UC Berkeley
UC Berkeley Previously Published Works

Title
Learning to optimize halide with tree search and random programs

Permalink
https://escholarship.org/uc/item/5h71f534

Journal
ACM Transactions on Graphics, 38(4)

ISSN
0730-0301

Authors
Adams, Andrew
Ma, Karima
Anderson, Luke
et al.

Publication Date
2019-08-31

DOI
10.1145/3306346.3322967

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Learning to Optimize Halide with Tree Search and Random Programs

ANDREW ADAMS, Facebook AI Research
KARIMA MA, UC Berkeley
LUKE ANDERSON, MIT CSAIL
RIYADH BAGHDADI, MIT CSAIL
TZU-MAO LI, MIT CSAIL
MICHAËL GHARBI, Adobe
BENOIT STEINER, Facebook AI Research
STEVEN JOHNSON, Google
KAYVON FATAHALIAN, Stanford University
FRÉDO DURAND, MIT CSAIL
JONATHAN RAGAN-KELLEY, UC Berkeley

We present a new algorithm to automatically schedule Halide programs for high-performance image processing and deep learning. We significantly improve upon the performance of previous methods, which considered a limited subset of schedules. We define a parameterization of possible schedules much larger than prior methods and use a variant of beam search to search over it. The search optimizes runtime predicted by a cost model based on a combination of new derived features and machine learning. We train the cost model by generating and featurizing hundreds of thousands of random programs and schedules. We show that this approach operates effectively with or without autotuning. It produces schedules which are on average almost twice as fast as the existing Halide autoscheduler without autotuning, or more than twice as fast with, and is the first automatic scheduling algorithm to significantly outperform human experts on average.

CCS Concepts: • **Computing methodologies → Image processing**; • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: optimizing compilers, Halide

ACM Reference Format:
Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

Authors' addresses: Andrew Adams, Facebook AI Research, andrew.b.adams@gmail.com; Karima Ma, UC Berkeley, karima_ma@berkeley.edu; Luke Anderson, MIT CSAIL, lukea@mit.edu; Riyadh Baghdadi, MIT CSAIL, baghdadi@mit.edu; Tzu-Mao Li, MIT CSAIL, tzumao@mit.edu; Michaël Gharbi, Adobe, mgharbi@adobe.com; Benoit Steiner, Facebook AI Research, benoitsteiner@fb.com; Steven Johnson, Google, srj@google.com; Kayvon Fatahalian, Stanford University, kayvonf@cs.stanford.edu; Frédo Durand, MIT CSAIL, fredo@csail.mit.edu; Jonathan Ragan-Kelley, UC Berkeley, jrk@berkeley.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0730-0301/2019/7-ART121 $15.00
https://doi.org/10.1145/3306346.3322967

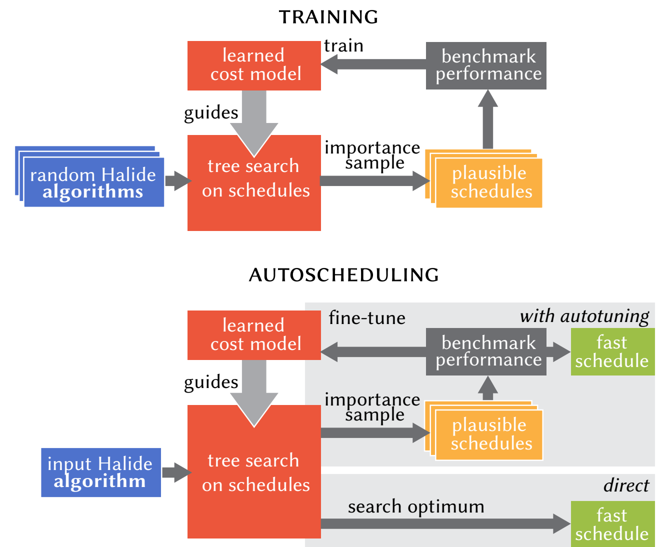

Fig. 1. We generate schedules for Halide programs using tree search over the space of schedules (Sec. 3) guided by a learned cost model and optional autotuning (Sec. 4). The cost model is trained by benchmarking thousands of randomly-generated Halide programs and schedules (Sec. 5). The resulting code significantly outperforms prior work and human experts (Sec. 6).

1 INTRODUCTION

Image processing and deep learning are pervasive. They are computationally intense, and implementations often have to be highly optimized by experts, at great cost, to be usable in practice. The Halide programming language has proven to be a powerful tool for this task because it separates the *algorithm* — what you want to compute — from the *schedule* — how you want to compute it, including choices about memory locality, redundant computation, and parallelism [Ragan-Kelley et al. 2012, 2013]. While Halide makes it easy to try different schedules, writing schedules that achieve high performance is hard: it requires expertise in hardware architecture and optimization, and even then, the space of possible schedules is enormous and their performance can be difficult to predict.

Automating the synthesis of high-performance schedules is sorely needed, but prior methods are limited in multiple ways [Mullapudi et al. 2016, 2015; Sioutas et al. 2018]. First, by design they only consider a small subset of all possible schedules. They generally work by modestly generalizing specific schedule templates or idioms, which are difficult to compose or extend to capture the countless other potentially fruitful choices for each application and target architecture. Second, they explore their key choices using specialized search procedures, which are tightly coupled to the family of schedules they consider. Third, they navigate this space using hand-designed cost models which struggle to accurately predict performance on real machines. These cost models can be tuned to guide the search to good performance on a few specific applications at a time, but they

struggle to generalize, often producing results many times slower than the best schedules in their search space on others.

In this paper, we present a new algorithm to automatically find high performance schedules for Halide programs that significantly outperforms previous automatic schedulers (Fig. 1). This is enabled by a number of key contributions, including:

- a new parameterization of the search space that can efficiently enumerate a much larger set of valid schedules than prior methods, increasing the available high-performance options;
- a more general search algorithm with backtracking and coarse-to-fine refinement;
- a new cost model that combines symbolic analysis with machine learning to predict performance;
- a robust methodology which trains the cost model on an infinite population of random programs;
- the optional use of sampling and benchmarking (autotuning) to further improve performance given extra time.

On a diverse set of applications spanning both imaging and learning, our algorithm outperforms the production Halide autoscheduler [Mullapudi et al. 2016] by 75% on average with no autotuning, and up to 135% on average with a few hours per application of autotuning. It is also more robust, never significantly underperforming the prior method, but at times outperforming it by an order of magnitude. With these results, it is the first automatic scheduling algorithm demonstrated to not just approach but significantly outperform human experts on average.

1.1 Our Approach

Parameterizing schedules. We define a general parameterization of the choice space which naturally subsumes most core Halide scheduling choices (Sec. 3). We enumerate choices from the output of a program backwards to the input, for each stage choosing both the granularity at which it will be computed with respect to its consumers, and how its own iteration space will be tiled across threads and SIMD lanes, and laid out in memory. This parameterization includes arbitrarily-nested loop tilings, with stages computed (fused) and stored (reused) at any granularity within them, encompassing important optimizations like line-buffering [Hegarty et al. 2014] not possible in most prior approaches. It mirrors the definition of the core scheduling operators themselves, makes both extending the space and reasoning about validity easy and local, and is suitable for generic tree search methods.

Search. We search this space using a general backtracking tree search algorithm based on beam search [Reddy 1977] (Sec. 3.2). We extend the classic algorithm to operate in a coarse-to-fine fashion, which we find provides more efficient backtracking over long-range decisions on this problem.

A hybrid learned cost model. We drive the search using a new cost model which combines symbolic analysis with learning to more accurately predict performance on real machines (Sec. 4). This is critical because our large search space increases the demands on the cost model: it includes not only many better schedules than prior methods, but also many *worse* choices. We featurize intermediate choices by treating the stages scheduled up to that point as a complete, smaller program, allowing the cost model to reason only about complete programs.

Training on random programs. We train the cost model to predict the benchmarked performance, on a given target architecture, of hundreds of thousands of random algorithms and schedules drawn from a distribution representative of key patterns in real programs (Sec. 5). This process is both automatic and generalizes more robustly than fitting to specific benchmarks, as most compilers do.

Trading off compile time for performance. We configurably scale our search based on the available compile time, from seconds to hours, to efficiently exploit both the fast cost model and ground-truth benchmarking (Sec. 4.3). Varying the beam size interpolates between fast greedy search and best-first. Given more time, randomizing the search by importance sampling the cost model, we can generate a near-infinite distribution of promising schedules. Compiling and benchmarking these to measure ground truth performance, in tens of minutes we can *autotune* to find better results than the model alone. Or, with hours to sample and benchmark, we can also iteratively fine-tune the cost model on the samples measured so far to accelerate autotuning.

Full code for the algorithm, random program generator, and test applications is available at https://autoscheduler2019.halide.io/.

2 RELATED WORK

There have been several prior attempts to automatically schedule Halide programs. However, these efforts either relied on searching a small space of schedule templates that exclude many higher-performance choices, or struggled to scale to nontrivial programs.

The original Halide autotuner [Ragan-Kelley et al. 2013] combined heuristic schedule templates with genetic search over random schedule rewrites, and relied entirely on ground-truth benchmarking to measure performance. It produced some results competitive with hand-written code, but finding schedules could take days for moderately-sized pipelines. Both its heuristics and its random rewrites frequently produced invalid schedules and were difficult to generalize. A simpler version of this approach was reimplemented in the OpenTuner framework [Ansel et al. 2014], but it can only effectively schedule the simplest pipelines [Mullapudi et al. 2015].

The current autoscheduler shipped as part of Halide is derived from Mullapudi et al.'s work [2016]. It performs greedy search over the possible applications of a simple schedule template using a simple cost model. This allows it to run quickly, with no benchmarking at all, but it only considers a single level of tiling and fusion, combined with fixed heuristic choices for things like parallelism, vectorization, and unrolling. This excludes many useful schedules (such as multi-level tiling and line-buffering), and has proven very difficult to generalize across applications or to new schedules. It can be autotuned by sampling the space of its three cost model parameters, but cannot sample its entire schedule space as our autotuner does.

The polyhedral compiler PolyMage [Mullapudi et al. 2015] uses an algorithm similar to that of the current Halide autoscheduler.

This has also been extended with backtracking and a richer cost model to some benefit [Jangda and Bondhugula 2018], but without changing the fundamentally restricted search space. Sioutas et al. [2018] improved local scheduling of individual Halide operations by defining a richer manual cost model to capture cache and prefetcher behaviour on CPUs. This was built on to schedule entire algorithms using a similar cost model coupled with a backtracking search over fusion strategies [Sioutas et al. 2019]. The space of schedules considered was broader than prior work in that it included sliding window strategies in addition to fusion in tiles. The space we search is broader still, and our cost model is more complex and learned, but we took substantial inspiration from their work in designing key terms in our model and featurization.

Many other polyhedral compilers use variants of the Pluto algorithm for automatic scheduling of affine loop nests [Baghdadi et al. 2015; Bondhugula et al. 2008; Grosser et al. 2012; Vasilache et al. 2018]. They formalize as an integer linear program (ILP) the problem of finding an affine loop transformation which maximizes outer loop parallelism and minimizes statement-to-statement reuse distance. The resulting ILP can be solved exactly, but the schedules considered exclude many key choices offered by Halide (introducing redundant computation to improve locality and parallelism), and the implicit cost model is only weakly correlated with actual performance. Changing either makes the problem no longer an ILP and intractable to solve.

TVM [Chen et al. 2018a] follows a similar philosophy to ours, combining generic search with learning and benchmarking to find good schedules [Chen et al. 2018b], but supports only semi-automatic scheduling: search is automated, but the *search space* must be manually defined by the programmer for each algorithm ("operator" in the parlance of deep learning frameworks) as a template. They also focus on locally optimizing individual operators in isolation, where we emphasize exploring long-range fusion through large programs.

There have also been many other attempts to use machine learning to improve prediction in compilers [Ashouri et al. 2018]. This includes predicting the best order for optimization passes [Fursin et al. 2008], the best tile sizes [Rahman et al. 2010], whether kernels should be mapped to CPUs or GPUs [Cummins et al. 2017], and the throughput of straight-line x86 code [Mendis et al. 2018].

3 NAVIGATING THE HALIDE SCHEDULING SPACE

To better understand the choice space we define, we begin with a brief introduction to core ideas in Halide.

A Halide program specifies an algorithm (which we often call a *pipeline*) as a directed acyclic graph (DAG) of stages producing multidimensional arrays of values. Each stage is defined as a function from any coordinate in an infinite grid to the value at that coordinate. Each stage can *consume* data from arbitrary points in prior stages, but — unlike in traditional languages like C++ — the order of evaluation of elements within each stage and across stages, their placement into arrays in memory, and even how many (and which) should be computed, are unspecified.

Consider a simple program:

```
h(x, y) = ...;
g(x, y) = pow(h(x, y), 1.8);
f(x, y) = g(x, y-1) + g(x, y+1);
```

It has three stages, with each point in `f` depending on a small window in `g`, which in turn depends on `h`. To evaluate this pipeline, we need to ask for a desired region of the output (say, $[0, w] \times [0, h]$). Given this, the Halide compiler can infer what regions are required of

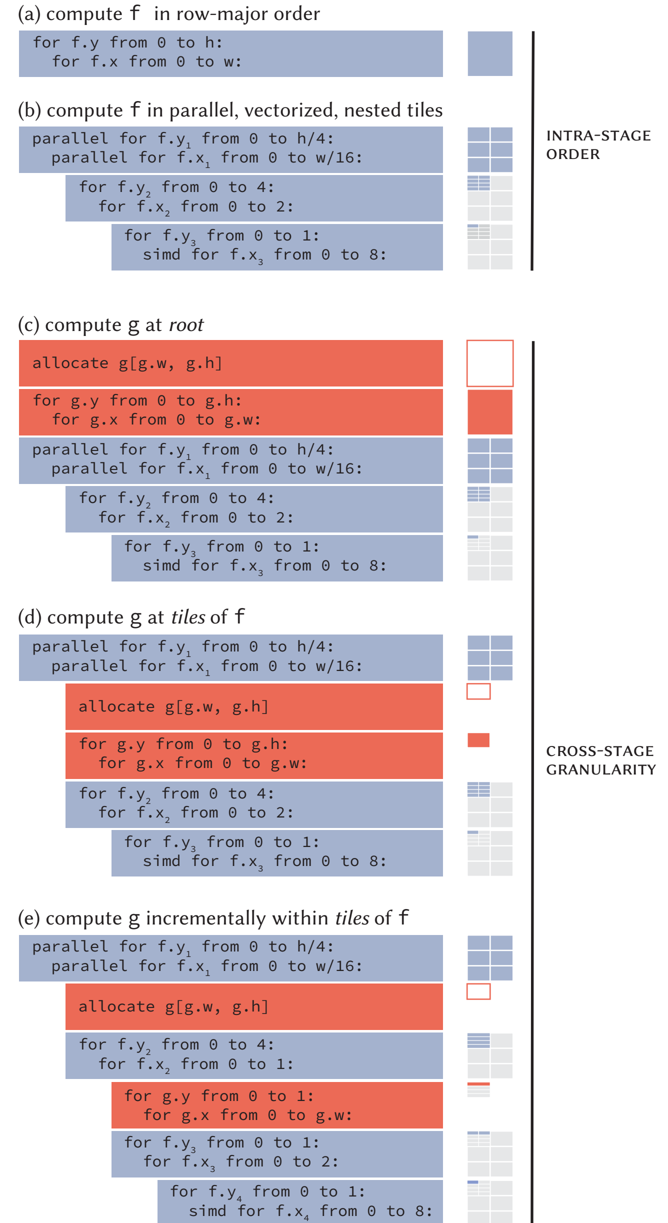

Fig. 2. Loop nests corresponding to different scheduling choices. *Top:* two different schedules of the same function, `f`, over the interval $[0, w] \times [0, h]$. (a) is a simple row-major loop nest, while (b) has been tiled two levels deep, with the outermost tile loop mapped across parallel threads, and the innermost across the lanes of a SIMD vector. *Bottom:* three different *cross-stage granularity* choices for the stage `g`, given a two-level tiling of its consumer `f`. The structure of the loop over `g` does not change, but the *size* of the regions computed and stored varies depending on the granularity.

the earlier stages `g` and `h` (here, $[0, w] \times [-1, h + 1]$), but we still do not know in what order all of these points should be computed, nor where they should be stored in memory. This is defined by the *schedule*.

A Halide schedule specifies how the computation of the pipeline should be compiled into a concrete *loop nest* which evaluates all of the points necessary in each stage to compute the requested output. Each stage is computed over a multidimensional rectangular region (a dense n-dimensional array or "tensor"), and the size to be computed (and the size of the corresponding memory allocation) is inferred by the compiler. The syntax of the Halide scheduling language is unimportant, but it is critical to understand the key choices it defines.

The generated loop nest is driven most fundamentally by two sets of choices:

Intra-stage order. We first choose in what order to evaluate the region of points within each stage. This includes arbitrary dimension order and tiling across the n-dimensional domain of that stage. Given this ordering and decomposition, any of the resulting loops can also be chosen to be processed in parallel across threads or SIMD lanes (or, equivalently, GPU blocks and threads), or unrolled. Examples of both a trivial row-major loop nest and a tiled parallel loop nest over the domain of `f` are shown in Fig. 2.

Cross-stage granularity. Given a separate loop nest over the local domain of each function, we then choose at what *granularity* each stage should be computed with respect to its consumers — how their loop nests should be combined. For example, when should we compute parts of `g` with respect to the parts of `f` that use them? The entire required region of a stage can be computed before moving on to the next stage (Fig. 2, lower top), or it can be computed over just the region required at any inner loop within its consumer's domain (Fig. 2, lower middle). At the extreme, it can be *inlined* directly into the computation of each point in its consumer. Finally, the required region of a stage can be computed *incrementally*, reusing values computed in previous iterations and stored at a coarser granularity (Fig. 2, lower bottom), producing a sliding window or "line-buffered" pattern.

The intra-stage order and decomposition interact with these cross-stage choices because they create different potential granularities at which computation and storage can be placed. Different choices about order and granularity make different tradeoffs between parallelism, locality, and the total amount of work performed.

3.1 Parameterizing the Space of Schedules

In order to search over them, we represent schedules directly as specialized loop nests. A loop nest is defined by a set of choices very similar to the visual language above. It contains, for each stage:

- a nested set of n-dimensional tilings;
- a compute and storage granularity;
- annotations of any levels of the tiling to be unrolled, or spread across parallel threads or SIMD lanes.

(For simplicity we capture arbitrary loop splittings and reorderings in a uniform way as recursive tilings, potentially including degenerate tiles of size 1 in some dimensions.)

Using this representation, we enumerate possible scheduling choices one stage at a time, starting from the last stage in the program and recursing back to the input. For each stage, we make two successive choices, each of which introduces new tilings:

1. We select a compute and storage granularity at which to insert the new stage, optionally adding an extra level of tiling to the consuming stage's existing loop nest to create additional options.
2. We add inner and outer tilings to the newly-added stage for parallelism. The outer tiling is annotated to be spread across parallel threads, and the inner tiling across SIMD lanes.

Each tiling added can take on arbitrarily many different tile sizes, yielding numerous successor states from each decision, and the recursion terminates when all stages have been scheduled ($2n$ steps for a program of n stages). Fig. 3 shows an enumeration of some granularity choices for a newly-added stage `h`.

3.2 Our Search Algorithm

We make this sequence of decisions using beam search, which maintains a set of k candidate states, all of which have made the same number of decisions, but in different ways. At each stage of the search, we expand all the ways each state could make the next decision to generate successor states, aggregate these in a list, sort it using the cost model, and take the top k.

Our cost model is trained on full pipelines, but here we apply it to states in which not all scheduling decisions have been made. We address this primarily by only costing pipeline stages that have been scheduled, which implicitly treats unscheduled stages as if they were inputs. However, whenever we compute features on a scheduled stage that depend on its relationship to an unscheduled stage, we use values that produce an optimistic underestimate of the true cost: we assume that all stages yet to be scheduled are computed as far inwards as possible given their consumer's compute_at locations, to maximize predicted locality, and we assume that their storage layout is such that all vector loads from them are dense.

Pruning. This procedure enumerates a large class of legal Halide schedules. To avoid wasting time on states unlikely to result in fast code, we further prune the generated states in several ways:

- All multi-core parallelism occurs at the outermost loop level.
- SIMD loop sizes are always the native vector width for the narrowest type used in an expression.
- The number of iterations of the parallel loop should be somewhere in between the number of cores and the number of cores times 16.
- With the exception of identity stages (a single load), no value may be redundantly recomputed more than ten times.
- Loops between the storage and compute sites for a stage should be single-dimensional (i.e. all loop dimensions except one are degenerate), as Halide's sliding window optimization works best along a single axis.

Coarse-to-fine Schedule Refinement. Standard beam search operates in a single pass. However, the large number of tiling sizes we generate tends to flood the beam with trivial variations on one basic schedule. We wish to enforce more diversity in the beam. We do

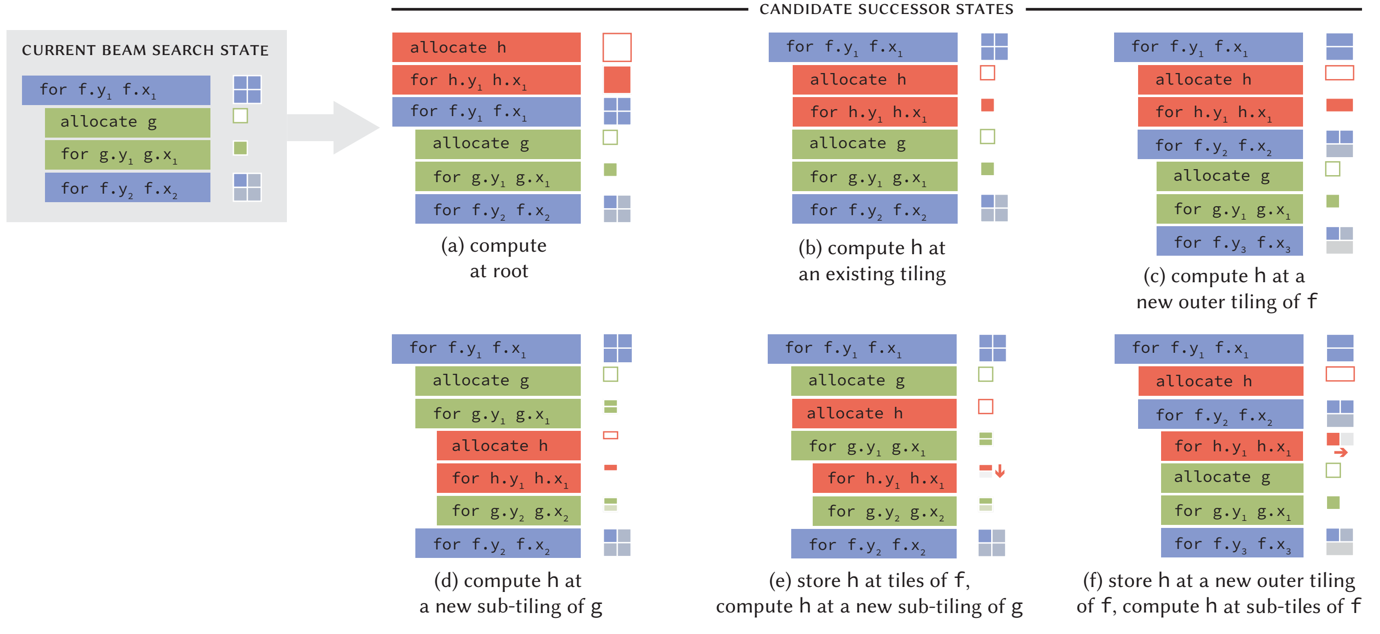

Fig. 3. We navigate the space of Halide schedules using beam search. Consider a three stage pipeline $h \to g \to f$. We begin scheduling at the end of the pipeline. In the state shown at top left, we have already scheduled g at some tiling of f, and must now generate a list of candidates for how to schedule h for the cost model to evaluate and rank. The existing Halide autoscheduler enforces a single level of tiling, so it only considers options a and b. Our search space includes different nested tilings (c, d), and tilings in which the storage and compute occur at different granularities (e, f), which in turn enables sliding window optimizations (shown as arrows). Each of these choices is parameterized by the tile sizes to use, and the choice space is further expanded by choice of storage layout for h, and the possibility of inlining it entirely.

this by defining a hash function $hash_d$ which hashes the loop nest up to some fixed depth d. We penalize the cost of all states in the beam for which a higher-rated state has the same hash. The effect of this is to preserve the backtracking ability granted by the beam for top-level scheduling decisions (e.g. which stages are computed at the root level) that are likely to have a large impact on runtime. However, fine-level decisions also benefit from beam search, so we perform a second pass in which we only consider those states with coarse-level hashes shared by states in the first pass that ultimately lead to a good final answer. The first pass finds coarse valleys in the search space, and the second pass explores them more thoroughly.

We then generalize this to an arbitrary number of passes by defining a permissible set of states. Pass p (for $p > 1$) only considers states whose hashes at depth $p - 1$ are in the permissible set, and penalizes duplicate states according to a hash at depth $p + 1$. At the end of the pass, all ancestors of the top few states in the beam have their hashes at depth p added to the permissible set. We use five passes for our results below. Please see the code in the supplemental material for the full details of the hash and the penalization procedure.

The full implementation of this search space and algorithm appear in `AutoSchedule.cpp` in the supplemental material.

3.3 Unexplored Schedules

Our algorithm explores a very large class of nested tilings, but the set of legal Halide schedules is larger still. Some scheduling features are handled using heuristics instead of search. First, we do not enumerate all possible ways to unroll loops, but instead just entirely unroll the innermost non-SIMD loop node if it is of constant size and has no more than 16 iterations total. Second, while we reorder the storage layout so that the vectorized dimension is innermost, we do not change the relative order of the other dimensions in the storage layout. For any multi-dimensional loop nodes that remain after tiling, we nest the generated loops in the same order as the storage layout, with any additional loops over a reduction domain innermost for serial loops, or outermost if the other loops were unrolled. We always `fuse` parallel loops into a single parallel loop when legal, avoiding the slight overhead involved in nested parallelism. We automatically select a strategy for tails of loops split by factors that do not divide their extent, and we automatically choose a memory space for each stage. Other Halide scheduling directives (e.g. `rfactor`, `memoize`, or `prefetch`), are not considered at all. However, we have designed our search such that any of these features could be added to the space by adding more decision points per stage.

4 PREDICTING RUNTIME

Having defined a state space and a method for generating successors within it, the final component required of beam search is a cost metric for evaluating states.

The most natural cost to minimize is actual recorded runtime, obtained via benchmarking. However, we wish to evaluate hundreds of thousands of potential schedules for each algorithm, and compiling and benchmarking a single schedule takes several seconds. Builds

should be fast and deterministic, and this setup would be neither. Instead, we train a small neural network to predict runtime based on features computed using Halide's symbolic analysis tools, and minimize this predicted runtime during the tree search.

4.1 Featurizing a Schedule

We separately featurize each stage in an algorithm (though many of the features describe the relationships between that stage and others), and we divide our featurization according to Halide's algorithm/schedule distinction.

Algorithm-specific features, which are invariant to the schedule, are histograms of the operations performed to compute a single point in the iteration domain of the stage. We classify operations performed into 40 separate buckets, using the Halide IR node types (add, multiply, subtract, load, etc). For each access made to another stage or to an input buffer, we compute *Jacobians* which describe how the coordinates accessed vary with respect to the loop dimensions of the consumer. For example, a stage that downsamples some input f by a factor of four may include a call of the form $f(4x, 4y)$, which has a Jacobian of $4I_2$. We use these Jacobians to classify memory accesses into several subcategories (for example, pointwise operations vs broadcasting). They are also used to compute several schedule-specific features below. Any non-constant terms in the Jacobian (for example due to data-dependent access) are treated as unknowns and worst case behavior is assumed.

Schedule-dependent features either count events of various types, or characterize memory footprints. The events counted include the number of times some region of a stage is evaluated, the number of times storage for it is allocated, the number of parallel tasks launched, the number of whole SIMD vectors computed, and the number of scalar values of it that are computed. We also use the Jacobians to determine the number of dense vectors and scalars loaded per vector computed, amortized across unrolled inner loops that share loaded values. This amortization captures the reuse of loaded values that is particularly important for fast matrix multiply inner loops.

For each stage, at each of several sites, we examine the shapes of all regions written to and read from. If any of the accessed stages have been scheduled inline, we look through them to their producers, multiplying Jacobians as necessary. To compute these shapes we use the bounds inference machinery built into Halide, which is based on symbolic interval arithmetic. Inspired by the work of Sioutas et al. [2018], for each region we characterize the footprint using the number of bytes touched, and also the number of contiguous segments of memory touched. The latter helps capture spatial locality, hardware prefetcher behavior, and false sharing of pages and cache lines.

The sites at which we encode these shapes include the root level of the loop nest, just inside the containing parallel loop, the site at which the stage has storage allocated, the site at which the stage is computed, and the innermost loop. At all of these sites but the last we also sum up the sizes of all allocations within that loop nest to compute a "working set" feature.

For stages which have been inlined, many of these features are zero. Their memory behavior is captured by the stage into which they were inlined.

This produces thirty-nine schedule-dependent features per stage. See Appendix A for an exhaustive list.

4.2 Cost Model Design

Attempting to predict the runtime of Halide programs directly using a neural network requires a large network and is hard to train, as this is a regression problem in which the target values have a very large dynamic range. Instead, our cost model outputs runtime as a dot product between a vector of hand-designed terms and a vector of coefficients predicted by a small neural network (Figure 4). These hand-designed terms are non-linear combinations of the schedule-specific features, crafted in such a way that one would expect them to scale proportionately to runtime. The network then just allocates more or less weight to each term, by predicting a coefficient on it. If any hand-designed term is irrelevant to performance on a particular architecture, the network is free to always predict a zero coefficient.

There are twenty-seven such terms. As an example, one of the terms is simply the number of allocations made. The network then predicts a cost per allocation for this stage. Another term estimates the total number of bytes read that were written on another core. A more complex term attempts to estimate the total number of page faults that may occur, taking into account serialization inside the kernel when multiple threads fault on the same page. The terms in our cost model are constructed procedurally from the schedule features in a differentiable program (Appendix B).

To train our network we minimize L2 error on throughput (the inverse of predicted runtime). We found that this encourages the network to spend its capacity parsing the differences between fast schedules, rather than predicting precisely how slow very bad schedules are. Training batches are composed of 32 schedules for a single random program. The throughputs are computed relative to the fastest schedule within each batch, to bring algorithms with very different runtimes into the same dynamic range. We implemented the network in Halide itself, and use the work of Li et al. [2018] for gradient descent. We use Adam [Kingma and Ba 2015] as our minimizer, with standard hyper-parameters.

4.3 Autotuning

Our cost model does not perfectly predict performance, and beam search is not guaranteed to globally minimize it. We can therefore find faster schedules by adding sampling and benchmarking to our search. We generate candidate schedules by importance sampling paths down the search tree using the cost model. At each stage of beam search, we expand the best predicted state with some probability p. Otherwise, we discard it, and repeat the procedure on the next-best state, to give an exponentially-decaying distribution. We typically set p such that we have a $\frac{1}{20}$ chance of operating entirely greedily and never discarding any states, so for an algorithm with s stages, $p = \frac{1}{20}^{\frac{1}{2s}}$.

Given a modest number of these samples, we can benchmark each to find the fastest. Given more, we use them to iteratively retrain the model in between batches of samples to specialize it to a particular

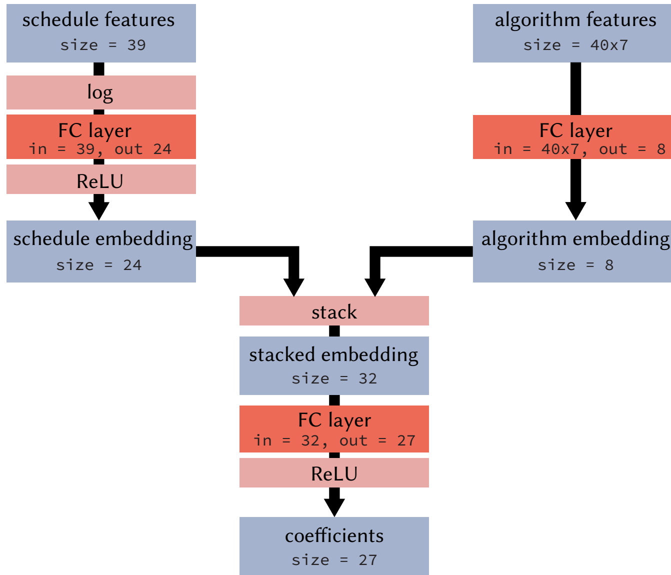

Fig. 4. Our cost model network architecture. The network has two heads, one for embedding schedule-specific features and the other for embedding algorithm-specific features. The schedule features are first log transformed, as they have a large dynamic range and are most naturally combined multiplicatively. We constrain the weights on the algorithm head to be positive by passing them through a sigmoid, to encourage cost to monotonically increase with operation count. Each embedding head outputs a low-dimensional vector. These are stacked and passed through a final linear layer and relu to produce 27 positive coefficients for our 27 hand-designed terms.

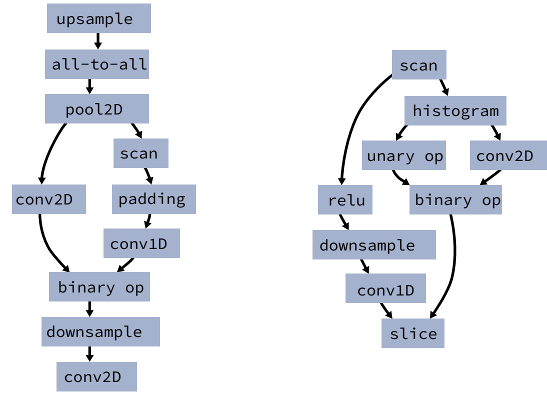

Fig. 5. Example Random Pipelines: Two random pipelines generated by our system. Our generator creates DAG-structured pipelines out of a library of stage types commonly used in image processing and deep learning. Each stage may use any of Halide's scalar types and may use randomly generated expressions of their inputs.

algorithm. In order to guarantee that the result of this procedure is always at least as good as just running beam search, we include beam search with no random dropout as one of the samples within each such batch.

5 GENERATING TRAINING DATA

Our training data comes from a random algorithm generator, which synthesizes a diverse range of programs that exhibit most of the patterns of computation present in real applications (Figure 5). For the full details of the vocabulary of this generator, see its source code in the supplemental material.

We found that the random algorithms did not need to be long, or even meaningful as algorithms. Our cost model reasons locally about stages in their immediate context, so it is sufficient to generate pipelines long enough to represent all the sorts of situations in which a stage could find itself. We do not rely on or exploit any coarse-scale structure or meaning.

One constraint that we did find necessary was that the programs must have some notion of the units on each spatial dimension to safely mix stages. Operations that mix across scales, for example adding an image to a smaller copy of itself, create shift-varying memory footprints, which invalidate some of our analysis (see Section 7). Fortunately, real imaging and learning pipelines tend to have these implicit units on their dimensions.

For each algorithm we generate in this way, we synthesize a batch of 32 schedules using the autotuning procedure described in Sec. 4.3, with the random dropout probability set so we have a 50% chance of rejecting any given state. We harvest the featurizations and runtimes produced in this way to form our training set.

To autotune we require weights for the cost model, so we seem to have begged the question. For generating an initial round of training data we use uniform random weights in $[\frac{-1}{2}, \frac{1}{2}]$. The structure of the network and the pruning during search are enough for random weights to almost always generate runnable schedules. There are exceptions, and so we kill any compilation or benchmarking job that takes more than ten minutes. We typically see a yield of 99.5%.

After this initial round of generating training data we train a model, and then repeat using the newer weights. This iteratively focuses the network on the sorts of schedules that beam search is likely to generate. We have found that five rounds of this bootstrapping procedure, with ten thousand random algorithms times thirty-two random schedules per round, is sufficient to converge.

Our data generation process is distributed over a cluster of machines, each of which searches for and compiles multiple programs in parallel, then benchmarks each one sequentially. For x86 weights, the cluster machines use a variety of server-class Xeon chips and for GPU samples we use a cluster of NVIDIA V100 GPUs.

6 RESULTS

Our system is designed to be able to produce good schedules in short compile times and better schedules given longer compile times. One way to exploit additional compile time is to increase the beam size. If more than a few minutes are available, it is worth actually compiling and benchmarking some promising candidate schedules, as described in Section 4.3. At the opposite extreme, for the shortest compile times our system can operate in greedy mode, using a beam size of 1.

We evaluate our system in these various compile-time regimes on a test set of random pipelines produced by our random pipeline generator, as well as a benchmark suite of realistic image processing pipelines. All performance results were generated under Ubuntu

18.04 on an Intel Core i9-7960X CPU. We compare our results to that of the current autoscheduler used by Halide, which we will refer to from now on as "master", on these two datasets. Results on random pipelines are shown and discussed in Figures 6 and 7.

6.1 Application suite

We also evaluate the system on a benchmark suite of real applications to ensure we do not overfit to randomly-generated pipelines. All algorithms were run on the standard 5 megapixel test image from the Halide repository. The applications are largely taken from the Halide repository or [Mullapudi et al. 2016], and the code for all is in the supplemental material. Here we will only describe the changes and additions relative to that work.

- *stencil chain* is a linear sequence of 32 non-separable 5×5 convolutions on a single channel of the test image.
- *camera pipe* has been augmented since the publication of previous autoscheduler [Mullapudi et al. 2016] by the addition of a sharpening filter.
- *conv + relu* is a single layer of a CNN at a size and shape for which we were able to write a manual schedule that beat the performance of MKLDNN [2016] (24 output channels, 120 input channels, 100x80 spatial size, and a batch size of 5).
- *IIR blur* is a blur implemented using a four pass (top to bottom, bottom to top, left to right, right to left) IIR filter.
- *BGU* is a bilateral-guided upsampling [Chen et al. 2016] pipeline. The human-written schedule performs poorly relative to the automatic methods. We note that the human-written schedule was published with the paper as the reference implementation, and was written by a co-creator of Halide. If we exclude this one result, human performance rises to 110% faster than the baseline, and is still inferior to our system when operating over similar time scales (hours).
- *Resnet50* is the popular deep residual network [He et al. 2016]. We do not compare to a human-written Halide schedule here. When compared to the most popular machine learning frameworks, we find our implementation to be competitive with MKLDNN-based frameworks. It is faster than TensorFlow [2016] and PyTorch [2017], but slower than MXNet [2015].

The results of this evaluation can be seen in Figure 8. On average, our system is significantly faster that prior work in all modes, and also significantly exceeds expert human performance when given the ability to benchmark. The raw runtimes and code to reproduce these results are available in the supplemental material.

6.2 Other architectures

In decoupling the featurization, the search space, and the search algorithm, we have designed our system to be adaptable to new architectures by extending any of these as necessary. However, this does not mean adding a new architecture is a trivial undertaking. Each new architecture requires going through a similar bring-up process as we did for x86 CPUs. We must train a cost model on hundreds of thousands of fresh random programs, and triage the results to ensure that the featurization correctly captures the factors that matter for performance on that architecture.

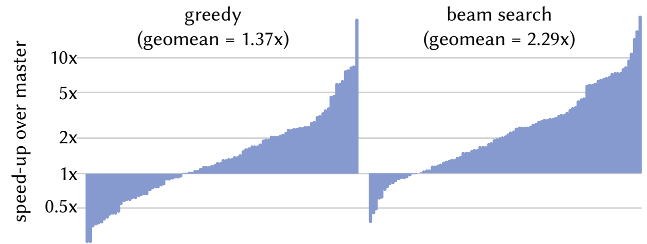

Fig. 6. Speed-up over the existing Halide autoscheduler on one hundred unseen random pipelines of lengths between 10 and 40 stages. Our algorithm operating in fast greedy mode is on average 1.37 times faster than the master autoscheduler. With beam search, our system is 2.29 times faster. The dynamic range is high — we are sometimes more than ten times faster, and sometimes less than half the speed. We generally see more outliers on the positive end, indicating that our method is more reliable than master. We found no benefit in autotuning on the random pipelines. While these random pipelines were not part of the training set, they are samples from the same distribution, and so the model can predict runtime accurately (see Figure 7) and gains little from benchmarking.

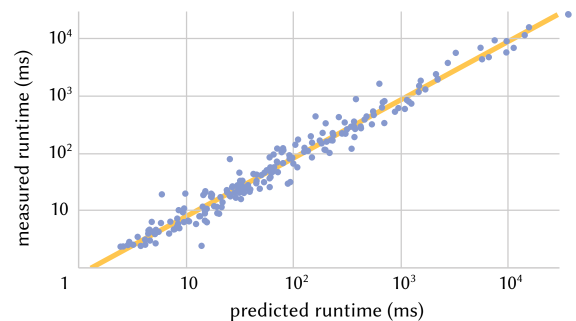

Fig. 7. Actual vs predicted runtimes for the random pipelines used in Figure 6. R^2 correlation is 0.96. The model was trained on a cluster of Intel Xeon servers in a managed Linux environment, yet these runtimes were measured on a desktop x86 CPU running vanilla Ubuntu, so the line of best-fit is slightly off the diagonal. For our scheduling algorithm to work we do not need to predict runtimes accurately, we just need to put them in the correct order, so minor changes in the target architecture are unimportant.

The most straight-forward new architecture to add is ARM CPUs. They mostly differ from high-end x86 quantitatively, rather than qualitatively. We evaluated our x86 model on quad-core Cortex A72 instances available on AWS. In greedy mode it is 9% faster than Halide master, and running beam search it is 23% faster, despite the fact that these weights were fit to results from processors on the opposite end of the performance spectrum. We lack enough ARM training samples to date to train network weights specifically for that architecture.

Adding CUDA GPUs requires changes to the search space. These are in flux, but at the time of writing they are as follows. For SIMD loop sizes used in the search, we use a value of 32 for all stages, to encourage a full warp of 32 threads along those dimensions. Using Halide's symbolic analysis tools, we reject any schedules with a per-block allocation size that would exceed the GPU's shared memory limit. We map the outermost parallel loop level to GPU blocks and the SIMD loop to GPU threads. We currently make

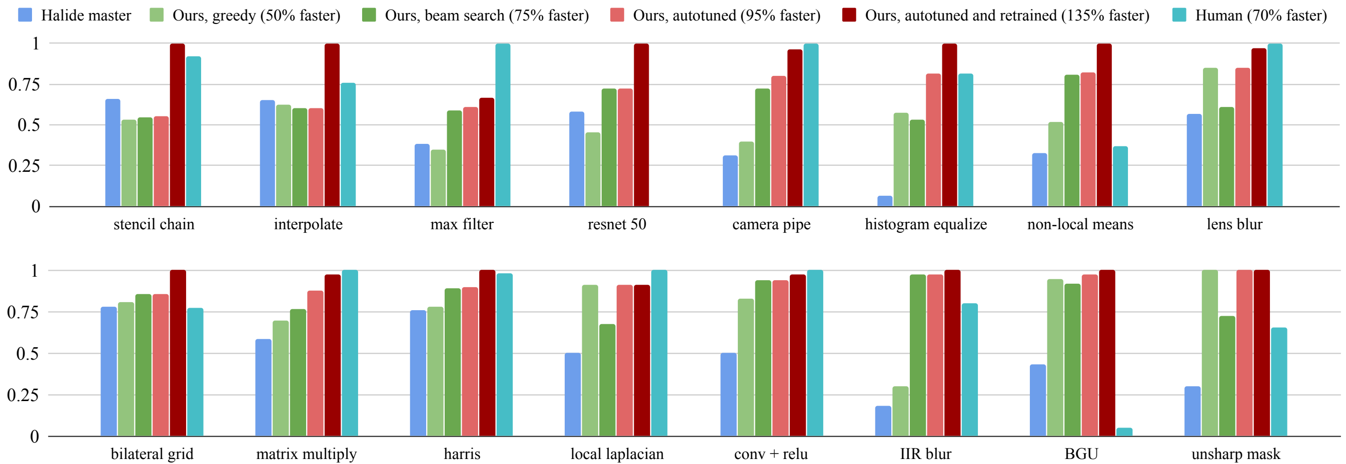

Fig. 8. Throughput relative to the throughput of the best known schedule on a suite of real applications, running on a high-end x86 CPU. Our model was not trained on these applications. From left to right within each cluster we have the implementation of Mullapudi et al. in the Halide open source repository, our system operating greedily (to minimize compile time), then our system with beam search using a beam of size 32. These three techniques all produce schedules without needing to compile and benchmark. Our system is 50% faster than the baseline when operating greedily, and 75% faster using beam search. Next we have our system operating in a mode where we importance sample our cost model to generate 32 probably-good schedules and benchmark to find the best. This takes about ten minutes and produces schedules twice as fast as the baseline. If we instead spend three hours generating and benchmarking more schedules, continuously using them to retrain the model for the specific application, we are on average 135% faster than the baseline. Note that beam search is not uniformly better than greedy search. Even if the cost model were perfect, beam search does not guarantee optimality. Neither do expert humans, who produced schedules 70% faster than the baseline.

no changes to the featurization. For a baseline here we use the GPU autoscheduler from Li et al.'s work [2018], which is to our knowledge the only functional GPU autoscheduler for Halide. On the application suite, our system operating greedily is currently 29% faster on average, and using beam search it is 33% faster. (We exclude lens blur and conv + relu from this comparison because the baseline algorithm would not run them and Resnet50 because the necessary baseline modifications were unavailable.) We consider these results preliminary but promising: careful comparison to hand-written baselines of standard applications is important for better absolute calibration and the current search restrictions to ensure valid GPU configurations are simplistic, likely excluding many faster choices.

6.3 Additional comparisons and ablations

We also performed a number of experiments on related systems and ablations of our system. None of these alternatives exceeded the performance of our system operating in greedy mode, so we mention them only briefly.

Autotuning Halide master. Mullapudi et al. report additional performance can be attained by searching over a set of small variations of their parameter space. We reproduced this result, and found that it was 21% faster than master on the application suite above.

PolyMage. The closest related language with a fully-automatic scheduler is PolyMage [Jangda and Bondhugula 2018]. The PolyMage repository reproduces a subset of this benchmark suite, though the code has decayed over time. Over the four applications that were functional at the time of writing (harris, unsharp, interpolate, and bilateral grid), PolyMage is 18% faster than Halide master.

No coarse-to-fine. We measure the benefit of doing coarse-to-fine passes (Section 3.2) by removing it, and instead increasing the beam size to 160 to maintain the same total compile time. This produced results 48% faster than Halide master, which is on par with our greedy result, indicating that a large beam is useless if it has no diversity.

No nested tiling. In order to test the relative importance of our better search algorithm and our expansion of the search space, we constrained beam search to only consider schedules within the space considered by Halide master. It produced schedules that were 28% faster than Halide master (compared to 75% on our expanded space), indicating that the improved search algorithm and cost model are important, and so is the expanded search space.

Random search vs retraining. A final natural question to ask is whether our fastest mode (autotuning with retraining) is fast due to the retraining, or simply due to taking more time to benchmark more random samples. To test this we ran beam search using the final set of weights produced during retraining. We found it to be 82% faster than Halide master, which is only slightly better than using the original, non-retrained weights. It seems most of the benefit comes from taking more random samples and taking the best schedule out of those samples.

6.4 Compiler performance

We intend this algorithm to work on very large programs, so scalability with respect to program size is critical. One reason we select beam search is because it scales linearly with the number of decisions to be made, which for our parameterization is bounded by twice the pipeline size. Beam search also scales linearly in the size of

the beam and the branching factor (the average number of successor states to a state). We use the size of the beam as a constant that we can tune according to available compile time. Search time also scales linearly with the number of coarse-to-fine passes of search we perform.

Above we report results at beam sizes of 1 (with one pass) and 32 (with five passes). The branching factor is dominated by the number of tiling choices we consider. This is the set of factorizations of a multi-dimensional domain, so it scales poorly with the size and dimensionality of the stages in the program. We keep this under control by logarithmically spacing the sizes, and automatically coarsening the sampling if more than a fixed number of candidates are generated.

At a beam size of 32, the slowest-to-compile application is *lens blur*, which evaluates 1.47 million potential schedules in 166 seconds, or nine microseconds per schedule. The fastest is *conv + relu*, which takes 334 milliseconds and considers just over ten thousand schedules. At a beam size of one, i.e. operating greedily, these times are 1.6 seconds and 15 milliseconds respectively. Compile times are dominated by generating candidate schedules and computing their featurization. The cost model itself is small and easy to parallelize (across batches of candidate schedules). Furthermore, the largest single layer in the network embeds the algorithm-specific features, which do not vary across a batch of schedules for the same algorithm, so it can be pre-computed, leaving very little work to do per distinct schedule.

7 LIMITATIONS AND FUTURE WORK

Expanding the training set. We have demonstrated that we can train a cost model on random programs and have it produce excellent schedules on real applications. A natural next step is to contribute samples harvested from real applications back to the training set. This will be particularly powerful when a user writes an algorithm for which our system produces a lousy schedule: our first response can be to ask them to donate it to our training set.

Expanding the search space. Our ability to grow the training set makes our algorithm extensible to new application domains. We have similarly designed the system to be extensible to new scheduling primitives and new architectures by expanding the search space and featurization. However, investigating slow schedules and crafting new features that capture what makes them slow is difficult work. A system that relied less on manual feature engineering (but was still capable of exploiting Halide's internal analysis passes) could adapt more gracefully to newer architectures, and could capture aspects of performance that we have missed on current architectures.

Reasoning in terms of constants. One major limitation of our system is that in order to evaluate a cost it must reason in terms of constant-valued properties of the algorithm and schedule. This means the user must provide an output size for which to optimize (though the resulting code will work at other sizes too). We also compute features of tiled loops by examining the memory footprints of a single representative iteration. We pick the middle-most. For some algorithms the middle-most iteration may be unusually cheap. For example when adding a matrix to its own transpose, the middle-most iteration only requires a single value of the input. This could lead beam search to a pathologically slow schedule.

Predicting performance through a deep compiler stack. Finally, our cost model deals with loop nests in an abstracted way in order to search over them quickly. When a final loop nest is selected as the one to compile, a heavy-duty compiler stack takes over. We are asking the cost model to not only predict the performance of the underlying hardware, but also predict the behavior of this compiler stack. This is notoriously difficult for humans, and sensitive to small changes in the underlying compiler.

8 CONCLUSION

We have introduced a new automatic scheduling algorithm for high-performance image processing and deep learning. For this, we have described a new parameterization of the space of valid Halide schedules, a beam search variant, manually-derived features, a trained cost model, and optional benchmarking and fine tuning. Our method significantly outperforms previous work on the automatic scheduling of high-performance image processing code, often by a factor of more than 2×. It can naturally span a trade-off between compile time and final performance and can be extended to handle new algorithms, scheduling directives, or architectures by adding features and retraining.

ACKNOWLEDGMENTS

This work was partially funded by Toyota, as well as the NSF/Intel Partnership on Computer Assisted Programming for Heterogeneous Architectures through grant CCF-1723445.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* (2016). http://arxiv.org/abs/1603.04467

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM. https://doi.org/10.1145/2628071.2628092

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5 (Sept. 2018). https://doi.org/10.1145/3197978

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society. https://doi.org/10.1109/PACT.2015.17

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM. https://doi.org/10.1145/1375581.1375595

Jiawen Chen, Andrew Adams, Neal Wadhwa, and Samuel W. Hasinoff. 2016. Bilateral Guided Upsampling. *ACM Trans. Graph.* 35, 6 (Nov. 2016). https://doi.org/10.1145/2980179.2982423

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* (2015). http://arxiv.org/abs/1512.01274

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* (2018). http://arxiv.org/abs/1802.04799

Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to Optimize Tensor Programs. *CoRR* (2018). http://arxiv.org/abs/1805.08166

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE.

Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. 2008. MILEPOST GCC: machine learning based research compiler. In *GCC summit*.

Tobias Grosser, Armin Groslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012).

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph. (Proc. SIGGRAPH)* 33, 4 (July 2014).

Intel. 2016. Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/intel/mkl-dnn

Abhinav Jangda and Uday Bondhugula. 2018. An effective fusion and tile size model for optimizing image processing pipelines. In *Symposium on Principles and Practice of Parallel Programming*.

Diederick P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*.

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4 (July 2018). https://doi.org/10.1145/3197517.3201383

Charith Mendis, Saman Amarasinghe, and Michael Carbin. 2018. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. *ArXiv e-prints* (Aug. 2018). arXiv:cs.DC/1808.07412 https://arxiv.org/pdf/1808.07412.pdf

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4 (July 2016). https://doi.org/10.1145/2897824.2925952

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015). https://doi.org/10.1145/2786763.2694364

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4 (July 2012). https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013). https://doi.org/10.1145/2499370.2462176

Mohammed Rahman, Louis-Noël Pouchet, and P Sadayappan. 2010. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*.

D. Raj. Reddy. 1977. *Speech Understanding Systems: A Summary of Results of the Five-Year Research Effort*. Department of Computer Science Technical Report. Carnegie Mellon University.

Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. 2018. Loop Transformations Leveraging Hardware Prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM. https://doi.org/10.1145/3168823

Savvas Sioutas, Sander Stuijk, Luc Waeijen, Twan Basten, Henk Corporaal, and Lou Somers. 2019. Schedule Synthesis for Halide Pipelines Through Reuse Analysis. *ACM Trans. Archit. Code Optim.* 16, 2 (April 2019). https://doi.org/10.1145/3310248

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* (2018). http://arxiv.org/abs/1802.04730

A FEATURIZATION

Here we list the schedule-dependent terms of the featurization used by the cost model (Sec 4.1). All features are computed per-stage. Note that for inlined stages, most of the features are zero, and their contribution is instead captured by the stage into which they are inlined. See the source code in the supplemental material for the details of how each feature is computed.

`num_realizations` The number of times storage for this stage is allocated. The product of the outer loop extents around the `store_at` site.

`num_productions` The number of times a tile of this stage is computed. The product of the outer loop extents around the `compute_at` site.

`points_computed_per_realization` The number of points computed of this stage per allocation. The product of the loop extents within the `store_at` site.

`points_computed_per_production` The number of points computed of this stage per region computed. The product of the loop extents within the `compute_at` site.

`points_computed_total` The number of points computed. Equal to `num_realizations` × `points_computed_per_realization`.

`points_computed_minimum` The minimum number of points that must be computed to produce a correct output. This is not a function of the schedule, but it is a useful reference point to evaluate redundant recomputation.

`innermost_loop_extent` The number of points in the iteration domain for a single tile of this stage.

`innermost_pure_loop_extent` The number of points in the iteration domain for a single tile of this stage, excluding loops corresponding to reductions. Measures the number of independent values of the stage computed per tile.

`unrolled_loop_extent` If the innermost loop over a tile is to be unrolled, this is equal to `innermost_pure_loop_extent`. If not, this feature is one.

`inner_parallelism` The number of parallel jobs launched while computing a region of this stage. Equal to the product of the parallel inner loops. Currently always one except for stages scheduled at root, because we force all parallelism to the outer loop.

`outer_parallelism` The maximum possible number of simultaneously-computed regions of this stage. Equal to the product of the parallel outer loops.

`bytes_at_realization` The size in bytes of the storage backing this stage.

`bytes_at_production` The size in bytes of the memory footprint written to per region computed of a stage.

`bytes_at_root` The size in bytes of the memory required to back this stage, if it were computed at the root level. Does not depend on the schedule, but acts as a useful point of reference.

`innermost_bytes_at_realization` The size in bytes of the storage for this stage along the innermost dimension, E.g. for a row-major matrix, this would be the length of a row in bytes.

`innermost_bytes_at_production` The size in bytes along the innermost dimension of the region written to per production.

`innermost_bytes_at_root` The size in bytes along the innermost dimension if this stage were computed at root.

`inlined_calls` For inlined Funcs, the total number of times the Func is evaluated. Otherwise zero.

`unique_bytes_read_per_realization` The total number of unique bytes loaded from all inputs per realization.

`unique_lines_read_per_realization` Similar to the above, but counts the number of contiguous-in-memory lines of data instead of bytes.

`allocation_bytes_read_per_realization` The sum of the sizes of all allocations accessed per realization of this stage.

`vector_size` The vectorization factor (SIMD width) used to compute this stage.

native_vector_size The native SIMD width for the narrowest type used when computing this stage. Does not depend on the schedule, but can be combined with the previous feature to detect unused SIMD lanes.

num_vectors The total number of entire SIMD vectors used to compute this stage.

num_scalars The total number of points of this stage computed as unvectorized code, typically due to loop tails.

scalar_loads_per_vector The number of scalar load instructions used per vector computed. These can arise from broadcast operations, vector gathers, or strided loads with stride greater than four.

vector_loads_per_vector The number of dense vector load instructions used per vector computed. Strided vector loads with stride less than five are treated as the equivalent number of dense loads.

scalar_loads_per_scalar The number of load instructions when computing a single scalar value of this stage.

unique_bytes_read_per_vector The number of unique bytes loaded to compute a single SIMD vector of this stage.

unique_lines_read_per_vector The number of contiguous memory spans accessed to compute a single SIMD vector of this stage.

unique_bytes_read_per_task The number of unique bytes accessed per parallel task. As with bytes_at_task, if the stage is computed at finer granularity than a single task, we measure the union of the regions accessed.

bytes_at_task The total number of bytes written of this stage per parallel task. If the stage is computed at a finer granularity than parallel tasks, then we take the union of the regions written to per parallel task of the containing parallelized stage.

innermost_bytes_at_task The same as the previous feature, but only considers the size of the innermost storage dimension.

unique_lines_read_per_task The number of contiguous memory spans accessed per parallel task.

working_set The sum of the sizes of all temporary allocations made while computing a region of this stage.

working_set_at_task The sum of the sizes of all in-scope allocations at the containing or inner parallel task site.

working_set_at_production The sum of the sizes of all in-scope allocations at the production site.

working_set_at_realization The sum of the sizes of all in-scope allocations at the realization site.

working_set_at_root The sum of the sizes of all allocations made for the entire algorithm. This feature is the same for all stages.

B MANUAL COST MODEL

This appendix describes the hand-designed tail end of our cost model from Section 4.2. The coefficients predicted by the neural network are written as c_i. We begin by defining two auxiliary functions:

$$m(x) = max(x, 1) \qquad select(x, t, f) = if\ t\ then\ x\ else\ f$$

The cost is then given by the following:

$$\textbf{compute_cost} = select(\texttt{inlined_calls} > 0,\ \texttt{vector_size} \times \texttt{num_vectors} \times c_0 + \texttt{num_scalars} \times c_1,\ \texttt{vector_size} \times \texttt{num_vectors} \times c_2 + \texttt{num_scalars} \times c_3)$$

$$\textbf{num_tasks} = m(\texttt{inner_parallelism} \times \texttt{outer_parallelism})$$

$$\begin{aligned}\textbf{load_cost} = &(\texttt{num_realizations} \times \texttt{unique_lines_read_per_realization} \times c_4 + \\ &\texttt{num_realizations} \times \texttt{unique_bytes_read_per_realization} \times c_5 + \\ &\texttt{num_vectors} \times \texttt{vector_loads_per_vector} \times c_6 + \\ &\texttt{num_scalars} \times \texttt{scalar_loads_per_scalar} \times c_7 + \\ &\texttt{num_vectors} \times \texttt{scalar_loads_per_vector} \times c_8 + \\ &\texttt{num_scalars} \times \texttt{unique_bytes_read_per_vector} \times c_9 + \\ &\texttt{num_vectors} \times \texttt{unique_bytes_read_per_vector} \times c_{10} + \\ &\texttt{num_scalars} \times \texttt{unique_lines_read_per_vector} \times c_{11} + \\ &\texttt{num_vectors} \times \texttt{unique_lines_read_per_vector} \times c_{12} + \\ &\texttt{num_tasks} \times \texttt{unique_bytes_read_per_task} \times c_{13} + \\ &\texttt{num_tasks} \times \texttt{unique_lines_read_per_task} \times c_{14})\end{aligned}$$

$$\textbf{tasks_per_core} = \texttt{num_tasks}/\texttt{num_cores}$$

$$\textbf{penalized_compute_cost} = \texttt{compute_cost} \times \frac{\lceil \texttt{tasks_per_core} \rceil}{m(\texttt{tasks_per_core})}$$

$$\textbf{lines_written_per_realization} = \frac{\texttt{inner_parallelism} \times \texttt{bytes_at_task}}{m(\texttt{innermost_bytes_at_task})}$$

$$\boldsymbol{\alpha} = select(\texttt{inner_parallelism} > 1, c_{15}, select(\texttt{is_output}, c_{16}, c_{17}))$$

$$\boldsymbol{\beta} = select(\texttt{inner_parallelism} > 1, c_{18}, select(\texttt{is_output}, c_{19}, c_{20}))$$

$$\textbf{store_cost} = \texttt{num_realizations} \times (\alpha \times \texttt{lines_written_per_realization} + \beta \times \texttt{bytes_at_realization})$$

$$\textbf{false_sharing_cost} = select(\texttt{inner_parallelism} > 1, \frac{(\texttt{num_vectors} + \texttt{num_scalars}) \times c_{21}}{m(\texttt{innermost_bytes_at_task})}, 0)$$

$$\textbf{max_threads_hitting_same_page_fault} = min(\texttt{inner_parallelism}, 4096/m(\texttt{innermost_bytes_at_task}))$$

$$\textbf{page_fault_cost} = \texttt{bytes_at_production} \times \texttt{max_threads_hitting_same_page_fault} \times \texttt{inner_parallelism} \times \texttt{outer_parallelism} \times c_{22}$$

$$\textbf{malloc_cost} = c_{23} \times \texttt{num_realizations}$$

$$\textbf{parallel_launch_cost} = \texttt{num_productions} \times select(\texttt{inner_parallelism} > 1, c_{24}, 0)$$

$$\textbf{parallel_task_cost} = \texttt{num_productions} \times (\texttt{inner_parallelism} - 1) \times c_{25}$$

$$\textbf{working_set_cost} = \texttt{working_set} \times c_{26}$$

$$\begin{aligned}\textbf{cost} = &\texttt{penalized_compute_cost} + \texttt{load_cost} + \\ &\texttt{store_cost} + \texttt{false_sharing_cost} + \texttt{page_fault_cost} + \\ &\texttt{malloc_cost} + \texttt{parallel_launch_cost} + \\ &\texttt{parallel_task_cost} + \texttt{working_set_cost}\end{aligned}$$