.2 ORGANIZATION OF COMPUTATION

The organization defines an order of evaluation, including choices of parallelism, as well as the allocation of storage/communication and placement of intermediate data. I represent the organization of computation as a *scheduled task graph*. The task graph is constructed by unrolling the algorithm into a set of nodes sufficient to compute the desired output. The nodes of the algorithm become tasks. Tasks compute a value based on their dependencies. For example, consider a simple two-stage box filter algorithm:

$$in(x) = \dots$$
$$blur_1(x) = (in(x-1) + in(x) + in(x+1)) \times \frac{1}{3}$$
$$blur_2(x) = (blur_1(x-1) + blur_1(x) + blur_1(x+1)) \times \frac{1}{3}$$

Unrolled to compute four pixels of output, its tasks and dependencies are:



This view remains *functional*, in that tasks are side effect-free: they yield a single output value, passed along dependence edges to any consumers. We can then *organize* the resulting computation in the task graph by imposing an order and granularity of evaluation, as well as an allocation of storage and placement of values within it.

Execution order and locality

To start, consider just the order of evaluation. Ignoring parallelism, a schedule defines a total order over the tasks. Valid schedules must respect the dependencies in the algorithm—visiting producers before their consumers in the DAG—and they must visit all required outputs.

The computation of a two-stage box filter can be organized into many different orders. For example, all nodes in a given stage can be computed before moving on to the next (Figure 4.5, top). Alternatively, each consumer can be computed as soon as all of its producers are complete (Figure 4.5, bottom).

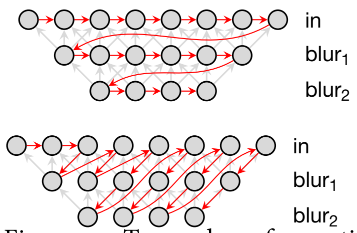


Figure 4.5: Two orders of execution for the task graph of the two-stage blur algorithm.

With just this choice, we can already see a key cost emerge: *locality* corresponds to the *use distance* along the path, from the node where a value is computed, to a node where it is consumed. This metric applies to every producer-consumer pair in the DAG; locality for a whole program corresponds to the distribution of use distances over all producer-consumer pairs, which can be summarized by scalar metrics like the *average use distance*, or the *maximum use distance* (equivalent to the maximum *working set* or *footprint* of intermediate data in the organization). In the box filter examples, the second organization, which evaluates consumers as soon as their inputs are complete, has much shorter average use distance, and therefore better overall locality (Figure 4.6).

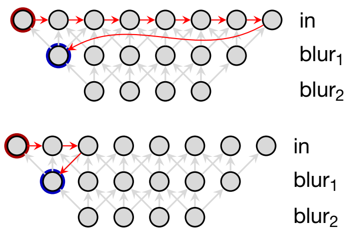
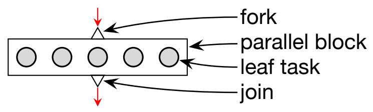


Figure 4.6: *Use distance* from a task to its consumer in two different organizations of the same blur algorithm.

Parallelism

I model parallelism in the task graph by ordering tasks to execute in parallel between special *fork* and *join* nodes:

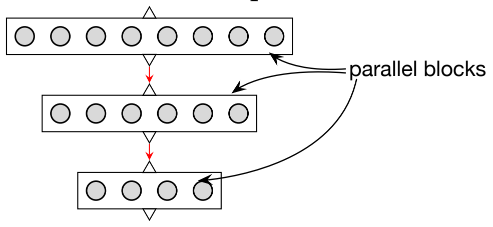


By allowing tasks to nest hierarchically, but requiring the children of each task to be either purely parallel or totally ordered at a given level of nesting, we can model the general *series-parallel* structure commonly assumed in the analysis of parallel algorithms[2]. Each node in a series-parallel task graph is composed of either a totally-ordered sequence of child tasks, or a *parallel block*, with a single fork point at the start, a set of parallel (unordered) child tasks, and a single join point at the

2. The nested series-parallel structure naturally represents any program consisting of parallel `for` loops, general nested data parallelism as in, e.g., NESL, and fork-join constructs, including nearly all patterns expressed by Cilk. (This structure is commonly assumed even in analysis of Cilk programs.)

end. The leaves of the hierarchy are the atomic tasks which perform actual computation (the same nodes which made up the entire task graph before introducing parallelism).

For example, the pixels in each stage of the chained box filter can be evaluated in parallel:



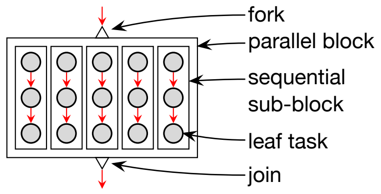


Figure 4.7: Nested parallel task blocks.

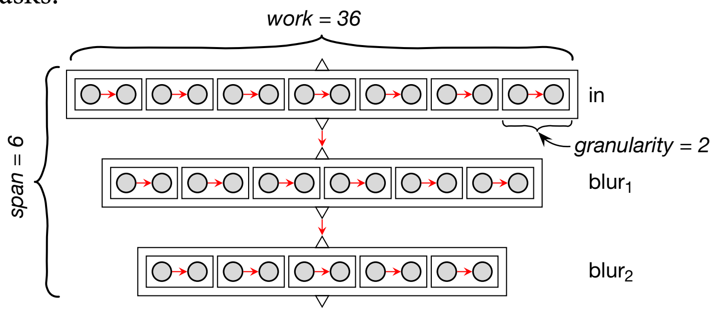
Following [23] and [13], I analyze the amount of parallelism as the ratio of two properties of a scheduled task graph:

1. The *work*, defined as the number of leaf tasks, gives the total size of the problem to be computed.
2. Conversely, the *span* (or *critical path*) of the whole graph—the length of the longest path which cannot be executed in parallel—gives an upper bound on the parallel speedup assuming infinite parallel processors.

Given these two terms, the amount of parallelism in a given organization can be understood as the $\frac{work}{span}$ ratio.

In practice, the amount of parallelism must be traded off against the *granularity* of parallel tasks. I define granularity as the number of subtasks contained by each task in a parallel block. Real parallel execution usually imposes some overhead for each task in a parallel block; granularity recognizes the tradeoff between the shortened span brought by increased parallelism, and the increased execution and synchronization overhead of organizing computation into more, finer-grained tasks.

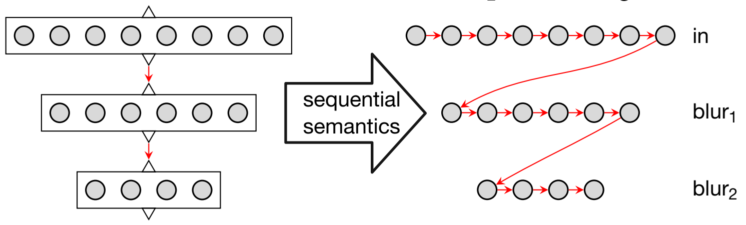
For example, the simple parallel organization of the two stage blur has six parallel tasks in the first stage, four in the second, and two in the third. For a similar organization over $O(n)$ output pixels, the work in each stage is $\approx n$. This organization reduces the span from $O(n)$ in the sequential organizations, to ≈ 3. The granularity of the tasks is extremely fine, each computing a single value. Alternatively, we can coarsen the parallel tasks, to trade parallelism for less per-task overhead, by hierarchically decomposing each parallel region into sets of tasks:



Locality in parallel task graphs

Considering locality, however, note that both of these organizations evaluate the entirety of each stage before moving on to the next. This is much like the breadth-first sequential organization, but for our understanding of *use distance*, the parallel edges in this graph do not capture the fact that all tasks in each stage are evaluated before any in the next (the meaning of the *join* point at the end of a parallel block). Instead, I use another common simplification from the analysis of parallel algorithms and define a *sequential semantics* on which to analyze locality. The sequential semantics are defined simply by trivially flattening the hierarchical composition of sequential tasks into a single sequential order, and flattening the sub-graph of each parallel block by imposing a simple (e.g., lexicographic) total order on the tasks within[3]. Under the sequential semantics, the average use distance between stages of both granularities of breadth-first parallel organization is identical to our initial breadth-first sequential organization:

3. Sequential semantics are a good fit for many real systems which exploit locality with caches, since caches are a fundamentally sequential construct. Any sequential ordering maintains the same *average* use distance from parallel sets of producers to consumers, though different sequential orderings can change the overall *distribution*. It is easy to see that, regardless of the permutation of tasks within a parallel block, the average use distance to consumers of any tasks within the block remains unchanged. By definition, in a valid organization, any consumers must come *after* the join point completes the parallel block, so the change in use distance corresponds only to the change in distance from a node to end of the parallel block, the average of which is unchanged by permutation.



Looking back at the alternative *locality-optimized* order (Figure 4.5), moving consumer tasks as soon after their producers as possible required a precise ordering of tasks. This fixed sequential ordering is fundamentally at odds with parallelism. The only parallel decomposition of this organization which does not affect the average use distance in the graph is only able to compute the first three pixels in parallel (Figure 4.8).

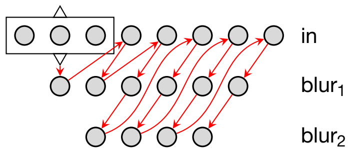


Figure 4.8: A locality-optimized interleaving allows virtually no parallelism.

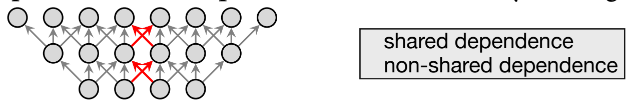
This organization has a maximum parallelism of three, a minimum parallelism of one, and a span just two shorter than the fully sequential version ($\approx n - 2$).

Here, we begin to see how the twin goals of parallelism and locality are in tension with each other: given a fixed task graph, an organization cannot simultaneously minimize use distance, which requires constraining the order of execution to move producers near consumers, and maximize parallelism, which requires unconstrained order over large collections of tasks. As a result, there is no meaningfully parallel organization equivalent to the locality-optimized sequential order.

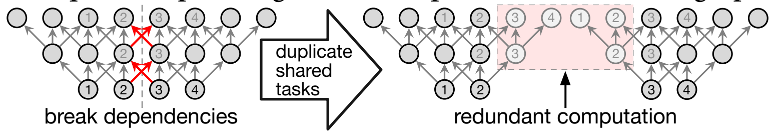
The tension between these goals is rooted in the dependencies between producers and consumers in the original algorithm. Maximizing locality constrains order, and particularly limits regular parallelism, because it requires executing a producer task as close as possible not just to *one* consumer, but to *all* of its consumers. In this way, maximizing locality for a producer interlocks the order of all of its consumer tasks. With stencil computations, so fundamental in image processing pipelines, tasks with multiple consumers are the common case. But this also points to a third way we can organize the computation: we can transform the task graph to break dependencies where multiple consumers depend on the value from a single producer. This opens up the possibility of different organizations, which maximize both locality and parallelism, but breaking dependencies requires duplicating nodes consumed by multiple downstream tasks.

Redundant computation

The third major tradeoff I consider in organizing computation is the possibility of intentionally introducing *redundant computation* to break dependencies in an algorithm, enabling simultaneous optimization of parallelism and locality. For example, returning to the two-stage box filter, notice that each task depends on a set of prior tasks also shared by its neighbors:



By breaking the dependence of neighboring consumers on a single producer, we can decouple their execution, splitting the computation into multiple independent subgraphs, but doing so requires duplicating the shared producer in each subgraph:



These subgraphs can be executed in parallel, and the worst-case use distance for the shared nodes is reduced by replicating them as near as possible to each consumer, but this comes at the expense of *redundantly recomputing* the duplicated nodes. In this way, we see it is also possible to trade off the total *amount of computation* to improve both *parallelism* and *locality*. I measure *excess computation* in the task graph as the ratio of the number of nodes computed to the minimum number required by the intrinsic algorithm (which is the set of all dependencies of the desired outputs).

When the four output, two-stage blur is split into two tiles, we introduced five pixels of redundant *excess computation*. In general, the amount of excess computation is proportional both to the amount of reuse in the original graph—the number of consumer tasks which depend on each producer, given by the size of the stencil—and also to the *granularity* at which the graph is split.

Together, parallelism, locality, and the total amount of computation are fundamentally in tension in the organization of computation. (This is not a limitation of the series-parallel structure, but applies similarly to any parallel organization.) In practice, the ideal organization for a given algorithm on a given architecture must balance all three, the possible choices for how to do so are combinatorially complex, and the best choice subtle and unpredictable.

Placement of data

The organization of computation is also responsible for mapping intermediate data passed between computations to locations in memory. So far, while focussed on ordering, I have assumed that each task is mapped to a unique storage location for potential access by any later consumer. Depending on the order of execution, however, it is usually possible to *reuse* memory locations for multiple values, reducing the total storage footprint. I formulate this last aspect of organization as a matter of choosing when and where storage (memory) is *reused* during execution. I model this in the task graph by grouping nodes into *allocations*, and overlaying *reuse* edges between allocations which map to the same storage. For example, in the breadth-first organization of the two-stage box filter, a simple allocation strategy groups each stage into its own allocation, and reuses the same storage for the first and third stages (Figure 4.9).

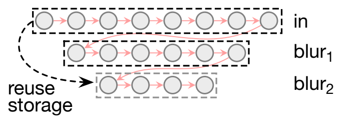


Figure 4.9: Storage reuse.

A legal schedule requires that the storage for a node not be reused until all of its consumers have executed. In the task graph, this means that the destination of a reuse edge must come strictly after the destinations of all dependence edges along the path given by the execution order edges.

I define the cost of storage as the *allocation footprint*, measured as the size of all allocations in the task graph which do not have any incident reuse edges. The size of each allocation is the maximum number of nodes covered by any of its uses (connected by reuse edges). For example, the locality-optimized ordering of the two-stage box filter enables a storage mapping with a much smaller footprint than the breadth-first organization (Figure 4.10). However, as with fine-grained parallel tasks, there is also cost associated with tracking many independent memory allocations. In real implementations, it is common to allocate and free large buffers shared statically by many grouped values. This model allows allocations to span groups of tasks. The *granularity* of storage allocation is given by the number of nodes shared within the allocation.

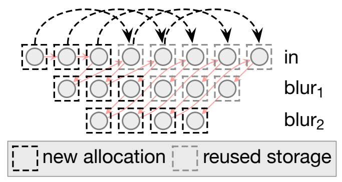


Figure 4.10: Fine-grained storage reuse in a locality-optimized ordering.

4.3 SUMMARY

This chapter proposes a model for understanding a wide class of programs as two separate concerns:

1. The *intrinsic algorithm* defines a basic graph of tasks connected by dependence edges.
2. The *organization* optionally duplicates nodes to split shared dependence edges; hierarchically groups tasks for nested parallel execution; overlays this graph with ordering edges; and finally overlays the graph with reuse edges, mapping the output of tasks to shared memory locations.

Taking these two concerns together, we can model programs as scheduled task graphs. The major characteristics which determine performance and efficiency on modern machines are directly visible in this graph:

- *Locality* relates to the reuse distance between dependent tasks along the order of computation.
- *Parallelism* is summarized by the $\frac{work}{span}$ ratio in the scheduled graph.
- *Task granularity* is given by the work within a single parallel task.
- *Redundant work* corresponds to the number of nodes executed relative to the minimum number required by the intrinsic algorithm.
- *Storage footprint* corresponds to the number of nodes without incoming reuse edges.
- *Allocation granularity* relates to the ratio of storage footprint to the number of unique allocation groups.

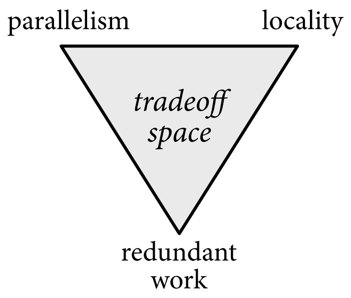


Figure 4.11: Parallelism, locality, and redundant work are often in tension, and must be traded off with each other to maximize the overall efficiency of an algorithm on a particular architecture.

I have shown how, for a given algorithm, these costs are determined by the organization of computation. In addition, because of the structure of dependencies in image processing algorithms, the key costs are often intrinsically in tension with each other. The next chapter will build on this view to define a model and language for compactly describing the organization of computation in the task graphs of image processing pipelines, independently of the algorithm definition.

5 SCHEDULES: MODELING THE ORGANIZATION OF COMPUTATION

This chapter introduces a model and language of *schedules*. Schedules compactly describe the organization of computation in image processing and stencil pipelines, independently of the algorithm definition. As we saw in the previous chapter, we are primarily concerned with the *order of execution* across the task graph making up the computation of a complete pipeline, where tasks represent individual operations on individual pixels. The Halide model is based first on defining an order of execution, and then determining the duplication of intermediate computations, and the allocation of storage for intermediate results, within the constraints of that order.

Treating each function on each pixel as a separate task, the space of possible organizations of a given image processing algorithm is enormous.[1] I argue, however, that the most fruitful organizations can be described compactly, by the composition of a few simple choices. These choices, and the rules by which they compose, form our model of schedules.

1. The space of execution orders is at least $O(2^n)$ for common task graphs.

Looking at a task graph formed from our restricted model of algorithms (graphs of functions over regular grids), we see an obvious division of the organization problem into two sets of choices:

1. Choices of organization *within* each stage (layer of the graph)
2. Choices of organization *across* stages

Halide's schedules model the organization of computation based on each stage choosing at what granularity to *compute* each of its inputs, at what granularity to *store* each for reuse, and, within those grains, in what *order* its domain should be traversed.

For simplicity and efficiency, schedules only model orders which can be compactly encoded as loop-nests which scan the required region of each stage, for all stages in the pipeline. The loops over the dimensions of each function, in isolation, must form a *perfect loop nest* (the *domain order*).

Schedules model *parallelism* by making some of the loops in this loop nest parallel; since tasks within a domain are always independent, they may be interleaved in any order by parallel execution. The granularity of parallelism corresponds to the number of tasks contained within each iteration of a parallel loop.

Schedules exploit *locality* by then *interleaving* the computation of subsets of interdependent tasks *across* multiple stages. Organization across stages is constrained by producer-consumer dependencies: for schedules to be *valid*, values must be computed and stored before they are consumed, and freed after. Halide's schedules define organization across stages as the *granularity* at which values are *grouped* and *interleaved* between producer and consumer stages, described in terms of the levels in the loop nest encapsulating a consuming stage at which a producer stage is computed and stored for reuse (the *call schedule*). This interleaves the perfect loop nests over the domains the individual functions into a single *semiperfect loop nest* which computes the required pixels of all stages in an entire pipeline.

Finally, the required region of each stage is inferred from its use, recursively back from the output, and tracked as an n-dimensional interval ("axis-aligned bounding box"). Neither the schedule nor the algorithm explicitly specifies the bounds to be computed; they are inferred from context using interval analysis[60].

The Domain Order: organization within *stages*

Halide's language of schedules first defines the order in which the required region of each function's domain should be traversed, which we call the *domain order*. Given a multidimensional interval (axis-aligned bounding box) specifying the region required of a function, the domain order defines a perfectly nested loop which subdivides the required region of the function and specifies the order in which it should be computed. Ordering *within* each stage is unconstrained for pure functions, since the definition of a stage does not allow dependence between values within it, so any ordering of the tasks within a single stage is valid.

The choice of loops is built up by construction from a default order which simply loops over the natural dimensions of the function in row-major order. For example, for the simple two-dimensional function $f(x, y)$, the default order traverses the y dimension outside the x dimension:

```
for $y = y_{min}$ to $y_{max}$
    for $x = x_{min}$ to $x_{max}$
        compute $f(x, y)$
```

Halide specifies the domain order for a function using a traditional set of loop transformation concepts, applied to the dimensions of the function. In the language, these choices are applied as annotations on each function.

- Each dimension can be traversed *sequentially* (the default) or in *parallel*. In the Halide language, this is written as, e.g., `f.parallel(y)`.
- Constant-size dimensions can be *unrolled* (`f.unroll(x)`) or *vectorized* (`f.vectorize(x)`).[2]
- Dimensions can be *reordered* (e.g., from column- to row-major: `f.reorder(y, x)`).
- Dimensions can be *split* by some factor, creating two new dimensions: an outer dimension, over the old range divided by the factor, and an inner dimension, which iterates within the factor (`f.split(x,outer,inner,factor)`). After splitting, references to the original index become *outer* × *factor* + *inner*.
- Finally, dimensions can be *fused*, turning two dimensions into one (`f.fuse(x, y)`). The new dimension iterates over the product of the range of the two original dimensions, and references to the original indices become $\frac{fused}{innerwidth}$ and *fused* mod *innerwidth*.

2. Vectorization and unrolling of general dimensions are modeled by first splitting a dimension by the vector width or unrolling factor, and then scheduling the new inner dimension as *vectorized* or *unrolled*: `f.vectorize(x, 4)` is syntactic sugar for:

```
split $f.x \rightarrow (f.x_o, f.x_i)$ by 4
vectorize $f.x_i$
```

Splitting recursively opens up further choices, and enables many common patterns like tiling when combined with other transformations.[3]

3. A 2D tiled loop over x, y corresponds to the domain order transformation:

```
split $x \rightarrow (x_o, x_i)$ by $t_{width}$
split $y \rightarrow (y_o, y_i)$ by $t_{height}$
reorder $x_i \rightarrow (y_i, x_o)$ by $y_o$
```

This gives the loop structure:

```
for $y_o$
    for $x_o$
        for $y_i$
            for $x_i$
                ...
```

Fusing dimensions, meanwhile, does not alter the order of traversal of the outer and inner loops, it simply allows them to be treated as a unit. This can be useful in several contexts. In particular, fusing two loops into one and evaluating it in *parallel* can have lower overhead than relying on nested parallelism applied to both loops simultaneously[4]. Fusing multiple dimensions also allows them to be *vectorized* together. For example, some computations are naturally organized with interleaved color channels, but the number of color channels is often smaller than the machine vector width; in this case, *fusing* the innermost (x) dimension with the dimension of color channels (c) allows them to be vectorized together, using any desired vector width up to the width of the *product* of both fused dimensions.

4. The overhead of nested parallel loops comes from additional tasks in the task system, while the overhead of *fused* loops comes from the added cost of the *div* and *mod* ops to compute the corresponding indices.

Because Halide's model of functions is data parallel by construction, dimensions can be interleaved in any order, and any dimension may be scheduled serial, parallel, or vectorized. For reduction functions, the dimensions of the reduction domain may only be reordered or parallelized if the reduction update is associative. The free variable dimensions of reduction functions may be scheduled in any order, just as with pure functions.

The Call Schedule: organization across *stages*

In addition to the order of evaluation *within* the domain of each function, the schedule also specifies the granularity with which to interleave the computation and storage of each function with the domain of the functions that call it. We call these choices the *call schedule*. Halide specifies a unique call schedule for each function in a pipeline. Each function's call schedule is defined by the points in the loop nest of its callers where it is computed and stored for reuse.

In the language of Halide's schedules, we control the call schedule of each function with two annotations:

- `f.compute_at(g, v)` specifies the granularity at which to realize regions of the function f as required at each iteration of the variable v in the loops over the domain of the function g. Each function must be computed in a scope at or enclosing the scope where it is consumed.
- `f.store_at(g, v)` similarly specifies the granularity at which to allocate memory to store and reuse values of f as within

each iteration of v in the loops over the domain of g. Each function must be stored in a scope at or enclosing the scope where it is computed.

Different choices of call schedule make different tradeoffs between locality and redundant computation for the values produced by one function (here, f) and consumed by another (here, g). The exact effect and interaction of domain order and call schedule choices are explained in depth in the remainder of this chapter, but intuitively call schedules make the following tradeoffs:

Computing at finer granularity improves locality, alternating between producing and consuming smaller collections of results, and minimizing reuse distance between where values are produced by one function and consumed by others. By default, the compute and storage granularity are the same: the results of a single region of f are allocated, produced, consumed by g, and then discarded, making the grains of computation of f independent. However, the region consumed in g by successive iterations often overlaps .[5] When this happens, each grain of f must redundantly *recompute* any values in its domain shared with other iterations.

5. This is always true for stencil access patterns.

Storing at coarser granularity keeps values around longer for potential reuse. This can avoid redundant computation of values shared between multiple computation grains of f. However, capturing reuse across multiple iterations requires constraining the order of execution across the grains of computation, so they can be synchronized to avoid redundantly recomputing shared values. This constrains the available parallelism. In Halide's semiperfect loop nests, it means that any dimension between the storage and computation granularities along which a function captures reuse over multiple compute iterations must be traversed *sequentially*.[6]

6. Splitting and reordering are perfectly valid, but parallelism—including vectorization—precludes reuse.

Finally, while most of my explanation focusses on scheduling a single function with respect to one other, real image processing pipelines contain graphs of dozens or hundreds of functions. Schedules are specified *per-function*, so the overall organization of computation is composed globally by the domain order and call schedule choices at each producer-consumer relationship in a large graph of functions.

Bounds inference

A schedule's specification of organization both within and across stages as a semiperfect loop nest is agnostic of the actual bounds of the resulting loops. All choices are specified independent of, and without reference to, the actual bounds of the required region of the function. Given a domain order for a function, the bounds of the resulting loops can all be symbolically inferred from the use of the function in the algorithm, and it's placement within the scheduled pipeline according to its call schedule. This dramatically simplifies schedule specification, and makes it impossible for any meaningful schedule to specify a loop nest which will not correctly satisfy each stage's producer-consumer relationships. [7]

7. The only interaction that may occur between the schedule and the meaningful bounds over which a pipeline may be computed is due to splitting: the minimum bounds of a split dimension must be rounded up to at least the split factor, but this effect is still inferred, not explicitly specified.

The Halide model only considers axis-aligned bounding regions, not general polytopes—a practical simplification for image processing and many other applications. This also allows the regions to be defined and analyzed using simple interval analysis. Since the simple model of domain order relies on later compiler inference to determine the actual bounds of evaluation and storage for each function and loop, it is essential that bounds analysis be capable of analyzing *every* expression and construct in the Halide language. Interval analysis is simpler than modern tools like polyhedral analysis, but it can effectively analyze a wider range of expressions, which is essential for this design.

5.1 UNDERSTANDING SCHEDULING AS LOOP SYNTHESIS

To understand Halide's schedules in more detail, it is helpful to work through the actual loops described by various scheduling choices. Consider the organization of the simple blur pipeline:

$in(x, y) = \ldots$
$bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)$
$by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1)$

First, we can define a simple tiled domain order for *by*:

split $x \rightarrow (x_o, x_i)$ by 4
split $y \rightarrow (y_o, y_i)$ by 4
reorder (x_i, y_i, x_o, y_o)

This defines a four-dimensional loop nest over the domain of *by*:

```
// a
for $y_o$
    // b
    for $x_o$
        // c
        for $y_i$
            // d
            for $x_i$
                // e
                compute $by(x_o \times 4 + x_i, y_o \times 4 + y_i)$
```

Throughout this section, we will use a loop pseudocode to help demonstrate the organization described by different schedules. For clarity and brevity, this pseudocode is written more as a sketch than a precise definition. Most significantly, coordinates used are the logical coordinates within functions' domains, wherever possible, including to address values stored in memory buffers. The vagaries of addressing values as offsets within arrays and similar details are ignored to better highlight the essential structure.

Computing a producer at a granularity of its consumer

Since *bx* is called by *by*, it must be computed somewhere before *by*. In the Halide model, the call schedule for *bx* allows it to be computed at any loop level enclosing its consumer (labeled *a* through *e*). The level at which we compute *bx* determines the granularity of pieces which are interleaved between the two stages in the pipeline. For example, *bx* may be computed at the granularity required for a single point in *by* (at *e*); at the granularity required for a single row of one tile of *by* (at *d*); at the granularity required for a full tile of *by* (at *c*); at the granularity required for a strip of four scanlines of *by* (at *b*); or at the granularity required for all uses by any points ever computed in *by* (at *a*). We describe the call schedule by saying that *bx* is "**computed at** *a dimension* **of** *a downstream function*." Each point in the loop nest is named by the dimension of the function to which it corresponds. The coarsest granularity, outside all loops, is given the special name "*root*."[8]

A common pattern in hand-optimized image processing pipelines is to interleave stages at the granularity of tiles. In this case, that corresponds to `bx.compute_at(by, xo)`. Given the default domain order for *bx*, that computes the two stages with the loop nest:

```
for $by.y_o$
    for $by.x_o$
        for $bx.y$
            for $bx.x$
                compute $bx(bx.x, bx.y)$
        for $by.y_i$
            for $by.x_i$
```

8. For example, we would say `bx.compute_at(by, xi)` for point *e*; `bx.compute_at(by, xo)` for point *c*; or `bx.compute_root()` for point *a*.

```
            compute $by(by.xo \times 4 + by.xi, by.yo \times 4 + by.yi)$
```

This also implies that we allocate storage for a whole tile of *bx* to store all the intermediate results from the loops which compute *bx* to where they are used to compute *by*:[9]

9. Implicitly, `bx.store_at(by, xo)`

```
**for** $by.y_o$
    **for** $by.x_o$
        allocate $bx[\dots]$
        **for** $bx.y$
            **for** $bx.x$
                $bx[bx.x, bx.y] \leftarrow$ compute $bx(bx.x, bx.y)$
        **for** $by.y_i$
            **for** $by.x_i$
                compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

Inferring the bounds of required regions

The sizes of the allocation and computation of *bx* implicitly depend on the bounds required for the corresponding uses in *by*. Recall that these bounds are not specified by the schedule, but are inferred as the minimum intervals required to satisfy the ordering and granularity specified in the schedule. Adding symbolic bounds to the loops and allocations gives the full structure of the generated loops for the sub-pipeline from *bx* to *by*:

```
**for** $by.y_o = by.y_o.min$ **to** $by.y_o.max$
    **for** $by.x_o = by.x_o.min$ **to** $by.x_o.max$
        allocate $bx[bx.x.extent \times bx.y.extent]$
        **for** $bx.y = bx.y.min$ **to** $bx.y.max$
            **for** $bx.x = bx.x.min$ **to** $bx.x.max$
                $bx[bx.x, bx.y] \leftarrow$ compute $bx(bx.x, bx.y)$
        **for** $by.y_i = by.y_i.min$ **to** $by.y_i.max$
            **for** $by.x_i = by.x_i.min$ **to** $by.x_i.max$
                compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

The actual values of the *min*, *max*, and *extent* (= *min* − *max*) terms are computed by bounds inference, recursively from the required region of the output. The actual intervals computed are unimportant, but it is worth noting one feature of this pipeline: because *by* accesses *bx* through a *stencil* ($by(x, y) = bx(x-1, y) + bx(x, y) + bx(x+1, y)$), neighboring points in the domain of *by* depend on overlapping points in the domain of *bx*. The region of *bx* required to compute a single tile of *by* is actually two pixels wider than the tile of *by*, and the region required by neighboring tiles of *by* overlaps in *bx* . Because *bx* is allocated and computed at the granularity of tiles of *by* (i.e., at $by.x_o$), for each tile (iteration of $by.x_o$), this organization recomputes an *overlapping* tile of *bx*, performing redundant work where the tile boundaries overlap.

Storing a function at a different granularity than its computation

From the view of Halide's schedules, this is not just because we interleaved the producer-consumer computation at this granularity, but because we also chose only to *store values of bx for reuse* at this granularity. This is the other choice made by a call schedule. In spite of the redundant computation, this is often an efficient organization; it completely decouples the computation of tiles, leaving them free to execute in parallel, and removing any intermediate data flowing between them, which could hurt locality. The model of call schedules allows another choice: we can allocate and store values of *bx* for reuse at another, *coarser* granularity in the computation of *by*. For example, we can store and reuse *bx* at $by.y_o$ (i.e., at the granularity of tile strips):

```
for $by.y_o$
    allocate $bx[bx.x.extent \times bx.y.extent]$
    for $by.x_o$
        for $bx.y$
            for $bx.x$
                $bx[bx.x, bx.y] \leftarrow$ compute $bx(bx.x, bx.y)$
        for $by.y_i$
            for $by.x_i$
                compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

The buffer allocated for intermediate results is larger because the extent required of *bx* is larger at this level (it now spans all iterations of $by.x_o$), but this larger buffer captures all uses of *bx* across a whole strip of *by*.

Because values of *bx* are no longer thrown away after each tile (iteration of $by.x_o$), each subsequent tile of *bx* can notice what was already computed and begin where the previous one left off. This still computes small tiles of *bx* immediately before they are consumed by the corresponding tiles of *by*, giving the locality benefit of interleaving producer and consumer with fine granularity, but it avoids redundantly recomputing values in *bx* reused by neighboring tiles of *by*. It is even possible to optimize the storage footprint for *bx* by recognizing that any value is only reused by three neighboring pixels in *by* ($bx(x, y)$ will never be used again after the computation of $by(x + 1, y)$ in this organization), so the storage can be remapped into a smaller circular buffer[10].

10. In the Halide design, this is left as a lower-level optimization to be discovered by the compiler, not explicitly described by the schedule. This optimization, which we call *storage folding* is discussed in Chapter 8

However, exploiting all of these patterns which emerge when a function is stored for reuse at a coarser granularity than it is computed depends on knowing the order of evaluation: each iteration can only statically know both what has been computed before, and what will never be needed again, given deterministic ordering of the loops between the storage and computation granularities (here, $by.x_o$). In practice, the Halide model requires that the loops between the storage and computation granularities be *sequential* in order to capture reuse and fold storage.

Inlined evaluation

The space of call schedules in Halide also includes a special case, called *inline*: inline computation uniquely computes a function as required at every separate call site. As the name implies, a call to a function scheduled to be computed inline is simply replaced with its definition. As a granularity, this is equivalent to computing and storing at the innermost loop of the caller's domain order, but does so separately for *every* caller. Inline functions do not have a meaningful domain order or storage granularity, since every point is inlined and evaluated independently. The Halide compiler treats this case specially: inline-computed results are passed through registers wherever possible, and never allocate storage on the heap.

In the case of the two-stage blur, computing *bx* inline effectively merges the two separate 1*D* convolutions into a single 2*D* convolution:

```
**for** $y$
  **for** $x$
    $by(x, y) = \frac{1}{9}(bx(x-1, y-1) + bx(x, y-1) + bx(x+1, y-1)$
                $+bx(x-1, y) + bx(x, y) + bx(x+1, y)$
                $+bx(x-1, y+1) + bx(x, y+1) + bx(x+1, y+1)\quad)$
```

Scheduling a third function

The call schedule defines the granularity of interleaving between each function and the subsequent stages of the pipeline which depend on it. In general, the available granularities are not limited to the domain of the immediately calling function, but include *any enclosing loop* in-scope where the function is used.

Continuing with our blur pipeline, consider also the input function, *in*, used in *bx*:

$$in(x,y) = \dots$$
$$bx(x,y) = in(x-1,y) + in(x,y) + in(x+1,y)$$
$$by(x,y) = bx(x,y-1) + bx(x,y) + bx(x,y+1)$$

Given our initial tiled schedule, where *bx* is *computed at* $by.x_o$, at what granularities could we *compute in* and *store it for reuse*? Returning to the loop structure, we see:

```
// a
for $by.y_o$
    // b
    for $by.x_o$
        // c
        for $bx.y$
            // d
            for $bx.x$
                // e
                compute $bx(bx.x, bx.y)$ // uses in
        for $by.y_i$
            // f
            for $by.x_i$
                // g
                compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

in is used to compute *bx*, so each grain must be computed before we compute *bx*. As with the call schedule for *bx* in *by*, *in* can be computed anywhere in the domain order of *bx* (points *d* and *e*), but it may also be computed at any enclosing granularity *within which bx* is computed: as required for every tile of *by* (point *c*); as required for every four scanlines of *by* (point *b*); or as required for all pixels in the output (point *a*). These latter three points relate to the domain order of the downstream function *by*, not the immediate caller of *in*. It may *not*, however, be computed at granularities which do not enclose the computation of *bx*, since these do not correspond to meaningful granularities at which the function *in* is actually used (here, points *f* and *g* actually come *after* the computation of *bx*, which requires the computation of *in*).

Scheduling functions with multiple consumers

Because we specify the call schedule per-function, while functions may be called in multiple different stages, a valid call schedule must compute every function at a level which is in scope for *all* of its callers. Consider the simplest case of a function which is called separately by two unrelated functions:

$in(x, y) = \ldots$
$f(x) = in(x) \times 2$
$g(x) = in(x-1) + in(x+1)$
$h(x) = f(x) + g(x \times 2)$

This forms a diamond dependence pattern: *in* is used independently by both f and g, which are then used together to compute h. Because *in* is used separately in both f and g, it must be computed and stored at some granularity which encompasses its uses in both functions. Concretely, it must be scheduled at some granularity at or after the point at which these two branches of the pipeline re-join in the evaluation of h.

We can still express unique call schedules for each separate call site of a single function by restructuring the graph: by inserting a separate identity function which proxies the shared function at each unique call site, we can schedule the shared function to be computed *inline*, and then separately control the computation granularity of each call site via its unique proxy function. For example, we can rewrite our simple example as:

$in(x, y) = \ldots$
$in_f(x) = in(x)$
$f(x) = in_f(x) \times 2$
$in_g(x) = in(x)$
$g(x) = in_g(x-1) + in_g(x+1)$
$h(x) = f(x) + g(x \times 2)$

Then, by computing *in inline*, we have effectively created two new functions, *inF* and *inG*, which may be computed independently within f and g, respectively.

Interaction between call schedules and domain order

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

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

The interleaving defined by the call schedule intertwines the order of execution of every function in an entire pipeline. One consequence of this is that choices of *parallelism* applied to the domain order of one function can implicitly influence other functions upstream. Consider again the blur pipeline. Scheduling both *bx* and *in* to be computed at the granularity of tiles of *by* means that any parallel execution of the tiles of *by* implicitly also evaluates *bx* and *in* within the same parallel tasks:

```
// parallel loops include in, bx
parallel for $by.y_o$
  parallel for $by.x_o$
    for $in.y$
      for $in.x$
        compute $in(in.x, in.y)$
    for $bx.y$
      for $bx.x$
        compute $bx(bx.x, bx.y)$
    for $by.y_i$
```

```
        for $by.x_i$
            compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

This occurs even though the domain order of neither *in* nor *bx* specifies any of their dimensions to be evaluated in parallel. Conversely, parallel execution of *by within* the granularity at which *in* and *bx* are computed creates finer-grained parallel tasks which do not include computation of *in* or *bx*, leaving them evaluated sequentially:

```
for $by.y_o$
    for $by.x_o$
        for $in.y$
            for $in.x$
                compute $in(in.x, in.y)$
        for $bx.y$
            for $bx.x$
                compute $bx(bx.x, bx.y)$
        // parallel loops do not include in, bx
        parallel for $by.y_i$
            parallel for $by.x_i$
                compute $by(by.x_o \times 4 + by.x_i, by.y_o \times 4 + by.y_i)$
```

Changing storage layout

Halide's schedules also allow controlling the mapping of each function's logical domain into linear memory addresses. This is done using the `reorder_storage` operator, defined much like the `reorder` operation on the domain order, but instead defining the precedence of dimensions in the storage layout. Storage layout is always defined in terms of the intrinsic dimensions of a function[11]. The default storage order is identical to the default domain order: the left-most dimension is innermost (the least-significant bits of the storage index), and the right-most dimension is outermost.

11. There is currently no equivalent to the *split* operator for storage layout. This is a straightforward extension, but has not yet proven important in practice. Contrary to historical belief, tiled *storage layouts*, in the program's virtual address, space are rarely important for common image sizes on modern processor architectures and memory systems.

For example, consider the 2D function $f(x, y)$. Regardless of its domain order, addresses in memory storing values of f will by default map 2D x, y coordinates into 1D offsets of the form $y \times f.x.extent + x$. That is, values of f are stored in *row-major order*. This can be changed to a *column-major order*—without changing any aspects of the domain order of f, its call schedule, or the schedules of its consumers or the functions it consumes—with the operation $f.reorderstorage(y, x)$.

5.2 SCHEDULING ITERATION DOMAINS

The biggest limitation of Halide's current model of organization is its dependence on the properties of pure functions. Specifically, the freedom to split, reorder, and parallelize dimensions depends on the *independence* of a pure function's meaning at each point in its domain. Iteration domains, as defined in Halide, do not share this property: because of the recursive nature of the *update* step, the final value of an iterative function depends on earlier computations in the iteration[12]. This means that we cannot safely allow the same general changes in the order of iteration dimensions as in the case of pure functions. (In practice, we allow changes, but warn the user that they may be unsafe unless we can statically prove otherwise.)

12. The update step can be viewed as a loop nest over the reduction domain which mutates the values of the function, at a mixture of pure and computed coordinates, in each iteration.

Much like pure functions, functions defined in multiple update steps default to being computed as required for the *innermost* dimension of the calling function. However, according to the semantics, they must evaluate all required values of any the computed coordinates of their own output domain (effectively breadth-first within their own update domain) within each iteration in their iteration domain.

```
// pure dimensions outermost
for $y$
    // reduction dimensions
    for $r.x$
        // free variables used in computed coordinates
        for $x$
            // update $f(y, x + r.x)$
```

Reorganizing general iterative updates

Reorganizing reduction functions more generally *is* often feasible, using generalizations of the same primitives described here. For example, in the case of associative reductions, splitting and parallelizing dimensions of the reduction domain requires duplicating the intermediate storage for each parallel task and then recombining the results from each.[13] Where splitting a pure variable produces two pure variables, splitting a reduction domain variable produces a pure outer variable, whose iterations are independent and whose storage is dupli-

13. The common case of point-wise reductions, applied independently to every pixel in the output domain, is very often associative. This pattern is built into the language via simple macros for point-wise *sum*, *product*, *minimum*, *maximum*, *argmin*, and *argmax*.

cated, and an iterative inner variable which walks over a subset of the prior reduction domain.

Reorganization becomes more complex in the case of general iterative updates with recursive dependence simultaneously across both pure dimensions (space) and the iteration domain (time), but is still feasible for many classes of scans and other operations. Formalizing our reorganization primitives for iterative reduction domains remains future work.

5.3 SCHEDULING GPU EXECUTION

Halide's schedules map portions of computation to GPU execution explicitly. There is a straightforward isomorphism between Halide's loop nests over regular rectangular grids and the grids and blocks in GPU compute programming models [16, 64]. Schedules describe GPU execution by labeling specific loops to correspond to specific GPU thread and block dimensions. Scheduling macros including `gpu(var)` and `gpu_tile(x, y, w, h)` provide syntactic sugar for common forms of this transformation.

Computation outside loops mapped to thread blocks map to host execution exactly as before; computation inslide each loop nest mapped to GPU threads are compiled into a corresponding GPU kernel and the host-side runtime logic to launch it over the required domain and manage its data. Multiple functions may be computed inside a single set of blocks by computing them at the granularity of the innermost thread block. This generates phases of thread-level computations separated by local, block-level barriers between sets of thread loops. Memory allocated at block granularity is mapped to the GPU's block-level scratchpad ("shared memory"). This mapping is shown in more detail, by example, in Appendix A.12.

6 ORGANIZING IMAGE PROCESSING PIPELINES WITH HALIDE SCHEDULES

This chapter explores how Halide's schedule primitives—its two-part model for organizing computation in image processing pipelines—can be composed to express a wide range of optimized organizations. The key underlying principle is the deep connection between computation and storage granularity, and the fundamental tension between parallelism, locality, and redundant work. For single-producer, single-consumer pipelines—or, more generally, the relationship between a single pair of stages—there is a direct correspondence between the valid half of the 2*D* space of computation and storage granularities, and the three qualities we wish to optimize: the extremes of the space each completely sacrifice one metric, while the interior of the space balances them all differently (Figure 6.1). In this sense, the space of compute and storage granularities is, in fact, *isomorphic* to the space of tradeoffs between parallelism, locality, and redundant work. This relationship becomes more complex for real applications, which contain many stages; the computation and storage granularity can be different for each stage, schedule choices interact across stages, and the overall balance is determined by the global organization. Finally, in Section 6.2, I will discuss two ways of finding *good* schedules: manual exploration, often by an optimization expert, and fully automatic search using autotuning.

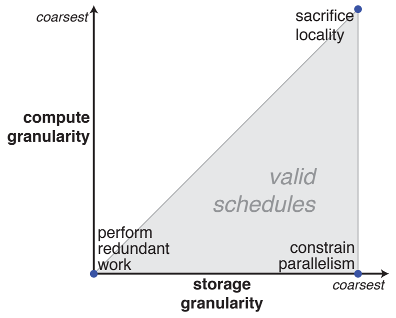


Figure 6.1: The space of schedules directly corresponds to the space of tradeoffs between parallelism, locality, and redundant work.

6.1 SCHEDULING A SINGLE PAIR OF FUNCTIONS

To start, we will look more deeply at the tradeoffs in a version of our recurring blur example, simplified by reduction to one dimension for easier visualization:

$$blur_1(x) = in(x-1) + in(x) + in(x+1)$$
$$blur_2(x) = blur_1(x-1) + blur_1(x) + blur_1(x+1)$$

Extreme points in the space of schedules

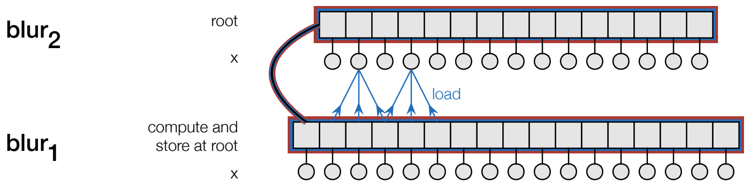
The obvious starting points are the extremes of the choice space for how the first stage ($blur_1$) can be interleaved with the second ($blur_2$):

- coarse-grained computation and storage interleaving,
- fine-grained computation and storage interleaving, or
- fine-grained computation interleaving with coarse-grained storage interleaving.

Breadth-first organization sacrifices locality.

Assuming a default domain order for each function (y outside x), $blur_1$ may first be computed and stored at the coarsest granularity (*root*):

```
allocate $blur_1$
for all $x = -1$ to $w + 1$
    $blur_1[x] = in[x-1] + in[x] + in[x+1]$
allocate $blur_2$
for all $x = 0$ to $w$
    $blur_2[x] = blur_1[x-1] + blur_1[x] + blur_1[x+1]$
```



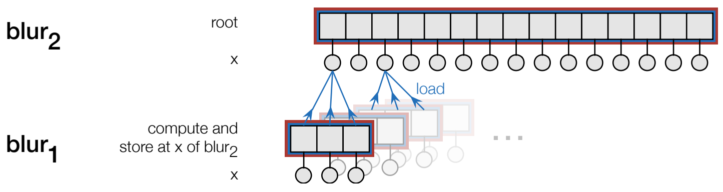
This computes and stores an entire intermediate image containing all results of $blur_1$ prior to computing any pixels in $blur_2$. This requires storage for an entire intermediate image throughout the pipeline, and it introduces a long ($\approx n$) reuse distance between where each value is computed in $blur_1$ and where it is used in $blur_2$. Still, it computes each value only once, performing no redundant work, and it allows total freedom to execute the computations in each stage in parallel. A simple choice of parallel domain order for a modern multicore would split each

x dimension to some granularity (`blur`$_i$`.split(x, tile_width, xo, xi)`), compute the outer dimension in parallel threads (`blur`$_i$`.parallel(xo)`), and vectorize the innermost component of the x dimension (`blur`$_i$`.vectorize(xi, vector_width)`). This would give relatively coarse-grained parallel tasks ($\approx$ *tile width*), but more than enough of them (also $\approx n/\textit{tile width}$) to saturate a reasonably-sized machine.

Fine-grained computation and storage wastes work.

At the other extreme, $blur_1$ may be computed and stored at the innermost granularity of $blur_2$ ($blur_2.x$ or *inline*):

```
allocate $blur_2$
for all $x = 0$ to $w$
    allocate $blur_1$
    for all $x' = x - 1$ to $x + 1$
        $blur_1[x'] = in[x' - 1] + in[x'] + in[x' + 1]$
    $blur_2[x] = blur_1[x - 1] + blur_1[x] + blur_1[x + 1]$
```



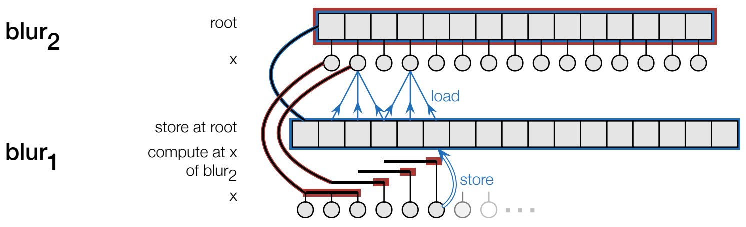
This computes each value of $blur_1$ immediately before consuming them to compute the corresponding value of $blur_2$, giving a maximum use distance of just three tasks, maximizing producer-consumer locality. Each value is freed as soon as it is consumed, requiring a storage footprint of three pixels for intermediate results of $blur_1$. Parallelism is unconstrained: the domain of $blur_2$ can be parallelized and vectorized arbitrarily, and the corresponding parallel tasks will include parallel computation of intermediate values from $blur_1$. Most significantly, however, the total amount of work is tripled: each value in $blur_1$ is computed, consumed, and thrown away by each of three separate consumers in $blur_2$. This is the cost of maximizing locality (minimizing use distance) while leaving no reuse dependence between tasks in $blur_2$.

Sliding-window interleaving constraints parallelism.

Lastly, we can compute $blur_1$ at the innermost granularity, while storing it for reuse at the coarsest granularity[1]:

1. compute at *by.x*; store at *root*.

```
allocate $blur_1$, $blur_2$
for $x = 0$ to $w$
    $x'_{min}$ = if $(x = 0)$ then $x - 1$ else $x + 1$ // skip already computed
    for all $x' = x'_{min}$ to $x + 1$
        $blur_1[x'] = in[x' - 1] + in[x'] + in[x' + 1]$
    $blur_2[x] = blur_1[x - 1] + blur_1[x] + blur_1[x + 1]$
```



This couples the execution of neighboring tasks in $blur_2$ which reuse common values in $blur_1$. This increases both the storage footprint and the maximum use distance for values in $blur_1$ by a small constant factor (≈ 3).[2] Values are still used in $blur_2$ nearly as soon as they are produced in $blur_1$, but they are computed only once and reused. This performs no redundant work, and retains very short use distance between tasks in $blur_1$ and their use in $blur_2$, but to do so it requires coupling the execution of neighboring tasks in $blur_2$ to exploit reuse. This constrains the order of execution: if neighboring tasks in $blur_2$ are to share inputs from $blur_1$ with minimal use distance, they must be executed together. Because of the overlapping nature of neighboring stencils, this order dependence propagates across the entire domain of $blur_2$, requiring it to be executed in sequential order. This constrains the possible parallelism in $blur_2$ and increases the critical path by $\approx n$.

2. Throughout this chapter, I assume automatic discovery of *storage folding* opportunities, as performed by the Halide compiler and described in Chapter 8. Storage folding only affects footprint, not use distance.

The space of schedules spans the space of tradeoffs fundamental to image processing pipelines.

These three organizations represent extremes of both the space of possible producer-consumer organizations for this two-stage pipeline, viewed along the axes of *computation* and *stor-*

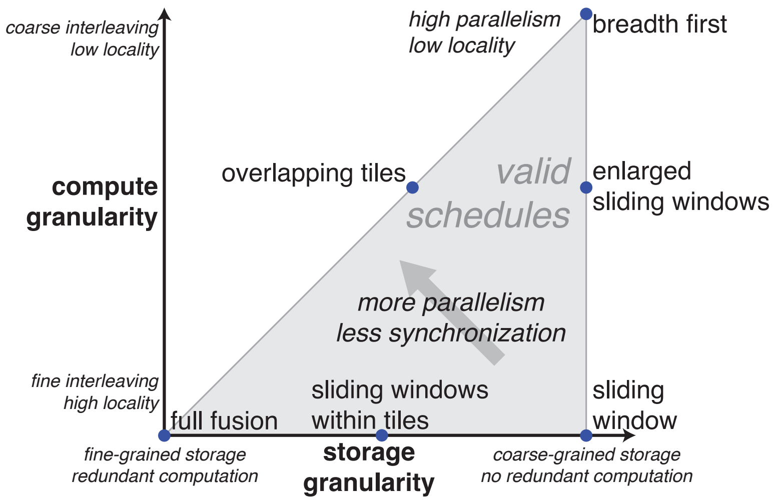


Figure 6.2: A natural way to visualize the space of scheduling choices is by granularity of storage (x-axis), and granularity of computation (y-axis). Breadth-first execution does coarse-grain computation into coarse-grain storage. Total fusion performs fine-grain computation into fine-grain storage (small temporary buffers). Sliding window strategies allocate enough space for the entire intermediate stage, but compute it in in fine-grain chunks as late as possible. These extremes each have their pitfalls. Breadth-first execution has poor locality, total fusion often does redundant work, and using sliding windows to avoid redundant recomputation constrains parallelism by introducing dependencies across loop iterations. The best strategies tend to be mixed, and lie somewhere in the middle of the space.

age granularity defined by our call schedule, and the space of tradeoffs between locality, redundant work, and parallelism:

- The *breadth-first* schedule both *stores* and *computes* $blur_1$ at the coarsest granularity (which we call the *root* level—outside any other loops). It computes every required value exactly once, and it does not constrain possible parallelism, but it completely sacrifices producer-consumer locality between stages.
- The *fused* schedule both *stores* and *computes* $blur_1$ at the finest granularity, inside the innermost (x) loop of $blur_2$. At this granularity, values are produced and consumed in the same iteration, maximizing locality, but they must be reallocated and recomputed on each iteration, independently, introducing redundant computation.
- The *sliding window* schedule *stores* at the *root* granularity, while *computing* at the finest granularity. With this interleaving, val-

ues of $blur_1$ are computed in the same iteration as their first use, but persist across iterations. To exploit this by reusing shared values in subsequent iterations, the loops between the *storage* and *computation* levels must be strictly ordered, so that a single unique first iteration exists for each point, which can compute it for later consumers. This precludes executing iterations in parallel; it maintains locality and avoids redundant computation at the expense of parallelism.

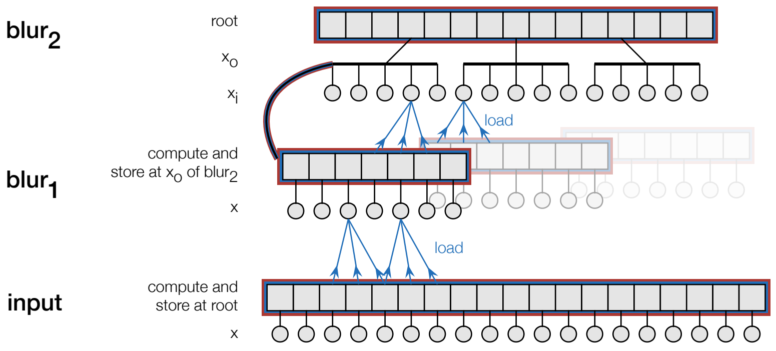
Composing organizations

These tradeoffs can be balanced by further composing our scheduling primitives to express compound organizations.

Interleaved tiles trade redundant work for locality.

For example, The first organization we considered for the blur pipeline in the previous chapter split the output domain into independent tiles, and interleaved the computation of intermediate stages at the granularity of those tiles. This is a common organization in optimized software image processing pipelines. This organization applies similarly in 1*D*. First we tile the domain of the final function by a tile width parameter (`tw`): `blur₂.split(x, xo, xi, tw)`. Then, we *compute and store* each intermediate function at the granularity of tiles of the output: `blur₁.compute_at(blur₂, xo)`. This completely decouples tiles of computation and storage. Tiles may be computed in parallel (`blur₂.parallel(xo)`). The resulting parallel tasks have granularity defined by the size of the tiles. Internally, tiles are also free to exploit fine-grained data parallelism (e.g., by *vectorizing* $blur_2.x_i$ and $blur_1.x$). Use distance and storage footprint for each task is constrained to the size of the tile ($\approx tilewidth$).

```
**allocate** $blur_2$
**for all** $x_o = 0$ **to** $w/tw$
    **allocate** $blur_1$
    **for all** $x' = x_o \times tw - 1$ **to** $(x_o + 1) \times tw - 1 + 1$
        $blur_1[x'] = in[x' - 1] + in[x'] + in[x' + 1]$
    **for all** $x_i = 0$ **to** $tw$
        **let** $x' = x_o \times tw + x_i$
        $blur_2[x'] = blur_1[x' - 1] + blur_1[x'] + blur_1[x' + 1]$
```


blur2
root
x_o
x_i
load
compute and store at x_o of blur2
blur1
x
load
compute and store at root
input
x

The tradeoff for the controlled locality and storage footprint, and the decoupled parallel execution across tiles, is redundant computation of overlapping values on the boundary between tiles. These tradeoffs are balanced by changing the granularity of tiles in the pipeline: large tiles minimize the fraction of redundant computation and provide low parallel scheduling overhead from coarse-grained tasks at the expense of locality and storage, while small tiles minimize use distance and storage footprint at the expense of more redundant computation. The amount of excess computation introduced is $\approx \frac{\textit{stencil width}-1}{\textit{tile width}}$ (in this case, $\frac{2}{\textit{tile width}}$).

Enlarged sliding windows trade locality for fine-grained parallelism.

We can open up opportunities for fine-grained parallelism in a sliding window organization by interleaving computation at a coarser granularity. In the 1*D* blur, this corresponds to splitting the second stage into segments of width `sw` (`blur₂.split(x, xo, xi, sw)`), computing the first stage at the granularity of these segments (`blur₁.compute_at(blur₂, xo)`), and storing it for reuse globally (`blur₁.store_root()`).

```
allocate $blur_1$, $blur_2$
for $x_o = 0$ to $w/sw$
    $x'_{min}$ = if $(x_o = 0)$ then $x_o \times sw - 1$ else $x_o \times sw + 1$
    for all $x' = x'_{min}$ to $(x_o + 1) \times sw + 1$
        $blur_1[x'] = in[x' - 1] + in[x'] + in[x' + 1]$
    for all $x_i = 0$ to $sw$
        $x = x_o \times sw + x_i$
        $blur_2[x] = blur_1[x - 1] + blur_1[x] + blur_1[x + 1]$
```

Compared to pixel-level sliding window interleaving of the two stages, this organization increases the use distance and storage footprint from three pixels to being proportional to the segment width ($\approx$ *segment width*). It still requires sequential iteration over interleaved segments, but it has several advantages over the per-pixel sliding window order. First, it tracks values for reuse at coarser granularity, which can have lower overhead than interleaving computation per-pixel. Second, because of this, it exposes fine-grained data-parallelism ($\approx$ *segment width*) *within* each segment; while scanlines must be executed sequentially, pixels within each scanline need not be. This fine-grained parallelism is naturally exploited on many processors by vectorization.

Sliding windows within tiles trade redundant work for coarse-grained parallelism.

We can expose coarse-grained parallelism by starting from a fine-grained sliding window schedule and moving in the opposite direction. Again splitting the second stage into segments(`blur₂.split(x, xo, xi, sw)`), but computing the first stage at the innermost granularity (`blur₁.compute_at(blur₂, xi)`), and storing it for reuse at the level of segments (`blur₁.store_at(blur₂, xo)`) performs fine-grained interleaving within a each of several segments, but decouples the computation of the segments by redundantly computing a fringe along their boundaries.

```
**allocate** $blur_2$
**for all** $x_o = 0$ **to** $w/sw$
    **allocate** $blur_1$
    **for** $x_i = 0$ **to** $sw$
        $x = x_o \times sw + x_i$
        $x'_{min} =$ **if** $(x = 0)$ **then** $x-1$ **else** $x+1$ // *skip already computed*

        **for all** $x' = x'_{min}$ **to** $x+1$
            $blur_1[x'] = in[x'-1] + in[x'] + in[x'+1]$
        $blur_2[x] = blur_1[x-1] + blur_1[x] + blur_1[x+1]$
```

Compared to pixel-level sliding window interleaving of the two stages, this organization retains the same minimal use distance, and still requires sequential iteration over the individual pixels within each segment, but it decouples execution across segments. Because of this, it exposes coarse-grained parallelism ($\approx \frac{width}{segment\ width}$) *across* segments. This coarse-grained parallelism is naturally exploited on many processors by multithreading.

Line-buffering is a special case of enlarged and tiled sliding windows.

A common variant of these organizations is widely used in image processing pipelines containing only small stencils, like the camera pipeline in mobile image signal processors (ISPs). A special case of enlarged sliding window interleaving, this organization is often called "line-buffering" [53, 83] and becomes clearer in 2*D*, so I will briefly return to the original 2*D* blur algorithm, consisting of functions *bh* and *bv*.

In a line-buffered organization, the granularity of interleaving is enlarged by one loop level, and intermediate data is buffered between stencil stages at the granularity of scanlines, with the minimal number of scanlines required to support each stage's stencil stored in circular buffers which slide down the image in a synchronous fashion between all stages. In the Halide representation, this organization is modeled by interleaving *computation* at the granularity of *scanlines* (here, computing *bh* at *bv.y*), while *storing* values for reuse at the *root* granularity:

```
allocate $bh$, $bv$
for $y = 0$ to $h$
    $y'_{min}$ = if $(y = 0)$ then $y - 1$ else $y + 1$ // skip already computed
    for all $y' = y'_{min}$ to $y + 1$
        for all $x = 0$ to $width$
            $bh[x, y'] = in[x - 1, y'] + in[x, y'] + in[x + 1, y']$
    for all $x = 0$ to $width$
        $bv[x, y] = bh[x, y - 1] + bh[x, y] + bh[x, y + 1]$
```

This design pattern is common in hardware image processing pipelines, where fine-grained control is relatively inexpensive, and abundant fine-grained data parallelism is sufficient to keep even very wide pipelines busy.

Multicore general-purpose processors, by contrast, often also require coarse-grained parallel tasks, with minimal coupling between them, to distribute across far-away cores. Even a single fixed-function hardware pipeline might want to limit the maximum size of scanlines buffered between stages, decoupling the execution beyond this granularity to limit buffering of intermediate data. In both cases, the line-buffering organization can be modified to introduce some redundant computation to decouple coarse-grained parallel tasks and control use distance and buffer footprint across stages. We can express this transformation by splitting and reordering the output domain to be computed in strips (*split* $by.x \rightarrow (by.x_o, by.x_i)$ by *strip width*; *reorder* $by.x_i$, $by.y$, $by.x_o$). Within each strip, the pipeline is still locally line-buffered (compute *bx* at $by.y$, and store for reuse at $by.x_o$), but across strips intermediate results are recomputed rather than being stored for reuse; strips are free to be executed in parallel (*parallel* $by.x_o$) . This limits producer-consumer use distance and buffer footprint to $\approx 3 \times$ *stripwidth*. It still allows $\approx$ *stripwidth* fine-grained data parallelism, but also yields $\approx$ *numstrips* coarse-grained parallel tasks. Decoupling strips introduces redundant work where stencils overlap on strip boundaries (here, $\approx 2 \times$ *numstrips* $\times \sqrt{n}$).

6.2 FINDING GOOD SCHEDULES

Given our model of the organization of computation, the last question that remains is: how can we determine *good* schedules? We have worked on two strategies: optimizing the schedule by hand, and automatically searching the space of possible schedules using autotuning.

Hand-optimized schedules

Developers can manually specify schedules as part of their Halide program. This was our focus for the first year of Halide's existence, and it remains the most common practice. Even with fully hand-tuned schedules, exploring a wide range of potentially fruitful organizations using our terse and composable description of algorithms and schedules is much easier than manually reorganizing computation in a traditional language like C or CUDA. This is for several reasons.

First, Halide's representation of the algorithm is much more terse, both because of domain-specific syntax, and because it ignores order of execution, and all bounds and memory allocations are implicit, and inferred automatically.

Second, the representation of even a complex optimized schedule is terse relative to the equivalent C. The description of common patterns like vectorization and multithreaded parallelism is extremely compact[3]. Sensible defaults for domain order and interleaving, combined with inferred indexing and bounds computation even for complex tiled and interleaved sequences of stages, mean that the schedule only specifies non-obvious choices in the organization. Even beyond this, real scheduling code is made even more compact by the combination of syntactic sugar for common patterns (e.g., the tile operator), and host-language metaprogramming to simultaneously apply related schedule directives to related functions.

3. `f.parallel(dim)`
`f.vectorize(dim, width)`

Third, even radical changes in organization expressible by our schedule operators do not change the algorithm code, or the resulting computation, at all. The algorithm generally represents the large majority of total code to describe an optimized Halide pipeline (around 75 – 90% in our benchmark suite, presented in Chapter 9). It *is* sometimes necessary to change the intrinsic algorithm definition during optimization, both to tune the actual computation as bottlenecks are revealed, and to better enable different organizations, but in practice we have found, once we have a good initial algorithm, the large majority of time optimizing is spent simply tuning the schedule.

Finally, changes in the schedule are often orthogonal to one another, and the generated code is correct by construction. It is often possible to rapidly explore many different organizations simply by changing one or a few schedule directives, re-compiling, and re-running a performance test.

Manual search

In practice, the way we generally do manual optimization in Halide proceeds in two steps.

First, relatively simple intuition and design patterns go a long way in pointing towards likely fruitful organizations. A common starting point is simply to inline functions with point-wise dependence, tile and interleave functions with small stencils, and compute functions with large stencils or unpredictable dependence patterns. In addition, for conventional multicores, we generally begin vectorizing the innermost dimensions of most functions, and parallelizing outer dimensions at relatively coarse granularity.

Second, given that starting point, we quickly test different hypotheses to improve the performance of the pipeline by changing features of the organization and re-running benchmarks. Effectively, this process is a heuristic search of the space of possible schedules, guided by expert intuition. The performance space for complex pipelines on modern architectures is complex and unpredictable, so, while this intuition is valuable, it is very common to be surprised by the performance of many seemingly promising optimizations.

A detailed discussion of our experience manually tuning the local Laplacian filters pipeline is presented in Chapter 9.

Portability

A key advantage of Halide's split representation is portability. The best optimized organizations vary from one architecture to another. Variation among general-purpose multicores, with different memory hierarchies, degrees of parallelism, and compute to bandwidth ratios can be modest; the difference between the best organization for more widely varied architectures, like mobile multicores and workstation GPUs, is often enormous. Reorganizing computations and data in a traditional language often requires rewriting nearly the entire code.

With the algorithm strictly separated from choices of organization encoded in the schedule, optimized implementations for many different targets can share the same algorithm code, while using different schedules to express different organizations for each target. In practice, we find that ranges of different general-purpose multicores (e.g., small ARM mobile SoCs, single-socket x86 PC CPUs, and large multi-core, multi-socket x86 server CPUs) are often well targeted by a common family of organizations, sharing most scheduling code, but with some parameters (e.g., vector width, unrolling factors, tile sizes) changing between targets. GPUs, by contrast, usually demand very different organizations, often relying less on staging intermediate data through caches and more on global memory, as well as often allowing a larger share of redundant computation to save bandwidth.

Autotuning

We can also automatically search the space of schedules using auto-tuning. For the same reasons that our schedule representation enables far easier, more rapid manual exploration of possible optimizations, it provides a natural representation in which to *automatically* search for efficient organizations. The space is far too large to search exhaustively[4]. We have focussed on stochastic search. Our autotuner is implemented using the OpenTuner framework .

4. For example in the local Laplacian filters pipeline, we estimate a lower bound of 10^{720} schedules. This is derived by labeling functions with three tilings per function and all possible store and compute granularities. The actual dimensionality of the space is likely much higher. The optimal schedule dependends on machine architecture, image dimensions, and code generation in complex ways, and exhibits global dependencies between choices due to loop fusion and caching behavior.

There are two key challenges in automatically searching the space of schedules.

Representing schedules for autotuning

The first challenge is representing and enumerating the space of possible organizations in a way that enables rapid exploration of all useful choices while also enabling heuristic search strategies like genetic optimization. The operators of our schedule algebra provide a natural foundation.

Concretely, many heuristic search algorithms require some notion of *combining* different points in the search space—often a linear (weighted) combination. This operation is sufficient to implement many genetic optimization algorithms and other heuristics. Our current implementation maps into the existing partially-ordered list parameter type in OpenTuner, which natively supports these operations, and so supports many different heuristic search strategies.

We map complete Halide schedules to OpenTuner primitives as follows. We map each loop of each function into a pair of tokens in the sorted list, and map the computation of each function into an additional token. We define partial-order constraint edges between the computation nodes of each pair of functions with a direct caller-callee relationship, and between each function's computation node and the open and close tokens of its loops (which must happen before and after, respectively). Loop type (parallelism, unrolling, vectorization) is a conventional discrete selection selection parameter for each loop. We also add a normalization step which ensures all schedule lists map to valid schedules.

As future work, we hope to explore direct search over algebraic transformations of the schedule tree representation defined in the previous chapter.

Domain-specific search heuristics

The second challenge is steering the search quickly towards useful parts of the space. We have worked on a number of domain-specific heuristics to bootstrap the search process.

One valid starting schedule is to label all functions as computed and stored breadth-first (at the outermost, *root* granularity). The tuner converges from this starting point, albeit slowly. We can often do better by seeding the initial population with reasonable schedules. For each function we find its rectangular footprint relative to the caller (via bounds inference) and inline functions with footprint one. Remaining functions are stochastically scheduled as either:

1. *fully parallelized and tiled*, or
2. simply parallelized over y.

We define *fully parallelized and tiled* as tiled over the first and second (x and y) dimensions, vectorized within the tile's inner (x) coordinate, and parallelized over the outer (y) tile dimension. This allows us to often discover good starting points for functions that vectorize well, or fall back to naive parallelism when that is not the case. The dimensions x and y are chosen from adjacent dimensions at random, except when there are optional bounds annotations provided by the Halide programmer (such as the number of color channels); dimensions with small bound are never tiled.

6.3 SUMMARY

In this chapter, I have shown how Halide's schedules can simply describe many different organizations of computation for image processing pipelines. The most important detail not to lose in my often simplified examples is that schedules are not specified once for a whole pipeline; rather, real pipelines are compositions of many functions, and these choices are made separately *for every function*, and together they determine how these many functions globally interact. Finally, I discussed the current methods by which users can find *good* schedules for their Halide programs: manual tuning, enabled by our compact and expressive model combined with automatic code synthesis, and autotuning using stochastic search.

7 AN ALGEBRA OF SCHEDULES

The key features of Halide's model of schedules are their *composability*, their *decoupling* from the intrinsic algorithm, and the ability to *transform* one schedule into another. In this chapter, we define an algebra on transformations of schedules. This algebra is largely equivalent to Halide's language of schedules, but it is not the same. This chapter focuses on a *transformational* perspective, and its algebraic properties, while the Halide language specifies call schedules *declaratively*, which is significantly more terse, but harder to reason about systematically.

7.1 REPRESENTING SCHEDULES

We represent the space of schedules as a family of trees, corresponding to the semiperfect loop nests that will compute the requested values of a given pipeline. A schedule tree has nodes of several types: *loop* nodes, *store* nodes, *compute* nodes, and a *root* node. We define them as an algebraic data type:

```
node = loop node
       | compute node
       | store node
       | root
root = { children : node list }
loop node = {
    func : Func
    var : Var
    type : enum { sequential | parallel | vectorized | unrolled }
    children : node list
}
```

```
store node = {
    func : Func
    children : node list
}
compute node = {
    func : Func
    children : compute node list
}
```

A schedule tree is always anchored by exactly one *root* node corresponding to its root, the outermost scope of the generated pipeline. Below this are recursive subtrees defining the storage allocation, loops, and computation of each function in the pipeline. Importantly, the children of each node are encoded as a *list* since they are ordered; the meaning of a node depends on the order of its children.

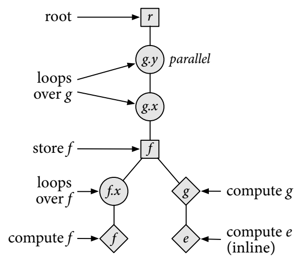


Figure 7.1: Schedule tree for a simple pipeline.

All non-root nodes are associated with a specific function. Loop nodes specify the variable in the domain of the function to which they correspond. They also carry a *type* attribute describing the nature of the loop over this dimension (whether it should be sequential, parallel, vectorized, or unrolled).

Store nodes specify the point in the tree at which a particular function is stored for reuse. Only the children of a function's store node may reference it, either to compute or consume its values (except in the special case of *inline* functions).

Compute nodes are generally terminal, forming the leaves of the tree where useful work is actually done. They are allowed to have only other compute nodes as children, to encode the special call schedule of *inline*, in which a called function's own domain is not traversed, but rather each call site is directly replaced by the definition of the function.

Each function in the algorithm has exactly one *compute node*. If it is computed *inline*, it has no other associated nodes. Otherwise, it has exactly one *store node*, and one loop node for each variable in its domain order (i.e., for each dimension of the function, after dimension splitting and fusion have been applied).

A function's store node must be an ancestor of its loop nodes and the compute nodes of any of its calling functions. A function's compute node must be a descendant of its loop nodes.

The output function, by definition, cannot be computed *inline*, so it always has a full set of loop nodes. By convention, the output function has no *store node*, since its storage is allocated and passed in by the caller; it is stored at *root* granularity, by definition.

An example schedule tree

Consider as an example the two-stage blur algorithm we have studied before. The tiled and interleaved schedule of these two stages can be represented as a simple tree with loop nodes for each of $blury.y_o$, $blury.x_o$, $blury.y_i$, $blury.x_i$, $blurx.y$, and $blurx.x$. The storage of *blurx* is the immediate child of the $blury.x_o$ loop. Below that, the loops over *blurx*'s and *blury*'s inner dimensions, and the computation of each function, are peer subtrees, with the *blurx* subtree before the *blury* subtree, to respect producer-consumer ordering.

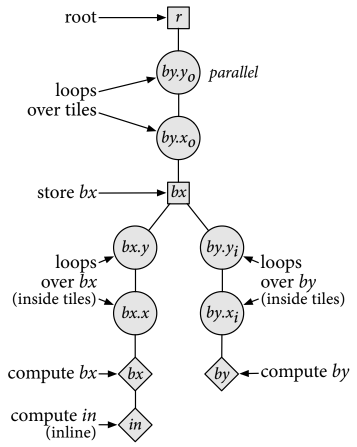


Figure 7.2: Schedule tree for tiled blur.

Legality rules

Within this schedule tree representation, several constraints must be met for a schedule to be legal. These are determined by the space of *meaningful* choices which can actually compute an algorithm as specified. In particular:

- A function must be computed before it is consumed: a function's compute node must occur before the compute nodes of its calling functions in a depth-first traversal of the schedule tree.
- Storage must be allocated and in scope to be used: a function's store node must be an ancestor both of its compute node and of its callers' compute nodes.
- Beyond the rules specified in the definitions above, constraints of practical code generation make certain patterns illegal. In particular, we only allow vectorization of an innermost loop (a loop node which is not the ancestor of any other loop nodes), and we only allow vectorization and unrolling of constant-width loops.

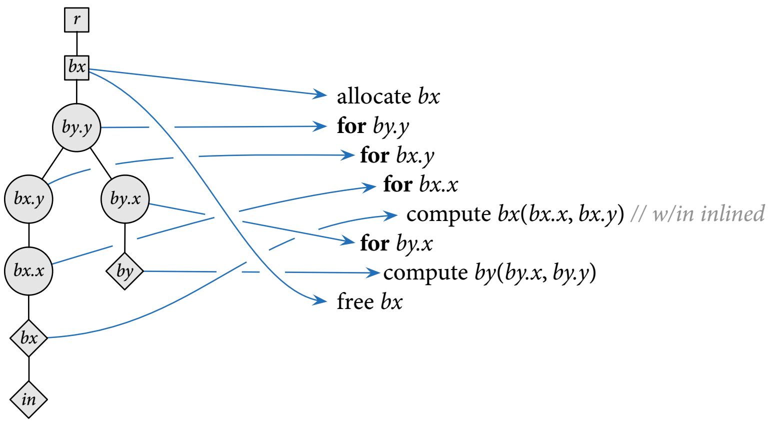
The semantics of schedules

The semantics of the schedule tree define its mapping into a loop nest, including stack-style allocation and deallocation of intermediate storage for each function. They are as follows.

The tree is visited in depth-first order, starting from the *root*, respecting the order of the child lists at each node. For each node,

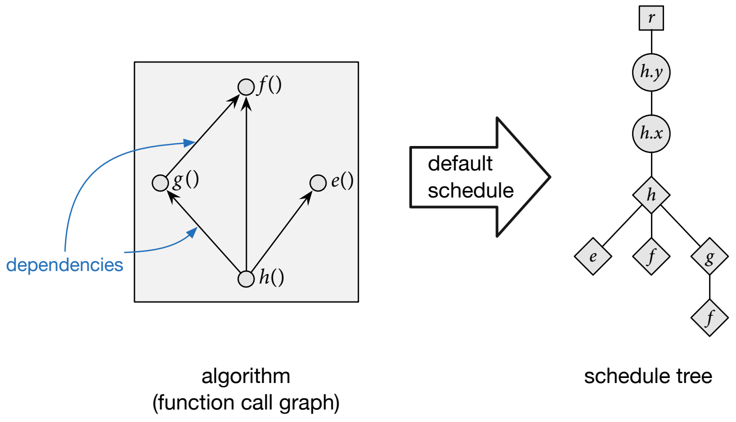
- if it is a *loop node*, it opens the corresponding loop with the appropriate attributes (parallel, vectorized, unrolled), recursively visits each child in order, and closes the corresponding loop;
- if it is a *store node*, it allocates the corresponding storage, recursively visits each child, and deallocates the storage;
- if it is a *compute node*, it computes the value of the corresponding function at the location in its domain defined by its loops and stores the value in its allocated storage, or, if it is inline, simply returns the required value to the appropriate site in the calling compute node's definition.

Considering again the tiled schedule for the two-stage blur algorithm, the semantics of the schedule tree map to the following loop nest:



The starting point: from an algorithm to a default schedule

Since this algebraic model of schedules is transformational, we need a well-defined starting point. The default schedule maps the dependence graph of functions in a Halide algorithm to its initial schedule tree. The default schedule has a root node, above a series of loop nodes for each dimension of the output function, from outermost to innermost, followed by a compute node for the output function. Every other function is scheduled to be computed *inline* by default, so the subtree rooted at the compute node for the output function directly mirrors the call graph of the original algorithm, with each function f called in a function g represented by a further subtree beginning at f. A function with multiple callers has has the subtree rooted at its compute node duplicated as an immediate child of each caller's compute node.



7.2 SCHEDULE TRANSFORMATIONS

To traverse the space of possible organizations, we define a family of operators with which to transform schedule trees.

Split variable

split(func, var) splits one variable in the tree into two. This replaces a single loop node into a pair of loop nodes, with the inner node (farther from the root) inheriting the children of the original node, the outer node (closer to the root) replacing the original node as the child of its parent, and the inner node as the only child of the outer node.

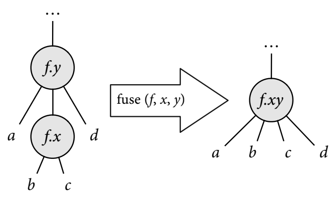


Figure 7.3: Split $f.x$. Note that each variable is uniquely identified using not just the variable name, but also the function in which it is used.

Fuse variables

fuse(func, var, var) merges two adjacent loop nodes from the same function in the tree into one loop node. The new node remains at the same location in the tree as the original outer loop node, and the children of each are concatenated, with the children of the original outer variable coming before the children of the original inner variable.

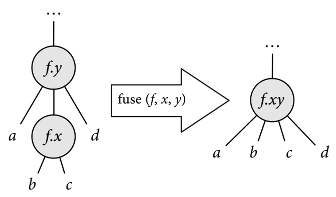


Figure 7.4: Fuse $f.x$ and $f.y$.

Reorder variables

reorder(func, var list) reorders (a subset of) the loop nodes associated with a given function. It simply exchanges the variables associated with each node in the list to match the desired order. Topology is unchanged, and for simplicity, the loop types are left in their original, topological locations, rather than being carried with the variable to which they had first been applied.

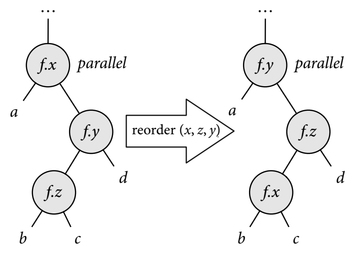


Figure 7.5: Reorder three variables of f.

Change loop type

loop type(func, var, type) changes the type of a given loop to one of the set *sequential*, *parallel*, *unrolled*, or *vectorized*. This simply changes an attribute on the corresponding loop node.

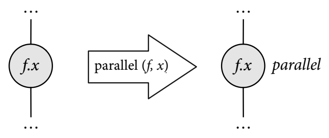


Figure 7.6: Change loop type of *f.x* to *parallel*.

Hoist computation

hoist compute(func) coarsens the granularity at which a function is computed by moving the subtree beginning at its outermost loop node one generation closer to the root of the tree. A function's computation cannot be hoisted past its storage node; in general, the subtree rooted at the outermost loop of a function cannot be hoisted past the storage nodes of *any* functions whose compute nodes are within the subtree.

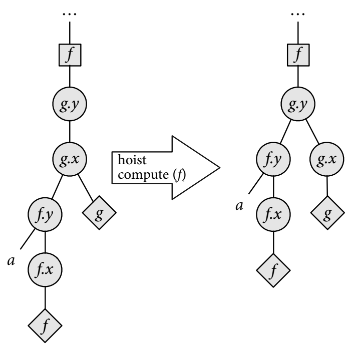


Figure 7.7: Hoist computation of *f*.

Lower computation

lower compute(func) is the inverse operation of *hoist compute*. It moves the subtree rooted at the function's outermost loop one generation farther from the root, closer to its consumer, making finer the granularity at which the function is computed.

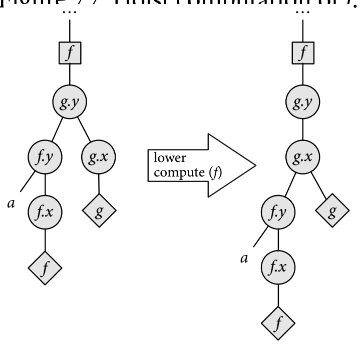


Figure 7.8: Lower computation of *f*.

Lowering is only legal when all functions computed within the corresponding subtree are consumed only within the subtree rooted at a single sibling of that subtree, making the direction of lowering obvious; in cases where there are multiple sibling subtrees which consume functions produced in the subtree being lowered, lowering would correspond to computing some function outside the scope of its consumer(s).

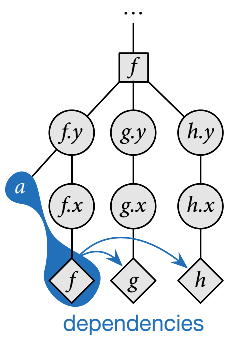


Figure 7.9: It is not possible to *lower compute(f)* if the union of f and anything in the subtree a are used in both g and h.

Hoist storage

hoist storage(func) coarsens the granularity at which a function is stored for reuse by moving its storage node one generation closer to the root of the tree. This swaps the function's storage node with its immediate parent in the tree. The storage node takes the place of its former parent in its former grandparent's child list, while the former parent becomes the sole child of the storage node. Former children of the storage node are inserted into the child list of the former parent in the location formerly occupied by the storage node.

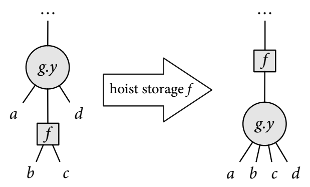


Figure 7.10: Hoist storage of f.

Lower storage

lower storage(func) is the inverse operation of *hoist storage*. It moves a function's storage node away from the root, towards both the function's own loop nodes and the compute nodes of functions which consume it. The operation exchanges a storage node with its child along the path towards its consumers and its own computation node. If multiple immediate children of the storage node contain either the computation node of the corresponding function, or computations which consume it, the storage may not be lowered any further. Storage may not be lowered past the outermost loop node of the corresponding function.

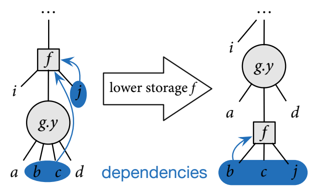


Figure 7.11: Lower storage of f.

Inline function

inline(func) transforms a function to be computed *inline*. This removes all storage and loop nodes for the function, and makes the function's compute node a child of its immediate caller's compute node. If a function has multiple callers, its compute

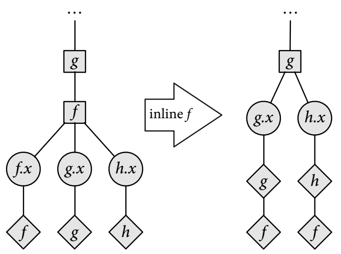


Figure 7.12: Inline computation of f.

node is duplicated as a child of each caller's compute node. If any of the removed nodes have children which are loop or storage nodes from other functions, these children become children of the next node closer to the root; they are effectively hoisted out of the deleted subtree, while maintaining the same relative topological order.

Deinline function

deinline(func) is the inverse operation of *inlining*. It reconstitutes the storage and default loop nodes for a function. It first places the storage node immediately between the compute node closest to the root in the subtree where the function was inlined, and that node's former parent. Then in makes the loop nodes for the function the immediately preceding child of the newly created storage node.

If a function has multiple callers, its storage node is placed at the nearest (non-compute node) dominator of all compute nodes which directly call it, unifying all uses of the function.

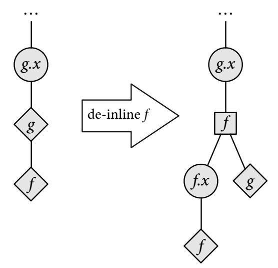


Figure 7.13: De-inline computation of f.

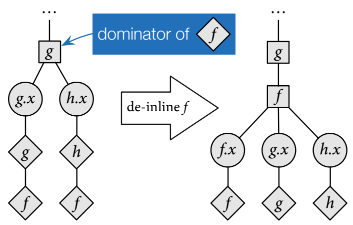


Figure 7.14: De-inline computation of a function with multiple callers.

7.3 SUMMARY

This representation forms an algebra of schedules: valid schedule trees define a space of possible schedules for an algorithm, while the transformations are operators which map between points in this space. Intuitively, the tree structure of this space corresponds directly to the semiperfect loop nests synthesized by Halide schedules, and the canonical inline-everything schedule tree corresponds to the default Halide schedule. The transformations can implement the schedule declarations in Halide's language of schedules[1]. Domain order operations all have direct corollaries in these transformations (*split*, *fuse*, *reorder*, and changing *loop type*). These operations in Halide are transformational in nature, so their correspondence is very direct. Call schedules in Halide, meanwhile, are specified in a *declarative* way, while compute and storage granularity are changed *transformationally* in this algebra. Intuitively, however, it is clear that any valid schedule can be modeled by a schedule tree, and that the composition of *inline/deinline*, *hoist/lower storage*, and *hoist/lower computation* operations can span the space of valid schedules for a given tree. Therefore, any valid Halide call schedule can be described by composing

1. The algebra, as presented, does not address iteration domains or multi-stage update functions. This is a straightforward extension.

a series of these operations, starting from the default schedule tree for the algorithm.

8 COMPILING SCHEDULED PIPELINES

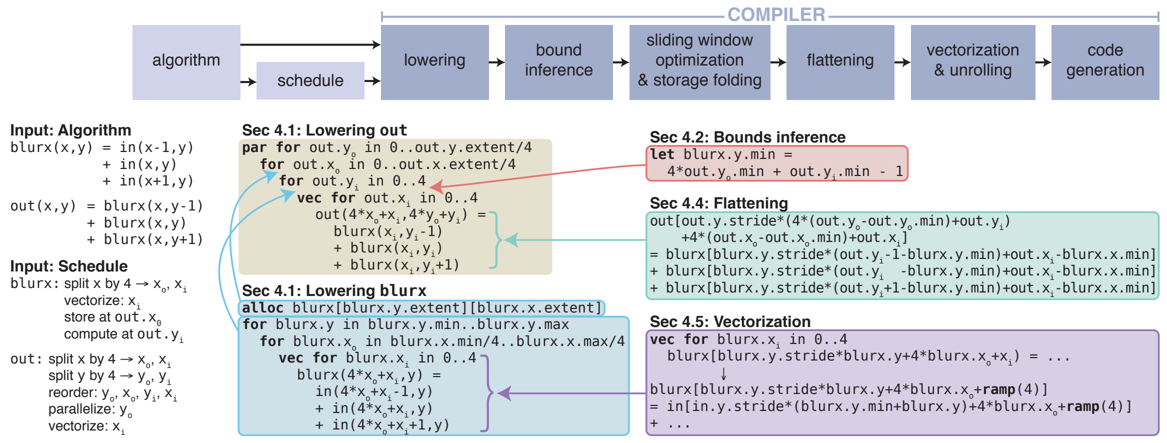
Our compiler combines the functions describing a Halide pipeline, with a fully-specified schedule for each function, to synthesize the machine code for a single procedure which implements the entire pipeline. The generated pipeline is exposed as a C ABI callable function which takes buffer pointers for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

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

Given the functions defining a Halide pipeline and a fully specified schedule as input (Figure 8.1, left), our compiler proceeds through the major steps below.

8.1 LOWERING AND LOOP SYNTHESIS

The first step of our compiler is a lowering process that synthesizes a single, complete set of loop nests and allocations, given a Halide pipeline and a fully-specified schedule (Figure 8.1, middle).



Input: Algorithm

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

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

Input: Schedule

```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i

out: split x by 4 → x_o, x_i
     split y by 4 → y_o, y_i
     reorder: y_o, x_o, y_i, x_i
     parallelize: y_o
     vectorize: x_i
```

Sec 4.1: Lowering out

```
par for out.y_o in 0..out.y.extent/4
  for out.x_o in 0..out.x.extent/4
    for out.y_i in 0..4
      vec for out.x_i in 0..4
        out(4*x_o+x_i,4*y_o+y_i) =
          blurx(x_i,y_i-1)
          + blurx(x_i,y_i)
          + blurx(x_i,y_i+1)
```

Sec 4.1: Lowering blurx

```
alloc blurx[blurx.y.extent][blurx.x.extent]
for blurx.y in blurx.y.min..blurx.y.max
  for blurx.x_o in blurx.x.min/4..blurx.x.max/4
    vec for blurx.x_i in 0..4
      blurx(4*x_o+x_i,y) =
        in(4*x_o+x_i-1,y)
        + in(4*x_o+x_i,y)
        + in(4*x_o+x_i+1,y)
```

Sec 4.2: Bounds inference

```
let blurx.y.min =
  4*out.y_o.min + out.y_i.min - 1
```

Sec 4.4: Flattening

```
out[out.y.stride*(4*(out.y_o-out.y_o.min)+out.y_i)
    +4*(out.x_o-out.x_o.min)+out.x_i]
= blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)+out.x_i-blurx.x.min]
+ blurx[blurx.y.stride*(out.y_i  -blurx.y.min)+out.x_i-blurx.x.min]
+ blurx[blurx.y.stride*(out.y_i+1-blurx.y.min)+out.x_i-blurx.x.min]
```

Sec 4.5: Vectorization

```
vec for blurx.x_i in 0..4
  blurx[blurx.y.stride*blurx.y+4*blurx.x_o+x_i) = ...
          ↓
blurx[blurx.y.stride*blurx.y+4*blurx.x_o+ramp(4)]
= in[in.y.stride*(blurx.y.min+blurx.y)+4*blurx.x_o+ramp(4)]
+ ...
```

Figure 8.1: The core of the Halide compiler lowers a functional representation of an imaging pipeline to imperative code using a *schedule*. It does this by first constructing a loop nest producing the final stage of the pipeline (in this case out), and then recursively injecting the storage and computation of earlier stages of the pipeline at the loop levels specified by the schedule. The locations and sizes of regions computed are symbolic at this point. They are resolved by the subsequent bound inference pass, which injects interval arithmetic computations in a preamble at each loop level that set the region produced of each stage to be at least as large as the region consumed by subsequent stages. Next, sliding window optimization and storage folding remove redundant computation and excess storage where the storage granularity is above the compute granularity. A simple flattening transform converts multidimensional coordinates in the infinite domain of each function into simple one-dimensional indices relative to the base of the corresponding buffer. Vectorization and unrolling passes replace loops of constant with k scheduled as *vectorized* or *unrolled* with the corresponding k-wide vector code or k copies of the loop body. Finally, backend code generation emits machine code for the scheduled pipeline via LLVM.

Lowering begins from the function defining the output (in this case, out). Given the function's domain order from the schedule, it generates a loop nest covering the required region of the output, whose body evaluates the function at a single point in that domain (Figure 8.1, middle-top). The order of loops is given by the schedule, and includes additional loops for split dimensions. Loops are defined by their minimum value and their extent, and all loops implicitly stride by 1. This process rounds up the total traversed domain of dimensions which have been split to the nearest multiple of the split factor, since all loops have a single base and extent expression.

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

Lowering then proceeds recursively up the pipeline, from callers to callees (here, from `out` to `blurx`). Callees (apart from those scheduled *inline*) are scheduled to be computed at the granularity of some dimension of some caller function. This corresponds to an existing loop in the code generated so far. This site is located, and code evaluating the callee is injected at the beginning of that loop body. This code takes the form of a loop nest constructed using the domain order of the callee. The allocation for the callee is similarly injected at some containing loop level specified by the schedule. In Figure 8.1, middle, `blurx` is allocated at the level of tiles ($\texttt{out.x}_o$), while it is computed as required for each scanline within the tile ($\texttt{out.y}_i$). The allocation and computation for `blurx` are inserted at the corresponding points in the loop nest.

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

8.2 BOUNDS INFERENCE

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

After bounds inference has recursed to the top of the pipeline, it walks back down to the output, injecting definitions for the bounds variables used as stand-ins during lowering. They are defined by expressions which compute concrete bounds as a preamble at each loop level (e.g., in Figure 8.1, right, the minimum bound of blurx.y is computed from interval analysis of the index expressions at which it is accessed combined with the bounds of the calling function, `out`). In practice, hoisting dynamic bounds evaluation expressions to the outermost loop level possible makes the runtime overhead of more complex bounds expressions negligible.

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

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

8.3 SLIDING WINDOW OPTIMIZATION AND STORAGE FOLDING

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

For example, in Figure 8.1, `blurx` is stored for reuse within each tile of `out`, but computed as needed, for each scanline within the tile. Because scanlines ($\texttt{out.y}_i$) are traversed sequentially, intermediate values of `blurx` are computed immediately before the first scanline of `out` which needs them, but may be reused my later scanlines within the tile. For each iteration of $\texttt{out.y}_i$, the range of `blurx.y` is computed to exclude the interval covered by all prior iterations computed within the tile.

Storage folding is a second similar optimization employed at this stage of lowering. If a region is allocated outside of a serial loop but only used within it, and the subregion used by each loop iteration marches monotonically across the region allocated, we can "fold" the storage, by rewriting indices used when accessing the region by reducing them modulo the maximum extent of the region used in any given iteration. For example, in Figure 8.1, each iteration of $\texttt{out.y}_i$ only needs access to the last 3 scanlines of `blurx`, so the storage of `blurx` can be reduced to just 3 scanlines, and the value `blurx(x,y+3)` will reuse the same memory address as `blurx(x,y)`, `blurx(x,y-3)`, and so on. This reduces peak memory use and working set size.

8.4 FLATTENING

Next, the compiler flattens multi-dimensional loads, stores, and allocations into their single-dimensional equivalent. This happens in the conventional way: a stride and a minimum offset are computed for each dimension, and the buffer index corresponding to a multidimensional site is the dot product of the site coordinates and the strides, minus the minimum. (Cf. Figure 8.1, right.) By convention, we always set the stride of the innermost dimension to 1, to ensure we can perform dense vector loads and stores in that dimension. For images, this lays them out in memory in scanline order. While our model of scheduling allows extreme flexibility in the order of execution, we do not support more unusual layouts memory, such as tiled or sparse storage. (We have found that modern caching memory hierarchies largely obviate the need for tiled storage layouts, in practice.)

8.5 VECTORIZATION AND UNROLLING

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

Vectorization completely replaces a loop of size n with a single statement. For example, in Figure 8.1 (lower right), the vector loop over `blurx.x`$_i$ is replaced by a single 4-wide vector expression. Any occurrences of the loop index (`blurx.x`$_i$) are replaced with a special value `ramp(`n`)` representing the vector $[0\ 1 \ldots n-1]$. A type coercion pass is then run over this to promote any scalars combined with this special value to n-wide broadcasts of the scalar expression. All of our IR nodes are meaningful for vector types: loads become gathers, stores become scatters, arithmetic becomes vector arithmetic, ternary expressions become vector selects, and so on. Later, during code generation, loads and stores of a linear expression of $k \times$ `ramp(`n`)` $+ o$ will become dense vector loads and stores if the coefficient $k = 1$, or strided loads and stores with stride k otherwise. In contrast to many languages, Halide has no divergent control flow, so this transformation is always well-defined and straight-forward to apply. In our representation, we never split a vector into a bundle of scalars. It is always a single expression containing ramps and broadcast nodes. We have found that this yields extremely efficient code *without* any sort of generalized loop auto-vectorization.

8.6 BACK-END CODE GENERATION

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

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

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

GPU Code Generation

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

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

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

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

RESULTS & EVALUATION 9

This chapter evaluates the effectiveness of the Halide language and compiler in implementing, optimizing, and compiling real applications. My primary focus is evaluating the experience of building and manually scheduling a range of real image processing pipelines relative to existing state-of-the-art equivalents implemented in traditional languages including C/C++, CUDA, MATLAB, and assembly. Section 9.1 presents case studies of different image processing applications on three classes of target architecture: x86-64 multicores, ARMv7 multicores, and CUDA GPUs. I will show how, for the same algorithm, different schedules can have very different performance. Optimizing schedules enables simple algorithm code to compile to state-of-the-art performance on a range of different architectures. I will show that it is feasible to do this *automatically* using stochastic search and autotuning to search the space: section 9.2 presents initial results automatically scheduling several of these applications with a prototype autotuner. Finally, I show how it's proving useful in real production applications: Section 9.3 discusses Halide's wider adoption and deployment in its first two years.

9.1 IMAGE PROCESSING APPLICATIONS IN HALIDE

To evaluate our representation and compiler, we applied them to a range of image processing applications. We reimplemented each in Halide, and compared both code complexity, and hand- and auto-tuned schedule performance generated by our compiler, to the best previously published expert implementation we could find. We selected this set of examples to cover a diversity of algorithms and communication patterns. It includes pipelines ranging from two to 99 stages, and including many different stencils, data-dependent access patterns, histograms, and reductions. We describe each application and our experiences implementing and optimizing it below.

Blur

Blur is the simple two-stage box filter example used throughout this thesis. It convolves an input image with two 3×1 box kernels in two steps, a horizontal 3×1 kernel followed by a vertical 1×3 kernel. This is a simple example of two consecutive stencils. Our reference comparison is a hand-optimized, manually fused and multithreaded loop nest defined almost entirely in SSE intrinsics [74]. This version is 36 lines of code, and is 12× faster than a simple pair of loops in C compiled by GCC 4.7. The Halide version expresses the same algorithm in two lines, and exact same organization as the optimized reference in two lines of schedule (five total directives).

The blur reference implementation was initially developed during early Halide development, to establish a baseline for optimized code performance on a simple chain of small stencils. It is hand-coded in SSE, with manually tiled and interleaved loops, OpenMP parallelism, and all pixel computation implemented as SSE intrinsics on 16-bit fixed point values. Starting from a trivial pair of scalar loop nests expressing the basic algorithm[1], many alternative organizations were explored over the course of several days, initially on a quad core Core 2 Xeon Mac Pro. Vectorization and multithreaded parallelism improved performance over the initial organization, but by far less than the theoretical factor of perhaps 16× (four cores, each with eight-way SIMD parallelism through the critical path, vs. 2-3 wide superscalar execution of the primary pixel computations on a single core). Rather, the pipeline was limited by memory bandwidth. The most efficient organization found computed the whole pipeline on 256×32 tiles, interleaving the computation of tiles between stages, and storing intermediate results of the bx stage only briefly in a per-thread tile buffer. The resulting code was 11× faster than the trivial pair of loop nests.

1.
```
for all $y$
    for all $x$
        compute $bh(x, y)$
for all $y$
    for all $x$
        compute $bv(x, y)$
```

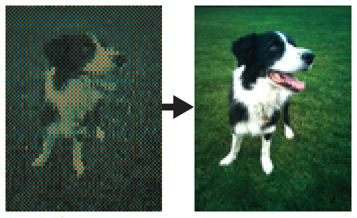
The Halide expression of the same algorithm and organization generates nearly identical machine code and performance on the same machine. Using it, we also quickly explored even more organizations. We found several alternatives which balanced locality and redundant computation in different ways. Several provided similar performance on the original Core 2 Mac Pro, but different organizations performed an additional 10 – 15% faster than the initial tiled organization on Sandybridge and Ivybridge-generation quad core CPUs.

Camera Pipeline

Camera pipeline transforms the raw data recorded by an image sensor into a usable image. It comprises four steps: hot-pixel suppression, demosaicking, color correction, and tone adjustment (i.e., gamma correction and contrast enhancement). This combination of processes mixes a wide variety of operations including complex, heterogeneous stencils and convolutions, and is optimized for a mixture of 16-bit fixed computation and floating point transcendentals. Its demosaicking, alone, is a complex combination of 21 interleaved and inter-dependent stencils.

The reference comparison comes from the Frankencamera and takes two forms, each expressing the same basic hand-optimized organization [2], one hard-coded for ARM NEON, the other in pure C++. Both versions use a single carefully tiled and fused loop nest. The pure C++ version is 306 lines in total. The hand-vectorized ARM version is a heavily optimized mixture of vector intrinsics and inline ARM originally assembly targeted at a Cortex A8 core (specifically, the OMAP3 processor in the Nokia N900), taking 463 lines in total. The tightly bounded stencil communication down the pipeline makes fusion of stages to save bandwidth and storage a critical optimization for this application. In both versions, all producer-consumer communication is staged through scratch buffers, and tiles are distributed over parallel threads using OpenMP. In the pure C++ version, the tight inner loops are at least partially autovectorized by GCC; in the ARM-specific version, virtually all operations performed are highly tuned 4- and 8-wide NEON vector instructions.

The Halide algorithm is 145 lines describing 32 functions and 22 different stencils. It is dramatically simpler than even the pure C++ version; it was literally translated from the pseudocode in the comments explaining the original source. We can express the same optimizations used in the Frankencamera assembly, separately from the algorithm: the output is



Camera Raw Pipeline

Optimized NEON ASM:	463 lines
Nokia N900:	772 ms
Halide algorithm:	145 lines
schedule:	23 lines
Nokia N900:	741 ms

2.75x shorter
5% *faster* than tuned assembly

Port to different architecture:

Quad-core x86:	51 ms

Figure 9.1

tiled, and each intermediate stage is *computed* and *stored* at tile granularity. The innermost dimension of each function is *vectorized*. This requires one line of scheduling choices (roughly two directives each) per function in the pipeline.

Our implementation takes 741 ms to process a 5 megapixel raw image on a Nokia N900 running the Frankencamera code, while the Frankencamera implementation takes 772 ms. Our implementation is also portable, whereas the Frankencamera assembly is entirely platform specific: the same Halide code compiles to multithreaded x86 SSE code, which takes 51 ms on our quad-core desktop.

The Frankencamera pure C++ fallback code takes 54 ms, multithreaded with OpenMP and autovectorized by GCC 4.7. the Halide version takes 14 ms.

The Halide implementation uses a schedule modeled directly after the optimized ARM version. Performance difference is largely due to vector code quality.

Multi-scale Interpolation

Interpolate uses an image pyramid to interpolate pixel data for seamless compositing. This requires dealing with data at many different resolutions. The resulting pyramids are chains of stages which locally resample over small stencils, but through which dependence propagates globally across the entire image. This algorithm is used in Adobe Photoshop and Camera Raw to implement the healing brush.

The reference implementation is a carefully-structured set of loop nests, written as 152 lines of C++, which were hand-tuned by an Adobe engineer to generate a vectorized implementation in GCC . The Halide algorithm is 7× simpler (21 lines). On a single core of a Core i7-3770, the two implementations deliver similar performance, requiring 25 ms/megapixel processed; on four cores, the Halide version scales almost perfectly, requiring 7 ms/megapixel.

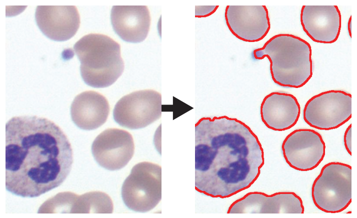
Fast Fourier Transform

The fast Fourier transform is widely used in image processing and other domains, but the algorithm is more similar to dense linear algebra than traditional image processing pipelines. To test the appicability of Halide to different application domains and computational patterns, we implemented a complement of 2D fast Fourier transform variants in Halide. The implementations share most logic and total under 350 lines of code for 2D complex-to-complex, real-to-complex, and complex-to-real, including optimized schedules for a Core i7-3770 x86 processor with AVX. Performance is competitive with autotuned FFTW, long regarded as the state-of-the-art FFT implementation for x86 processors. For a 32×32 complex-to-complex 2D FFT on floating point data, the Halide implementation currently outperforms the best autotuned FFTW implementation by 15% (29.7 GFLOP/s for Halide, vs. 26.2 GFLOP/s for FFTW). On a 64×64 FFT, the same Halide schedule thrashes the cache hierarchy, delivering just over half the performance (15.3 GFLOP/s), where FFTW scales smoothly (25.1 GFLOP/s). A different schedule for the larger block size could likely close the gap. The Halide implementation of complex-to-real transformation differs slightly in semantics from FFTW (it returns a transposed result), so it is less perfectly comparable, but for both 32×32 and 64×64 the Halide implementations are nearly twice as fast as the FFTW variants tuned on this machine, starting from far simpler code.

Level Set Image Segmentation

Active contour selection (a.k.a., *snake* [47]) is a method for segmenting objects from a background (Figure 9.2). Level-set methods [15] are an effective way to implement such techniques when the objects of interest are likely to be smooth and when the number of connected components is not known a priori. It is well suited for medical applications (e.g., to segment cells). We implemented the algorithm proposed by Li et al. [52]. The algorithm is iterative, and can be interpreted as a gradient-descent optimization of a 2D function. Each update of this function is composed of three terms, each of them being a combination of differential quantities computed with small 3×1 and 1×3 stencils, and point-wise nonlinear operations, such as normalizing the gradients.

The reference comparison is the original authors' implementation, which is 67 lines of MATLAB. MATLAB is notoriously slow when misused, but this code expresses all operations in the array-wise notation that MATLAB executes most efficiently.



***Snake* Image Segmentation**

Vectorized MATLAB:	67 lines
Quad-core x86:	3800 ms
Halide algorithm:	148 lines
schedule:	7 lines
Quad-core x86:	55 ms

2.2x longer
70x faster

Schedule for different architecture:

CUDA GPU:	3 ms (1250x)

Figure 9.2

In Halide, we factored this algorithm into three feed-forward pipelines. Two pipelines create images that are invariant to the optimization loop, and one primary pipeline performs a single iteration of the optimization loop. While Halide can represent bounded iteration over the outer loop using a reduction, it is more naturally expressed in the imperative host language. At 148 lines, the Halide implementation is longer, but this is largely due to the combination of syntactic overhead in the C++ embedding (e.g., each function or variable must be declared before it is used, while the MATLAB syntax does not require explicit declaration), and the absence of some built-in MATLAB operations which must instead be expressed explicitly as a sequence of multiple operations in Halide.

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

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

Bilateral Grid

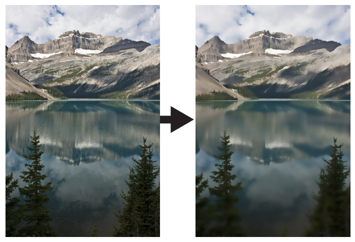
The bilateral filter smoothes an image while preserving edges [68]. It is used for denoising or to decompose images into local and global details. It is efficiently computed with the *bilateral grid* algorithm [20, 66]. This algorithm first scatters the image data into a 3D grid, effectively building a windowed histogram in each column of the grid, then blurs the grid along each of is axes with three 5-point stencils. Finally, the output image is constructed by trilinear interpolation within the grid at locations determined by the input image.

The CPU reference code is a tuned but clean implementation from the original authors in 122 lines of C++. It is partially autovectorized by GCC, but is nontrivial to multithread (a naive OpenMP parallelization of major stages results in a slowdown on our benchmark CPU), so the reference is single-threaded. The Halide algorithm is 34 lines.

We implemented this algorithm in Halide and found that the best schedule for the CPU simply parallelizes each stage across an appropriate axis. The only stage regular enough to benefit from vectorization is the small-footprint blur, but for commonly used filter sizes the time taken by the blur is insignificant. Using this schedule on our quad-core x86 desktop, we compute a bilateral filter of a four megapixel input using typical filter parameters (spatial standard deviation of 8 pixels, range standard deviation of 0.1) in 80 ms. In comparison, the moderately-optimized C++ version provided by Paris and Durand [66] takes 472 ms using a single thread on the same machine. Our single-threaded runtime is 254 ms; some of our speedup is due to parallelism, and some is due to generating superior scalar code. We use 34 lines of code to describe the algorithm, and 6 for its schedule, compared to 122 lines in the C++ reference.

We first tried running the same algorithm on the GPU using a schedule which performs the reduction over each tile of the input image on a single CUDA block, with each thread responsible for one input pixel. Halide detected the parallel reduction, and automatically inserted atomic floating point adds to memory. The runtime was 40 ms—only 2× faster than our optimized CPU code, due to atomic contention. The latest hand-written GPU implementation by Chen et al. [20] expresses the same algorithm and a similar schedule in 370 lines of CUDA C++, and takes 24 ms on the same GPU.

With the rapid schedule exploration enabled by Halide, we quickly found a better schedule that trades off some parallelism to reduce atomics contention. We modified the schedule to use one thread per tile of the input, with each thread walking serially over the reduction domain. This one-line change in schedule gives us a runtime of 11 ms for the same image. When we rewrite the hand-tuned CUDA implementation to match the schedule found with Halide, it takes 8 ms. The 3 ms improvement over Halide comes from the use of texture units for the slicing stage. Halide does not currently use tex-



Bilateral Grid

Tuned C++:	122 lines
Quad-core x86:	472ms
Halide algorithm:	34 lines
schedule:	6 lines
Quad-core x86:	80 ms

3x shorter
5.9x faster

Schedule for different architecture:

CUDA GPU:	11 ms (42x)
Hand-written CUDA: [Chen et al. 2007]	23 ms

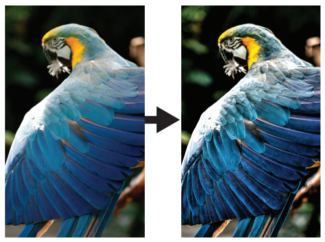
Figure 9.3

ture hardware. In general, hand-tuned CUDA can surpass the performance Halide achieves when there is a significant win from clever use of specific CUDA features not expressible in our schedule, but exploring different optimization strategies is much harder than in Halide. Compared to the original CUDA bilateral grid, the schedule found with Halide saved 13 ms, while the clever use of texture units saved 3 ms.

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

Local Laplacian Filters

One of the most important tasks in producing compelling photographic images is adjusting local contrast. *Local Laplacian filters* uses a multi-scale approach to tone map images and enhance local contrast in an edge-respecting fashion [67, 10]. It is used in the clarity, tone mapping, and other filters in Adobe Photoshop and Lightroom. It works by building a set of multiple Gaussian and Laplacian image pyramids, with complex dependencies between them. The filter output is ultimately produced by a data-dependent resampling from several pyramids. The resulting pipeline mixes many images at many different resolutions with a complex network of dependencies. With the parameters we used, the pipeline contains 99 different stages, operating at many scales, and with different computational patterns.



Local Laplacian Filter

C++, OpenMP+IPP:	262 lines
Quad-core x86:	335 ms
Halide algorithm:	62 lines
schedule:	7 lines
Quad-core x86:	158 ms

3.7x shorter
2.1x faster

Schedule for different architecture:

CUDA GPU:	48 ms (7x)

Figure 9.4

The reference implementation is 262 lines of C++, developed at Adobe, and carefully parallelized with OpenMP, and offloading most intensive kernels to tuned assembly routines from Intel Performance Primitives [65, 40]. It has very similar performance to a version deployed in their products, which took several months to develop, including at least 2-3 weeks dedicated to optimization. It is 10× faster than an algorithmically identical reference version written by the authors in pure C++, without IPP or OpenMP. The Halide version was written in two days, in 52 lines of code. A third implementation, in ispc [70], using OpenMP to distribute the work across multiple cores, used 288 lines of code. It is longer than in Halide due to explicit boundary handling, memory management, and C-style kernel syntax.

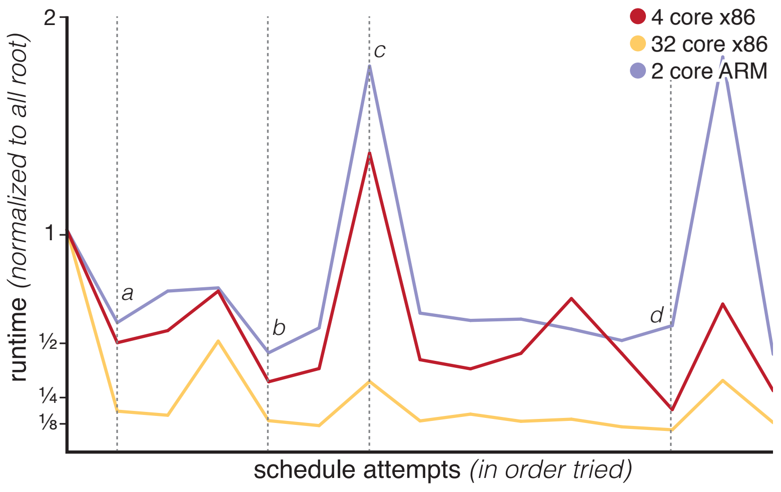


Figure 9.5: Results from manual tuning of the local Laplacian filters schedule across two x86 machines and a dual-core ARM machine, in the order schedules were tried.

We implemented local Laplacian filters in Halide, and explored multiple strategies for scheduling it efficiently on several different machines. The statement of the algorithm did not change during the exploration of plausible schedules. We found effective schedules for the local Laplacian filter by manually testing and refining a small, hand-tuned schedule, across a range of multicore CPUs. The overall progress of relative performance on each platform during this process is plotted in Figure 9.5, and some major steps are highlighted. To begin, all functions were scheduled to be computed sequentially, at *root* granularity. Then, each stage was parallelized over its outermost dimension (*a*). Computing the Laplacian pyramid levels *inline* improves locality, at the cost of redundant computation (*b*). However, excessive inlining is dangerous: the high spike in runtimes results from additionally inlining every other Gaussian pyramid level (*d*). The best performance on the x86 processors required additionally inlining only the bottom-most Gaussian pyramid level, and vectorizing across the *x* dimension (*e*). The ARM performs slightly better with a similar schedule, but no vectorization. The entire optimization process took only a couple of hours.

Ultimately, we found that on several x86 platforms, the best performance came from a complex schedule involving inlining certain stages, and vectorizing and parallelizing the rest.

The schedule is specified using seven lines of code. Using this schedule on our quad-core laptop, processing a 4 megapixel image takes 158 ms. On the same processor the hand-optimized, OpenMP/IPP version developed by Aubry et al. takes 335 ms. A third implementation, in ispc, takes 327 ms to process the 4-megapixel image. The Halide implementation is faster due to fusion down the pipeline. The ispc implementation can be manually fused by rewriting it, but this would significantly lengthen and complicate the code, and .

A schedule equivalent to naive parallel C, with all major stages scheduled as *root* but evaluated in parallel over the outer dimensions, performs much less redundant computation than the fastest schedule, but takes 296 ms because it sacrifices producer-consumer locality and is limited by memory bandwidth. This organization is roughly equivalent to the OpenMP/IPP and ispc implementations.

The best schedule on a dual core ARM OMAP4 processor is slightly different. While the same stages should be inlined, vectorization is not worth the extra instructions, as the algorithm is bandwidth-bound rather than compute-bound. On the ARM processor, the algorithm takes 5.5 seconds with vectorization and 4.2 seconds without. Naive evaluation takes 9.7 seconds. The best schedule for the ARM takes 278 ms on the x86 laptop—75% longer than the best x86 schedule.

This algorithm maps well to the GPU, where processing the same four-megapixel image takes only 49 ms. The best schedule evaluates most stages as *root*, but fully fuses (*inlines*) all of the Laplacian pyramid levels wherever they are used, trading increased computation for reduced bandwidth and storage, similar to the x86 and ARM schedules. Each stage is split into 32×32 tiles that each map to a single CUDA block. The same algorithm statement then compiles to 83 total invocations of 25 distinct CUDA kernels, combined with host CPU code that precomputes lookup tables, manages device memory and data movement, and synchronizes the long chain of kernel invocations. Writing such code by hand is a daunting prospect, and would not allow for the rapid performance-space exploration that Halide provides.

9.2 AUTOMATIC SCHEDULING USING AUTOTUNING

To test the feasibility of automatically discovering good schedules using stochastic search over the space of schedules, we tested our prototype autotuner on five applications on an x86 target, and three on a CUDA GPU. The resulting performance relative to our expert-tuned handwritten reference applications is shown in Figure 9.6. The autotuner consistently finds schedules which match or exceed the performance of these hand-optimized C++, CUDA, and intrinsics implementations on the same machine. Autotuned performance is generally similar to or slightly better than the best hand-coded schedules we found on the specific tests for which the schedules were tuned, but they generally used more total schedule parameters than the hand-written schedules. These examples took between 2 hours and 2 days to tune (from 10s to 100s of generations). In all cases, the tuner converged to within 15% of the final performance after less than one day tuning on a single machine. Improvements to the compiling and tuning infrastructure (for example, distributing tests across a cluster) could reduce these times significantly.

Over-fitting

One challenge with autotuning is *over-fitting* to the exact benchmark used in tuning. In image processing, a major variable which affects the performance of different schedules is image size; common image sizes vary from thousands to tens of millions of pixels. To evaluate the potential over-fitting effect of training on very different image sizes, we cross-tested three algorithms on two image sizes, varying by at least an order of magnitude. The smaller size (640 × 480) was chosen to easily fit in the on-chip caches of the target processor. The larger size varied by application, but was chosen to be larger than the largest on-chip cache.

We generally found the tuned schedules to be insensitive to moderate changes in resolution or architecture, but extreme changes can cause the best schedule to change dramatically. Figure 9.7 shows experiments in cross-testing schedules tuned at these different resolutions. We observe that schedules generalize better from low resolutions to high resolutions. We tested

x86

	Halide autotuned	*Expert hand-tuned*	*Speedup*
Blur	11 ms	13 ms	1.2×
Bilateral grid	36 ms	158 ms	4.4×
Camera pipeline	14 ms	49 ms	3.4×
Interpolation	32 ms	54 ms	1.7×
Local Laplacian	113 ms	189 ms	1.7×

CUDA

	Halide autotuned	*Expert hand-tuned*	*Speedup*
Bilateral grid	8.1 ms	18 ms	2.3×
Interpolation	9.1 ms	*54 ms**	5.9×
Local Laplacian	21 ms	*189 ms**	9×

Figure 9.6: Comparison of autotuned Halide program running times to hand-optimized programs created by domain experts in C, intrinsics, and CUDA. Halide programs are both faster and require fewer lines of code. *(*No GPU reference available, compared to CPU reference.)*

the degree to which schedules fit the architecture on which they are tuned by mapping the best GPU schedule for local Laplacian filter to the CPU, and found that this is 7× slower than the best CPU schedule.

9.3 DEPLOYMENT

At the time of writing, Halide has been used by students and engineers for research, products, and teaching many companies and institutions. Here, I highlight four publicly-known examples.

Several groups within Google are heavy users of Halide. Several dozen engineers write image processing code in Halide in their day-to-day work. Over 10,000 lines of Halide code are in use in production in various products. The largest user is Google+ Photos. The image processing of their auto-enhance pipeline is written in Halide. For every user photo uploaded to Google's servers, a large amount of Halide code runs in a data center (primarily on x86/SSE multicores) to enhance

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

Figure 9.7: Cross-testing of autotuned schedules across resolutions. Each program is autotuned on a source image size. The resulting schedule is tested on a target image size giving a "cross-tested time." This is compared to the result of running the autotuner directly on the target resolution. We report the ratio of the cross-tested time to the autotuned-on-target time as the "slowdown." Note that schedules generalize better from low resolutions to high resolutions. In theory the slow-down should always be at least one, but due to the stochastic nature of the search some schedules were slower when autotuned on the target.

it for display. On Android cell phones, auto-enhance in the photos app runs the same Halide algorithm, compiled with a schedule optimized for mutlicore ARM/NEON in mobile devices. Google's HDR+ pipeline is also written partially in Halide. This code processes every photo taken on Google Glass, and implements the HDR+ mode in the official camera application on recent Android devices, notably the Nexus 5. Of note, the fact that Halide is embedded in C++, rather than being a "new" language or being embedded in something less established, was essential to its widespread adoption being allowed at Google.

Multiple research and product teams at Adobe are also experimenting with Halide. In July 2012, I worked with the engineer responsible for the Camera Raw pipeline in Photoshop and at the core of Lightroom to prototype replacement of his implementation of local Laplacian filters with a Halide equivalent. The Halide version took less than a day to adapt from our existing local Laplacian filters implementation and integrate into Lightroom. It required 60 lines of code, and represented less than two days of total implementation effort, relative to several thousand lines and about three month of work for the hand-tuned original. We quickly found a schedule which performed twice as fast as the hand-tuned version on the same 8 core Nehalem x86 workstation, and another which performed 9× faster on a Tesla C2070 GPU. So far, this remains a prototype. Production deployment was initially limited by the need to integrate with complex custom tile scheduling and memory management common in Adobe's systems, which limited Halide integration to kernels operating within individual tiles; it is now feasible to fuse complex Halide pipelines directly with such systems thanks to the addition of extern Image functions in Halide, but doing so at Adobe remains ongoing work.

Researchers on the Department of Energy X-STACK project have implemented several kernels from the CloverLeaf hydrodynamics and HPGMG multigrid benchmarks. Initial results outperform simple OpenMP equivalents.

Finally, Halide was used last fall by over 50 MIT undergraduate and graduate students in the 6.815/6.865 computational photography course to implement and schedule several algorithms, including basic convolution and Harris corner detection. As the final part of their assignment, students even implemented basic autotuning by automatically generating and testing many different schedules. The class was otherwise taught in Python (using the NumPy library for efficient array storage and processing [43]), and many students surprisingly commented that they found Halide enjoyably concise, even relative to unoptimized Python. (This extra concision in algorithm expression was largely due to the removal of explicit loops, boundary handling, and complex indexing thanks to Halide's function notation and automatic bounds inference.) Reasoning about call schedules and computation granularity while scheduling was the most significant challenge for most students, but the best students quickly mastered scheduling and were able to exceed the performance of a vectorized NumPy implementation of Harris corner detection by up to two orders of magnitude.

RELATED WORK 10

This chapter highlights connections between the design and implementation of Halide and several areas of prior work. Halide draws on decades of ideas, systems, languages, and compilers. Particular emphasis is placed on the concept of explicitly modeling the schedule as a first-class part of the language, as well as the areas of stencil computation, stream processing, loop transformation, and high-performance image processing and graphics systems.

Split languages

One of the more unusual features of the Halide language is its simultaneous *decoupling* of the definition of algorithms from the organization of their computation, and promotion of the organization to a first-class, programmer-controlled part of the language in the form of explicit *schedules*. This idea is mirrored in a few other notable systems.

SPIRAL

The SPIRAL system [73] uses a domain-specific language to specify linear signal processing operations at a very high level of abstraction, independent of many implementation choices. Given that, a series of transformation algebras progressively describes how an optimized implementation should be generated. First, a family of equivalence transformations maps from signal processing operations to more efficient equivalent operations. Then, separate mapping functions describe how these operations should be turned into efficient code for a particular architecture. For general-purpose processors, these mappings are described using a loop synthesis algebra which maps from high-level linear signal processing operations to concrete loop nests which implement them. These latter transformations are particularly similar to our own scheduling algebra. Altogether, SPIRAL enables high performance across a range of architectures by making deep use of mathematical identities on linear filters. Image processing and computational photography algorithms often do not fit within a strict linear filtering model. Our work can be seen as an attempt to generalize this approach to a broader class of programs.

Sequoia & Legion

The Sequoia language defines a model where a user-defined "mapping" describes how to execute tasks on a tree-like memory hierarchy [29]. This parallels our model of scheduling, but focuses on hierarchical problems like blocked matrix multiply, rather than pipelines of images. Sequoia's mappings, which are highly explicit, are also more verbose than our schedules, which are designed to infer many details not specified by the programmer.

Legion is a successor to Sequoia, focussed on dynamic scheduling and irregular computation where Sequoia was initially limited to static scheduling [11]. Legion carries over the concept of mappings, but implements them as user-defined subroutines which execute as part of the dynamic scheduler at runtime.

Sequoia's and Legion's mappings, and SPIRAL's loop synthesis and other transformation algebras, echo Halide's separation of the model of scheduling from the description of the algorithm, and its lifting outside our compiler. Legions mappings are executed repeatedly at runtime, while Sequoia's mappings, SPIRAL's transformation algebras, and Halide's schedules all direct the structure of code synthesized at compile time.

A key difference between Sequoia's mappings and Halide's schedules is the computational generality of their expression. In Sequoia, mappings are expressed by statically enumerating all choices for a given program in what amounts to an exhaustive configuration file. The key challenge with this split representation is the rapid increase in complexity of mappings as the algorithms, themselves, become more complex.

At the outset, I expected to face similar challenges in Halide. By nature, the space of possible choices about the organization of computation grows combinatorially with program size. In practice, however, I was surprised to find how easily direct specification of Halide schedules scaled, within the scope of image processing pipelines we have studied so far. I believe there are three fundamental reasons:

1. The richer features of the general-purpose programming language (C++) in which our schedule specifications were embedded—including powerful facilities for abstraction, composition, and control flow—often made schedules of a given complexity much more concise to express than when statically enumerated.[1] Syntactic sugar for common scheduling patterns even proved useful enough to include in the core language.[2]
2. Halide programs, in practice, are limited in size. We have mostly focussed on algorithms and pipelines up to about 100 stages. These individual algorithms may be composed into even more complex systems, but we are never likely to face a single end-to-end pipeline of hundreds of thousands of different functions which must be scheduled together.
3. Even for our most complex pipelines, the actual complexity of optimized schedules seemed to grow at worst linearly with program size. The space of possible choices is certainly combinatorial, but while interactions between functions in a pipeline graph are, in principle, global and unbounded, most are, in practice, relatively local. This limits the complexity of the schedules we have ever actually needed to write.

1. As a simple example, it is common for fragments of similar scheduling choices to be applied to many related functions. Rather than enumerating every choice, we can simply abstract the common choices into a subroutine, and apply it to each Halide function using a simple `for` loop.

2. `tile`, `cuda`, and other scheduling choices are implemented as simple macros atop lower-level schedule primitives.

Stencil optimization

Stencils are a common computational pattern in numerical algorithms . They have been well studied in the form of *iterated stencil computations*, where one or a few small stencils are applied to the same grid over many iterations [30, 49, 63]. Critically, many optimizations for iterated stencil computations are based on the assumption that the time dimension of iteration is large relative to the spatial dimension of the grid. In contrast, Halide was designed to target other applications, in image processing and computer graphics, where stencils are common, but often in a very different form: stencil pipelines. Stencil pipelines are graphs of different stencil computations. Iteration of the same stencil occurs, but it is the exception, not the rule; most stages apply their stencil only once before passing data to the next stage, which performs different data parallel computation over a different stencil. In image processing pipelines, most individual stencils are applied only once, while images are millions of pixels in size. Image processing pipelines also include more types of computation than stencils alone, and scheduling them requires choices not only of different parameters, but of entirely different strategies, for each of many heterogeneous stages, which is infeasible with either exhaustive search or polyhedral optimization.

Iterated stencil computations are important to many scientific applications, and have been studied for decades. Frigo and Strumpen proposed a cache oblivious traversal for efficient stencil computation [30]. Their view of locality optimization by interleaving the application of stencils in space and time inspired our model of scheduling. The Pochoir compiler automatically transforms C++ stencil algorithms from serial loop form into a parallel cache oblivious form using similar algorithms [84].

Overlapping tiling (also called tiling with "ghost zones") is a strategy which divides a stencil computation into tiles, and trades off redundant computation along tile boundaries to improve locality and parallelism [49, 58]. This is a common pattern in optimizing both iterated stencil computations and image processing algorithms, and was a key early target for our design[3]. Many other tiling strategies represent different points in the tradeoff space modeled by our representation [63].

3. Overlapped tiling is modeled in our schedule representation as interleaving both the storage and computation of producer stages inside the tile loops of their consumer.

Past compilers have automatically synthesized parallel loop nests with overlapped tiling on CPUs and GPUs [49, 36]. These compilers focussed on synthesizing high quality code given a single, user-defined set of overlapped tiling parameters. Autotuning has also been applied to iterated stencil computations, but past tuning work has focussed on exhaustive search of small parameter spaces for one or a few hard-coded strategies [44, 21].

Stream processing

Graph-structured programs have been studied in the context of streaming languages [85, 31, 17]. Streaming languages encode data and task parallelism in graphs of kernels. Compilers automatically schedule these graphs using tiling, fusion, and fission [45]. Static communication analysis allows stream compilers to simultaneously optimize data parallelism and producer-consumer locality by interleaving computation and communication between kernels. Most stream compilation research has focussed on 1D streams, where sliding window communication allows 1D stencil patterns [87, 72, 33]. Intel's Concurrent Collections system provides a stream programming model in C++ [19].

Image processing pipelines can be thought of as programs on 2D and 3D streams and stencils. As I have shown, even with 1D streams and stencils, sliding window schedules are only one extreme point in a complex family of tradeoffs between locality, parallelism, and the amount of computation. The model of computation required by image processing is also more general than stencils, alone. While most stages are point or stencil operations over the results of prior stages, some stages gather from arbitrary data-dependent addresses, while others scatter to arbitrary addresses to compute operations like histograms.

StreamIt

The StreamIt language models programs as graphs of kernels connected by streams. Data parallelism and communication are made explicit by split-join operators on streams. Work on StreamIt focussed heavily on communication optimization of cyclostatic dataflow within this framework. To implement stencils over streams, a StreamIt kernel peeks back some distance into its input stream. Research focussed on efficiently implementing peeking kernels using sliding window schedules, where producer-consumer rates and intermediate storage were automatically optimized for overlapping stencil data accesses in 1D streams [31]. This model fundamentally relies on synchronous execution of producer and consumer stages, constraining the order of computation and limiting parallelism. It also requires all data be cast as 1D streams. Implementing 2D stencils over image data flattened into 1D streams would create very large ($O(n)$) peeking windows, requiring large intermediate buffering and limiting producer-consumer locality. In this case, it is often more efficient to decouple producer and consumer computations by redundantly recomputing some values within the stencil, reducing its effective size. Our model of scheduling addresses the problem of overlapping multidimensional (2D and 3D) stencils, where storage footprint, locality, parallelism, and the total amount of work become a critical but complex choice.

Brook

The Brook stream processing language similarly models stream programs as graphs of kernels connected by streams [17]. Brook modeled the complex dependencies between streams introduced by stencils using a family of explicit "stream operators." However, because of the complex tradeoffs and global schedule transformations required by stencil pipelines, this feature was never fully implemented or automatically optimized by the Brook compilers. The most widely-deployed Brook compiler, targeting GPUs, did not allow any stream operators, only map and reduce-style kernels.

Multidimensional streaming

The synchronous dataflow model at the root of streaming programming models was generalized to multidimensional streams by the long-running Ptolemy project [26, 61, 18]. Multidimensional dataflow framework is a powerful framework for analyzing a subset of the computations expressible in Halide, and would be a natural starting point for further reasoning about and automatically optimizing Halide programs. However, it does not model arbitrary scatter and gather dependencies between stages, which are a critical in many image processing pipelines. Ptolemy also does not consider the potential of introducing redundant recomputation of intermediate values to enable a wider range of optimized dataflow schedules.

Loop optimization

Pipelines of simple map operations can be optimized by traditional loop fusion: merging multiple successive operations on each point into a single compound operation improves arithmetic intensity by maximizing producer-consumer locality, keeping intermediate data values in fast local memory (caches or registers) as it flows through the pipeline [90, 6, 7]. But traditional loop fusion does not apply to stencil operations, where neighboring points in a consumer stage depend on overlapping regions of a producer stage. Instead, stencils require a complex tradeoff between producer-consumer locality, synchronization, and redundant computation. Because this tradeoff is made by interleaving the order of allocation, execution, and communication of each stage, we call it the pipeline's *schedule*. These tradeoffs exist in scheduling individual iterated stencil computations in scientific applications, and the complexity of the choice space is reflected by the many different tiling and scheduling strategies introduced in past work [49, 30, 63]. In image processing pipelines, this tradeoff must be made for each producer-consumer relationship between stages in the graph—often dozens or hundreds—and the ideal schedule depends on the global interaction among every stage, often requiring the composition of many different strategies.

Data parallel & collection-oriented languages

Closely related to the classes of looping programs most often targeted by loop optimizers are numerous data parallel and collection-oriented languages, dating back at least to APL [42]. Since then, many data-parallel languages have been proposed. *Lisp and C* languages for the Connection Machines were notable early examples [50, 25]. Intel's Array Building Blocks provides an embedded language for data-parallel array processing in C++ [62]. Like with Halide's image functions, whole pipelines of operations are built up and optimized globally by a JIT compiler.

Particularly relevant in graphics, CUDA and OpenCL expose an imperative, single program-multiple data programming model which can target both GPUs and multicore CPUs with SIMD units [16, 64]. ispc provides a similar abstraction for SIMD processing on x86 CPUs [70]. Their semantics closely model the underlying machine. Like C, they allow the specification of very high performance implementations for many algorithms. But because parallel work distribution, synchronization, kernel fusion, and memory are all explicitly managed by the programmer, complex algorithms are often not composable in these languages, and the optimizations required are often specific to an architecture, so code must be rewritten for different platforms. An OpenCL pipeline optimized for a modern GPU will often be quite different from the same algorithm optimized for a multicore CPU, which will be different still from a mobile CPU version. Even optimizing for different GPUs can require different programs.

Image processing languages & systems

Domain-specific languages for image processing go back at least as far as Bell Labs' Pico and POPI [38, 37]. Most prior image processing languages and systems have focused on efficient expression of individual kernels. The most notable recent commercial examples are Apple's CoreImage system and Adobe's PixelBender [22, 71]. Both focus on high performance for individual kernels by exposing a simple data-parallel programming model which can be easily compiled to multithreaded SIMD code or GPU shaders. Optimizing compilers for PixelBender can perform traditional kernel fusion on stages with trivial dependence, in the absence of stencils. Neon embeds a similar kernel language in C# [32]. Recently, Cornwall et al. demonstrated fast GPU code generation for image processing code using polyhedral optimization [24].

At the other extreme, extensible image processing frameworks used in compositing systems and Adobe Photoshop [77, 3] do not address the performance of individual kernels, and instead focus entirely on providing a runtime model for composing individual image processing operations into large graphs while guaranteeing a fixed memory footprint. They do this by requiring all operations to conform to an explicitly tiled interface. Each operation must be able to produce an arbitrary-sized tile of output pixels requested by its caller, and for this region, it additionally reports what regions it requires of each of its inputs. A centralized scheduler manages resources by producing a tiled execution plan for the whole graph, querying each operation to determine its resource requirements under the plan, and reducing tile sizes until it can meet its memory budget across all operations in the graph.

Spreadsheets for Images extended the spreadsheet metaphor as a functional programming model for imaging operations [51]. Spreadsheets correspond to the functional-reactive programming model. Spreadsheets of image computations allowed lazy reevaluation of only the parts of large processing graphs which are changing, not unlike the compositing graphs of Shantzis [77].

Pan introduced a functional model for image processing much like our own, in which images are functions from coordinates to values [27]. Modest differences exist (Pan's images are functions over a *continuous* coordinate domain, while in ours the domain is *discrete*), but Pan is a close sibling of our *intrinsic algorithm* representation. However, it has no corollary to our model of *scheduling* and ultimate compilation. It exists as an interpreted embedding within Haskell, and as source to source compiler to C containing basic scalar and loop optimizations [28].

At its core, the representation of algorithms in the Halide language is similar to Pan, an embedded DSL for functional image synthesis in Haskell [27]. There are two major differences, both of which arise from our focus on generating high performance code:

1. In Halide, functions are defined over a discrete (integer) domain, while Pan's functions apply to continuous (floating point) coordinates. Pan supplements this with a built-in notion of sampling from continuous functions to discrete images. We chose to encode images on a discrete domain because it maps more directly to machine operation and storage (iteration over pixels; discrete memory locations). This makes the translation from algorithm to machine code performed by the code generator more direct, and makes it easier for the programmer to reason about performance. Resampling operations and other functionality may still be added as library functionality, or even as future syntactic sugar, but we have fundamentally chosen to make sampling part of the intrinsic definition of imaging operations, rather than trying to make it orthogonal to other choices.
2. While Pan is interpreted by the full Haskell runtime, Halide uses a staged compilation model to ultimately emit static code. A programmer's Halide code first *constructs* a complete Halide program (a graph of functions), and then *compiles* it into static machine code. In practice, this means that, while Pan supports much of Haskell's high-level functionality directly in a programmer's image processing code, Halide programs may only use the full functionality of the host language (C++) at *elaboration time*, while constructing the program graph; during execution of the generated Halide code, only the native

operations and types provided by Halide may be used.

Image processing architectures & optimization

As important an application as image processing is, enormous effort has gone into implementing many efficient and high-performance image processing pipelines. Several common patterns emerge from the fundamental structure of these pipelines in many different designs, both of software systems and of hardware architectures:

- The greatest single pressure is to maximize producer-consumer locality between stages.
- Computational parallelism comes first from exploiting highly regular, fine-grained data parallelism at the pixel level.
- Additional coarse-grained parallelism can be exposed, without sacrificing locality, by introducing some redundant work to decouple sets of pixels for independent computation.

The hardware image signal processors (ISPs) at the core of most cameras perform most processing using the *line buffering* computational pattern [53, 83]. In a line-buffered pipeline, intermediate data flowing between stages is buffered in small circular buffers, just large enough to hold a few scanlines of data. Line buffers are sized to fit just enough data to cover the stencil footprint of each successive stage, minimizing intermediate storage and maximizing producer-consumer locality. This core design optimizes locality. From here, exploiting parallelism requires tradeoffs. Fine-grained data parallelism can be exploited within scanlines by computing and buffering batches larger than the minimum required by the stencil footprint of each stage. Coarse-grained parallelism can be exposed by decoupling line buffered regions and redundantly recomputing shared values along their boundaries Halide was specifically designed to model patterns and tradeoffs like these.

Shading languages & the graphics pipeline

Elsewhere in graphics, the real-time graphics pipeline, as embodied by OpenGL and Direct3D, has been a hugely successful abstraction for high performance parallel programming precisely because the *schedule* is separated from the specification of user code in *programmable shaders* [14, 48, 54, 56]. This allows GPUs and drivers to efficiently execute a wide range of programs with little programmer control over parallelism and memory management. This separation of concerns is extremely effective, but it is specific to the design of a single pipeline. That pipeline also exhibits different characteristics than image processing pipelines, where reductions and stencil communication are common, and kernel fusion is essential for efficiency.

Embedded DSLs

Halide is designed as an embedded domain-specific language in C++. The *language* is built using a *type-directed embedding*: Halide functions and expressions are constructed using Halide-provided types, which overload common arithmetic operators to provide natural, native syntax for basic expressions. This is *not* to be confused with "template metaprogramming," commonly discussed in C++ [4], for example to embed shaders directly inside a C++ OpenGL program [57]; Halide's implementation uses virtually no templates. Template metaprogramming approaches rely on statically expanding programs at C++ compile-time (specifically, at template expansion-time); Halide programs are constructed dynamically, at C++ *run-time*, by the dynamic construction of C++ objects and execution of C++ functions. Internally, these objects construct an abstract syntax tree (AST) for the Halide program.

This design allows programmers to use the entire C++ "host language" as a metaprogramming layer while constructing Halide pipelines. For example, C++ control flow logic can define parameterized pipeline constructors; C++ for-loops can tersely describe the construction of whole families of related Halide computations; the construction of common pipeline functionality can be abstracted behind C++ functions. The same host language metaprogramming can also be applied to the description of Halide pipelines' *schedules*. Again, all of this host language logic is evaluated at run-time of the C++ program which defines the Halide pipeline, which is actually program-construction time for the Halide pipeline (equivalent to template-expansion time in C++, itself). This is similar to macro evaluation-time in a Lisp [34].

Halide is *compiled* using a multi-stage programming approach [82]. Rather than being directly executed while or after being constructed, the Halide program, built by the user at run-time of their C++ program defining the Halide algorithm, is then compiled directly from this representation into optimized machine code. The resulting machine code can either be emitted to an object file on disk (along with a C header declaring its interface), or directly emitted into memory to be called by the compiling process via JIT-compilation.

11 CONCLUSION

I have argued that the performance of image processing pipelines is limited by fundamental tradeoffs between parallelism, locality, and the amount of computation performed. These tradeoffs are determined not just by the algorithms used, but by the way their computations and data are *organized*. My thesis is that we can explicitly model the space of possible organizations for image processing pipelines at once generally enough to describe and compose many state-of-the-art optimizations, and precisely enough to automatically generate code competitive with hand-tuned expert implementations. By constraining our model of organization to a set of choices on *semiperfect loop nests*, we can describe an enormous space of useful organizations with the composition of a few primitive concepts, while also enabling predictably high-performance code generation for a wide range of architectures—from CPU SIMD units to GPUs—without the complexity of autoparallelization or autovectorization. We can then explicitly *decouple* the definition of the intrinsic algorithms from the way they are organized. Programming image processing pipelines this way is *better* than with existing languages: it enables dramatically simpler code to run faster than hand-tuned implementations, portably across a wide range of different architectures, while maintaining modularity and composability far greater than comparably optimized implementations in a traditional language C or CUDA.

The results achieved so far with real image processing pipelines in Halide demonstrate the feasibility and power of separating algorithms from their schedules. Changing the schedule enables a single algorithm definition to achieve high performance on a diversity of machines. On a single machine, it enables rapid performance space exploration. The algorithm specification also becomes considerably more concise once scheduling concerns are separated.

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

The complexity of optimized organizations is compositional.

It is widely accepted that writing and tuning optimized code is challenging, and that future architectures—with their ever-growing demand for parallelism and memory system optimization—will likely only make it harder. The challenge comes from the large and complex the space of plausible optimizations, the complexity of the code necessary to express each point in this space, and the often extreme change is this code required to move from one point to another. But is this complexity fundamental to the problem of optimization on modern architectures?

This thesis suggests not. The success of the Halide approach reveals a profound characteristic of the apparent complexity which makes optimization challenging: at least in the domain of image processing pipelines, the complexity in the space of useful organizations is mostly *compositional.* That is, most useful reorganizations to balance parallelism, locality, and the total amount of computation can be cast in terms of a few fundamental patterns. In Halide's schedules, we decompose these into the order of evaluation *within* each stage, and the granularity of interleaving *between* stages. Individually, these patterns are simple; the complexity emerges from their composition to globally optimize an entire pipeline.

This compositional nature makes sense. While inner-loop code will always matter, it is easily optimized locally, and after decades of compiler research, it is effectively addressed by existing techniques; the key challenges in optimization unsolved by existing compilers come at the points of interaction between pieces of a larger algorithm or system. The set of most useful ways individual pairs of pieces can be composed is small, but the complexity emerges from the combinatoric space of choices for all pairs in a large system.

Why a DSL?

Traditional libraries, based on subroutine composition, are no longer sufficient for high performance programming, because parallelism and locality—the key to performance and efficiency on modern hardware—are determined by global program structure, not local optimizations. Because of this, I argue that a DSL are the natural successor to a traditional library in providing flexible computation atop a given class of data structures.

Compiled DSLs provide two critical advantages over libraries of subroutines and data types:

1. Global program transformations require extensive code generation. To optimize parallelism and locality, an efficient system must defer computation until an entire program has been built, transform the result, and then compile it into efficient code. *Compilers* are the tool for mechanizing the transformation of computations.
2. Controlling the implementation of the core data structure, and deeply understanding the dependence across computations on this data structure, are essential to being able to perform global optimization. These are the key things we get from defining our own *language*, and making it aware of our particular data structures (domain-specific).

The functional model we chose for Images is both more flexible and easier to reason about than the traditional model of expressing image processing operations by mutating arrays of intermediate data. Even after starting down the path of building a DSL to perform global program transformations for parallelism and locality, we first tried to describe image processing pipelines as loops mutating buffers of intermediate data, which connected producer to consumer stages. Transformations in this model quickly became too complex to reason about. In particular, the freedom to expand the bounds of a computation or redundantly recompute values comes trivially when doing global loop synthesis from the functional model, but reasoning about guard bands and redundant computation explodes in complexity when successively applying individual transformations on imperative loops and memory allocations.

11.1 LIMITATIONS & FUTURE WORK

While initial results are promising, and Halide is seeing significant adoption, our work is far from done. I expect the greatest impact to come from work on a few key outstanding problems.

Automatic schedule inference

While manual scheduling is often feasible—even desirable—for optimization experts, and our initial results suggest that stochastic search combined with domain-specific heuristics can make autotuning feasible for realistic Halide programs, more traditional *model-driven* optimization is still important. Many programmers do not want to learn how to schedule their algorithms, and autotuning alone is not sufficient for many uses. First, where traditional compiler optimizations take as input just the program itself, autotuning requires a test harness and benchmark data, requiring more work from the programmer. Above all, even when final code will be optimized using autotuning, it is important for algorithms in development to deliver reasonable performance with seconds—not hours or days—of compile time.

Fortunately, Halide's model of schedules is just as suited to model-driven optimization as it is to stochastic search. Schedules provide a powerful parameterization of the space of choices to be made during global optimization of a pipeline, and the the schedule algebra (Chapter 7) provides a representation within which to reason about the composition of transformations. The other key component of a model-driven optimizer is a cost model, for which the task graph analysis of metrics related to parallelism, locality, and redundant (Chapter 4) work offers a strong foundation.

Modularity & composition

While Halide works well for individual algorithms and pipelines on the order of dozens of stages, several limitations make it challenging to build systems past this scale.

First, while Halide's decoupling of schedules from the definition of algorithms makes *algorithm* fragments easily composable, the semantics of schedules are still global and highly inter-dependent—both on the composition of the algorithm and on other choices made in the schedule. This limits modularity, and poses a particular challenge for building libraries of generic image processing algorithms: how can programmers schedule library modules without knowledge of their implementation, and how can library implementers schedule their modules without knowledge of the pipeline into which is is composed? Automatic schedule inference is one natural solution. Another interesting direction is improving support for composition in the scheduling language, itself.

Compile time for very large pipelines is another issue. Global synthesis of a single loop nest, and in particular static bounds inference via interval analysis through an entire pipeline, have costs which scale super-linearly with program complexity. A practical solution is to allow limited dynamic dispatch and bounds inference to decouple large pipelines into smaller pieces, and allow separate compilation.

Irregular algorithms & data structures

My choice of the phrase "image processing" to describe Halide's existing target domain is deliberate: so far, I think we have most effectively addressed only those algorithms which actually focus on processing at the level of pixels. However, many algorithms related to images—especially those in higher-level computer vision—require irregular algorithms and irregular data structures. It is easy to see how the Halide programming model can extend from dense regular grids to grids with sparse dimensions, and from static to dynamic dependence-based scheduling. It is much harder to imagine how the existing Halide language can generalize to encapsulate trees and other data structures in a unified and efficient way. At its core, this points to the issue of DSL composability: if domain-specific languages replace traditional libraries for interaction with important individual data structure classes, how can algorithms written in multiple different DSLs, using complementary data structures, compose into an efficient application?

Algorithms which depend on data-dependent early termination conditions, like sliding windows and boosted feature cascades common in state-of-the-art feature detectors [89, 88], are another challenge for the existing Halide model, which requires non-output-dependent bounds on all iterative updates. Extending the Halide model to allow isolated stream compaction steps appears to be a fruitful direction for efficiently bounding many types of irregularity [39].

New target architectures

New target architectures are a natural direction for future research and development. In particular, targets for distributed memory systems could allow Halide algorithms to process gigapixel- and terapixel-scale images like those in used in satellite imaging and mapping applications; targeting digital signal processors (DSPs) and future programming image signal processors (ISPs) could provide greater efficiency than existing CPU and GPU architectures for common image processing workloads; and directly synthesizing specialized logic for FPGAs or ASICs could provide even greater efficiency, still. In separate work, we have also shown that, by further restricting the Halide model to pure stencil operations without resampling, it is feasible both to automatically schedule *line-buffered pipelines* with optimal buffering, and to automatically synthesize efficient FPGA and ASIC implementations [35].

REFERENCES

[1] International technology roadmap for semiconductors. Technical report, 2013. Cited on p. 20.

[2] A. Adams, E. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy. The Frankencamera: An experimental platform for computational photography. *ACM Transactions on Graphics*, 29(4), 2010. Cited on p. 96.

[3] Adobe. The adobe photoshop CC SDK. http://www.adobe.com/devnet/photoshop/sdk.html, 2013. Cited on p. 115.

[4] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley Professional, 2001. Cited on pp. 32 and 118.

[5] A. Paul Alivisatos, Miyoung Chun, George M. Church, Ralph J. Greenspan, Michael L. Roukes, and Rafael Yuste. The brain activity map project and the challenge of functional connectomics. *Neuron*, 74, 2012. Cited on p. 19.

[6] Saman Amarasinghe. *Parallelizing Compiler Techniques based on Linear Inequalities*. PhD thesis, Stanford University, 1997. Cited on p. 113.

[7] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. IEEE Computer Society Press, 1991. Cited on p. 113.

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

[9] J. Ansel, S. Kamil, K. Veeramachaneni, U.M. O'Reilly, and S. Amarasinghe. OpenTuner: An extensible framework for program autotuning. Technical Report MIT-CSAIL-TR-2013-026, Massachusetts Institute of Technology, 2013. Cited on p. ii.

[10] M. Aubry, S. Paris, S. W. Hasinoff, J. Kautz, and F. Durand. Fast and robust pyramid-based image processing. Technical Report MIT-CSAIL-TR-2011-049, Massachusetts Institute of Technology, 2011. Cited on pp. 1, 9, and 101.

[11] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *ACM/IEEE Conference on Supercomputing*, 2012. Cited on p. 109.

[12] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989. Cited on p. 28.

[13] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996. Cited on p. 39.

[14] David Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, pages 724–734, 2006. Cited on p. 118.

[15] David Breen, Ron Fedkiw, Ken Museth, Stanley Osher, Guillermo Sapiro, and Ross Whitaker. *Level Set and PDE Methods for Computer Graphics*, 2004. Course at ACM SIGGRAPH. Cited on p. 98.

[16] I. Buck. GPU computing: Programming a massively parallel processor. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2007. Cited on pp. 60 and 114.

[17] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004. Cited on pp. 111 and 113.

[18] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Multirate signal processing in Ptolemy. In *Proceedings of the Internationl Conference on Acoustics, Speech, and Signal Processing*, 1991. Cited on p. 113.

[19] Z. Budimlić, M. Burke, V. Cavél, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto1, V. Sarkar, F. Schlimbach, and S. Taşırlar. Concurrent collections. *Scientific Programming*, 18(3–4):203–217, 2010. Cited on p. 112.

[20] J. Chen, S. Paris, and F. Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph.*, 26(3), 2007. Cited on pp. 99 and 100.

[21] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011. Cited on p. 111.

[22] CoreImage. Apple CoreImage programming guide, 2006. Cited on p. 115.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. Cited on p. 39.

[24] J. L. T. Cornwall, L. Howes, P. H. J. Kelly, P. Parsonage, and B. Nicoletti. High-performance SIMT code generation in an active visual effects library. In *Conference on Computing Frontiers*, 2009. Cited on p. 115.

[25] Thinking Machines Corporation. *C* reference manual.* 1987. Cited on p. 114.

[26] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 91(2), January 2003. Cited on p. 113.

[27] C. Elliott. Functional image synthesis. In *Proceedings of Bridges*, 2001. Cited on p. 116.

[28] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. Updated version of paper by the same name that appeared in SAIG '00 proceedings. Cited on p. 116.

[29] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *ACM/IEEE Conference on Supercomputing*, 2006. Cited on p. 109.

[30] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, 2005. Cited on pp. 110, 111, and 114.

[31] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002. Cited on pp. 111 and 112.

[32] Brian Guenter and Diego Nehab. The neon image processing language. Technical Report MSR-TR-2010-175, Microsoft Research, 2010. Cited on p. 115.

[33] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal – A data flow-oriented language for signal processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, 1986. Cited on p. 112.

[34] Timothy P. Hart. MACRO definitions for Lisp. Technical Report AIM-057, Massachusetts Institute of Technology, 1963. Cited on p. 118.

[35] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics*, 33(4), 2014. Cited on p. 125.

[36] J. Holewinski, L. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ICS*, 2012. Cited on p. 111.

[37] Gerard Holzmann. *Beyond Photography: The Digital Darkroom*. Prentice Hall, 1988. Cited on p. 115.

[38] Gerard J. Holzmann. Pico—a picture editor. *AT&T Technical Journal*, 66(2):2–13, March/April 1987. Cited on p. 115.

[39] Daniel Horn. Stream reduction operations for gpgpu applications. In Matt Pharr, editor, *GPU Gems 2*, chapter 36. 2006. Cited on p. 125.

[40] IPP. Intel Integrated Performance Primitives. http://software.intel.com/en-us/articles/intel-ipp/. Cited on p. 101.

[41] François Irigoin. *Partitionnement des boucles imbriquées - Une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Université PARIS-VI, 1989. Cited on p. 14.

[42] Kenneth Iverson. Chapter 6: a programming language. In Fred Brook and Kenneth Iverson, editors, *DRAFT copy for Automatic Data Processing*. 1960. Cited on p. 114.

[43] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001–. Cited on p. 107.

[44] S. Kamil, C. Chan, L. Oliker, J. Shalf, , and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, 2010. Cited on p. 111.

[45] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. Concurrent VLSI Architecture Tech Report 122, Stanford University, March 2002. Cited on p. 111.

[46] N. Karlsson, E. Di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M.E. Munich. The vSLAM algorithm for robust localization and mapping. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005. Cited on p. 19.

[47] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4), 1988. Cited on p. 98.

[48] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL shading language*. 2004. Cited on p. 118.

[49] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, 2007. Cited on pp. 110, 111, and 114.

[50] Clifford Lasser and Stephen M Omohundro. The essential Star-lisp manual. *Thinking Machines Corporation*, 1986. Cited on p. 114.

[51] Marc Levoy. Spreadsheets for images. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 139–146, July 1994. Cited on p. 115.

[52] C. Li, C. Xu, C. Gui, and M. D. Fox. Distance regularized level set evolution and its application to image segmentation. *IEEE Transactions on Image Processing*, 19(12):3243–3254, December 2010. Cited on p. 98.

[53] Tao Lin. Color interpolator and horizontal/vertical edge enhancer using two line buffer and alternating even/odd filters for digital camera, November 1999. Cited on pp. 69 and 117.

[54] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A user-programmable vertex engine. In *In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH 2001*. ACM Press, 2001. Cited on p. 118.

[55] Richard F. Lyon. The optical mouse, and an architectural methodology for smart digital sensors. 1981. Cited on p. 19.

[56] William R. Mark, R. Steven, Glanville Kurt, Akeley Mark, and J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22:896–907, 2003. Cited on p. 118.

[57] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Graphics Hardware 2002*, pages 57–68, September 2002. Cited on p. 118.

[58] J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. In *IJPP*, 2011. Cited on pp. 7 and 111.

[59] Thomas B. Moeslund and Erik Granum. A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding*, 81, 2001. Cited on p. 19.

[60] R. Moore. *Interval Analysis*. 1966. Cited on p. 46.

[61] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(7), July 2002. Cited on p. 113.

[62] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2011. Cited on p. 114.

[63] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Supercomputing*, 2010. Cited on pp. 110, 111, and 114.

[64] OpenCL. The OpenCL specification, version 1.2. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2011. Cited on pp. 60 and 114.

[65] OpenMP. OpenMP. http://openmp.org/. Cited on p. 101.

[66] S. Paris and F. Durand. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision*, 81(1), 2009. Cited on pp. 99 and 100.

[67] S. Paris, S. W. Hasinoff, and J. Kautz. Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid. *ACM Trans. Graph.*, 30(4), 2011. Cited on pp. 1 and 101.

[68] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand. Bilateral filtering: Theory and applications. Foundations and Trends in Computer Graphics and Vision, 2009. Cited on p. 99.

[69] Sylvain Paris. personal communication. Cited on p. 2.

[70] Matt Pharr and William R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing (InPar)*, 2012. Cited on pp. 101 and 114.

[71] PixelBender. Adobe PixelBender reference, 2010. Cited on p. 115.

[72] Harry Printz. *Automatic Mapping of Large Signal Processing Systems to a Parallel Machine*. Ph.D. Thesis, Carnegie Mellon University, 1991. Cited on p. 112.

[73] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. In *Proceedings of the IEEE*, volume 93, 2005. Cited on p. 108.

[74] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4), 2012. Cited on pp. ii and 95.

[75] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013. Cited on p. ii.

[76] John E. Savage. *Models of Computation*. Addison-Wesley, 1998. Cited on p. 35.

[77] M. A. Shantzis. A model for efficient and flexible image computing. In *ACM SIGGRAPH*, 1994. Cited on p. 115.

[78] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. Real-time human pose recognition in parts from a single depth image. In *Conference on Computer Vision and Pattern Recongition*, 2011. Cited on p. 19.

[79] Lloyd M. Smith, Jane Z. Sanders, Robert H. Kaiser, Peter Hughes, Chris Dodd, Charles R. Connell, Cheryl Heiner, Stephen B. H. Kent, and Leroy E. Hood. Fluorescence detection in automated DNA sequence analysis. *Nature*, 321, 1986. Cited on p. 19.

[80] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006. ACM Press. Cited on p. 19.

[81] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, 2005. Cited on p. 20.

[82] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Notices*, 32(12):203–217, December 1997. Cited on pp. 32 and 119.

[83] Tsutomu Takayama. White balance control for still image sensing apparatus, April 1990. Cited on pp. 69 and 117.

[84] Y. Tang, R. Chowdhury, B. Kuszmaul, C-K Luk, and C. Leiserson. The Pochoir stencil compiler. In *SPAA*, 2011. Cited on p. 111.

[85] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, 2002. Cited on p. 111.

[86] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 23, 2006. Cited on p. 19.

[87] Ping-Sheng Tseng. *A Parallelizing Compiler for Disributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, 1989. Cited on p. 112.

[88] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Conference on Computer Vision and Pattern Recongition*, 2001. Cited on p. 125.

[89] Paul Viola and Michael Jones. Robust real-time object detection. 2001. Cited on p. 125.

[90] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), Oct 1991. Cited on p. 113.

[91] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua. Hierarchical overlapped tiling. In *Proc. CGO*, 2012. Cited on p. 7.

APPENDICES

WRITING HALIDE PROGRAMS

A

A.1 BASICS

```
// Halide tutorial lesson 1.

// This lesson demonstrates basic usage of Halide as a JIT compiler for imaging.

// On linux, you can compile and run it like so:
// g++ lesson_01*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_01
// LD_LIBRARY_PATH=../bin ./lesson_01

// On os x:
// g++ lesson_01*.cpp -g -I ../include -L ../bin -lHalide -o lesson_01
// DYLD_LIBRARY_PATH=../bin ./lesson_01

// The only Halide header file you need is Halide.h. It includes all of Halide.
#include <Halide.h>

// We'll also include stdio for printf.
#include <stdio.h>

int main(int argc, char **argv) {

    // This program defines a single-stage imaging pipeline that
    // outputs a grayscale diagonal gradient.

    // A 'Func' object represents a pipeline stage. It's a pure
    // function that defines what value each pixel should have. You
    // can think of it as a computed image.
    Halide::Func gradient;

    // Var objects are names to use as variables in the definition of
    // a Func. They have no meaning by themselves.
    Halide::Var x, y;

    // Funcs are defined at any integer coordinate of its variables as
    // an Expr in terms of those variables and other functions.
    // Here, we'll define an Expr which has the value x + y. Vars have
    // appropriate operator overloading so that expressions like
    // 'x + y' become 'Expr' objects.
    Halide::Expr e = x + y;

    // Now we'll add a definition for the Func object. At pixel x, y,
    // the image will have the value of the Expr e. On the left hand
    // side we have the Func we're defining and some Vars. On the right
    // hand side we have some Expr object that uses those same Vars.
    gradient(x, y) = e;

    // This is the same as writing:
    //
    //   gradient(x, y) = x + y;
    //
    // which is the more common form, but we are showing the
    // intermediate Expr here for completeness.

    // That line of code defined the Func, but it didn't actually
    // compute the output image yet. At this stage it's just Funcs,
    // Exprs, and Vars in memory, representing the structure of our
    // imaging pipeline. We're meta-programming. This C++ program is
    // constructing a Halide program in memory. Actually computing
    // pixel data comes next.

    // Now we 'realize' the Func, which JIT compiles some code that
    // implements the pipeline we've defined, and then runs it.  We
    // also need to tell Halide the domain over which to evaluate the
    // Func, which determines the range of x and y above, and the
    // resolution of the output image. Halide.h also provides a basic
    // templatized Image type we can use. We'll make an 800 x 600
    // image.
    Halide::Image<int32_t> output = gradient.realize(800, 600);

    // Halide does type inference for you. Var objects represent
    // 32-bit integers, so the Expr object 'x + y' also represents a
    // 32-bit integer, and so 'gradient' defines a 32-bit image, and
    // so we got a 32-bit signed integer image out when we call
    // 'realize'. Halide types and type-casting rules are equivalent
    // to C.

    // Let's check everything worked, and we got the output we were
    // expecting:
    for (int j = 0; j < output.height(); j++) {
        for (int i = 0; i < output.width(); i++) {
            // We can access a pixel of an Image object using similar
            // syntax to defining and using functions.
            if (output(i, j) != i + j) {
                printf("Something went wrong!\n"
                       "Pixel %d, %d was supposed to be %d, but instead it's %d\n",
                       i, j, i+j, output(i, j));
                return -1;
            }
        }
    }

    // Everything worked! We defined a Func, then called 'realize' on
    // it to generate and run machine code that produced an Image.
    printf("Success!\n");

    return 0;
}
```

A.2 INPUT IMAGES

```
// Halide tutorial lesson 2.

// This lesson demonstrates how to pass in input images.

// On linux, you can compile and run it like so:
// g++ lesson_02*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -lpthread -ldl -o lesson_02
// LD_LIBRARY_PATH=../bin ./lesson_02

// On os x:
// g++ lesson_02*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -o lesson_02
// DYLD_LIBRARY_PATH=../bin ./lesson_02

// The only Halide header file you need is Halide.h. It includes all of Halide.
#include <Halide.h>

// Include some support code for loading pngs. It assumes there's an
// Image type, so we'll pull the one from Halide namespace;
using Halide::Image;
#include "image_io.h"

int main(int argc, char **argv) {

    // This program defines a single-stage imaging pipeline that
    // brightens an image.

    // First we'll load the input image we wish to brighten.
    Halide::Image<uint8_t> input = load<uint8_t>("images/rgb.png");

    // Next we define our Func object that represents our one pipeline
    // stage.
    Halide::Func brighter;

    // Our Func will have three arguments, representing the position
    // in the image and the color channel. Halide treats color
    // channels as an extra dimension of the image.
    Halide::Var x, y, c;

    // Normally we'd probably write the whole function definition on
    // one line. Here we'll break it apart so we can explain what
    // we're doing at every step.

    // For each pixel of the input image.
    Halide::Expr value = input(x, y, c);

    // Cast it to a floating point value.
    value = Halide::cast<float>(value);

    // Multiply it by 1.5 to brighten it. Halide represents real
    // numbers as floats, not doubles, so we stick an 'f' on the end
    // of our constant.
    value = value * 1.5f;

    // Clamp it to be less than 255, so we don't get overflow when we
    // cast it back to an 8-bit unsigned int.
    value = Halide::min(value, 255.0f);

    // Cast it back to an 8-bit unsigned integer.
    value = Halide::cast<uint8_t>(value);

    // Define the function.
    brighter(x, y, c) = value;

    // The equivalent one-liner to all of the above is:
    //
    // brighter(x, y, c) = Halide::cast<uint8_t>(min(input(x, y, c) * 1.5f, 255));
    //
    // In the shorter version:
    // - I skipped the cast to float, because multiplying by 1.5f does
    //   that automatically.
    // - I also used integer constants in clamp, because they get cast
    //   to match the type of the first argument.
    // - I left the Halide:: off clamp. It's unnecessary due to Koenig
    //   lookup.

    // Remember. All we've done so far is build a representation of a
    // Halide program in memory. We haven't actually processed any
    // pixels yet. We haven't even compiled that Halide program yet.

    // So now we'll realize the Func. The size of the output image
    // should match the size of the input image. If we just wanted to
    // brighten a portion of the input image we could request a
    // smaller size. If we request a larger size Halide will throw an
    // error at runtime telling us we're trying to read out of bounds
    // on the input image.
    Halide::Image<uint8_t> output = brighter.realize(input.width(), input.height(), input.channels());

    // Save the output for inspection. It should look like a bright parrot.
    save(output, "brighter.png");

    printf("Success!\n");
    return 0;
}
```

A.3 DEBUGGING, PART 1

```
// Halide tutorial lesson 3

// This lesson demonstrates how to inspect what the Halide compiler is producing.

// On linux, you can compile and run it like so:
// g++ lesson_03*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_03
// LD_LIBRARY_PATH=../bin ./lesson_03

// On os x:
// g++ lesson_03*.cpp -g -I ../include -L ../bin -lHalide -o lesson_03
// DYLD_LIBRARY_PATH=../bin ./lesson_03

#include <Halide.h>
#include <stdio.h>

// This time we'll just import the entire Halide namespace
using namespace Halide;

int main(int argc, char **argv) {

    // We'll start by defining the simple single-stage imaging
    // pipeline from lesson 1.

    // This lesson will be about debugging, but unfortunately in C++,
    // objects don't know their own names, which makes it hard for us
    // to understand the generated code. To get around this, you can
    // pass a string to the Func and Var constructors to give them a
    // name for debugging purposes.
    Func gradient("gradient");
    Var x("x"), y("y");
    gradient(x, y) = x + y;

    // Realize the function to produce an output image. We'll keep it
    // very small for this lesson.
    Image<int> output = gradient.realize(8, 8);

    // That line compiled and ran the pipeline. Try running this
    // lesson with the environment variable HL_DEBUG_CODEGEN set to
    // 1. It will print out the various stages of compilation, and a
    // pseudocode representation of the final pipeline.

    // If you set HL_DEBUG_CODEGEN to a higher number, you can see
    // more and more details of how Halide compiles your pipeline.
    // Setting HL_DEBUG_CODEGEN=2 shows the Halide code at each stage
    // of compilation, and also the llvm bitcode we generate at the
    // end.

    // If you'd prefer to read C code, the compile_to_c method emits C
    // code that implements the Halide pipeline. It can't compile
    // as-is without you also implementing some support functions, but
    // it can be helpful for understanding what the Halide pipeline is
    // doing. You pass it the name of the file, a list of arguments
    // the generated function should take (none in this case), and the
    // name of the generated function. Have a look inside gradient.cpp
    // after compiling and running this lesson.
    gradient.compile_to_c("gradient.cpp", std::vector<Argument>(), "gradient");

    // Using these two tricks -- setting HL_DEBUG_CODEGEN and calling
    // compile_to_c -- you can usually figure out what code Halide is
    // generating. In the next lesson we'll see how to snoop on Halide
    // at runtime.

    printf("Success!\n");
    return 0;
}
```

A.4 DEBUGGING, PART 2

```cpp
// Halide tutorial lesson 4

// This lesson demonstrates how to follow what Halide is doing at runtime.

// On linux, you can compile and run it like so:
// g++ lesson_04*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_04
// LD_LIBRARY_PATH=../bin ./lesson_04

// On os x:
// g++ lesson_04*.cpp -g -I ../include -L ../bin -lHalide -o lesson_04
// DYLD_LIBRARY_PATH=../bin ./lesson_04

#include <Halide.h>
#include <stdio.h>
using namespace Halide;

int main(int argc, char **argv) {

    Func gradient("gradient");
    Var x("x"), y("y");

    // We'll define our gradient function as before.
    gradient(x, y) = x + y;

    // And tell Halide that we'd like to be notified of all
    // evaluations.
    gradient.trace_stores();

    // Realize the function over an 8x8 region.
    printf("Evaluating gradient\n");
    Image<int> output = gradient.realize(8, 8);

    // This will print out all the times gradient(x, y) gets
    // evaluated.

    // Now that we can snoop on what Halide is doing, let's try our
    // first scheduling primitive. We'll make a new version of
    // gradient that processes each scanline in parallel.
    Func parallel_gradient("parallel_gradient");
    parallel_gradient(x, y) = x + y;

    // We'll also trace this function.
    parallel_gradient.trace_stores();

    // Things are the same so far. We've defined the algorithm, but
    // haven't said anything about how to schedule it. In general,
    // exploring different scheduling decisions doesn't change the code
    // that describes the algorithm.

    // Now we tell Halide to use a parallel for loop over the y
    // coordinate. On linux we run this using a thread pool and a task
    // queue. On os x we call into grand central dispatch, which does
    // the same thing for us.
    parallel_gradient.parallel(y);

    // This time the printfs should come out of order, because each
    // scanline is potentially being processed in a different
    // thread. The number of threads should adapt to your system, but
    // on linux you can control it manually using the environment
    // variable HL_NUMTHREADS.
    printf("\nEvaluating parallel_gradient\n");
    parallel_gradient.realize(8, 8);

    printf("Success!\n");
    return 0;
}
```

A.5 SCHEDULING, PART 1

```
// Halide tutorial lesson 5

// This lesson demonstrates how to manipulate the order in which you
// evaluate pixels in a Func, including vectorization,
// parallelization, unrolling, and tiling.

// On linux, you can compile and run it like so:
// g++ lesson_05*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_05
// LD_LIBRARY_PATH=../bin ./lesson_05

// On os x:
// g++ lesson_05*.cpp -g -I ../include -L ../bin -lHalide -o lesson_05
// DYLD_LIBRARY_PATH=../bin ./lesson_05

#include <Halide.h>
#include <stdio.h>
using namespace Halide;

int main(int argc, char **argv) {

    // We're going to define and schedule our gradient function in
    // several different ways, and see what order pixels are computed
    // in.

    Var x("x"), y("y");

    // First we observe the default ordering.
    {
        Func gradient("gradient");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // By default we walk along the rows and then down the columns.
        printf("Evaluating gradient row-major\n");
        Image<int> output = gradient.realize(4, 4);

        // The equivalent C is:
        printf("Equivalent C:\n");
        for (int y = 0; y < 4; y++) {
            for (int x = 0; x < 4; x++) {
                printf("Evaluating at %d, %d: %d\n", x, y, x + y);
            }
        }
        printf("\n\n");
    }

    // Reorder variables.
    {
        Func gradient("gradient_col_major");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // If we reorder x and y, we can walk down the columns
        // instead. The reorder call takes the arguments of the func,
        // and sets a new nesting order for the for loops that are
        // generated. The arguments are specified from the innermost
        // loop out, so the following call puts y in the inner loop:
        gradient.reorder(y, x);

        printf("Evaluating gradient column-major\n");
        Image<int> output = gradient.realize(4, 4);

        printf("Equivalent C:\n");
        for (int x = 0; x < 4; x++) {
            for (int y = 0; y < 4; y++) {
                printf("Evaluating at %d, %d: %d\n", x, y, x + y);
            }
        }
        printf("\n\n");
    }

    // Split a variable into two.
    {
        Func gradient("gradient_split");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // The most powerful primitive scheduling operation you can do
        // to a var is to split it into inner and outer sub-variables:
        Var x_outer, x_inner;
        gradient.split(x, x_outer, x_inner, 2);

        // This breaks the loop over x into two nested loops: an outer
        // one over x_outer, and an inner one over x_inner. The last
        // argument to split was the "split factor". The inner loop
        // runs from zero to the split factor. The outer loop runs
        // from zero to the extent required of x (4 in this case)
        // divided by the split factor. Within the loops, the old
        // variable is defined to be outer * factor + inner. If the
        // old loop started at a value other than zero, then that is
        // also added within the loops.

        printf("Evaluating gradient with x split into x_outer and x_inner \n");
        Image<int> output = gradient.realize(4, 4);

        printf("Equivalent C:\n");
        for (int y = 0; y < 4; y++) {
            for (int x_outer = 0; x_outer < 2; x_outer++) {
                for (int x_inner = 0; x_inner < 2; x_inner++) {
                    int x = x_outer * 2 + x_inner;
                    printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                }
            }
        }
        printf("\n\n");

        // Note that the order of evaluation of pixels didn't actually
        // change! Splitting by itself does nothing, but it does open
        // up all of the scheduling possibilities that we will explore
        // below.
    }

    // Fuse two variables into one.
    {
        Func gradient("gradient_fused");
        gradient(x, y) = x + y;

        // The opposite of splitting is 'fusing'. Fusing two variables
        // merges the two loops into a single for loop over the
        // product of the extents. Fusing is less important that
        // splitting, but it also sees use (as we'll see later in this
        // lesson). Like splitting, fusing by itself doesn't change
        // the order of evaluation.
        Var fused;
        gradient.fuse(x, y, fused);

        printf("Evaluating gradient with x and y fused\n");
        Image<int> output = gradient.realize(4, 4);

        printf("Equivalent C:\n");
        for (int fused = 0; fused < 4*4; fused++) {
            int y = fused / 4;
            int x = fused % 4;
            printf("Evaluating at %d, %d: %d\n", x, y, x + y);
        }
    }

    // Evaluating in tiles.
    {
        Func gradient("gradient_tiled");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // Now that we can both split and reorder, we can do tiled
        // evaluation. Let's split both x and y by a factor of two,
        // and then reorder the vars to express a tiled traversal.
        //
        // A tiled traversal splits the domain into small rectangular
        // tiles, and outermost iterates over the tiles, and within
        // that iterates over the points within each tile. It can be
        // good for performance if neighboring pixels use overlapping
        // input data, for example in a blur. We can express a tiled
        // traversal like so:
        Var x_outer, x_inner, y_outer, y_inner;
        gradient.split(x, x_outer, x_inner, 2);
        gradient.split(y, y_outer, y_inner, 2);
        gradient.reorder(x_inner, y_inner, x_outer, y_outer);

        // This pattern is common enough that there's a shorthand for it:
        // gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);

        printf("Evaluating gradient in 2x2 tiles\n");
        Image<int> output = gradient.realize(4, 4);

        printf("Equivalent C:\n");
        for (int y_outer = 0; y_outer < 2; y_outer++) {
            for (int x_outer = 0; x_outer < 2; x_outer++) {
                for (int y_inner = 0; y_inner < 2; y_inner++) {
                    for (int x_inner = 0; x_inner < 2; x_inner++) {
                        int x = x_outer * 2 + x_inner;
                        int y = y_outer * 2 + y_inner;
                        printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                    }
                }
            }
        }
        printf("\n\n");
    }

    // Evaluating in vectors.
    {
        Func gradient("gradient_in_vectors");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // The nice thing about splitting is that it guarantees the
        // inner variable runs from zero to the split factor. Most of
        // the time the split-factor will be a compile-time constant,
        // so we can replace the loop over the inner variable with a
        // single vectorized computation. This time we'll split by a
        // factor of four, because on X86 we can use SSE to compute in
        // 4-wide vectors.
        Var x_outer, x_inner;
        gradient.split(x, x_outer, x_inner, 4);
        gradient.vectorize(x_inner);

        // Splitting and then vectorizing the inner variable is common
        // enough that there's a short-hand for it. We could have also
        // said:
        //
        // gradient.vectorize(x, 4);
        //
        // which is equivalent to:
        //
        // gradient.split(x, x, x_inner, 4);
        // gradient.vectorize(x_inner);
        //
        // Note that in this case we reused the name 'x' as the new
        // outer variable. Later scheduling calls that refer to x
        // will refer to this new outer variable named x.
        //
        // Our snoop function isn't set-up to print out vectors, this
        // is why we included one called snoopx4 above.

        // This time we'll evaluate over an 8x4 box, so that we have
        // more than one vector of work per scanline.
        printf("Evaluating gradient with x_inner vectorized \n");
        Image<int> output = gradient.realize(8, 4);

        printf("Equivalent C:\n");
        for (int y = 0; y < 4; y++) {
            for (int x_outer = 0; x_outer < 2; x_outer++) {
                // The loop over x_inner has gone away, and has been
                // replaced by a vectorized version of the
                // expression. On x86 processors, Halide generates SSE
                // for all of this.
                int x_vec[] = {x_outer * 4 + 0,
                               x_outer * 4 + 1,
                               x_outer * 4 + 2,
                               x_outer * 4 + 3};
                int val[] = {x_vec[0] + y,
                             x_vec[1] + y,
                             x_vec[2] + y,
                             x_vec[3] + y};
                printf("Evaluating at <%d, %d, %d, %d>, <%d, %d, %d, %d>: <%d, %d, %d, %d>\n",
                       x_vec[0], x_vec[1], x_vec[2], x_vec[3],
                       y, y, y, y,
                       val[0], val[1], val[2], val[3]);
            }
        }
        printf("\n\n");
    }

    // Unrolling a loop.
    {
        Func gradient("gradient_in_vectors");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // If multiple pixels share overlapping data, it can make
        // sense to unroll a computation so that shared values are
        // only computed or loaded once. We do this similarly to how
        // we expressed vectorizing. We split a dimension and then
        // fully unroll the loop of the inner variable. Unrolling
        // doesn't change the order in which things are evaluated.
        Var x_outer, x_inner;
        gradient.split(x, x_outer, x_inner, 2);
        gradient.unroll(x_inner);

        // The shorthand for this is:
        // gradient.unroll(x, 2);

        printf("Evaluating gradient unrolled by a factor of two\n");
        Image<int> result = gradient.realize(4, 4);

        printf("Equivalent C:\n");
        for (int y = 0; y < 4; y++) {
            for (int x_outer = 0; x_outer < 2; x_outer++) {
                // Instead of a for loop over x_inner, we get two
                // copies of the innermost statement.
                {
                    int x_inner = 0;
                    int x = x_outer * 2 + x_inner;
                    printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                }
                {
                    int x_inner = 1;
                    int x = x_outer * 2 + x_inner;
                    printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                }
            }
        }


    }

    // Splitting by factors that don't divide the extent.
    {
        Func gradient("gradient_split_5x4");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // Splitting guarantees that the inner loop runs from zero to
        // the split factor, which is important for the uses we saw
        // above. So what happens when the total extent we wish to
        // evaluate x over isn't a multiple of the split factor? We'll
        // split by a factor of two again, but now we'll evaluate
        // gradient over a 5x4 box instead of the 4x4 box we've been
        // using.
        Var x_outer, x_inner;
        gradient.split(x, x_outer, x_inner, 2);

        printf("Evaluating gradient over a 5x4 box with x split by two \n");
        Image<int> output = gradient.realize(5, 4);

        printf("Equivalent C:\n");
        for (int y = 0; y < 4; y++) {
            for (int x_outer = 0; x_outer < 3; x_outer++) { // Now runs from 0 to 3
                for (int x_inner = 0; x_inner < 2; x_inner++) {
                    int x = x_outer * 2;
                    // Before we add x_inner, make sure we don't
                    // evaluate points outside of the 5x4 box. We'll
                    // clamp x to be at most 3 (5 minus the split
                    // factor).
                    if (x > 3) x = 3;
                    x += x_inner;
                    printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                }
            }
        }
        printf("\n\n");

        // If you read the output, you'll see that some coordinates
        // were evaluated more than once! That's generally OK, because
        // pure Halide functions have no side-effects, so it's safe to
        // evaluate the same point multiple times. If you're calling
        // out to C functions like we are, it's your responsibility to
        // make sure you can handle the same point being evaluated
        // multiple times.

        // The general rule is: If we require x from x_min to x_min + x_extent, and
        // we split by a factor 'factor', then:
        //
        // x_outer runs from 0 to (x_extent + factor - 1)/factor
        // x_inner runs from 0 to factor
        // x = min(x_outer * factor, x_extent - factor) + x_inner + x_min
        //
        // In our example, x_min was 0, x_extent was 5, and factor was 2.
        // However, if you write a Halide function with an update
        // definition (see lesson 9), then it is not safe to evaluate
        // the same point multiple times, so we won't apply this
        // trick. Instead the range of values computed will be rounded
        // up to the next multiple of the split factor.
    }

    // Fusing, tiling, and parallelizing.
    {
        // We saw in the previous lesson that we can parallelize
        // across a variable. Here we combine it with fusing and
        // tiling to express a useful pattern - processing tiles in
        // parallel.

        // This is where fusing shines. Fusing helps when you want to
        // parallelize across multiple dimensions without introducing
        // nested parallelism. Nested parallelism (parallel for loops
        // within parallel for loops) is supported by Halide, but
        // often gives poor performance compared to fusing the
        // parallel variables into a single parallel for loop.

        Func gradient("gradient_fused_tiles");
        gradient(x, y) = x + y;
        gradient.trace_stores();

        // First we'll tile, then we'll fuse the tile indices and
        // parallelize across the combination.
        Var x_outer, y_outer, x_inner, y_inner, tile_index;
        gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);
        gradient.fuse(x_outer, y_outer, tile_index);
        gradient.parallel(tile_index);

        // The scheduling calls all return a reference to the Func, so
        // you can also chain them together into a single statement to
        // make things slightly clearer:
        //
        // gradient
        //     .tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2)
        //     .fuse(x_outer, y_outer, tile_index)
        //     .parallel(tile_index);


        printf("Evaluating gradient tiles in parallel\n");
        Image<int> output = gradient.realize(4, 4);

        // The tiles should occur in arbitrary order, but within each
        // tile the pixels will be traversed in row-major order.

        printf("Equivalent (serial) C:\n");
        // This outermost loop should be a parallel for loop, but that's hard in C.
        for (int tile_index = 0; tile_index < 4; tile_index++) {
            int y_outer = tile_index / 2;
            int x_outer = tile_index % 2;
            for (int y_inner = 0; y_inner < 2; y_inner++) {
                for (int x_inner = 0; x_inner < 2; x_inner++) {
                    int y = y_outer * 2 + y_inner;
                    int x = x_outer * 2 + x_inner;
                    printf("Evaluating at %d, %d: %d\n", x, y, x + y);
                }
            }
        }
        printf("\n\n");
    }

    // Putting it all together.
    {
        // Are you ready? We're going to use all of the features above now.
```

```
Func gradient_fast("gradient_fast");
gradient_fast(x, y) = x + y;

// We'll process 256x256 tiles in parallel.
Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient_fast
    .tile(x, y, x_outer, y_outer, x_inner, y_inner, 256, 256)
    .fuse(x_outer, y_outer, tile_index)
    .parallel(tile_index);

// We'll compute two scanlines at once while we walk across
// each tile. We'll also vectorize in x. The easiest way to
// express this is to recursively tile again within each tile
// into 4x2 subtiles, then vectorize the subtiles across x and
// unroll them across y:
Var x_inner_outer, y_inner_outer, x_vectors, y_pairs;
gradient_fast
    .tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 2)
    .vectorize(x_vectors)
    .unroll(y_pairs);

// Note that we didn't do any explicit splitting or
// reordering. Those are the most important primitive
// operations, but mostly they are buried underneath tiling,
// vectorizing, or unrolling calls.

// Now let's evaluate this over a range which is not a
// multiple of the tile size.

// If you like you can turn on tracing, but it's going to
// produce a lot of printfs. Instead we'll compute the answer
// both in C and Halide and see if the answers match.
Image<int> result = gradient_fast.realize(800, 600);

printf("Checking Halide result against equivalent C...\n");
for (int tile_index = 0; tile_index < 4 * 3; tile_index++) {
    int y_outer = tile_index / 4;
    int x_outer = tile_index % 4;
    for (int y_inner_outer = 0; y_inner_outer < 256/2; y_inner_outer++) {
        for (int x_inner_outer = 0; x_inner_outer < 256/4; x_inner_outer++) {
            // We're vectorized across x
            int x = std::min(x_outer * 256, 800-256) + x_inner_outer*4;
            int x_vec[4] = {x + 0,
                            x + 1,
                            x + 2,
                            x + 3};

            // And we unrolled across y
            int y_base = std::min(y_outer * 256, 600-256) + y_inner_outer*2;
            {
                // y_pairs = 0
                int y = y_base + 0;
                int y_vec[4] = {y, y, y, y};
                int val[4] = {x_vec[0] + y_vec[0],
                              x_vec[1] + y_vec[1],
                              x_vec[2] + y_vec[2],
                              x_vec[3] + y_vec[3]};

                // Check the result.
                for (int i = 0; i < 4; i++) {
                    if (result(x_vec[i], y_vec[i]) != val[i]) {
                        printf("There was an error at %d %d!\n", x_vec[i], y_vec[i]);
                        return -1;
                    }
                }
            }
            {
                // y_pairs = 1
```

```
                    int y = y_base + 1;
                    int y_vec[4] = {y, y, y, y};
                    int val[4] = {x_vec[0] + y_vec[0],
                                  x_vec[1] + y_vec[1],
                                  x_vec[2] + y_vec[2],
                                  x_vec[3] + y_vec[3]};

                    // Check the result.
                    for (int i = 0; i < 4; i++) {
                        if (result(x_vec[i], y_vec[i]) != val[i]) {
                            printf("There was an error at %d %d!\n", x_vec[i], y_vec[i]);
                            return -1;
                        }
                    }
                }
            }
        }
    }
}

// Note that in the Halide version, the algorithm is specified
// once at the top, separately from the optimizations, and there
// aren't that many lines of code total. Compare this to the C
// version. There's more code (and it isn't even parallelized or
// vectorized properly). More annoyingly, the statement of the
// algorithm (the result is x plus y) is buried in multiple places
// within the mess. This C code is hard to write, hard to read,
// hard to debug, and hard to optimize further. This is why Halide
// exists.

printf("Success!\n");
return 0;
}
```

A.6 REALIZING OVER SHIFTED DOMAINS

```
// Halide tutorial lesson 6.

// This lesson demonstrates how to evaluate a Func over a domain that
// does not start at (0, 0).

// On linux, you can compile and run it like so:
// g++ lesson_06*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_06
// LD_LIBRARY_PATH=../bin ./lesson_06

// On os x:
// g++ lesson_06*.cpp -g -I ../include -L ../bin -lHalide -o lesson_06
// DYLD_LIBRARY_PATH=../bin ./lesson_06

#include <Halide.h>
#include <stdio.h>

using namespace Halide;

int main(int argc, char **argv) {

    // The last lesson was quite involved, and scheduling complex
    // multi-stage pipelines is ahead of us. As an interlude, let's
    // consider something easy: evaluating funcs over rectangular
    // domains that do not start at the origin.

    // We define our familiar gradient function.
    Func gradient("gradient");
    Var x("x"), y("y");
    gradient(x, y) = x + y;

    // And turn on tracing so we can see how it is being evaluated.
    gradient.trace_stores();

    // Previously we've realized gradient like so:
    //
    // gradient.realize(8, 8);
    //
    // This does three things internally:
    // 1) Generates code than can evaluate gradient over an arbitrary
    // rectangle.
    // 2) Allocates a new 8 x 8 image.
    // 3) Runs the generated code to evaluate gradient for all x, y
    // from (0, 0) to (7, 7) and puts the result into the image.
    // 4) Returns the new image as the result of the realize call.

    // What if we're managing memory carefully and don't want Halide
    // to allocate a new image for us? We can call realize another
    // way. We can pass it an image we would like it to fill in. The
    // following evaluates our Func into an existing image:
    printf("Evaluating gradient from (0, 0) to (7, 7)\n");
    Image<int> result(8, 8);
    gradient.realize(result);

    // Let's check it did what we expect:
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            if (result(x, y) != x + y) {
                printf("Something went wrong!\n");
                return -1;
            }
        }
    }

    // Now let's evaluate gradient over a 5 x 7 rectangle that starts
    // somewhere else -- at position (100, 50). So x and y will run
    // from (100, 50) to (104, 56) inclusive.
    // We start by creating an image that represents that rectangle:
    Image<int> shifted(5, 7); // In the constructor we tell it the size.
    shifted.set_min(100, 50); // Then we tell it the top-left corner.

    printf("Evaluating gradient from (100, 50) to (104, 56)\n");

    // Note that this won't need to compile any new code, because when
    // we realized it the first time, we generated code capable of
    // evaluating gradient over an arbitrary rectangle.
    gradient.realize(shifted);

    // From C++, we also access the image object using coordinates
    // that start at (100, 50).
    for (int y = 50; y < 57; y++) {
        for (int x = 100; x < 105; x++) {
            if (shifted(x, y) != x + y) {
                printf("Something went wrong!\n");
                return -1;
            }
        }
    }
    // The image 'shifted' stores the value of our Func over a domain
    // that starts at (100, 50), so asking for shifted(0, 0) would in
    // fact read out-of-bounds and probably crash.

    // What if we want to evaluate our Func over some region that
    // isn't rectangular? Too bad. Halide only does rectangles :)

    printf("Success!\n");
    return 0;
}
```

A.7 MULTI-STAGE PIPELINES

```
// Halide tutorial lesson 7

// This lesson demonstrates how express multi-stage pipelines.

// On linux, you can compile and run it like so:
// g++ lesson_07*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -lpthread -ldl -o lesson_07
// LD_LIBRARY_PATH=../bin ./lesson_07

// On os x:
// g++ lesson_07*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -o lesson_07
// DYLD_LIBRARY_PATH=../bin ./lesson_07

#include <Halide.h>
#include <stdio.h>

using namespace Halide;

// Support code for loading pngs.
#include "image_io.h"

int main(int argc, char **argv) {
    // First we'll declare some Vars to use below.
    Var x("x"), y("y"), c("c");

    // Now we'll express a multi-stage pipeline that blurs an image
    // first horizontally, and then vertically.
    {
        // Take a color 8-bit input
        Image<uint8_t> input = load<uint8_t>("images/rgb.png");

        // Upgrade it to 16-bit, so we can do math without it overflowing.
        Func input_16("input_16");
        input_16(x, y, c) = cast<uint16_t>(input(x, y, c));

        // Blur it horizontally:
        Func blur_x("blur_x");
        blur_x(x, y, c) = (input_16(x-1, y, c) + 2*input_16(x, y, c) + input_16(x+1, y, c))/4;

        // Blur it vertically:
        Func blur_y("blur_y");
        blur_y(x, y, c) = (blur_x(x, y-1, c) + 2*blur_x(x, y, c) + blur_x(x, y+1, c))/4;

        // Convert back to 8-bit.
        Func output("output");
        output(x, y, c) = cast<uint8_t>(blur_y(x, y, c));

        // Each Func in this pipeline calls a previous one using
        // familiar function call syntax (we've overloaded operator()
        // on Func objects). A Func may call any other Func that has
        // been given a definition. This restriction prevents
        // pipelines with loops in them. Halide pipelines are always
        // feed-forward graphs of Funcs.

        // Now let's realize it...

        // Image<uint8_t> result = output.realize(input.width(), input.height(), 3);

        // Except that the line above is not going to work. Uncomment
        // it to see what happens.

        // Realizing this pipeline over the same domain as the input
        // image requires reading pixels out of bounds in the input,
        // because the blur_x stage reaches outwards horizontally, and
        // the blur_y stage reaches outwards vertically. Halide
        // detects this by injecting a piece of code at the top of the
        // pipeline that computes the region over which the input will
        // be read. When it starts to run the pipeline it first runs
        // this code, determines that the input will be read out of
        // bounds, and refuses to continue. No actual bounds checks
        // occur in the inner loop; that would be slow.
        //
        // So what do we do? There are a few options. If we realize
        // over a domain shifted inwards by one pixel, we won't be
        // asking the Halide routine to read out of bounds. We saw how
        // to do this in the previous lesson:
        Image<uint8_t> result(input.width()-2, input.height()-2, 3);
        result.set_min(1, 1);
        output.realize(result);

        // Save the result. It should look like a slightly blurry
        // parrot, and it should be two pixels narrower and two pixels
        // shorter than the input image.
        save(result, "blurry_parrot_1.png");

        // This is usually the fastest way to deal with boundaries:
        // don't write code that reads out of bounds :) The more
        // general solution is our next example.
    }

    // The same pipeline, with a boundary condition on the input.
    {
        // Take a color 8-bit input
        Image<uint8_t> input = load<uint8_t>("images/rgb.png");

        // This time, we'll wrap the input in a Func that prevents
        // reading out of bounds:
        Func clamped("clamped");

        // Define an expression that clamps x to lie within the the
        // range [0, input.width()-1].
        Expr clamped_x = clamp(x, 0, input.width()-1);
        // Similarly clamp y.
        Expr clamped_y = clamp(y, 0, input.height()-1);
        // Load from input at the clamped coordinates. This means that
        // no matter how we evaluated the Func 'clamped', we'll never
        // read out of bounds on the input. This is a clamp-to-edge
        // style boundary condition, and is the simplest boundary
        // condition to express in Halide.
        clamped(x, y, c) = input(clamped_x, clamped_y, c);

        // Upgrade it to 16-bit, so we can do math without it
        // overflowing. This time we'll refer to our new Func
        // 'clamped', instead of referring to the input image
        // directly.
        Func input_16("input_16");
        input_16(x, y, c) = cast<uint16_t>(clamped(x, y, c));

        // The rest of the pipeline will be the same...

        // Blur it horizontally:
        Func blur_x("blur_x");
        blur_x(x, y, c) = (input_16(x-1, y, c) + 2*input_16(x, y, c) + input_16(x+1, y, c))/4;

        // Blur it vertically:
        Func blur_y("blur_y");
        blur_y(x, y, c) = (blur_x(x, y-1, c) + 2*blur_x(x, y, c) + blur_x(x, y+1, c))/4;

        // Convert back to 8-bit.
        Func output("output");
        output(x, y, c) = cast<uint8_t>(blur_y(x, y, c));

        // This time it's safe to evaluate the output over the some
        // domain as the input, because we have a boundary condition.
        Image<uint8_t> result = output.realize(input.width(), input.height(), 3);
```

```
        // Save the result. It should look like a slightly blurry
        // parrot, but this time it will be the same size as the
        // input.
        save(result, "blurry_parrot_2.png");
    }

    printf("Success!\n");
    return 0;
}
```

A.8 SCHEDULING, PART 2

```
// Halide tutorial lesson 8

// This lesson demonstrates how schedule multi-stage pipelines.

// On linux, you can compile and run it like so:
// g++ lesson_08*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_08
// LD_LIBRARY_PATH=../bin ./lesson_08

// On os x:
// g++ lesson_08*.cpp -g -I ../include -L ../bin -lHalide -o lesson_08
// DYLD_LIBRARY_PATH=../bin ./lesson_08

#include <Halide.h>
#include <stdio.h>

using namespace Halide;

int main(int argc, char **argv) {
    // First we'll declare some Vars to use below.
    Var x("x"), y("y");

    // Let's examine various scheduling options for a simple two stage
    // pipeline. We'll start with the default schedule:
    {
        Func producer("producer_default"), consumer("consumer_default");

        // The first stage will be some simple pointwise math similar
        // to our familiar gradient function. The value at position x,
        // y is the sqrt of product of x and y.
        producer(x, y) = sqrt(x * y);

        // Now we'll add a second stage which adds together multiple
        // points in the first stage.
        consumer(x, y) = (producer(x, y) +
                          producer(x, y+1) +
                          producer(x+1, y) +
                          producer(x+1, y+1));

        // We'll turn on tracing for both functions.
        consumer.trace_stores();
        producer.trace_stores();

        // And evaluate it over a 5x5 box.
        printf("\nEvaluating producer-consumer pipeline with default schedule\n");
        consumer.realize(4, 4);

        // There were no messages about computing values of the
        // producer. This is because the default schedule fully
        // inlines 'producer' into 'consumer'. It is as if we had
        // written the following code instead:

        // consumer(x, y) = (sqrt(x * y) +
        //                   sqrt(x * (y + 1)) +
        //                   sqrt((x + 1) * y) +
        //                   sqrt((x + 1) * (y + 1)));

        // All calls to 'producer' have been replaced with the body of
        // 'producer', with the arguments subtituted in for the
        // variables.

        // The equivalent C code is:
        float result[4][4];
        for (int y = 0; y < 4; y++) {
            for (int x = 0; x < 4; x++) {
                result[y][x] = (sqrt(x*y) +
                                sqrt(x*(y+1)) +
                                 sqrt((x+1)*y) +
                                 sqrt((x+1)*(y+1)));
            }
        }
        printf("\n");
    }

    // Next we'll examine the next simplest option - computing all
    // values required in the producer before computing any of the
    // consumer. We call this schedule "root".
    {
        // Start with the same function definitions:
        Func producer("producer_root"), consumer("consumer_root");
        producer(x, y) = sqrt(x * y);
        consumer(x, y) = (producer(x, y) +
                          producer(x, y+1) +
                          producer(x+1, y) +
                          producer(x+1, y+1));

        // Tell Halide to evaluate all of producer before any of consumer.
        producer.compute_root();

        // Turn on tracing.
        consumer.trace_stores();
        producer.trace_stores();

        // Compile and run.
        printf("\nEvaluating producer.compute_root()\n");
        consumer.realize(4, 4);

        // Reading the output we can see that:
        // A) There were stores to producer.
        // B) They all happened before any stores to consumer.


        // Equivalent C:

        float result[4][4];

        // Allocate some temporary storage for the producer.
        float producer_storage[5][5];

        // Compute the producer.
        for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                producer_storage[y][x] = sqrt(x * y);
            }
        }

        // Compute the consumer. Skip the prints this time.
        for (int y = 0; y < 4; y++) {
            for (int x = 0; x < 4; x++) {
                result[y][x] = (producer_storage[y][x] +
                                producer_storage[y+1][x] +
                                producer_storage[y][x+1] +
                                producer_storage[y+1][x+1]);
            }
        }

        // Note that consumer was evaluated over a 4x4 box, so Halide
        // automatically inferred that producer was needed over a 5x5
        // box. This is the same 'bounds inference' logic we saw in
        // the previous lesson, where it was used to detect and avoid
        // out-of-bounds reads from an input image.
    }

    // Let's compare the two approaches above from a performance
    // perspective.
```

```
// Full inlining (the default schedule):
// - Temporary memory allocated: 0
// - Loads: 0
// - Stores: 16
// - Calls to sqrt: 64

// producer.compute_root():
// - Temporary memory allocated: 25 floats
// - Loads: 64
// - Stores: 39
// - Calls to sqrt: 25

// There's a trade-off here. Full inlining used minimal temporary
// memory and memory bandwidth, but did a whole bunch of redundant
// expensive math (calling sqrt). It evaluated most points in
// 'producer' four times. The second schedule,
// producer.compute_root(), did the mimimum number of calls to
// sqrt, but used more temporary memory and more memory bandwidth.

// In any given situation the correct choice can be difficult to
// make. If you're memory-bandwidth limited, or don't have much
// memory (e.g. because you're running on an old cell-phone), then
// it can make sense to do redundant math. On the other hand, sqrt
// is expensive, so if you're compute-limited then fewer calls to
// sqrt will make your program faster. Adding vectorization or
// multi-core parallelism tilts the scales in favor of doing
// redundant work, because firing up multiple cpu cores increases
// the amount of math you can do per second, but doesn't increase
// your system memory bandwidth or capacity.

// We can make choices in between full inlining and
// compute_root. Next we'll alternate between computing the
// producer and consumer on a per-scanline basis:
{
    // Start with the same function definitions:
    Func producer("producer_y"), consumer("consumer_y");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
                      producer(x, y+1) +
                      producer(x+1, y) +
                      producer(x+1, y+1));

    // Tell Halide to evaluate producer as needed per y coordinate
    // of the consumer:
    producer.compute_at(consumer, y);

    // This places the code that computes the producer just
    // *inside* the consumer's for loop over y, as in the
    // equivalent C below.

    // Turn on tracing.
    producer.trace_stores();
    consumer.trace_stores();

    // Compile and run.
    printf("\nEvaluating producer.compute_at(consumer, y)\n");
    consumer.realize(4, 4);

    // Reading the log you should see that producer and consumer
    // alternate on a per-scanline basis. Let's look at the
    // equivalent C:

    float result[4][4];

    // There's an outer loop over scanlines of consumer:
    for (int y = 0; y < 4; y++) {
        // Allocate space and compute enough of the producer to
        // satisfy this single scanline of the consumer. This
        // means a 5x2 box of the producer.
        float producer_storage[2][5];
        for (int py = y; py < y + 2; py++) {
            for (int px = 0; px < 5; px++) {
                producer_storage[py-y][px] = sqrt(px * py);
            }
        }

        // Compute a scanline of the consumer.
        for (int x = 0; x < 4; x++) {
            result[y][x] = (producer_storage[0][x] +
                            producer_storage[1][x] +
                            producer_storage[0][x+1] +
                            producer_storage[1][x+1]);
        }
    }

    // The performance characteristics of this strategy are in
    // between inlining and compute root. We still allocate some
    // temporary memory, but less that compute_root, and with
    // better locality (we load from it soon after writing to it,
    // so for larger images, values should still be in cache). We
    // still do some redundant work, but less than full inlining:

    // producer.compute_at(consumer, y):
    // - Temporary memory allocated: 10 floats
    // - Loads: 64
    // - Stores: 56
    // - Calls to sqrt: 40
}

// We could also say producer.compute_at(consumer, x), but this
// would be very similar to full inlining (the default
// schedule). Instead let’s distinguish between the loop level at
// which we allocate storage for producer, and the loop level at
// which we actually compute it. This unlocks a few optimizations.
{
    Func producer("producer_store_root_compute_y"), consumer("consumer_store_root_compute_y");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
                      producer(x, y+1) +
                      producer(x+1, y) +
                      producer(x+1, y+1));


    // Tell Halide to make a buffer to store all of producer at
    // the outermost level:
    producer.store_root();
    // ... but compute it as needed per y coordinate of the
    // consumer.
    producer.compute_at(consumer, y);

    producer.trace_stores();
    consumer.trace_stores();

    printf("\nEvaluating producer.store_root().compute_at(consumer, y)\n");
    consumer.realize(4, 4);

    // Reading the log you should see that producer and consumer
    // again alternate on a per-scanline basis. It computes a 5x2
    // box of the producer to satisfy the first scanline of the
    // consumer, but after that it only computes a 5x1 box of the
    // output for each new scanline of the consumer!
    //
    // Halide has detected that for all scanlines except for the
    // first, it can reuse the values already sitting in the
    // buffer we've allocated for producer. Let's look at the
    // equivalent C:

    float result[4][4];

    // producer.store_root() implies that storage goes here:
    float producer_storage[5][5];

    // There's an outer loop over scanlines of consumer:
    for (int y = 0; y < 4; y++) {

        // Compute enough of the producer to satisfy this scanline
        // of the consumer.
        for (int py = y; py < y + 2; py++) {

            // Skip over rows of producer that we've already
            // computed in a previous iteration.
            if (y > 0 && py == y) continue;

            for (int px = 0; px < 5; px++) {
                producer_storage[py][px] = sqrt(px * py);
            }
        }

        // Compute a scanline of the consumer.
        for (int x = 0; x < 4; x++) {
            result[y][x] = (producer_storage[y][x] +
                            producer_storage[y+1][x] +
                            producer_storage[y][x+1] +
                            producer_storage[y+1][x+1]);
        }
    }

    // The performance characteristics of this strategy are pretty
    // good! The numbers are similar compute_root, except locality
    // is better. We're doing the minimum number of sqrt calls,
    // and we load values soon after they are stored, so we're
    // probably making good use of the cache:

    // producer.store_root().compute_at(consumer, y):
    // - Temporary memory allocated: 10 floats
    // - Loads: 64
    // - Stores: 39
    // - Calls to sqrt: 25

    // Note that my claimed amount of memory allocated doesn't
    // match the reference C code. Halide is performing one more
    // optimization under the hood. It folds the storage for the
    // producer down into a circular buffer of two
    // scanlines. Equivalent C would actually look like this:

    {
        // Actually store 2 scanlines instead of 5
        float producer_storage[2][5];
        for (int y = 0; y < 4; y++) {
            for (int py = y; py < y + 2; py++) {
                if (y > 0 && py == y) continue;
                for (int px = 0; px < 5; px++) {
                    // Stores to producer_storage have their y coordinate bit-masked.
                    producer_storage[py & 1][px] = sqrt(px * py);
                }
            }

            // Compute a scanline of the consumer.
            for (int x = 0; x < 4; x++) {
                // Loads from producer_storage have their y coordinate bit-masked.
                result[y][x] = (producer_storage[y & 1][x] +
                                producer_storage[(y+1) & 1][x] +
                                producer_storage[y & 1][x+1] +
                                producer_storage[(y+1) & 1][x+1]);
            }
        }
    }
}

// We can do even better, by leaving the storage outermost, but
// moving the computation into the innermost loop:
{
    Func producer("producer_store_root_compute_y"), consumer("consumer_store_root_compute_y");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
                      producer(x, y+1) +
                      producer(x+1, y) +
                      producer(x+1, y+1));


    // Store outermost, compute innermost.
    producer.store_root().compute_at(consumer, x);

    producer.trace_stores();
    consumer.trace_stores();

    printf("\nEvaluating producer.store_root().compute_at(consumer, x)\n");
    consumer.realize(4, 4);

    // Reading the log, you should see that producer and consumer
    // now alternate on a per-pixel basis. Here's the equivalent C:

    float result[4][4];

    // producer.store_root() implies that storage goes here, but
    // we can fold it down into a circular buffer of two
    // scanlines:
    float producer_storage[2][5];

    // For every pixel of the consumer:
    for (int y = 0; y < 4; y++) {
        for (int x = 0; x < 4; x++) {

            // Compute enough of the producer to satisfy this
            // pixel of the consumer, but skip values that we've
            // already computed:
            if (y == 0 && x == 0)
                producer_storage[y & 1][x] = sqrt(x*y);
            if (y == 0)
                producer_storage[y & 1][x+1] = sqrt((x+1)*y);
            if (x == 0)
                producer_storage[(y+1) & 1][x] = sqrt(x*(y+1));
            producer_storage[(y+1) & 1][x+1] = sqrt((x+1)*(y+1));

            result[y][x] = (producer_storage[y & 1][x] +
                            producer_storage[(y+1) & 1][x] +
                            producer_storage[y & 1][x+1] +
                            producer_storage[(y+1) & 1][x+1]);
        }
    }

    // The performance characteristics of this strategy are the
    // best so far. One of the four values of the producer we need
    // is probably still sitting in a register, so I won't count
    // it as a load:
    // producer.store_root().compute_at(consumer, x):
    // - Temporary memory allocated: 10 floats
    // - Loads: 48
    // - Stores: 56
    // - Calls to sqrt: 40
```

```
}

// So what's the catch? Why not always do
// producer.store_root().compute_at(consumer, x) for this type of
// code?
//
// The answer is parallelism. In both of the previous two
// strategies we've assumed that values computed on previous
// iterations are lying around for us to reuse. This assumes that
// previous values of x or y happened earlier in time and have
// finished. This is not true if you parallelize or vectorize
// either loop. Darn. If you parallelize, Halide won't inject the
// optimizations that skip work already done if there's a parallel
// loop in between the store_at level and the compute_at level,
// and won't fold the storage down into a circular buffer either,
// which makes our store_root pointless.

// We're running out of options. We can make new ones by
// splitting. We can store_at or compute_at at the natural
// variables of the consumer (x and y), or we can split x or y
// into new inner and outer sub-variables and then schedule with
// respect to those. We'll use this to express fusion in tiles:
{
    Func producer("producer_store_root_compute_y"), consumer("consumer_store_root_compute_y");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
                      producer(x, y+1) +
                      producer(x+1, y) +
                      producer(x+1, y+1));


    // Tile the consumer using 2x2 tiles.
    Var x_outer, y_outer, x_inner, y_inner;
    consumer.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);

    // Compute the producer per tile of the consumer
    producer.compute_at(consumer, x_outer);

    // Notice that I wrote my schedule starting from the end of
    // the pipeline (the consumer). This is because the schedule
    // for the producer refers to x_outer, which we introduced
    // when we tiled the consumer. You can write it in the other
    // order, but it tends to be harder to read.

    // Turn on tracing.
    producer.trace_stores();
    consumer.trace_stores();

    printf("\nEvaluating:\n"
           "consumer.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);\n"
           "producer.compute_at(consumer, x_outer);\n");
    consumer.realize(4, 4);

    // Reading the log, you should see that producer and consumer
    // now alternate on a per-tile basis. Here's the equivalent C:

    float result[4][4];

    // For every tile of the consumer:
    for (int y_outer = 0; y_outer < 2; y_outer++) {
        for (int x_outer = 0; x_outer < 2; x_outer++) {
            // Compute the x and y coords of the start of this tile.
            int x_base = x_outer*2;
            int y_base = y_outer*2;

            // Compute enough of producer to satisfy this tile. A
            // 2x2 tile of the consumer requires a 3x3 tile of the
            // producer.
            float producer_storage[3][3];
            for (int py = y_base; py < y_base + 3; py++) {
                for (int px = x_base; px < x_base + 3; px++) {
                    producer_storage[py-y_base][px-x_base] = sqrt(px * py);
                }
            }

            // Compute this tile of the consumer
            for (int y_inner = 0; y_inner < 2; y_inner++) {
                for (int x_inner = 0; x_inner < 2; x_inner++) {
                    int x = x_base + x_inner;
                    int y = y_base + y_inner;
                    result[y][x] = (producer_storage[y - y_base][x - x_base] +
                                    producer_storage[y - y_base + 1][x - x_base] +
                                    producer_storage[y - y_base][x - x_base + 1] +
                                    producer_storage[y - y_base + 1][x - x_base + 1]);
                }
            }
        }
    }

    // Tiling can make sense for problems like this one with
    // stencils that reach outwards in x and y. Each tile can be
    // computed independently in parallel, and the redundant work
    // done by each tile isn't so bad once the tiles get large
    // enough.
}

// Let's try a mixed strategy that combines what we have done with
// splitting, parallelizing, and vectorizing. This is one that
// often works well in practice for large images. If you
// understand this schedule, then you understand 95% of scheduling
// in Halide.
{
    Func producer("producer_mixed"), consumer("consumer_mixed");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
                      producer(x, y+1) +
                      producer(x+1, y) +
                      producer(x+1, y+1));

    // Split the y coordinate of the consumer into strips of 16 scanlines:
    Var yo, yi;
    consumer.split(y, yo, yi, 16);
    // Compute the strips using a thread pool and a task queue.
    consumer.parallel(yo);
    // Vectorize across x by a factor of four.
    consumer.vectorize(x, 4);

    // Now store the producer per-strip. This will be 17 scanlines
    // of the producer (16+1), but hopefully it will fold down
    // into a circular buffer of two scanlines:
    producer.store_at(consumer, yo);
    // Within each strip, compute the producer per scanline of the
    // consumer, skipping work done on previous scanlines.
    producer.compute_at(consumer, yi);
    // Also vectorize the producer (because sqrt is vectorizable on x86 using SSE).
    producer.vectorize(x, 4);

    // Let's leave tracing off this time, because we're going to
    // evaluate over a larger image.
    // consumer.trace_stores();
    // producer.trace_stores();

    Image<float> halide_result = consumer.realize(800, 600);

    // Here's the equivalent (serial) C:
```

```
float c_result[600][800];

// For every strip of 16 scanlines
for (int yo = 0; yo < 600/16 + 1; yo++) { // (this loop is parallel in the Halide version)

    // 16 doesn't divide 600, so push the last slice upwards to fit within [0, 599] (see lesson 05).
    int y_base = yo * 16;
    if (y_base > 600-16) y_base = 600-16;

    // Allocate a two-scanline circular buffer for the producer
    float producer_storage[2][801];

    // For every scanline in the strip of 16:
    for (int yi = 0; yi < 16; yi++) {
        int y = y_base + yi;

        for (int py = y; py < y+2; py++) {
            // Skip scanlines already computed *within this task*
            if (yi > 0 && py == y) continue;

            // Compute this scanline of the producer in 4-wide vectors
            for (int x_vec = 0; x_vec < 800/4 + 1; x_vec++) {
                int x_base = x_vec*4;
                // 4 doesn't divide 801, so push the last vector left (see lesson 05).
                if (x_base > 801 - 4) x_base = 801 - 4;
                // If you're on x86, Halide generates SSE code for this part:
                int x[] = {x_base, x_base + 1, x_base + 2, x_base + 3};
                float vec[4] = {sqrtf(x[0] * py), sqrtf(x[1] * py), sqrtf(x[2] * py), sqrtf(x[3] * py)};
                producer_storage[py & 1][x[0]] = vec[0];
                producer_storage[py & 1][x[1]] = vec[1];
                producer_storage[py & 1][x[2]] = vec[2];
                producer_storage[py & 1][x[3]] = vec[3];
            }
        }

        // Now compute consumer for this scanline:
        for (int x_vec = 0; x_vec < 800/4; x_vec++) {
            int x_base = x_vec * 4;
            // Again, Halide's equivalent here uses SSE.
            int x[] = {x_base, x_base + 1, x_base + 2, x_base + 3};
            float vec[] = {
                (producer_storage[y & 1][x[0]] +
                 producer_storage[(y+1) & 1][x[0]] +
                 producer_storage[y & 1][x[0]+1] +
                 producer_storage[(y+1) & 1][x[0]+1]),
                (producer_storage[y & 1][x[1]] +
                 producer_storage[(y+1) & 1][x[1]] +
                 producer_storage[y & 1][x[1]+1] +
                 producer_storage[(y+1) & 1][x[1]+1]),
                (producer_storage[y & 1][x[2]] +
                 producer_storage[(y+1) & 1][x[2]] +
                 producer_storage[y & 1][x[2]+1] +
                 producer_storage[(y+1) & 1][x[2]+1]),
                (producer_storage[y & 1][x[3]] +
                 producer_storage[(y+1) & 1][x[3]] +
                 producer_storage[y & 1][x[3]+1] +
                 producer_storage[(y+1) & 1][x[3]+1])};

            c_result[y][x[0]] = vec[0];
            c_result[y][x[1]] = vec[1];
            c_result[y][x[2]] = vec[2];
            c_result[y][x[3]] = vec[3];
        }

    }
}
// Look on my code, ye mighty, and despair!
```

```
        // Let's check the C result against the Halide result. Doing
        // this I found several bugs in my C implementation, which
        // should tell you something.
        for (int y = 0; y < 600; y++) {
            for (int x = 0; x < 800; x++) {
                float error = halide_result(x, y) - c_result[y][x];
                // It's floating-point math, so we'll allow some slop:
                if (error < -0.001f || error > 0.001f) {
                    printf("halide_result(%d, %d) = %f instead of %f\n",
                           x, y, halide_result(x, y), c_result[y][x]);
                    return -1;
                }
            }
        }

    }

    // This stuff is hard. We ended up in a three-way trade-off
    // between memory bandwidth, redundant work, and
    // parallelism. Halide can't make the correct choice for you
    // automatically (sorry). Instead it tries to make it easier for
    // you to explore various options, without messing up your
    // program. In fact, Halide promises that scheduling calls like
    // compute_root won't change the meaning of your algorithm -- you
    // should get the same bits back no matter how you schedule
    // things.

    // So be empirical! Experiment with various schedules and keep a
    // log of performance. Form hypotheses and then try to prove
    // yourself wrong. Don't assume that you just need to vectorize
    // your code by a factor of four and run it on eight cores and
    // you'll get 32x faster. This almost never works. Modern systems
    // are complex enough that you can't predict performance reliably
    // without running your code.

    // We suggest you start by scheduling all of your non-trivial
    // stages compute_root, and then work from the end of the pipeline
    // upwards, inlining, parallelizing, and vectorizing each stage in
    // turn until you reach the top.

    // Halide is not just about vectorizing and parallelizing your
    // code. That's not enough to get you very far. Halide is about
    // giving you tools that help you quickly explore different
    // trade-offs between locality, redundant work, and parallelism,
    // without messing up the actual result you're trying to compute.

    printf("Success!\n");
    return 0;
}
```

A.9 UPDATE DEFINITIONS

```
// Halide tutorial lesson 9

// This lesson demonstrates how to define a Func in multiple passes, including scattering.

// On linux, you can compile and run it like so:
// g++ lesson_09*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -lpthread -ldl -fopenmp -o lesson_09
// LD_LIBRARY_PATH=../bin ./lesson_09

// On os x (will only work if you actually have g++, not Apple's pretend g++ which is actually clang):
// g++ lesson_09*.cpp -g -I ../include -L ../bin -lHalide `libpng-config --cflags --ldflags` -fopenmp -o lesson_09
// DYLD_LIBRARY_PATH=../bin ./lesson_09

#include <Halide.h>
#include <stdio.h>

// We're going to be using x86 SSE intrinsics later on in this lesson.
#ifdef __SSE2__
#include <emmintrin.h>
#endif

// We'll also need a clock to do performance testing at the end.
#include "clock.h"

using namespace Halide;

// Support code for loading pngs.
#include "image_io.h"

int main(int argc, char **argv) {
    // Declare some Vars to use below.
    Var x("x"), y("y");

    // Load a grayscale image to use as an input.
    Image<uint8_t> input = load<uint8_t>("images/gray.png");

    // You can define a Func in multiple passes. Let's see a toy
    // example first.
    {
        // The first definition must be one like we have seen already
        // - a mapping from Vars to an Expr:
        Func f;
        f(x, y) = x + y;
        // We call this first definition the "pure" definition.

        // But the later definitions can include computed expressions on
        // both sides. The simplest example is modifying a single point:
        f(3, 7) = 42;

        // We call these extra definitions "update" definitions, or
        // "reduction" definitions. A reduction definition is an
        // update definition that recursively refers back to the
        // function's current value at the same site:
        f(x, y) = f(x, y) + 17;

        // If we confine our update to a single row, we can
        // recursively refer to values in the same column:
        f(x, 3) = f(x, 0) * f(x, 10);

        // Similarly, if we confine our update to a single column, we
        // can recursively refer to other values in the same row.
        f(0, y) = f(0, y) / f(3, y);

        // The general rule is: Each Var used in an update definition
        // must appear unadorned in the same position as in the pure
        // definition in all references to the function on the left-
        // and right-hand sides. So the following definitions are
        // legal updates:
        f(x, 17) = x + 8; // x is used, so all uses of f must have x as the first argument.
        f(0, y) = y * 8;  // y is used, so all uses of f must have y as the second argument.
        f(x, x + 1) = x + 8;
        f(y/2, y) = f(0, y) * 17;

        // But these ones would cause an error:
        // f(x, 0) = f(x + 1, 0) <- First argument to f on the right-hand-side must be 'x', not 'x + 1'.
        // f(y, y + 1) = y + 8   <- Second argument to f on the left-hand-side must be 'y', not 'y + 1'.
        // f(y, x) = y - x;      <- Arguments to f on the left-hand-side are in the wrong places.
        // f(3, 4) = x + y;      <- Free variables appear on the right-hand-side but not the left-hand-side.

        // We'll realize this one just to make sure it compiles. The
        // second-to-last definition forces us to realize over a
        // domain that is taller than it is wide.
        f.realize(100, 101);

        // For each realization of f, each step runs in its entirety
        // before the next one begins. Let's trace the loads and
        // stores for a simpler example:
        Func g("g");
        g(x, y) = x + y;   // Pure definition
        g(2, 1) = 42;      // First update definition
        g(x, 0) = g(x, 1); // Second update definition

        g.trace_loads();
        g.trace_stores();

        g.realize(4, 4);

        // Reading the log, we see that each pass is applied in turn. The equivalent C is:
        int result[4][4];
        // Pure definition
        for (int y = 0; y < 4; y++) {
            for (int x = 0; x < 4; x++) {
                result[y][x] = x + y;
            }
        }
        // First update definition
        result[1][2] = 42;
        // Second update definition
        for (int x = 0; x < 4; x++) {
            result[0][x] = result[1][x];
        }
    }

    // Putting update passes inside loops.
    {
        // Starting with this pure definition:
        Func f;
        f(x, y) = x + y;

        // Say we want an update that squares the first fifty rows. We
        // could do this by adding 50 update definitions:

        // f(x, 0) = f(x, 0) * f(x, 0);
        // f(x, 1) = f(x, 1) * f(x, 1);
        // f(x, 2) = f(x, 2) * f(x, 2);
        // ...
        // f(x, 49) = f(x, 49) * f(x, 49);

        // Or equivalently using a compile-time loop in our C++:
        // for (int i = 0; i < 50; i++) {
        //   f(x, i) = f(x, i) * f(x, i);
        // }

        // But it's more manageable and more flexible to put the loop
        // in the generated code. We do this by defining a "reduction
        // domain" and using it inside an update definition:
        RDom r(0, 50);
        f(x, r) = f(x, r) * f(x, r);
        Image<int> halide_result = f.realize(100, 100);

        // The equivalent C is:
        int c_result[100][100];
        for (int y = 0; y < 100; y++) {
            for (int x = 0; x < 100; x++) {
                c_result[y][x] = x + y;
            }
        }
        for (int x = 0; x < 100; x++) {
            for (int r = 0; r < 50; r++) {
                // The loop over the reduction domain occurs inside of
                // the loop over any pure variables used in the update
                // step:
                c_result[r][x] = c_result[r][x] * c_result[r][x];
            }
        }

        // Check the results match:
        for (int y = 0; y < 100; y++) {
            for (int x = 0; x < 100; x++) {
                if (halide_result(x, y) != c_result[y][x]) {
                    printf("halide_result(%d, %d) = %d instead of %d\n",
                           x, y, halide_result(x, y), c_result[y][x]);
                    return -1;
                }
            }
        }
    }

    // Now we'll examine a real-world use for an update definition:
    // computing a histogram.
    {

        // Some operations on images can't be cleanly expressed as a pure
        // function from the output coordinates to the value stored
        // there. The classic example is computing a histogram. The
        // natural way to do it is to iterate over the input image,
        // updating histogram buckets. Here's how you do that in Halide:
        Func histogram("histogram");

        // Histogram buckets start as zero.
        histogram(x) = 0;

        // Define a multi-dimensional reduction domain over the input image:
        RDom r(0, input.width(), 0, input.height());

        // For every point in the reduction domain, increment the
        // histogram bucket corresponding to the intensity of the
        // input image at that point.
        histogram(input(r.x, r.y)) += 1;

        Image<int> halide_result = histogram.realize(256);

        // The equivalent C is:
        int c_result[256];
        for (int x = 0; x < 256; x++) {
            c_result[x] = 0;
        }
        for (int r_y = 0; r_y < input.height(); r_y++) {
            for (int r_x = 0; r_x < input.width(); r_x++) {
                c_result[input(r_x, r_y)] += 1;
            }
        }
```

```
        // Check the answers agree:
        for (int x = 0; x < 256; x++) {
            if (c_result[x] != halide_result(x)) {
                printf("halide_result(%d) = %d instead of %d\n",
                       x, halide_result(x), c_result[x]);
                return -1;
            }
        }
    }

    // Scheduling update steps
    {
        // The pure variables in an update step and can be
        // parallelized, vectorized, split, etc as usual.

        // Vectorizing, splitting, or parallelize the variables that
        // are part of the reduction domain is trickier. We'll cover
        // that in a later lesson.

        // Consider the definition:
        Func f;
        f(x, y) = x*y;
        // Set the second row to equal the first row.
        f(x, 1) = f(x, 0);
        // Set the second column to equal the first column plus 2.
        f(1, y) = f(0, y) + 2;

        // The pure variables in each stage can be scheduled
        // independently. To control the pure definition, we schedule
        // as we have done in the past. The following code vectorizes
        // and parallelizes the pure definition only.
        f.vectorize(x, 4).parallel(y);

        // We use Func::update(int) to get a handle to an update step
        // for the purposes of scheduling. The following line
        // vectorizes the first update step across x. We can't do
        // anything with y for this update step, because it doesn't
        // use y.
        f.update(0).vectorize(x, 4);

        // Now we parallelize the second update step in chunks of size
        // 4.
        Var yo, yi;
        f.update(1).split(y, yo, yi, 4).parallel(yo);

        Image<int> halide_result = f.realize(16, 16);

        // Here's the equivalent (serial) C:
        int c_result[16][16];

        // Pure step. Vectorized in x and parallelized in y.
        for (int y = 0; y < 16; y++) { // Should be a parallel for loop
            for (int x_vec = 0; x_vec < 4; x_vec++) {
                int x[] = {x_vec*4, x_vec*4+1, x_vec*4+2, x_vec*4+3};
                c_result[y][x[0]] = x[0] * y;
                c_result[y][x[1]] = x[1] * y;
                c_result[y][x[2]] = x[2] * y;
                c_result[y][x[3]] = x[3] * y;
            }
        }

        // First update. Vectorized in x.
        for (int x_vec = 0; x_vec < 4; x_vec++) {
            int x[] = {x_vec*4, x_vec*4+1, x_vec*4+2, x_vec*4+3};
            c_result[1][x[0]] = c_result[0][x[0]];
            c_result[1][x[1]] = c_result[0][x[1]];
            c_result[1][x[2]] = c_result[0][x[2]];
            c_result[1][x[3]] = c_result[0][x[3]];
        }

        // Second update. Parallelized in chunks of size 4 in y.
        for (int yo = 0; yo < 4; yo++) { // Should be a parallel for loop
            for (int yi = 0; yi < 4; yi++) {
                int y = yo*4 + yi;
                c_result[y][1] = c_result[y][0] + 2;
            }
        }

        // Check the C and Halide results match:
        for (int y = 0; y < 16; y++) {
            for (int x = 0; x < 16; x++) {
                if (halide_result(x, y) != c_result[y][x]) {
                    printf("halide_result(%d, %d) = %d instead of %d\n",
                           x, y, halide_result(x, y), c_result[y][x]);
                    return -1;
                }
            }
        }
    }

    // That covers how to schedule the variables within a Func that
    // uses update steps, but what about producer-consumer
    // relationships that involve compute_at and store_at? Let's
    // examine a reduction as a producer, in a producer-consumer pair.
    {
        // Because an update does multiple passes over a stored array,
        // it's not meaningful to inline them. So the default schedule
        // for them does the closest thing possible. It computes them
        // in the innermost loop of their consumer. Consider this
        // trivial example:
        Func producer, consumer;
        producer(x) = x*17;
        producer(x) += 1;
        consumer(x) = 2 * producer(x);
        Image<int> halide_result = consumer.realize(10);

        // The equivalent C is:
        int c_result[10];
        for (int x = 0; x < 10; x++)  {
            int producer_storage[1];
            // Pure step for producer
            producer_storage[0] = x * 17;
            // Update step for producer
            producer_storage[0] = producer_storage[0] + 1;
            // Pure step for consumer
            c_result[x] = 2 * producer_storage[0];
        }

        // Check the results match
        for (int x = 0; x < 10; x++) {
            if (halide_result(x) != c_result[x]) {
                printf("halide_result(%d) = %d instead of %d\n",
                       x, halide_result(x), c_result[x]);
                return -1;
            }
        }

        // For all other compute_at/store_at options, the reduction
        // gets placed where you would expect, somewhere in the loop
        // nest of the consumer.
    }

    // Now let's consider a reduction as a consumer in a
    // producer-consumer pair. This is a little more involved.
    {
        {
```

```
    // Case 1: The consumer references the producer in the pure step only.
    Func producer, consumer;
    // The producer is pure.
    producer(x) = x*17;
    consumer(x) = 2 * producer(x);
    consumer(x) += 1;

    // The valid schedules for the producer in this case are
    // the default schedule - inlined, and also:
    //
    // 1) producer.compute_at(x), which places the computation of
    // the producer inside the loop over x in the pure step of the
    // consumer.
    //
    // 2) producer.compute_root(), which computes all of the
    // producer ahead of time.
    //
    // 3) producer.store_root().compute_at(x), which allocates
    // space for the consumer outside the loop over x, but fills
    // it in as needed inside the loop.
    //
    // Let's use option 1.

    producer.compute_at(consumer, x);

    Image<int> halide_result = consumer.realize(10);

    // The equivalent C is:
    int c_result[10];
    // Pure step for the consumer
    for (int x = 0; x < 10; x++)  {
        // Pure step for producer
        int producer_storage[1];
        producer_storage[0] = x * 17;
        c_result[x] = 2 * producer_storage[0];
    }
    // Update step for the consumer
    for (int x = 0; x < 10; x++) {
        c_result[x] += 1;
    }

    // All of the pure step is evaluated before any of the
    // update step, so there are two separate loops over x.

    // Check the results match
    for (int x = 0; x < 10; x++) {
        if (halide_result(x) != c_result[x]) {
            printf("halide_result(%d) = %d instead of %d\n",
                   x, halide_result(x), c_result[x]);
            return -1;
        }
    }
}

{
    // Case 2: The consumer references the producer in the update step only
    Func producer, consumer;
    producer(x) = x * 17;
    consumer(x) = x;
    consumer(x) += producer(x);

    // Again we compute the producer per x coordinate of the
    // consumer. This places producer code inside the update
    // step of the producer, because that's the only step that
    // uses the producer.
    producer.compute_at(consumer, x);

    // Note however, that we didn't say:
    //
    // producer.compute_at(consumer.update(0), x).
    //
    // Scheduling is done with respect to Vars of a Func, and
    // the Vars of a Func are shared across the pure and
    // update steps.

    Image<int> halide_result = consumer.realize(10);

    // The equivalent C is:
    int c_result[10];
    // Pure step for the consumer
    for (int x = 0; x < 10; x++)  {
        c_result[x] = x;
    }
    // Update step for the consumer
    for (int x = 0; x < 10; x++) {
        // Pure step for producer
        int producer_storage[1];
        producer_storage[0] = x * 17;
        c_result[x] += producer_storage[0];
    }


    // Check the results match
    for (int x = 0; x < 10; x++) {
        if (halide_result(x) != c_result[x]) {
            printf("halide_result(%d) = %d instead of %d\n",
                   x, halide_result(x), c_result[x]);
            return -1;
        }
    }
}

{
    // Case 3: The consumer references the producer in
    // multiple steps that share common variables
    Func producer, consumer;
    producer(x) = x * 17;
    consumer(x) = producer(x) * x;
    consumer(x) += producer(x);

    // Again we compute the producer per x coordinate of the
    // consumer. This places producer code inside both the
    // pure and the update step of the producer. So there ends
    // up being two separate realizations of the producer, and
    // redundant work occurs.
    producer.compute_at(consumer, x);

    Image<int> halide_result = consumer.realize(10);

    // The equivalent C is:
    int c_result[10];
    // Pure step for the consumer
    for (int x = 0; x < 10; x++)  {
        // Pure step for producer
        int producer_storage[1];
        producer_storage[0] = x * 17;
        c_result[x] = producer_storage[0] * x;
    }
    // Update step for the consumer
    for (int x = 0; x < 10; x++) {
        // Another copy of the pure step for producer
        int producer_storage[1];
        producer_storage[0] = x * 17;
        c_result[x] += producer_storage[0];
    }
    // Check the results match
    for (int x = 0; x < 10; x++) {
        if (halide_result(x) != c_result[x]) {
            printf("halide_result(%d) = %d instead of %d\n",
                   x, halide_result(x), c_result[x]);
            return -1;
        }
    }
}

{
    // Case 4: The consumer references the producer in
    // multiple steps that do not share common variables
    Func producer, consumer;
    producer(x, y) = x*y;
    consumer(x, y) = x + y;
    consumer(x, 0) = producer(x, x-1);
    consumer(0, y) = producer(y, y-1);

    // In this case neither producer.compute_at(consumer, x)
    // nor producer.compute_at(consumer, y) will work, because
    // either one fails to cover one of the uses of the
    // producer. So we'd have to inline producer, or use
    // producer.compute_root().

    // Let's say we really really want producer to be
    // compute_at the inner loops of both consumer update
    // steps. Halide doesn't allow multiple different
    // schedules for a single Func, but we can work around it
    // by making two wrappers around producer, and scheduling
    // those instead:

    // Attempt 2:
    Func producer_wrapper_1, producer_wrapper_2, consumer_2;
    producer_wrapper_1(x, y) = producer(x, y);
    producer_wrapper_2(x, y) = producer(x, y);

    consumer_2(x, y) = x + y;
    consumer_2(x, 0) += producer_wrapper_1(x, x-1);
    consumer_2(0, y) += producer_wrapper_2(y, y-1);

    // The wrapper functions give us two separate handles on
    // the producer, so we can schedule them differently.
    producer_wrapper_1.compute_at(consumer_2, x);
    producer_wrapper_2.compute_at(consumer_2, y);

    Image<int> halide_result = consumer_2.realize(10, 10);

    // The equivalent C is:
    int c_result[10][10];
    // Pure step for the consumer
    for (int y = 0; y < 10; y++) {
        for (int x = 0; x < 10; x++) {
            c_result[y][x] = x + y;
        }
    }
    // First update step for consumer
    for (int x = 0; x < 10; x++) {
        int producer_wrapper_1_storage[1];
        producer_wrapper_1_storage[0] = x * (x-1);
        c_result[0][x] += producer_wrapper_1_storage[0];
    }
    // Second update step for consumer
    for (int y = 0; y < 10; y++) {
        int producer_wrapper_2_storage[1];
        producer_wrapper_2_storage[0] = y * (y-1);
        c_result[y][0] += producer_wrapper_2_storage[0];
    }
    // Check the results match
    for (int y = 0; y < 10; y++) {
        for (int x = 0; x < 10; x++) {
            if (halide_result(x, y) != c_result[y][x]) {
                printf("halide_result(%d, %d) = %d instead of %d\n",
                       x, y, halide_result(x, y), c_result[y][x]);
                return -1;
            }
        }
    }
}

{
    // Case 5: Scheduling a producer under a reduction domain
    // variable of the consumer.

    // We are not just restricted to scheduling producers at
    // the loops over the pure variables of the consumer. If a
    // producer is only used within a loop over a reduction
    // domain (RDom) variable, we can also schedule the
    // producer there.

    Func producer, consumer;

    RDom r(0, 5);
    producer(x) = x * 17;
    consumer(x) = x + 10;
    consumer(x) += r + producer(x + r);

    producer.compute_at(consumer, r);

    Image<int> halide_result = consumer.realize(10);

    // The equivalent C is:
    int c_result[10];
    // Pure step for the consumer.
    for (int x = 0; x < 10; x++)  {
        c_result[x] = x + 10;
    }
    // Update step for the consumer.
    for (int x = 0; x < 10; x++) {
        for (int r = 0; r < 5; r++) { // The loop over the reduction domain is always the inner loop.
            // We've schedule the storage and computation of
            // the producer here. We just need a single value.
            int producer_storage[1];
            // Pure step of the producer.
            producer_storage[0] = (x + r) * 17;

            // Now use it in the update step of the consumer.
            c_result[x] += r + producer_storage[0];
        }
    }

    // Check the results match
    for (int x = 0; x < 10; x++) {
        if (halide_result(x) != c_result[x]) {
            printf("halide_result(%d) = %d instead of %d\n",
                   x, halide_result(x), c_result[x]);
            return -1;
        }
    }


}
}
```

```
// A real-world example of a reduction inside a producer-consumer chain.
{
    // The default schedule for a reduction is a good one for
    // convolution-like operations. For example, the following
    // computes a 5x5 box-blur of our grayscale test image with a
    // clamp-to-edge boundary condition:

    // First add the boundary condition.
    Func clamped;
    Expr x_clamped = clamp(x, 0, input.width()-1);
    Expr y_clamped = clamp(y, 0, input.height()-1);
    clamped(x, y) = input(x_clamped, y_clamped);

    // Define a 5x5 box that starts at (-2, -2)
    RDom r(-2, 5, -2, 5);

    // Compute the 5x5 sum around each pixel.
    Func local_sum;
    local_sum(x, y) = 0; // Compute the sum as a 32-bit integer
    local_sum(x, y) += clamped(x + r.x, y + r.y);

    // Divide the sum by 25 to make it an average
    Func blurry;
    blurry(x, y) = cast<uint8_t>(local_sum(x, y) / 25);

    Image<uint8_t> halide_result = blurry.realize(input.width(), input.height());

    // The default schedule will inline 'clamped' into the update
    // step of 'local_sum', because clamped only has a pure
    // definition, and so its default schedule is fully-inlined.
    // We will then compute local_sum per x coordinate of blurry,
    // because the default schedule for reductions is
    // compute-innermost. Here's the equivalent C:

    Image<uint8_t> c_result(input.width(), input.height());
    for (int y = 0; y < input.height(); y++) {
        for (int x = 0; x < input.width(); x++) {
            int local_sum[1];
            // Pure step of local_sum
            local_sum[0] = 0;
            // Update step of local_sum
            for (int r_y = -2; r_y <= 2; r_y++) {
                for (int r_x = -2; r_x <= 2; r_x++) {
                    // The clamping has been inlined into the update step.
                    int clamped_x = std::min(std::max(x + r_x, 0), input.width()-1);
                    int clamped_y = std::min(std::max(y + r_y, 0), input.height()-1);
                    local_sum[0] += input(clamped_x, clamped_y);
                }
            }
            // Pure step of blurry
            c_result(x, y) = (uint8_t)(local_sum[0] / 25);
        }
    }

    // Check the results match
    for (int y = 0; y < input.height(); y++) {
        for (int x = 0; x < input.width(); x++) {
            if (halide_result(x, y) != c_result(x, y)) {
                printf("halide_result(%d, %d) = %d instead of %d\n",
                       x, y, halide_result(x, y), c_result(x, y));
                return -1;
            }
        }
    }
}

// Reduction helpers.
{
    // There are several reduction helper functions provided in
    // Halide.h, which compute small reductions and schedule them
    // innermost into their consumer. The most useful one is
    // "sum".
    Func f1;
    RDom r(0, 100);
    f1(x) = sum(r + x) * 7;

    // Sum creates a small anonymous Func to do the reduction. It's equivalent to:
    Func f2;
    Func anon;
    anon(x) = 0;
    anon(x) += r + x;
    f2(x) = anon(x) * 7;

    // So even though f1 references a reduction domain, it is a
    // pure function. The reduction domain has been swallowed to
    // define the inner anonymous reduction.

    Image<int> halide_result_1 = f1.realize(10);
    Image<int> halide_result_2 = f2.realize(10);

    // The equivalent C is:
    int c_result[10];
    for (int x = 0; x < 10; x++) {
        int anon[1];
        anon[0] = 0;
        for (int r = 0; r < 100; r++) {
            anon[0] += r + x;
        }
        c_result[x] = anon[0] * 7;
    }

    // Check they all match.
    for (int x = 0; x < 10; x++) {
        if (halide_result_1(x) != c_result[x]) {
            printf("halide_result_1(%d) = %d instead of %d\n",
                   x, halide_result_1(x), c_result[x]);
            return -1;
        }
        if (halide_result_2(x) != c_result[x]) {
            printf("halide_result_2(%d) = %d instead of %d\n",
                   x, halide_result_2(x), c_result[x]);
            return -1;
        }
    }
}


// A complex example that uses reduction helpers.
{
    // Other reduction helpers include "product", "minimum",
    // "maximum", "argmin", and "argmax". Using argmin and argmax
    // requires understanding tuples, which come in a later
    // lesson. Let's use minimum and maximum to compute the local
    // spread of our grayscale image.

    // First, add a boundary condition to the input.
    Func clamped;
    Expr x_clamped = clamp(x, 0, input.width()-1);
    Expr y_clamped = clamp(y, 0, input.height()-1);
    clamped(x, y) = input(x_clamped, y_clamped);

    RDom box(-2, 5, -2, 5);
    // Compute the local maximum minus the local minimum:
    Func spread;
    spread(x, y) = (maximum(clamped(x + box.x, y + box.y)) -
                    minimum(clamped(x + box.x, y + box.y)));
```

```
// Compute the result in strips of 32 scanlines
Var yo, yi;
spread.split(y, yo, yi, 32).parallel(yo);

// Vectorize across x within the strips. This implicitly
// vectorizes stuff that is computed within the loop over x in
// spread, which includes our minimum and maximum helpers, so
// they get vectorized too.
spread.vectorize(x, 16);

// We'll apply the boundary condition by padding each scanline
// as we need it in a circular buffer (see lesson 08).
clamped.store_at(spread, yo).compute_at(spread, yi);

Image<uint8_t> halide_result = spread.realize(input.width(), input.height());

// The C equivalent is almost too horrible to contemplate (and
// took me a long time to debug). This time I want to time
// both the Halide version and the C version, so I'll use sse
// intrinsics for the vectorization, and openmp to do the
// parallel for loop (you'll need to compile with -fopenmp or
// similar to get correct timing).
#ifdef __SSE2__

// Don't include the time required to allocate the output buffer.
Image<uint8_t> c_result(input.width(), input.height());

double t1 = current_time();

// Run this one hundred times so we can average the timing results.
for (int iters = 0; iters < 100; iters++) {

    #pragma omp parallel for
    for (int yo = 0; yo < (input.height() + 31)/32; yo++) {
        int y_base = std::min(yo * 32, input.height() - 32);

        // Compute clamped in a circular buffer of size 8
        // (smallest power of two greater than 5). Each thread
        // needs its own allocation, so it must occur here.

        int clamped_width = input.width() + 4;
        uint8_t *clamped_storage = (uint8_t *)malloc(clamped_width * 8);

        for (int yi = 0; yi < 32; yi++) {
            int y = y_base + yi;

            uint8_t *output_row = &c_result(0, y);

            // Compute clamped for this scanline, skipping rows
            // already computed within this slice.
            int min_y_clamped = (yi == 0) ? (y - 2) : (y + 2);
            int max_y_clamped = (y + 2);
            for (int cy = min_y_clamped; cy <= max_y_clamped; cy++) {
                // Figure out which row of the circular buffer
                // we're filling in using bitmasking:
                uint8_t *clamped_row = clamped_storage + (cy & 7) * clamped_width;

                // Figure out which row of the input we're reading
                // from by clamping the y coordinate:
                int clamped_y = std::min(std::max(cy, 0), input.height()-1);
                uint8_t *input_row = &input(0, clamped_y);

                // Fill it in with the padding.
                for (int x = -2; x < input.width() + 2; x++) {
                    int clamped_x = std::min(std::max(x, 0), input.width()-1);
                    *clamped_row++ = input_row[clamped_x];
                }
            }

            // Now iterate over vectors of x for the pure step of the output.
            for (int x_vec = 0; x_vec < (input.width() + 15)/16; x_vec++) {
                int x_base = std::min(x_vec * 16, input.width() - 16);

                // Allocate storage for the minimum and maximum
                // helpers. One vector is enough.
                __m128i minimum_storage, maximum_storage;

                // The pure step for the maximum is a vector of zeros
                maximum_storage = (__m128i)_mm_setzero_ps();

                // The update step for maximum
                for (int max_y = y - 2; max_y <= y + 2; max_y++) {
                    uint8_t *clamped_row = clamped_storage + (max_y & 7) * clamped_width;
                    for (int max_x = x_base - 2; max_x <= x_base + 2; max_x++) {
                        __m128i v = _mm_loadu_si128((__m128i const *)(clamped_row + max_x + 2));
                        maximum_storage = _mm_max_epu8(maximum_storage, v);
                    }
                }

                // The pure step for the minimum is a vector of
                // ones. Create it by comparing something to
                // itself.
                minimum_storage = (__m128i)_mm_cmpeq_ps(_mm_setzero_ps(),
                                                        _mm_setzero_ps());

                // The update step for minimum.
                for (int min_y = y - 2; min_y <= y + 2; min_y++) {
                    uint8_t *clamped_row = clamped_storage + (min_y & 7) * clamped_width;
                    for (int min_x = x_base - 2; min_x <= x_base + 2; min_x++) {
                        __m128i v = _mm_loadu_si128((__m128i const *)(clamped_row + min_x + 2));
                        minimum_storage = _mm_min_epu8(minimum_storage, v);
                    }
                }

                // Now compute the spread.
                __m128i spread = _mm_sub_epi8(maximum_storage, minimum_storage);

                // Store it.
                _mm_storeu_si128((__m128i *)(output_row + x_base), spread);

            }
        }

        free(clamped_storage);
    }
}

double t2 = current_time();

// Skip the timing comparison if we don't have openmp
// enabled. Otherwise it's unfair to C.
#ifdef _OPENMP

// Now run the Halide version again without the
// jit-compilation overhead. Also run it one hundred times.
for (int iters = 0; iters < 100; iters++) {
    spread.realize(halide_result);
}

double t3 = current_time();

// Report the timings. On my machine they both take about 3ms
// for the 4-megapixel input (fast!), which makes sense,
// because they're using the same vectorization and
// parallelization strategy. However I find the Halide easier
```

```
        // to read, write, debug, modify, and port.
        printf("Halide spread took %f ms. C equivalent took %f ms\n",
               (t3 - t2)/100, (t2 - t1)/100);

        #endif // _OPENMP

        // Check the results match:
        for (int y = 0; y < input.height(); y++) {
            for (int x = 0; x < input.width(); x++) {
                if (halide_result(x, y) != c_result(x, y)) {
                    printf("halide_result(%d, %d) = %d instead of %d\n",
                           x, y, halide_result(x, y), c_result(x, y));
                    return -1;
                }
            }
        }

        #endif // __SSE2__

    }

    printf("Success!\n");
    return 0;
}
```

A.10 AHEAD-OF-TIME COMPILATION

```
// Halide tutorial lesson 10.

// This lesson demonstrates how to use Halide as an more traditional
// ahead-of-time (AOT) compiler.

// This lesson is split across two files. The first (this one), builds
// a Halide pipeline and compiles it to an object file and header. The
// second (lesson_10_aot_compilation_run.cpp), uses that object file
// to actually run the pipeline. This means that compiling this code
// is a multi-step process.

// On linux, you can compile and run it like so:
// g++ lesson_10*generate.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_10_generate
// LD_LIBRARY_PATH=../bin ./lesson_10_generate
// g++ lesson_10*run.cpp lesson_10_halide.o -lpthread -o lesson_10_run
// ./lesson_10_run

// On os x:
// g++ lesson_10*generate.cpp -g -I ../include -L ../bin -lHalide -o lesson_10_generate
// DYLD_LIBRARY_PATH=../bin ./lesson_10_generate
// g++ lesson_10*run.cpp lesson_10_halide.o -o lesson_10_run
// ./lesson_10_run

// The benefits of this approach are that the final program:
// - Doesn't do any jit compilation at runtime, so it's fast.
// - Doesn't depend on libHalide at all, so it's a small, easy-to-deploy binary.

#include <Halide.h>
#include <stdio.h>
using namespace Halide;

int main(int argc, char **argv) {

    // We'll define a simple one-stage pipeline:
    Func brighter;
    Var x, y;

    // The pipeline will depend on one scalar parameter.
    Param<uint8_t> offset;

    // And take one grayscale 8-bit input buffer. The first
    // constructor argument gives the type of a pixel, and the second
    // specifies the number of dimensions (not the number of
    // channels!). For a grayscale image this is two; for a color
    // image it's three. Currently, four dimensions is the maximum for
    // inputs and outputs.
    ImageParam input(type_of<uint8_t>(), 2);

    // If we were jit-compiling, these would just be an int and an
    // Image, but because we want to compile the pipeline once and
    // have it work for any value of the parameter, we need to make a
    // Param object, which can be used like an Expr, and an ImageParam
    // object, which can be used like an Image.

    // Define the Func.
    brighter(x, y) = input(x, y) + offset;

    // Schedule it.
    brighter.vectorize(x, 16).parallel(y);

    // This time, instead of calling brighter.realize(...), which
    // would compile and run the pipeline immediately, we'll call a
    // method that compiles the pipeline to an object file and header.
    //
    // For AOT-compiled code, we need to explicitly declare the
    // arguments to the routine. This routine takes two. Arguments are
    // usually Params or ImageParams.
    std::vector<Argument> args(2);
    args[0] = input;
    args[1] = offset;
    brighter.compile_to_file("lesson_10_halide", args);

    // If you're using C++11, you can just say:
    // brighter.compile_to_file("lesson_10_halide", {input, offset});

    printf("Halide pipeline compiled, but not yet run.\n");

    // To continue this lesson, look in the file lesson_10_aot_compilation_run.cpp

    return 0;
}
// Before reading this file, see lesson_10_aot_compilation_generate.cpp

// This is the code that actually uses the Halide pipeline we've
// compiled. It does not depend on libHalide, so we won't be including
// Halide.h.
//
// Instead, it depends on the header file that lesson_10_generate
// produced when we ran it:
#include "lesson_10_halide.h"

#include <stdio.h>

int main(int argc, char **argv) {
    // Have a look in the header file above (it won't exist until you've run
    // lesson_10_generate).

    // It starts with a definition of a buffer_t:
    //
    // typedef struct buffer_t {
    //     uint64_t dev;
    //     uint8_t* host;
    //     int32_t extent[4];
    //     int32_t stride[4];
    //     int32_t min[4];
    //     int32_t elem_size;
    //     bool host_dirty;
    //     bool dev_dirty;
    // } buffer_t;
    //
    // This is how Halide represents input and output images in
    // pre-compiled pipelines. There's a 'host' pointer that points to the
    // start of the image data, some fields that describe how to access
    // pixels, and some fields related to using the GPU that we'll ignore
    // for now (dev, host_dirty, dev_dirty).

    // Let's make some input data to test with:
    uint8_t input[640 * 480];
    for (int y = 0; y < 480; y++) {
        for (int x = 0; x < 640; x++) {
            input[y * 640 + x] = x ^ (y + 1);
        }
    }

    // And the memory where we want to write our output:
    uint8_t output[640 * 480];

    // In AOT-compiled mode, Halide doesn't manage this memory for
    // you. You should use whatever image data type makes sense for
    // your application. Halide just needs pointers to it.

    // Now we make a buffer_t to represent our input and output. It's
    // important to zero-initialize them so you don't end up with
    // garbage fields that confuse Halide.
```

```
buffer_t input_buf = {0}, output_buf = {0};

// The host pointers point to the start of the image data:
input_buf.host  = &input[0];
output_buf.host = &output[0];

// To access pixel (x, y) in a two-dimensional buffer_t, Halide
// looks at memory address:

// host + elem_size * ((x - min[0])*stride[0] + (y - min[1])*stride[1])

// The stride in a dimension represents the number of elements in
// memory between adjacent entries in that dimension. We have a
// grayscale image stored in scanline order, so stride[0] is 1,
// because pixels that are adjacent in x are next to each other in
// memory.
input_buf.stride[0] = output_buf.stride[0] = 1;

// stride[1] is the width of the image, because pixels that are
// adjacent in y are separated by a scanline's worth of pixels in
// memory.
input_buf.stride[1] = output_buf.stride[1] = 640;

// The extent tells us how large the image is in each dimension.
input_buf.extent[0] = output_buf.extent[0] = 640;
input_buf.extent[1] = output_buf.extent[1] = 480;

// We'll leave the mins as zero. This is what they typically
// are. The host pointer points to the memory location of the min
// coordinate (not the origin!).  See lesson 6 for more detail
// about the mins.

// The elem_size field tells us how many bytes each element
// uses. For the 8-bit image we use in this test it's one.
input_buf.elem_size = output_buf.elem_size = 1;

// To avoid repeating all the boilerplate above, We recommend you
// make a helper function that populates a buffer_t given whatever
// image type you're using.

// Now that we've setup our input and output buffers, we can call
// our function. Looking in the header file, it's signature is:

// int lesson_10_halide(buffer_t *_input, const int32_t _offset, buffer_t *_brighter);

// The return value is an error code. It's zero on success.

int offset = 5;
int error = lesson_10_halide(&input_buf, offset, &output_buf);

if (error) {
    printf("Halide returned an error: %d\n", error);
    return -1;
}

// Now let's check the filter performed as advertised. It was
// supposed to add the offset to every input pixel.
for (int y = 0; y < 480; y++) {
    for (int x = 0; x < 640; x++) {
        uint8_t input_val = input[y * 640 + 480];
        uint8_t output_val = output[y * 640 + 480];
        uint8_t correct_val = input_val + offset;
        if (output_val != correct_val) {
            printf("output(%d, %d) was %d instead of %d\n",
                   x, y, output_val, correct_val);
            return -1;
        }
    }
    }

    // Everything worked!
    printf("Success!\n");
    return 0;
}
```

A.11 CROSS COMPILATION

```
// Halide tutorial lesson 11.

// This lesson demonstrates how to use Halide as a cross-compiler.

// On linux, you can compile and run it like so:
// g++ lesson_11*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_11
// LD_LIBRARY_PATH=../bin ./lesson_11

// On os x:
// g++ lesson_11*.cpp -g -I ../include -L ../bin -lHalide -o lesson_11
// DYLD_LIBRARY_PATH=../bin ./lesson_11

#include <Halide.h>
#include <stdio.h>
using namespace Halide;

int main(int argc, char **argv) {

    // We'll define the simple one-stage pipeline that we used in lesson 10.
    Func brighter;
    Var x, y;

    // Declare the arguments.
    Param<uint8_t> offset;
    ImageParam input(type_of<uint8_t>(), 2);
    std::vector<Argument> args(2);
    args[0] = input;
    args[1] = offset;

    // Define the Func.
    brighter(x, y) = input(x, y) + offset;

    // Schedule it.
    brighter.vectorize(x, 16).parallel(y);

    // The following line is what we did in lesson 10. It compiles an
    // object file suitable for the system that you're running this
    // program on.  For example, if you compile and run this file on
    // 64-bit linux on an x86 cpu with sse4.1, then the generated code
    // will be suitable for 64-bit linux on x86 with sse4.1.
    brighter.compile_to_file("lesson_11_host", args);

    // We can also compile object files suitable for other cpus and
    // operating systems. You do this with an optional third argument
    // to compile_to_file which specifies the target to compile for.

    // Let's use this to compile a 32-bit arm android version of this code:
    Target target;
    target.os = Target::Android; // The operating system
    target.arch = Target::ARM;   // The CPU architecture
    target.bits = 32;            // The bit-width of the architecture
    std::vector<Target::Feature> arm_features; // A list of features to set
    target.set_features(arm_features);
    brighter.compile_to_file("lesson_11_arm_32_android", args, target); // Pass the target as the last argument.

    // And now a Windows object file for 64-bit x86 with AVX and SSE 4.1:
    target.os = Target::Windows;
    target.arch = Target::X86;
    target.bits = 64;
    std::vector<Target::Feature> x86_features;
    x86_features.push_back(Target::AVX);
    x86_features.push_back(Target::SSE41);
    target.set_features(x86_features);
    brighter.compile_to_file("lesson_11_x86_64_windows", args, target);

    // And finally an iOS mach-o object file for one of Apple's 32-bit
    // ARM processors - the A6. It’s used in the iPhone 5. The A6 uses
    // a slightly modified ARM architecture called ARMv7s. We specify
    // this using the target features field.  Support for Apple’s
    // 64-bit ARM processors is very new in llvm, and still somewhat
    // flaky.
    target.os = Target::IOS;
    target.arch = Target::ARM;
    target.bits = 32;
    std::vector<Target::Feature> armv7s_features;
    armv7s_features.push_back(Target::ARMv7s);
    target.set_features(armv7s_features);
    brighter.compile_to_file("lesson_11_arm_32_ios", args, target);


    // Now let’s check these files are what they claim, by examining
    // their first few bytes.

    // 32-arm android object files start with the magic bytes:
    uint8_t arm_32_android_magic[] = {0x7f, ’E’, ’L’, ’F’, // ELF format
                                      1,       // 32-bit
                                      1,       // 2’s complement little-endian
                                      1,       // Current version of elf
                                      3,       // Linux
                                      0, 0, 0, 0, 0, 0, 0, 0, // 8 unused bytes
                                      1, 0,    // Relocatable
                                      0x28, 0};  // ARM


    FILE *f = fopen("lesson_11_arm_32_android.o", "rb");
    uint8_t header[32];
    if (!f || fread(header, 32, 1, f) != 1) {
        printf("Object file not generated\n");
        return -1;
    }
    fclose(f);

    if (memcmp(header, arm_32_android_magic, sizeof(arm_32_android_magic))) {
        printf("Unexpected header bytes in 32-bit arm object file.\n");
        return -1;
    }

    // 64-bit windows object files start with the magic 16-bit value 0x8664
    // (presumably referring to x86-64)
    uint8_t win_64_magic[] = {0x64, 0x86};

    f = fopen("lesson_11_x86_64_windows.o", "rb");
    if (!f || fread(header, 32, 1, f) != 1) {
        printf("Object file not generated\n");
        return -1;
    }
    fclose(f);

    if (memcmp(header, win_64_magic, sizeof(win_64_magic))) {
        printf("Unexpected header bytes in 64-bit windows object file.\n");
        return -1;
    }

    // 32-bit arm iOS mach-o files start with the following magic bytes:
    uint32_t arm_32_ios_magic[] = {0xfeedface, // Mach-o magic bytes
                                   12,  // CPU type is ARM
                                   11,  // CPU subtype is ARMv7s
                                   1};  // It’s a relocatable object file.
    f = fopen("lesson_11_arm_32_ios.o", "rb");
    if (!f || fread(header, 32, 1, f) != 1) {
        printf("Object file not generated\n");
        return -1;
    }
    fclose(f);
```

```
    if (memcmp(header, arm_32_ios_magic, sizeof(arm_32_ios_magic))) {
        printf("Unexpected header bytes in 32-bit arm ios object file.\n");
        return -1;
    }

    // It looks like the object files we produced are plausible for
    // those targets. We'll count that as a success for the purposes
    // of this tutorial. For a real application you'd then need to
    // figure out how to integrate Halide into your cross-compilation
    // toolchain. There are several small examples of this in the
    // Halide repository under the apps folder. See HelloAndroid and
    // HelloiOS here:
    // https://github.com/halide/Halide/tree/master/apps/
    printf("Success!\n");
    return 0;
}
```

A.12 USING THE GPU

```
// Halide tutorial lesson 12.

// This lesson demonstrates how to use Halide to run code on a GPU.

// On linux, you can compile and run it like so:
// g++ lesson_12*.cpp -g -I ../include -L ../bin -lHalide -lpthread -ldl -o lesson_12
// LD_LIBRARY_PATH=../bin ./lesson_12

// On os x:
// g++ lesson_12*.cpp -g -I ../include -L ../bin -lHalide -lpng -o lesson_12
// DYLD_LIBRARY_PATH=../bin ./lesson_12

#include <Halide.h>
#include <stdio.h>
using namespace Halide;

// Include some support code for loading pngs.
#include "image_io.h"

// Include a clock to do performance testing.
#include "clock.h"

// Define some Vars to use.
Var x, y, c, i;

// We're going to want to schedule a pipeline in several ways, so we
// define the pipeline in a class so that we can recreate it several
// times with different schedules.
class Pipeline {
public:
    Func lut, padded, padded16, sharpen, curved;
    Image<uint8_t> input;

    Pipeline(Image<uint8_t> in) : input(in) {
        // For this lesson, we'll use a two-stage pipeline that sharpens
        // and then applies a look-up-table (LUT).

        // First we'll define the LUT. It will be a gamma curve.

        lut(i) = cast<uint8_t>(clamp(pow(i / 255.0f, 1.2f) * 255.0f, 0, 255));

        // Augment the input with a boundary condition.
        padded(x, y, c) = input(clamp(x, 0, input.width()-1),
                                clamp(y, 0, input.height()-1), c);

        // Cast it to 16-bit to do the math.
        padded16(x, y, c) = cast<uint16_t>(padded(x, y, c));

        // Next we sharpen it with a five-tap filter.
        sharpen(x, y, c) = (padded16(x, y, c) * 2-
                            (padded16(x - 1, y, c) +
                             padded16(x, y - 1, c) +
                             padded16(x + 1, y, c) +
                             padded16(x, y + 1, c)) / 4);

        // Then apply the LUT.
        curved(x, y, c) = lut(sharpen(x, y, c));
    }

    // Now we define methods that give our pipeline several different
    // schedules.
    void schedule_for_cpu() {
        // Compute the look-up-table ahead of time.
        lut.compute_root();

        // Compute color channels innermost. Promise that there will
        // be three of them and unroll across them.
        curved.reorder(c, x, y)
              .bound(c, 0, 3)
              .unroll(c);

        // Look-up-tables don't vectorize well, so just parallelize
        // curved in slices of 16 scanlines.
        Var yo, yi;
        curved.split(y, yo, yi, 16)
              .parallel(yo);

        // Compute sharpen as needed per scanline of curved, reusing
        // previous values computed within the same strip of 16
        // scanlines.
        sharpen.store_at(curved, yo)
               .compute_at(curved, yi);

        // Vectorize the sharpen. It's 16-bit so we'll vectorize it 8-wide.
        sharpen.vectorize(x, 8);

        // Compute the padded input at the same granularity as the
        // sharpen. We'll leave the cast to 16-bit inlined into
        // sharpen.
        padded.store_at(curved, yo)
              .compute_at(curved, yi);

        // Also vectorize the padding. It's 8-bit, so we'll vectorize
        // 16-wide.
        padded.vectorize(x, 16);

        // JIT-compile the pipeline for the CPU.
        curved.compile_jit();
    }

    // Now a schedule that uses CUDA or OpenCL.
    void schedule_for_gpu() {
        // We make the decision about whether to use the GPU for each
        // Func independently. If you have one Func computed on the
        // CPU, and the next computed on the GPU, Halide will do the
        // copy-to-gpu under the hood. For this pipeline, there's no
        // reason to use the CPU for any of the stages. Halide will
        // copy the input image to the GPU the first time we run the
        // pipeline, and leave it there to reuse on subsequent runs.

        // As before, we'll compute the LUT once at the start of the
        // pipeline.
        lut.compute_root();

        // Let's compute the look-up-table using the GPU in 16-wide
        // one-dimensional thread blocks. First we split the index
        // into blocks of size 16:
        Var block, thread;
        lut.split(i, block, thread, 16);
        // Then we tell cuda that our Vars 'block' and 'thread'
        // correspond to CUDA's notions of blocks and threads, or
        // OpenCL's notions of thread groups and threads.
        lut.gpu_blocks(block)
           .gpu_threads(thread);

        // This is a very common scheduling pattern on the GPU, so
        // there's a shorthand for it:

        // lut.gpu_tile(i, 16);

        // Func::gpu_tile method is similar to Func::tile, except that
        // it also specifies that the tile coordinates correspond to
        // GPU blocks, and the coordinates within each tile correspond
        // to GPU threads.
        // Compute color channels innermost. Promise that there will
        // be three of them and unroll across them.
        curved.reorder(c, x, y)
              .bound(c, 0, 3)
              .unroll(c);

        // Compute curved in 2D 8x8 tiles using the GPU.
        curved.gpu_tile(x, y, 8, 8);

        // This is equivalent to:
        // curved.tile(x, y, xo, yo, xi, yi, 8, 8)
        //       .gpu_blocks(xo, yo)
        //       .gpu_threads(xi, yi);

        // We'll leave sharpen as inlined into curved.

        // Compute the padded input as needed per GPU block, storing the
        // intermediate result in shared memory. Var::gpu_blocks, and
        // Var::gpu_threads exist to help you schedule producers within
        // GPU threads and blocks.
        padded.compute_at(curved, Var::gpu_blocks());

        // Use the GPU threads for the x and y coordinates of the
        // padded input.
        padded.gpu_threads(x, y);

        // JIT-compile the pipeline for the GPU. CUDA or OpenCL are
        // not enabled by default. We have to construct a Target
        // object, enable one of them, and then pass that target
        // object to compile_jit. Otherwise your CPU will very slowly
        // pretend it's a GPU, and use one thread per output pixel.

        // Start with a target suitable for the machine you're running
        // this on.
        Target target = get_host_target();

        // Then enable OpenCL or CUDA.

        // We'll enable OpenCL here, because it tends to give better
        // performance than CUDA, even with NVidia's drivers, because
        // NVidia's open source LLVM backend doesn't seem to do all
        // the same optimizations their proprietary compiler does.
        target.set_feature(Target::OpenCL);

        // Uncomment the next line and comment out the line above to
        // try CUDA instead.
        // target.set_feature(Target::CUDA);

        // If you want to see all of the OpenCL or CUDA API calls done
        // by the pipeline, you can also enable the Debug
        // flag. This is helpful for figuring out which stages are
        // slow, or when CPU -> GPU copies happen. It hurts
        // performance though, so we'll leave it commented out.
        // target.set_feature(Target::Debug);

        curved.compile_jit(target);
    }

    void test_performance() {
        // Test the performance of the scheduled Pipeline.

        // If we realize curved into a Halide::Image, that will
        // unfairly penalize GPU performance by including a GPU->CPU
        // copy in every run. Halide::Image objects always exist on
        // the CPU.

        // Halide::Buffer, however, represents a buffer that may
```

```
        // exist on either CPU or GPU or both.
        Buffer output(UInt(8), input.width(), input.height(), input.channels());

        // Run the filter once to initialize any GPU runtime state.
        curved.realize(output);

        // Now take the best of 3 runs for timing.
        double best_time;
        for (int i = 0; i < 3; i++) {

            double t1 = current_time();

            // Run the filter 100 times.
            for (int j = 0; j < 100; j++) {
                curved.realize(output);
            }

            // Force any GPU code to finish by copying the buffer back to the CPU.
            output.copy_to_host();

            double t2 = current_time();

            double elapsed = (t2 - t1)/100;
            if (i == 0 || elapsed < best_time) {
                best_time = elapsed;
            }
        }

        printf("%1.4f milliseconds\n", best_time);
    }

    void test_correctness(Image<uint8_t> reference_output) {
        Image<uint8_t> output = curved.realize(input.width(), input.height(), input.channels());

        // Check against the reference output.
        for (int c = 0; c < input.channels(); c++) {
            for (int y = 0; y < input.height(); y++) {
                for (int x = 0; x < input.width(); x++) {
                    if (output(x, y, c) != reference_output(x, y, c)) {
                        printf("Mismatch between output (%d) and "
                               "reference output (%d) at %d, %d, %d\n",
                               output(x, y, c),
                               reference_output(x, y, c),
                               x, y, c);
                    }
                }
            }
        }

    }
};

int main(int argc, char **argv) {
    // Load an input image.
    Image<uint8_t> input = load<uint8_t>("images/rgb.png");

    // Allocated an image that will store the correct output
    Image<uint8_t> reference_output(input.width(), input.height(), input.channels());

    printf("Testing performance on CPU:\n");
    Pipeline p1(input);
    p1.schedule_for_cpu();
    p1.test_performance();
    p1.curved.realize(reference_output);

    printf("Testing performance on GPU:\n");
    Pipeline p2(input);
    p2.schedule_for_gpu();
    p2.test_performance();
    p2.test_correctness(reference_output);

    return 0;
}
```