Schedule Synthesis for Halide Pipelines through Reuse Analysis

SAVVAS SIOUTAS, SANDER STUIJK, and LUC WAEIJEN, Eindhoven University of Technology, The Netherlands
TWAN BASTEN, Eindhoven University of Technology and ESI, TNO, The Netherlands
HENK CORPORAAL, Eindhoven University of Technology, The Netherlands
LOU SOMERS, Océ Technologies and Eindhoven University of Technology, The Netherlands

Efficient code generation for image processing applications continues to pose a challenge in a domain where high performance is often necessary to meet real-time constraints. The inherently complex structure found in most image-processing pipelines, the plethora of transformations that can be applied to optimize the performance of an implementation, as well as the interaction of these optimizations with locality, redundant computation and parallelism, can be indentified as the key reasons behind this issue. Recent domain-specific languages (DSL) such as the Halide DSL and compiler attempt to encourage high-level design-space exploration to facilitate the optimization process. We propose a novel optimization strategy that aims to maximize producer-consumer locality by exploiting reuse in image-processing pipelines. We implement our analysis as a tool that can be used alongside the Halide DSL to automatically generate schedules for pipelines implemented in Halide and test it on a variety of benchmarks. Experimental results on three different multi-core architectures show an average performance improvement of 40% over the Halide Auto-Scheduler and 75% over a state-of-the art approach that targets the PolyMage DSL.

CCS Concepts: • **Software and its engineering → Compilers**; *Domain specific languages*;

Additional Key Words and Phrases: Loop optimizations, image-processing pipelines, compiler optimizations, Halide

ACM Reference format:
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, Article 10 (April 2019), 22 pages.
https://doi.org/10.1145/3310248

1 INTRODUCTION

High-tech systems such as wide-format printers, radars, and health-care monitoring applications execute complex image-processing algorithms on a target platform that is typically a multi-core CPU (e.g., from ARM or Intel) with SIMD extensions. To meet the real-time constraints, the

Authors' addresses: S. Sioutas, S. Stuijk, L. Waeijen, and H. Corporaal, Eindhoven University of Technology, Eindhoven, The Netherlands; emails: {s.sioutas, s.stuijk, l.j.w.waeijen, h.corporaal}@tue.nl; T. Basten, Eindhoven University of Technology, ESI, TNO, Eindhoven, The Netherlands; email: a.a.basten@tue.nl; L. Somers, Océ Technologies, Eindhoven University of Technology, The Netherlands; email: l.j.a.m.somers@tue.nl.

1544-3566/2019/04-ART10
https://doi.org/10.1145/3310248

final implementation needs to be highly optimized for the target platform. Traditional optimizations usually include a series of loop transformations, such as tiling and loop fusion, as well as vectorization and parallelization and aim to exploit locality (spatial and temporal), data-level parallelism, and task-level parallelism, respectively. However, manually applying these transformations significantly reduces code readability and portability and discourages high-level design-space exploration.

Recently, domain-specific languages (DSLs) such as Halide [21] and PolyMage [16] were introduced to facilitate the optimization process in high-performance image-processing applications. These DSLs allow developers to express applications in a more abstract format while maintaining the ability to apply low-level optimizations and transformations on the final code. The benefits of these approaches can be invaluable in the case of image-processing pipelines where a combination of optimizations including stage interleaving or stage fusion, tiling, vectorization, and parallelization are necessary to achieve high performance.

An image-processing pipeline can be defined as a series of functional stages, where each stage contains an arbitrary number of nested loops and depends on data produced during an earlier stage. As a result, interleaving the computation of these stages can offer significant performance improvement by exploiting producer/consumer locality and ensuring that intermediate buffers are kept inside the local caches or registers. Both Halide and PolyMage employ techniques that allow for the automatic optimization of such imaging pipelines. The Halide Auto-Scheduler [15] attempts to group stages together and evaluates an effective tiling in each group. PolyMage can use both auto-tuning to search parts of the design space as well as a recently introduced model-driven approach [10]. This new approach quickly attempts to fuse stages and extends the search space to cover more solutions than the previous auto-tuning method. However, all three techniques [10, 15, 16] focus on the interleaving of the computation of each stage using overlapping tiles and therefore lead to solutions with limited reuse possibilities and often miss sliding window opportunities.

In this article, we present a novel optimization strategy for image-processing pipelines that considers stage fusion for maximum producer/consumer locality in conjunction with tile size selection while evaluating reuse possibilities not considered in previous state-of-the-art approaches. Our technique is driven by an analytical model that takes relevant application and architecture-specific parameters (such as the number of cores/threads, cache size, interaction with hardware prefetching) into account and is capable of producing optimized schedules within seconds, even for complex pipelines with a large number of stages. We implement it as a tool to be used with the Halide DSL as an alternative cost model and analysis to the Halide Auto-Scheduler and evaluate it across a variety of benchmarks and target platforms. We compare our solutions to the ones produced by the Halide Auto-Scheduler, the manual solutions given for the Halide DSL on the same benchmarks when applicable, as well as the ones produced by PolyMage (using both the original auto-tuned method, as well as the DP-fusion technique implemented in Reference [10]) on the same target architectures. We observe a substantial performance improvement across all platforms and architectures.

It is important to remark that our technique is not restricted to Halide. It can be used with other DSLs and general-purpose compilers that target image processing or tensor or linear algebra applications and offer control over the production and consumption of pipeline stages as well as the allocation of intermediate buffers.

The rest of this work is organized as follows: Section 2 discusses related work. Section 3 gives a motivational example, while Section 4 presents our proposed optimization technique in detail. Section 5 showcases the experimental results that were obtained. Conclusive remarks are finally discussed in Section 6.

2 RELATED WORK

In this section, we discuss prior related work. We identify the limitations of traditional loop fusion and tiling techniques used in general-purpose languages when optimizing image-processing pipelines and investigate some of the benefits of recent image-processing domain-specific languages.

2.1 General-purpose Languages

Loop fusion in conjunction with tiling has been extensively studied in the past, especially in the case of general-purpose compilers. Most of these approaches focus on exploiting data locality while maintaining parallelism in applications dominated by linear algebra or stencil computations [11, 17, 25, 27, 28]. More specifically, the authors in Reference [28] propose a hierarchical tiling technique for iterative solver applications to reduce communication overhead without introducing severe redundant computation. In Reference [17], the effects of various inter-loop optimization strategies on PDE solvers are investigated.

In Reference [4], the authors propose an optimization strategy for compute-intensive multi-dimensional summations that involve products of several arrays. They investigate the effects of loop fusion and tiling in such applications while also reducing the memory footprint of intermediate temporary buffer requirements.

Other approaches have focused on enabling loop fusion in applications with complex data dependencies between loop iterations [14, 26]. The authors of Reference [26] propose a technique that eliminates fusion-preventing dependencies by means of loop tiling and array copying. After iteratively applying the aforementioned method to multiple loop nests, a single equivalent nested loop can be formed that can be tiled for cache locality. In a similar fashion, Reference [14] proposes a way of mitigating the presence of fusion-preventing dependencies, while maintaining parallelism and eliminating cache conflicts in the subsequent fused loops.

However, all aforementioned methods involve traditional loop fusion techniques that target time-iterated stencils, the scope of which differs from the complex multi-dimensional problems defined in the context of image-processing pipelines, a term that covers all applications within the scope of this work. Stages in these pipelines perform various data-parallel computations before having their output consumed by the next stage, which in turn executes a different computation or stencil.

2.2 Domain-specific Languages

Recently, DSLs have emerged that enable quick design-space exploration in the image-processing domain. These DSLs provide high-level abstractions in the definitions of the functional steps inside the pipeline, as well as the ability to apply optimizations on the generated code to ensure high performance on the final implementation.

Tensor Comprehensions [24] is an example of a recent DSL that targets deep learning applications such as convolutional and recurrent neural networks. It consists of a high-level language with syntax that resembles the mathematics of deep learning and a Just-In-Time polyhedral compiler for CUDA-based GPU architectures. It employs an auto-tuner to automatically generate efficient polyhedral schedules.

PolyMage [16] is another DSL for image-processing applications that uses a dataflowlike language to describe pipelines. It employs polyhedral transformations [9, 13, 19] to optimize the computations performed by the functional stages of the pipeline with a grouping-then-tiling approach. More specifically, it relies on auto-tuning over various tile dimensions, which are all powers of two, to decide which stages of the pipeline will be grouped together. It then applies polyhedral optimizations on each group to generate the final nested loops. An alternative optimization strategy

(a) Pipeline graph

```
1  Func blurx, blury
2  Var x, y, xi, yi;
3
4  // The algorithm
5  blurx(x,y)=(input(x-1,y)+input(x,y)+input(x+1,y))/3;
6  blury(x,y)=(blurx(x,y-1)+blurx(x,y)+blurx(x,y+1))/3;
7
8  // The schedule
9  blury.tile(x,y,xo,yo,xi,yi,256,32)
10       .vectorize(xi,8).parallel(yo);
11 blurx.compute_at(blury,xo).vectorize(x,8);
12
13 blury.realize(1024,1024);
```

(b) Halide algorithm definition & schedule

```
1  parallel yo in [0,1024/32):
2    for xo in [0,1024/256):
3    allocate blurx[34][256]
4      for y in [-1,33):
5        for x.o in [0,32):
6          vectorized x.vector in [0,8)
7            blurx(...) = ...
8      for yi in [0, 32):
9        for xi.o in [0,32):
10         vectorized xi.vector in [0,8)
11           blury(...) = ...
```

(c) Equivalent C-like loop-nest

Fig. 1. 3 x 3 Blur pipeline.

for pipelines implemented in PolyMage was introduced in Reference [10]. This method introduced a dynamic fusion and tiling model that extends the search space to tile sizes that are not powers of two and resolves the need for auto-tuning. However, due to the nature of the analysis that is used in the PolyMage compiler, its application scope is limited to stencil computations and up/down sampling.

Halide [21] is perhaps the most prominent of the DSL attempts. Halide separates the algorithmic description of a pipeline from its optimization schedule. Image-processing pipelines in Halide are defined as directed acyclic graphs, where each node of the graph represents a functional stage. Each stage is equivalent to a Halide function, which specifies all producer/consumer relations at the specific stage. Furthermore, the functional description of the pipeline is independent of its optimization schedule. In other words, the Halide functions define the relations and dependencies between the stage of the pipeline but do not influence the way the stages will get executed. As a result, the optimization schedule can control both the order of execution within a single stage, as well as the way the computation of stages gets interleaved during the execution of the pipeline. Figure 1(b) shows a simple two-stage blur filter implemented in Halide, along with its optimization schedule. Given this schedule, the compiler will tile the loop of the *blury* filter using a tile size of 256×32, vectorize the innermost intra-tile loop (xi) using vectors of size 8, and parallelize its outermost inter-tile loop (yo). Furthermore, the computation of the blurx stage will be interleaved on a per-tile basis and its innermost loop will also be vectorized using vectors of size 8. In other words, before each intra-tile loop iteration, Halide will first allocate buffer space and compute all pixels of blurx that will get consumed during this iteration. The equivalent loop-nest in pseudo-C can be seen in Figure 1(c).

Halide initially employed an auto-tuning framework to automatically generate optimized schedules for pipelines [21] that required an extensive amount of time to derive an adequate schedule. A more generic auto-tuning approach that is driven by genetic algorithms was proposed in the auto-tuning framework Opentuner [2]. This framework was able to generate efficient schedules in less time for small pipelines (e.g., bilateral grid), but fails to converge to a good solution for larger, more-complex problems.

Currently, Halide uses a heuristic-based Auto-Scheduler that was initially proposed in [15] but then received an updated cost model by the Halide community [7]. This method uses a greedy grouping algorithm to group stages of the pipeline together to maximize producer/consumer locality and applies tiling to the output stage of each group independently. However, the grouping

```
allocate blurx[By+2][Bx]
for y in [0,By+2):
  for x in [0,Bx):
    blurx(...) = ...

for y in [0,By):
  for x in [0,Bx):
    blury(...) = ...
```

(a) store/compute root

```
for yo in [0,By/Ty):
  for xo in [0,Bx/Tx):
    allocate blurx[Ty+2][Tx]
    for y in [-1,Ty+1):
      for x in [0,Tx):
        blurx(...) = ...
    for yi in [0, Ty):
      for xi in [0, Tx):
        blury(...) = ...
```

(b) overlapping tiles

```
for yo in [0,By/Ty):
  for xo in [0,Bx/Tx):
    allocate blurx[3][Tx]
    for yi in [-2, Ty):
      for x in [0,Tx):
        blurx(...) = ...
      if (yi<0): continue
      for xi in [0, Tx):
        blury(...) = ...
```

(c) sliding windows inside tiles

Fig. 2. 3×3 Blur pipeline—scheduling options.

strategy excludes parts of the design space and considers only a limited number of tile sizes, and its analysis does not cover buffer allocation and storage scheduling. As a result, while it can quickly produce schedules within seconds, it misses interesting solutions of the design space that may benefit from sliding window opportunities. Those missed solutions may, however, allow for better SIMD vector unit utilization and better exploitation of the hardware prefetchers.

Recently, another analytical model was introduced in Reference [23] that automatically schedules kernels in Halide. This method takes hardware prefetching into account and involves a hierarchical tiling approach, but it is limited to single-stage pipeline and -stage fusion falls outside its analysis.

Our method considers both the compute as well as the storage levels of a stage while determining its final optimization schedule. We show that by taking both compute and store level into account, we can reduce the amount of intermediate temporary buffer space required, which in return allows for different grouping and tiling options as well as increased producer/consumer locality. Furthermore, our analytical model takes hardware prefetching inherently into account and investigates tile sizes in a larger scope.

3 MOTIVATIONAL EXAMPLE AND PROBLEM FORMULATION

In this section, we use the blur pipeline seen in Figure 1 as a motivational example to demonstrate the limitations of current state-of-the-art approaches, as well as the idea behind our work.

As already mentioned, optimizing an image-processing pipeline usually involves dealing with a complex tradeoff among parallelism, locality, and recomputation. The transformations that are often considered include a combination of loop interchange, splitting, fusion, parallelization, and vectorization. Choosing a proper fusion strategy for each stage in a pipeline has a significant effect on the performance of the final implementation. Figure 2 shows three example schedules for the blur pipeline in pseudo-C syntax. The amount of reuse or recomputation, as well as the size of the intermediate buffer that is required can be controlled through the combination of various loop permutations, tile sizes, and levels at which we compute and store each stage of the pipeline. For example, the solution shown in Figure 2(a) computes all necessary pixels in *blurx* before consuming them to compute *blury*. Such a schedule avoids all recomputation but suffers from poor locality and a large intermediate buffer (depending on the problem size). However, fully inlining the producer (*blurx*) into its consumer (*blury*) increases locality but at the cost of the highest recomputation.

Current state-of-the-art approaches (e.g., the current Halide Auto-Scheduler) only consider scheduling options where compute and store are set to the same level of a loop nest. As an example, consider the schedule seen in Figure 2(b). In this case, tiling the iteration space of *blury* and fusing its producer into the innermost inter-tile loop (*xo*) allows for an intermediate solution that offers increased locality compared to the fully stored implementation and less recomputation than the fully inlined one. Furthermore, it does not hinder parallelization of the outermost

inter-tile loop, since the computation of *blurx* is interleaved at a lower level than the parallel loop. We can quantify the amount of intermediate storage (B_{blurx}) needed as well as the amount of recomputation (R_{blurx}) in such a schedule for arbitrary tile dimensions:

$$B_{blurx} = T_x(T_y + v_y) \cdot sizeof(DataType), \tag{1}$$

where T_x, T_y are the tile sizes in the x and y dimensions and v_y is the amount of overlap between *blurx* and its consumer *blury* in the y dimension (in this example $v_y = 2$),

$$R_{blurx} = Cp_{blurx}^{blury_{xo}^{xo}} - C_{blurx}^{root}, \tag{2}$$

where $Cp_{blurx}^{blury_{xo}^{xo}}$ is the total computation cost of *blurx* in this fusion scenario and C_{blurx}^{root} is the cost when all of its pixels are computed and stored before being consumed (as in Schedule (a)).

More specifically, $Cp_{blurx}^{blury_{xo}^{xo}}$ is the cost of *blurx* when fused into *blury* with its computation (subscript *xo*) and allocation (superscript *xo*) set to the *xo* level/index of the loop-nest of *blury*. Similarly, C_{blurx}^{root} is the cost of computing and storing *blurx* outside the loop nest of the consuming stage *blury*,

$$Cp_{blurx}^{blury_{xo}^{xo}} = T_x(T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y}, \tag{3}$$

$$C_{blurx}^{root} = B_x(B_y + 2), \tag{4}$$

where B_x, B_y are the problem sizes (loop bounds) in the x and y dimensions, respectively.[1]

Similar equations can be used to calculate the load cost (C_l) for *blurx*, which is equivalent to the load cost for the input data (C_{input}^{blurx}). In detail:

$$Cl_{blurx}^{blury_{xo}^{xo}} = C_{input}^{blurx} = (T_x + 2)(T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y}. \tag{5}$$

In the presence of a streaming hardware prefetcher,[2] the previous equation becomes

$$Cl_{blurx}^{blury_{xo}^{xo}} = (T_y + 2)\frac{B_x}{T_x}\frac{B_y}{T_y}, \tag{6}$$

where we eliminate the sequential accesses across cache lines. Finally, the amount of data that needs to stay in the cache to benefit from input data reuse is

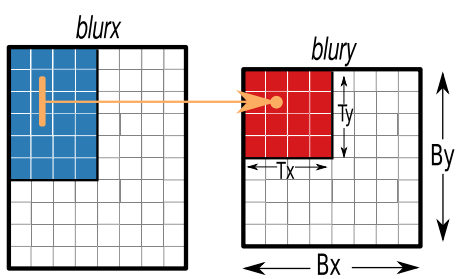
$$B_{input} = (T_x + 2)(T_y + 2) \cdot sizeof(DataType). \tag{7}$$

Such a schedule benefits from increased locality compared to the one with root storage. Furthermore, as seen in the above equations, the tradeoff between redundant computation and locality can be controlled by tuning the applied tile dimensions. It should also be noted that a different inter or intra-tile loop permutation leads to different buffer requirements and cost-functions. Figure 3(a) shows a visual representation of the schedule for an 8×8 output image with 4×4 tiling ($B_x = 8$, $B_y = 8$, $T_x = 4$, and $T_y = 4$). As seen in the figure, all blue pixels of *blurx* are evaluated and stored before being consumed to produce one red tile of *blury*.

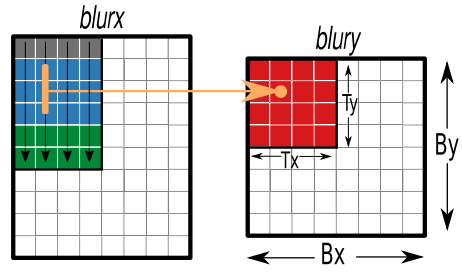
The third schedule (Figure 2(c)) shows an implementation where computation and storage are set to different levels. Such schedules benefit from sliding window opportunities that usually enable the folding of intermediate buffers without reducing the amount of data reuse. As an example, consider the buffer requirements for this schedule (all of the other costs will be the same as in

[1]To keep the equations and the example clear, we assume that the loop bounds in each dimension are a multiple of the tile size.

[2]We assume that the problem size in the column dimension is larger than the size of a physical page, and therefore the constant stride prefetchers cannot follow the stride of the non-consecutive load operations.



(a) Overlapping tiles: all blue pixels of $blurx$ are evaluated before being consumed to produce an area equal to the output tile in $blury$ (red pixels). The memory allocation for $blurx$ should be as large as the blue area.



(b) Sliding windows inside tiles: values of $blurx$ are computed per line of the output tile as needed. Pixels that are no longer needed are discarded (gray area). The intermediate buffer requirement is equal to the blue area. Green pixels will get evaluated in future intra-tile iterations

Fig. 3. Overlapping tiles and sliding window implementations.

schedule 2(b)). Starting from Equation (1) (since the store level remains the same), we can calculate the memory footprint of the producer $blurx$ stage as follows:

$$B_{blurx} = T_x(T_y + 2) \cdot sizeof(DataType). \tag{8}$$

Since pixels of $blurx$ are now computed per line of the output tile (yi), we do not need to keep all of them in the intermediate buffer but only those that can be reused across intra-tile iterations (or across one xo iteration). Therefore, B_{blurx} can be folded down to a circular buffer of size:

$$B_{blurx} = T_x(1 + 2) \cdot sizeof(DataType) = 3T_x \cdot sizeof(DataType). \tag{9}$$

The same holds for the input data buffer, which will now be

$$B_{input} = (T_x + 2)(T_y + 2) \cdot sizeof(DataType) \tag{10}$$

and will be folded down to:

$$B_{input} = (T_x + 2)(1 + 2) \cdot sizeof(DataType) = 3(T_x + 2) \cdot sizeof(DataType). \tag{11}$$

Figure 3(b) shows a visual representation for a small 8×8 output image with an applied tile size of 4×4 ($B_x = 8$, $B_y = 8$, $T_x = 4$, and $T_y = 4$). Unlike Figure 3(a), pixels of the producing stage $blurx$ are produced per line (yi) of the consuming stage $blury$ as needed. For example, three lines of width equal to Tx will be computed during the first yi iteration to produce one tile row, but only one line of $blurx$ will need to be computed for $yi > 0$, since two lines may be reused. Pixels that are no longer needed (cannot be reused across one iteration of xo) are discarded.

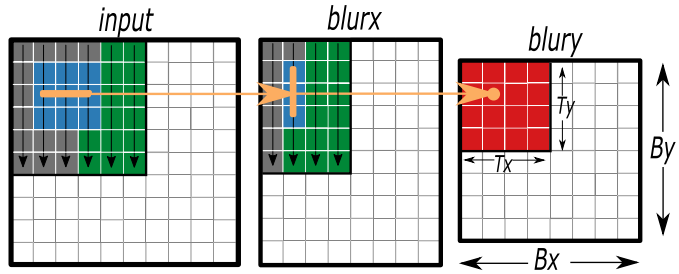
Note that the above schedule does not ensure maximum folding of the intermediate buffer allocated for $blurx$. For example, consider the schedule seen in Figure 4(a), interchanging the loop such that the ordering (from innermost to outermost) is (yi, xi, yo, xo), setting the compute level of $blurx$ to yi, and its storage to y would allow the buffer to get folded down to just the amount of overlap across y without any extra recomputation compared to the previous schedule:

$$B_{blurx} = (1 + v_y) \cdot sizeof(DataType) = 3 \cdot sizeof(DataType). \tag{12}$$

However, as it can also be seen from Figure 4(b), computing $blurx$ at the innermost level of its consumer causes the loading of the input buffer to be much less efficient. In detail, since $input$ is accessed in a column major order (three horizontal pixels at a time are needed to produce one pixel of $blurx$), prefetched (consecutive) cache lines will only be used after Ty iterations or will

```
for xo in [0,Bx/Tx):
  for yo in [0,By/Ty):
    for xi in [0, Tx):
      allocate blurx[3]
        for yi in [−2, Ty):
          blurx(...) = ...
        if (yi<0): continue
          blury(...) = ...
```

(a) maximum folding of the intermediate buffer



(b) Sliding windows inside tiles: values of $blurx$ are computed per pixel of the output tile as needed. Pixels that are no longer needed are discarded (gray area). The intermediate buffer requirement is equal to the blue area. Green pixels will get evaluated in future intra-tile iterations

Fig. 4. Maximum folding of the intermediate buffer $blurx$.

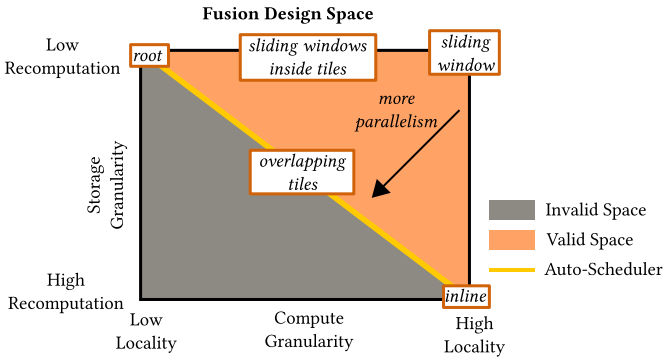


Fig. 5. Fusion solution space, adapted from Reference [21].

not even be used at all if Ty is too large, and they get evicted from the cache. As a result, the input load cost is now equal to:

$$Cl_{blurx}^{blury_{xi}^{yi}} = C_{input}^{blurx} = 3(T_y + 2)T_x \frac{B_x}{T_x}\frac{B_y}{T_y} = 3(T_y + 2)B_x \frac{B_y}{T_y}, \tag{13}$$

where the T_x factor can no longer be simplified, since accesses to *input* are not consecutive and the schedule does not benefit from hardware prefetching (as much as the previous one). For reference, the previous schedule (Figure 2(c)) performs twice as fast compared to this one, even though it does not maximize folding.

Based on the above (Equations (9) and (11) and Figure 3), we can conclude that folding the intermediate buffers leads to much smaller local memory requirements without sacrificing data reuse or increasing the amount of redundant computations. As a result, solutions that were previously not considered, e.g., tile sizes that led to large memory footprints can now easily be captured by separating the computation and storage of a stage. However, as seen by comparing Equations (13) and (6), maximum folding does not always ensure exploitation of the spatial locality or the hardware prefetching mechanisms of the platform. Due to this, a tradeoff analysis among reuse, recomputation, input loading cost, and memory requirements has to be conducted.

Figure 5 shows an abstract representation of the design space when considering stage fusion. As already mentioned, previous state-of-the-art techniques only consider solutions that reside within a small area of this space. The Halide Auto-Scheduler only produces solutions where the compute

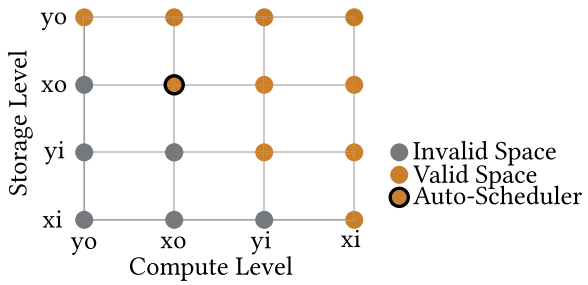


Fig. 6. Fusion solution space for the blur example.

granularity of a stage is the same as its storage granularity. As a result, the generated schedules are limited to fully inlined, fully stored, and tiled implementations with redundant computations where the computation and storage are set to the innermost inter-tile level (overlapping tiles). Figure 6 shows the distinct solutions for the above loop permutation of the blur example. The root and inlined solutions have been excluded due to limited reuse, parallelism, or locality as explained in the above example. We can notice that most solutions of the design space are currently not considered and all sliding window opportunities are missed.

Our method enables fast exploration of this new design space. We will show that through the use of heuristics, we can quickly prune the space down to a single solution (e.g., of the 10 valid schedules in Figure 6, we only need to evaluate 1). This is achieved by automatically eliminating most uninteresting schedules that are pareto dominated by other more efficient solutions. Dominant schedules are considered the following:

(1) Schedules that offer more reuse with the same buffer requirements, e.g., consider the schedules (yi, yi) and (yi, xo). The second schedule provides more reuse while the sliding window optimization allows for the same memory requirements.
(2) Schedules that offer the same amount of reuse with the smaller buffer requirements. e.g., schedules (xo, xo) and (yi, xo) as explained in the previous example.

The final solution is then evaluated through a cost function to pick the tile sizes. The analysis has to be repeated for different loop permutations, since that leads to a different design space with different solutions.

4 PROPOSED METHOD

In this section, we present each major step in our optimization flow. We follow a grouping-then-tiling technique that only attempts to split the pipeline into smaller segments if the initial solution does not fit within the memory constraints. More specifically, Section 4.1 discusses the algorithms responsible for choosing the compute and store level of a stage inside a pipeline (or a segment of a pipeline). Section 4.2 presents the tiling analysis that determines the proper tile sizes for a pipeline/segment. Section 4.3 demonstrates the procedure that is followed to split a pipeline into smaller segments. Some final optimizations are discussed in Section 4.4, and an overview of the optimization flow is given in Section 4.5

4.1 Fusion Strategy

This subsection introduces the analysis and heuristics that are used to determine which single point of the fusion space should be chosen for further evaluation. More specifically, this section

addresses the problem of choosing the computation and allocation level of each stage inside a pipeline (or a segment of it).

As already mentioned, our goal is to eliminate inefficient schedules without evaluating their costs. Algorithms 1 and 2 show the procedure that is followed to accomplish that. In detail, Algorithm 1 takes a pipeline (P) as an input that can be either the whole DAG of the initial pipeline or a sub-graph of it and identifies the compute and store level for each stage (K) in P. However, Algorithm 2 attempts to inline stages with trivial computational costs.

The pipeline can be described as a DAG of m connected nodes such that $P = \{K_0, K_1, \ldots, K_m\}$, where K_m is the output/final stage of P. Furthermore, to be able to describe all necessary dependencies between the nodes of the DAG, as well as the schedule of each stage, we perform the following definitions for all $i \leq m$:

- A linearly ordered set $D_i = \{xi_0, xi_1, \ldots, xi_{n1}, xo_0, xo_1, \ldots xo_{n2}\}$ that represents the tiled loop nest of K_i where the xi and xo are the intra and inter-tile loop indices, respectively.
- A list of tuples $W_i = \{Y_0, Y_1, \ldots, Y_l, \ldots, Y_t\}, 0 \leq t < m$, with $Y_l = \{K_l, I_l\}$, K_l the consuming stage and $I_l = \{E_0, E_1, \ldots, E_z\}$, z the number of unique indices in the loop nest of K_i, $E = (x, v), v \epsilon \mathbb{N}$, while $x \epsilon D_{K_l}$ is the dimension where the dependency exists and v the amount of overlap.
- A list of producers $L_i = \{K_0, K_1, \ldots, K_p\}, 0 \leq p < m$.
- A tuple $S_i = (x_{compute}, x_{store}), x \epsilon D_m$, which will partially define the final schedule and where $x_{compute}$ and x_{store} are indices of the output domain.

ALGORITHM 1: Stage Fusion Analysis

Input: $P, D_0, D_1, \ldots, D_m, W_0, W_1, \ldots, W_m$
Output: $S_0, S_1, \ldots, S_m$
$i \leftarrow m$
repeat
  if $W_i \neq \emptyset$ **then**
    $c_i \leftarrow max(\bigcup_{Y \epsilon W_i} select(\bigcup_{E \epsilon I} select(x, E), Y))$
    if $c_i \epsilon D_m$ **then**
      $s_i = next(D_m, c_i)$
      if $c_i = min(D_m)$ **or** $is\_reduction(c_i)$ **then**
      $c_i = next(D_m, c_i)$
      $S_i \leftarrow (c_i, s_i)$
    end
    else $S_i \leftarrow (min(D_m), min(D_m))$
  end
  else $S_i \leftarrow (inline, -)$
  $i \leftarrow i - 1$
until $i = 0$

ALGORITHM 2: Inline Trivial Stages

Input: $P, L_0, L_1, \ldots, L_m, S_0, S_1, \ldots, S_m$
Output: $S_0, S_1, \ldots, S_m$
$i \leftarrow m$
repeat
  if $S_i \neq (inline, -)$ **then**
    if $trivial(K_i) = True$ **then**
      for j *in* $0 \leq j \leq p_i$ **do**
        if $trivial(Kj) = False \ \ \& \ \ S_j = (inline, -)$ **then**
          $S_j \leftarrow S_i$
        end
      end
      $S_i \leftarrow (inline, -)$
    end
  end
  $i \leftarrow i - 1$
until $i=0$

Algorithms 1 and 2 determine the compute and store level of a stage inside a pipeline. More specifically, Algorithm 1 first checks whether a stage has overlap with any of its consumers (whether any of its values can be reused across iterations). If that is true, then the algorithm searches for the dependency index with the highest intra-tile order. If that index is also present in the loop nest of the output stage and is not the innermost one, then it is set as the compute level of the stage. Its store level is set to one level higher to benefit from sliding-window opportunities and ensure that all possible reuse is captured as explained in Section 3. While there might be cases

where maximum folding and therefore even smaller buffer requirements can only be obtained by either moving the store level higher or the compute level lower than what Algorithm 1 considers, the above heuristics allow us to quickly choose a single point in the design space while ensuring maximum reuse. Furthermore, if the chosen index corresponds to the innermost intra-tile loop of the loop nest or is a reduction dimension that offers full reuse, then the compute level is also set to one level higher. This decision is made to better exploit spatial locality and hardware prefetching in cases where the compute level is set to the column (as also explained in the motivational example) or vector index (which often corresponds to the innermost intra-tile loop) of a loop-nest and avoid redundant computation in reductions that have full overlap with their consumers. If, however, the chosen index is not found in the output loop nest, then it means that there is no direct overlap between the output domain and the stage that is being scheduled, but dependencies exist across intermediate stages. Its compute and store levels are therefore set to the innermost intra-tile loop of the output stage of that segment. The above method allows us to quickly choose a compute/store level for each stage of the group/segment. Furthermore, as explained in Section 3, moving the compute level even higher (to be the same as the store level) would lead to dominated solutions that require larger buffers only to achieve the same reuse. Finally, if a stage has zero overlap with its consumers, then its computation is inlined. We should note that all stages are scheduled with respect to the output stage of the pipeline. This eliminates any possibilities of nested loop fusion, which would add extra recomputation between the loops. We also introduce notation for two helper functions (*select* and *next*) where:

- *select* returns the first subset (or element) denoted by the first argument that belongs to the tuple (or pair) denoted by the second argument.
- *next* also takes two arguments and returns the element that belongs to the ordered set specified by the first argument and the position equal to the second argument plus one.

Algorithm 2 uses the partially defined output schedules of Algorithm 1 and attempts to inline the trivial stages of the pipeline. A stage is considered trivial only if its computational cost is equivalent to its load cost (similarly to the analysis followed by the Halide Auto-Scheduler) and only if all of its producers are non-inlined. After finding that a stage is trivial, the algorithm checks whether any of its direct producers that were previously inlined (due to zero overlap) may now have to be scheduled. In such a case, the compute and store levels of the newly found non-trivial stage is set to be the same as the ones of the now inlined stage.

4.2 Group Tiling

This section presents the analysis that chooses a proper tiling for a given pipeline/group. Algorithm 3 shows the procedure in detail.

The algorithm requires a pipeline (or segment) P as well as the linearly ordered set D_m as inputs. The latter represents the ordering of the tiled loop nest of the output stage and can initially be any (arbitrary) permutation of the loop nest as long as the intra-tile loops (xi) do not mix with the inter-tile ones (xo). The cost of evaluating each stage without any recomputation ($C_{Ki}^{root}, S_{Ki} = (root, root)$) is computed to be able to calculate the amount of recomputation for a given schedule. While this factor will be constant and will not alter the analysis within a group, it can affect the total cost of the pipeline when a different grouping is considered (and different stages have zero recomputation). As explained in the previous sections, the various discrete points in the fusion space depend on the loop permutation of the pipeline. As a result, the algorithm needs to try all possible intra and inter-tile loop permutations. Since the number of possible schedules explodes for large pipelines with multiple nested loops (such as convolutional neural networks), we do not attempt to interchange the kernels or other loops that only perform a few iterations. This decision

ALGORITHM 3: Tiling Analysis

Input: P, D_m
Output: $Tm_0, Tm_1, \ldots, Tm_{n1}, P_{fused}$
for *all i in* $0 \leq i < m$ **do** Evaluate C_{Ki}^{root}
repeat
  Perform Stage Fusion Analysis
  Inline trivial Stages
  repeat
    $C_{total} \leftarrow 0$
    for *all i in* $0 \leq i < m$ **do**
    $C_{total} += w \cdot Cl_{Ki}^{Km_{ci}^{si}} + Cp_{Ki}^{Km_{ci}^{si}} - C_{Ki}^{root}$
  until *all valid tile sizes evaluated*
until *all valid loop permutations evaluated*

Table 1. Notation

Tm_n	Tile Size in nth dimension
$Cl_{Ki}^{Km_{ci}^{si}}$	Load Cost of K_i
$Cp_{Ki}^{Km_{ci}^{si}}$	Compute Cost of K_i
C_{Ki}^{root}	Root Compute Cost of K_i
C_{total}	Total Cost of all stages
Bm_n	Problem Size in nth dimension
w	Relative cost of load operation

allows us to easily eliminate the loop overhead in many cases by unrolling those loops. For each possible loop permutation, we perform the fusion analysis described in Algorithm 1 and then attempt to inline any trivial stages (Algorithm 2). At this point we can evaluate all relevant costs presented in Section 3 for each stage of the pipeline individually, for an arbitrary tiling dimension. We iterate over all possible tile sizes that fit into specific constraints:

- The tile size of the innermost intra-tile dimension (which is not part of the kernel) has to be a multiple of the cache-line size, as well as a multiple of the native vector width.
- The tile size of the outermost inter-tile dimension has to fulfill:

$$\frac{Bm_{no}}{Tm_{no}} \geq N_{threads}. \tag{14}$$

- The tile size in a dimension where a dependency exists has to be at least as large as the amount of maximum overlap in that dimension such that, if x is the dimension of interest then:

$$Tm_x \geq max\left(\bigcup_{Y \epsilon W_i} select\left(\bigcup_{E \epsilon I} select(v, E), Y\right)\right). \tag{15}$$

In detail the first constraint is imposed to maintain vectorization in conjunction with spatial locality across cache lines. The second constraint ensures that the final schedule will have enough parallelism to utilize the multi-threaded aspects of the target architecture. The third constraint avoids invalid tile sizes that would lead to redundant computations without extra buffer benefits (since due to the inter-stage dependencies the memory allocation would be at least equal to the amount of overlap anyway). Finally, we do not consider tiles where the total footprint of stages without folded storage (compute and store level are the same) does not fit into the L2 cache. This constraint ensures that values that will be immediately consumed and cannot be reused in future iterations stay local. We calculate the costs defined in Section 3 for each stage individually using a weighted cost function and sum them together to compute the total cost of the pipeline. Our cost function uses the load cost of a stage ($Cl_{Ki}^{Km_{ci}^{si}}$) multiplied by the relevant overhead of a load operation compared to a computation (w) plus the amount of recomputation of that stage ($Cp_{Ki}^{Km_{ci}^{si}}$ - C_{Ki}^{root}). The combination of loop permutation, fusion choice, and tile size ($Tm_0, Tm_1, \ldots, Tm_{n1}$) that minimizes the total cost (C_{total}) of the pipeline is chosen as the final schedule.

ALGORITHM 4: Group Stages

Input: P, D_m
Output: $H_0, H_1, \ldots, H_w$
$n_P \leftarrow 0$
for *all* i *in* $0 \leq i < m$ **do**
  if $size(W_i) > 1$ **then**
    $S_i = \{root, root\}$
    $H_{n_P} \leftarrow$
    $\{K_{0i}, \ldots, K_{ti}, K_i\}$, n_P++
    $P \leftarrow erase(\{K_{0i}, \ldots, K_{ti}, K_i\})$
  end
end
$H_n \leftarrow P$
for j *in* $0 \leq j \leq n_P$ **do**
  $SplitSegment(H_j)$
end

ALGORITHM 5: Split Segments

Input: H
Output: $H_0, H_1, \ldots, H_w$
$TilingAnalysis(H)$
if $wset_H > csize$ **then**
  $n_H \leftarrow max\_split$
  repeat
    $S_{n_H} = \{root, root\}$
    $H_n \leftarrow \{K_0, \ldots, K_n\}$, n_H++
    $H \leftarrow erase(\{K_0, \ldots, K_n\})$
    $TilingAnalysis(H_n)$
    if $wset_{H_n} < csize$ **then**
      $SplitSegment(H)$
    end
  until $wset_{H_n} > csize$
end

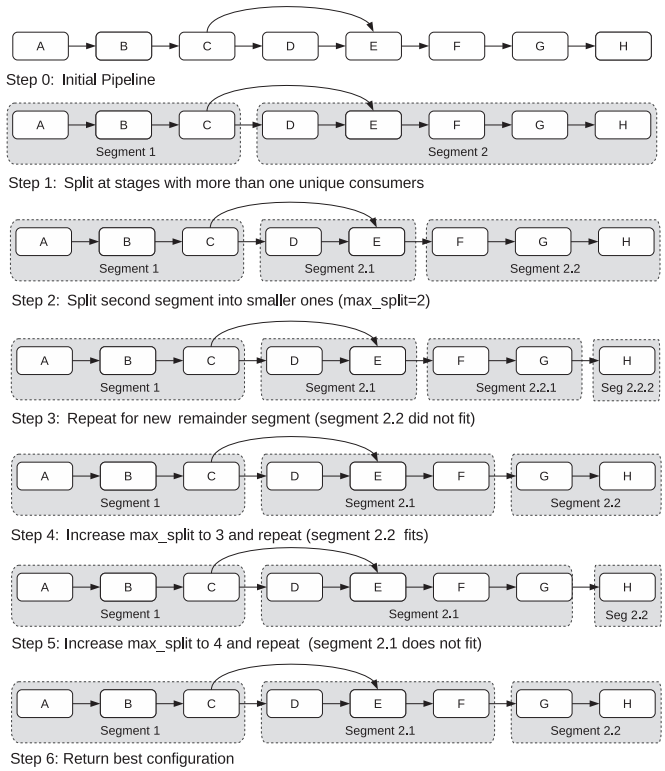


Fig. 7. Pipeline segmentation strategy.

4.3 Stage Grouping

If the memory footprint of the final schedule is larger than the size of the last-level-cache, then the pipeline is split into segments, and each segment is scheduled independently of the others. Given the fact that current multi-core architectures contain caches of many MBs in size that will likely fit many stages, our strategy attempts to reduce design time by only attempting to split the pipeline if the initial solution (where all stages are either fused or inlined into the output stage) does not fit into the cache. The memory footprint of the pipeline is equal to the amount of memory required/allocated for all intermediate stages of the pipeline (or segment). Data from intermediate stages will either be stored to be reused in future intra-tile iterations (in circular/folded buffers) or will immediately be consumed in the current intra-tile iteration and are not needed afterward. Buffers in the former category are folded down to the maximum amount of overlap (only in the dimension specified by the compute level of the stage), and their total size needs to fit into the last-level cache for future use, while buffers that will not get folded need to fit inside the L2 cache (such that their data stay local between production/consumption). All buffers are calculated based on the areas required (allocated) by the compiler for a given schedule.

Algorithms 4 and 5 show the steps that are followed to split the pipeline P into non-overlapping segments $(H_0, H_1, \ldots, H_w)$. Algorithm 4 takes the initial pipeline as an input and first checks whether any stages have more than one consumers. In that case, these stages form a new pipeline, along with their producers, and are erased from the initial DAG. This is done to limit the design space and enable faster optimization runtime. While as a result we may end up end up with multiple smaller segments in some pipelines, we did not notice any significant performance degradation due to this fact. Further investigation of performance benefits that may be captured by merging those smaller segments is left as future work. During the next step, we attempt to schedule all new pipelines, along with the remainder of the previous step (remaining stages of the original pipeline). Algorithm 5 checks whether the footprint of the new segments is still larger than the available cache size and then recursively splits those into smaller segments using the following process. Starting from the nth stage of the pipeline, where n is set to max_split (an integer value that controls the minimum size of a segment), we schedule all stages up to the nth. If the segment fits, then we attempt to schedule the remaining stages by recursively repeating the same algorithm. After having evaluated all possible configurations for a specific n, we increase it by 1, and the process is repeated until the working set of the segment no longer fits. This ensures that we skip configurations with invalid segment sizes. Each (unique) valid solution generated by Algorithm 5 where all stages of the original pipeline (P) have been successfully scheduled is cached, and the sum of all independent sub-pipelines' cost is evaluated. The configuration that results in the minimum (summed) cost is chosen as the final solution. We should also note that if the initial value of max_split is set to 1, then the algorithm will evaluate all valid grouping configurations for the pipeline.

Evaluating all possible configurations may require an extensive amount of time for larger pipelines (such as deep neural networks). Our method reduces the runtime of the grouping process by eliminating non-interesting segmentations. This is achieved in two ways:

- Upon identifying that the memory footprint of a segment is larger than the available cache size, we do not attempt to fuse more stages into the same segment. This choice can be explained as follows: A segment with a memory requirement larger than the available cache size will only grow larger if more stages are included into it, especially if the newly included stage has extra dependencies.
- We do not attempt to split the final segment of a pipeline into smaller ones, since that would only add external load costs from the previous root stages to the subsequent consuming ones. This choice allows us to significantly reduce the time needed to find the final configuration, especially in the context of modern multiprocessor architectures with large cache sizes.

The steps followed for an example pipeline can be seen in Figure 7.

4.4 Final Optimizations

Upon finding the final configuration of a pipeline, we have groups of stages with a specified tiling and loop permutation per group. We vectorize the innermost intra-tile loop of a group that is not part of a reduction (or a kernel) and parallelize its outermost inter-tile loop among the platform's threads/cores as explained in Section 4.2 (Equation (14)). However, we have not yet considered any changes in the permutation of individual stages within a stage. We optimize the loop nest of each producing stage within a segment through loop interchange that improves reuse distance by reordering loop indices with minimum strides to be innermost. Moreover, the loop that corresponds to the compute level of a stage is always set as outermost, since that loop will always iterate once

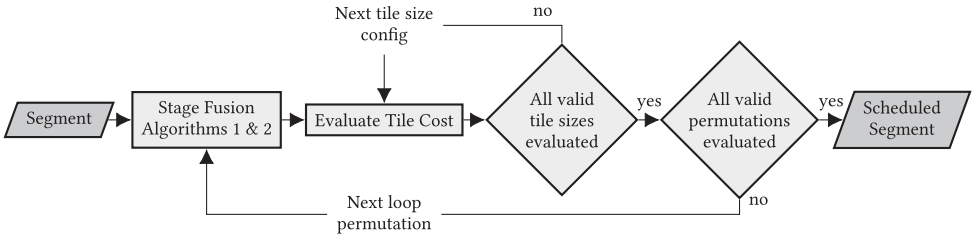


Fig. 8. Optimization flow for a pipeline segment—Algorithm 3.

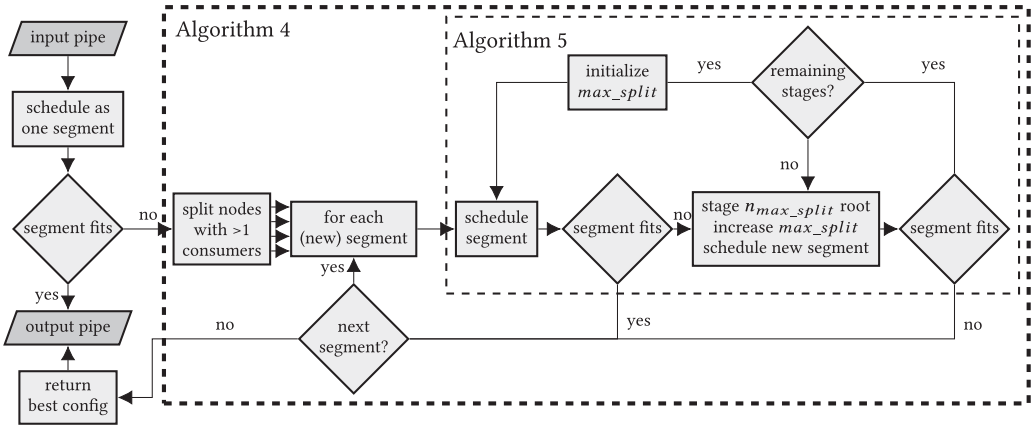


Fig. 9. Optimization flow for an arbitrary pipeline.

(or once plus the equivalent overlap with its consumer) and would add extra loop overhead in any other position.

4.5 Optimization Flow Overview

Figure 9 shows the optimization flow for an input pipeline along with all iteration steps involved, while Figure 8 shows the steps followed to schedule each segment (or the initial whole pipeline if it fits in the cache). In detail, for all valid permutations of the tiled loop nest, Algorithm 3 calls Algorithms 1 and 2 to determine the compute/store levels of each stage. It then evaluates the total cost of the pipeline for all valid tile sizes and the combination of D_m, T_m (loop permutation and tile sizes respectively) that minimizes C_{total} is chosen. If the memory footprint of the final schedule is larger than the constraints imposed by the last-level cache of the target system, then Algorithms 4 and 5 are used to split the pipeline into smaller segments, with Algorithm 3 (and subsequently Algorithms 1 and 2) used again to schedule each new segment. Every valid configuration (where all stages of the original pipeline have been successfully scheduled) is cached to be evaluated at the end of the process. The segmentation/configuration that minimizes the total cost of the original pipeline is chosen as the final, now scheduled pipeline.

5 EXPERIMENTAL RESULTS

This section demonstrates the results obtained across a wide variety of image-processing applications on three different architectures.

Table 2. Platform Features

Platform	LLC size (MB)	L2 cache size (KB)	$N_{threads}$
Intel i7-6700	8	256	8
Intel i7-5930K	12	256	12
ARM Cortex A15	2	512	4

5.1 Experimental Setup

The architectural details of each platform used in the experiments are listed in Table 2. Table 3 provides a description of each benchmark along with the problem size considered as well as the optimization runtime. Most of the descriptions were found in Reference [15]. The chosen benchmarks include image-processing pipelines used in Reference [15], the pyramid blending algorithm used in Reference [10], as well as a popular recent Deep-Neural-Network used for single image super-resolution (VDSR) that was introduced in Reference [12]. All problem sizes are chosen to be the same as the ones found either on the official Halide repository on GitHub [7] or as the ones used in Reference [10]. The optimization flow of most benchmarks is automated, with the exception of three pipelines (that are marked with an asterisk). These are only partially automated due to complex reductions and inter-stage dependencies such as extensive helper-function usage (argmin, maximum, lerp, etc.) in these applications. As a result, some steps of the algorithms defined in Section 4 had to be manually implemented for these pipelines. However, such dependencies can easily be derived by the Halide compiler and therefore will easily be captured when our framework has access to all information available to the compiler.

In the following graphs, the manual implementations refer to the manual schedules found in the Halide repository (the only exceptions being the pyramid benchmark, the manual schedule of which was found in Reference [10], as well as the VDSR network that we implemented in Halide). The PolyMage-A and PolyMage-DP implementations refer to the results replicated using the artifacts and instructions provided by the authors of References [10] and [20]. However, implementations were provided for only six benchmarks, which are also the ones considered in Reference [10].

We compare our results to the equivalent ones produced by the other methods: Since all of our applications are implemented in Halide, we can use the Halide Auto-Scheduler to produce schedules for all benchmarks. Each benchmark is executed 100 times, and the average execution time per run is measured. This process is repeated multiple times per benchmark and the minimum average among those is used as the final average execution time. Furthermore, we properly adjust the optimizations parameters of both the Auto-Scheduler and PolyMage before our experiments for the solutions to be tuned to the target platforms. Since PolyMage cannot explicitly vectorize loops (unlike Halide), the performance of the PolyMage implementations is highly influenced by the efficiency of the auto-vectorizer of the back-end compiler [10]. Finally, the problem size used in Reference [10] for the harris and unsharp benchmarks differs from the one in the Halide repository. We therefore repeat the experiments for this problem size as well and the results of this comparison can be seen in Table 4. Halide was built using llvm 4.0.0, while the PolyMage implementations were compiled using icpc on the i7-6700 platform and gcc on the i7-5930K and ARM platforms.

At this point, it is important to emphasize that, as seen in Figures 8 and 9, all of the proposed algorithms are tightly coupled. As explained in the motivational example ofs Section 3, sliding windows and circular buffers allow for tile sizes that would otherwise be impossible to consider (e.g., large tile strips that otherwise would never fit into the local buffer constraints imposed by the cache size). As a result, evaluating each algorithm independently is not possible; they should all be considered together.

Table 3. List of Benchmarks

Benchmark	Description	Optimization runtime
blur 2 stages 6,400 × 4,800	Simple two-pass 3 × 3 blur filter	3ms
bilateral 5 stages 2,560 × 1,536 × 3	Fast bilateral filter using the bilateral grid [3]. Constructs the grid using a histogram reduction, followed by stencil and sampling operations.	4ms
unsharp 6 stages 2,560 × 1,536 × 3	Enhances local contrast by smoothing an image with a small support Gaussian and subtracting it from the original to isolate the high-frequency content, which is then combined with the original image.	3ms
harris 13 stages 1,920 × 1,024 × 3	Implementation of the popular harris corner detection algorithm [8] that combines multiple stencils and point-wise operations.	5ms
camera 30 stages 2,560 × 1,936 × 3	The Frankencamera pipeline for processing raw data from an image sensor into a color image [1]. The pipeline performs hot-pixel suppression, demosaicing, color correction, gamma correction, and contrast.	5ms
interpolate 52 stages 1,536 × 2,560 × 3	Interpolation of image pixel values using an image pyramid for seamless compositing, based on the newest healing brush in Photoshop. Pyramid construction deals with image data at multiple resolutions and creates chains of stages with complex dependencies	152ms
laplacian 99 stages 1,536 × 2,560 × 3	A local Laplacian filter: an edge-aware, multi-scale approach for enhancing local contrast [18]. The pipeline builds multiple image pyramids with complex dependencies and performs data-dependent sampling.	912ms
lensblur 74 stages 1,536 × 2,560 × 3	Given a rectified stereo pair of images, produces a synthetic shallow-depth-of-field image. It first solves for depth by constructing and filtering a cost volume [22] using a convolution pyramid [6], then renders the synthetically defocused image by randomly sampling the source image over a virtual aperture.	(617ms)*
nlmeans 13 stages 614 × 1,024 × 3	Fast non-local means image denoising using the method of [5]. Computes a 7×7 image blur with weights determined by 7 × 7 patch similarity	(4ms)*
maxfilter 9 stages 1,920 × 1,024 × 3	Computes the maximum-brightness pixel within a circular region around each target pixel. Uses a precomputed table of differently-sized vertical max filters to reduce complexity from O(radius2) per output pixel to O(radius).	(4ms)*
pyramid 52 stages 1,920 × 1,024 × 3	Pyramid blending that blends two input images into one using a mask and a Laplacian pyramid of 4 levels.	116ms
VDSR 24 stages 256 × 256 × 64	VDSR [12] is an end-to-end network with 20 convolutional layers for single image super-resolution.	32s

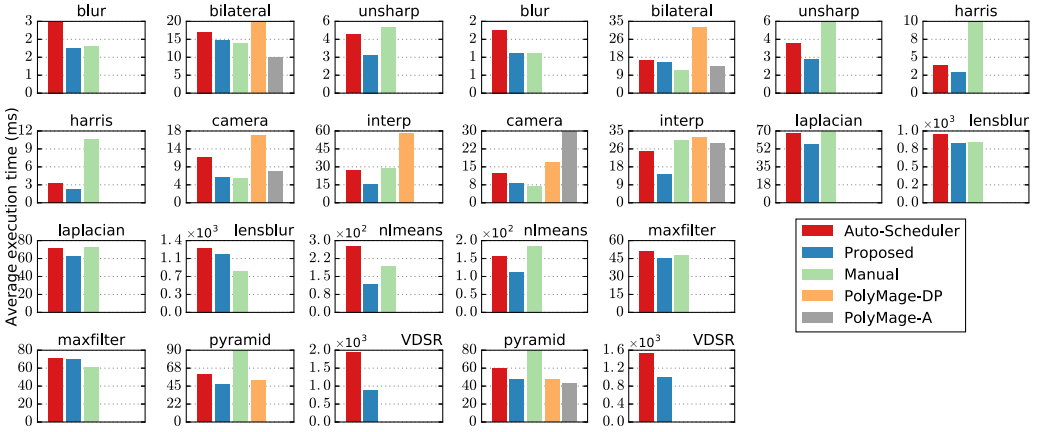
Table extended from Reference [15].

5.2 Performance Results

Figure 10 shows the average execution time (in ms) for each benchmark on the two Intel platforms listed in Table 2. The results for the harris and unsharp benchmarks on the problem size of the PolyMage implementations can be seen in Table 4.

Table 4. Average Execution Time (ms) for Harris and Unsharp Benchmarks—Problem Size 4,256×2,832

Method	Intel i7-6700		Intel i7-5930K		ARM Cortex A15	
	Harris	Unsharp	Harris	Unsharp	Harris	Unsharp
PolyMage-A	45	27	27	27	377	254
PolyMage-DP	15	21	21	21	304	388
Auto-Scheduler	16	22	14	22	164	206
Proposed	11	20	10	20	171	189



(a) Intel i7-6700 (b) Intel i7-5830K

Fig. 10. Performance results on the two Intel platforms.

Our schedules outperform the Auto-Scheduler solutions in almost all cases, with the Laplacian benchmark being the only exception on the Intel i7-5930K, where the difference in execution time is still within ≈2%. We noticed that while the initial Auto-Scheduler paper optimizes for L2 cache size, the currently used and updated one is targeting the shared last-level cache. We conducted multiple experiments for both choices and noticed that while some benchmarks experience a slight performance improvement when the memory footprint constraint is set to the size of L2, other ones suffer a dramatic performance degradation. As a result, we choose to use the currently advised method of optimizing for last-level cache in the results. Finally, our schedules are also comparable or even better than the manual ones in many cases.

PolyMage-DP performance is similar to the Auto-Scheduler in almost all cases. The constant updates and focus on the Auto-Scheduler by the Halide community may explain the difference in the results presented here and the ones in Reference [10] between the two methods. The efficiency of auto-tuned PolyMage-A solutions vary per benchmark and platform: The raw camera and bilateral grid implementations of the auto-tuned solutions on the intel i7-6700 are close to (or slightly better than) the manual Halide schedules. However, they are much less efficient compared to the other implementations of the harris filter on both Intel platforms.

The results for the ARM Cortex platform can be seen in Figure 11 and Table 4, where a similar pattern can be discerned. Our schedules outperform both the manual and auto-scheduled ones with the largest differences observed in the interp, laplacian, and VDSR benchmarks. The performance of the PolyMage solutions varies per benchmark. For example, it performs significantly worse than the Halide solutions on the camera pipeline, slightly better than both the Auto-Scheduler and the

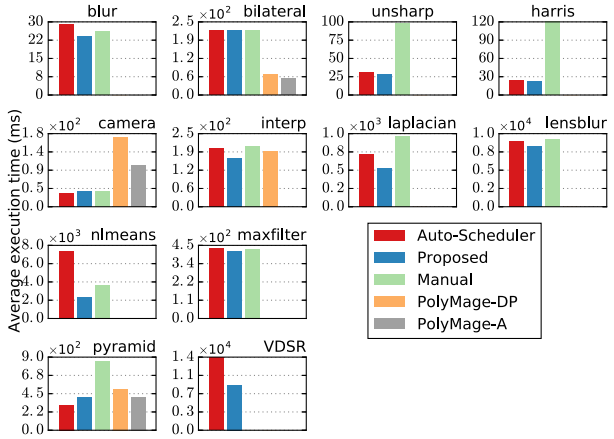


Fig. 11. Performance results on the ARM platform.

manual Halide schedule on the interp benchmark, and much faster than all Halide solutions in the bilateral pipeline. The main reasons behind this result are the differences in the functional description of the pipeline between the Halide and PolyMage implementations: Halide uses a built-in linear interpolation function that performs more complex computations than the PolyMage implementation of it. Upon forcing Halide to use a simpler approach, performance was improved by up to 40% in all three cases (Auto-Scheduler, Manual, and Proposed). Furthermore, Halide requires all expressions used as indices in functions/stages to be bounded and therefore performs extra clamping in two stages for the compiler to be able and derive the bounds of the equivalent producers. These extra computations are the main bottlenecks in the performance of the Halide bilateral pipeline on the ARM platform. However, finding an efficient description is outside the scope of this work.

Finally, to test the efficiency of our grouping strategy (Algorithms 4 and 5), we repeat the experiments for the VDSR network and investigate the performance for various problem sizes. We choose VDSR, since it consists of sequential stages where each subsequent stage consumes the output of the previous one (except for the input image that is consumed twice). This benchmark is therefore a good candidate for such an experiment, since various problem sizes will lead to different tiling choices and therefore different memory footprints, which, due to a constant memory constraint will require new segmentations. The results of this experiment on the Intel i7-6700 platform for five different dimensions of the output image are presented in Figure 12. Our schedules perform more than 2× better than the equivalent Auto-Scheduled solutions for large problem sizes.

To demonstrate the robustness of our method, the same experiment was conducted on another platform (with an Intel i7-6560U processor) once with the hardware prefetcher enabled and once with the hardware prefetcher disabled. The experiments followed a similar trend as in Figure 12 when comparing the two implementations. Furthermore, the performance degradation when the hardware prefetcher was disabled in our solutions was close to 20% while for the Auto-Scheduler solutions it was more than 2×. Upon further investigation, we noticed that the loop permutation chosen by the Auto-Scheduler (which attempts to reorder loops based on their stride, i.e., placing the loop with the smallest stride innermost) interleaves the column, row, and kernel dimensions, limiting the amount of spatial reuse that can be captured in the process. This incurs a high penalty when the hardware prefetchers are disabled. However, our proposed method does not reorder loops with low iterations (similar to the 3×3 convolution kernels) and only attempts to exploit prefetching when determining the tile size dimensions (Algorithm 3). Setting the kernel inner to

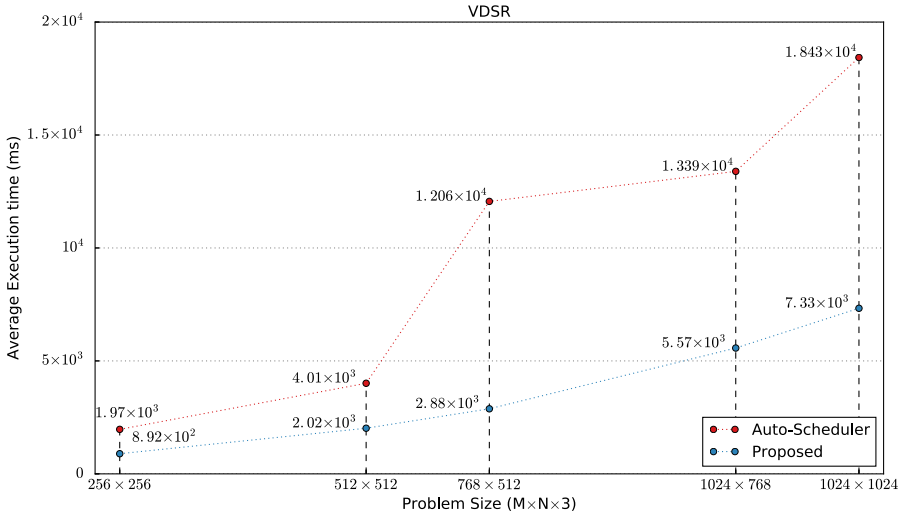


Fig. 12. VDSR performance on the Intel i7-6700 for various problem sizes.

the column and row dimensions allows data to stay in the local caches (or even registers) before they are reused. As a result, self-spatial reuse can still be exploited across kernel iterations, and this explains why our schedules do not suffer as much when hardware prefetchers are disabled.

Finally, based on the above results (Figures 10–12), we can observe that larger pipelines with multiple stages such as the interp, laplacian, and VDSR benchmarks benefit the most from our schedules, where sliding window opportunities are easily captured, buffers are folded down to smaller memory footprints, and new tiling opportunities are considered. Moreover, even in cases where the pipeline does not offer such opportunities (e.g., bilateral, nlmeans), our solutions remain similar to (or in many cases even better than) both the Auto-Scheduler, and the manual solutions.

6 CONCLUSIONS

In this work, we present a novel platform-aware algorithm for the optimization of image-processing pipelines running on multi-core CPU-based architectures. We show that our method captures solutions of the design space that were not covered in previous state-of-the-art techniques by effectively considering combinations of loop tiling, interchange, and stage fusion with independent computation and allocation per stage. Our model takes into account multiple architecture-specific parameters such as multithreading, vectorization, and hardware prefetching. We evaluate our proposed method across a variety of image-processing applications implemented in the Halide DSL and compiler and compare it to both previous state-of-the-art techniques that target the Halide and PolyMage DSLs, as well as manually optimized Halide solutions. Experimental results show significant average performance improvements compared to previous related work as well as the manually optimized implementations of Halide pipelines.

REFERENCES

[1] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. 2010. The Frankencamera: An experimental platform for computational photography. In *ACM SIGGRAPH 2010 Papers (SIGGRAPH'10)*. ACM, New York, NY, Article 29, 12 pages. DOI: https://doi.org/10.1145/1833349.1778766

[2] 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, New York, NY, 303–316. DOI: https://doi.org/10.1145/2628071.2628092

[3] Jiawen Chen, Sylvain Paris, and Frédo Durand. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph.* 26 (2007), 103.

[4] D. Cociorva, J. W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. 2001. Loop optimization for a class of memory-constrained computations. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*. ACM, New York, NY, 103–113. DOI: https://doi.org/10.1145/377792.377814

[5] J. Darbon, A. Cunha, T. F. Chan, S. Osher, and G. J. Jensen. 2008. Fast nonlocal filtering applied to electron cryomicroscopy. In *Proceedings of the 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 1331–1334. DOI: https://doi.org/10.1109/ISBI.2008.4541250

[6] Zeev Farbman, Raanan Fattal, and Dani Lischinski. 2011. Convolution pyramids. In *Proceedings of the 2011 SIGGRAPH Asia Conference (SA'11)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2024156.2024209

[7] Halide. 2018. Halide GitHub Repository (MIT License). Retrieved from https://github.com/halide/Halide (commit 402171e7a4dfacb0bd93297cbdfb600a325fe745).

[8] Chris Harris and Mike Stephens. 1988. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*. 147–151.

[9] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 311–320. DOI: https://doi.org/10.1145/2304576.2304619

[10] Abhinav Jangda and Uday Bondhugula. 2018. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, NY, 261–275. DOI: https://doi.org/10.1145/3178487.3178507

[11] Ken Kennedy and Kathryn S. McKinley. 1994. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer, Berlin, 301–320.

[12] J. Kim, J. K. Lee, and K. M. Lee. 2016. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 1646–1654. DOI: https://doi.org/10.1109/CVPR.2016.182

[13] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2007. Effective automatic parallelization of stencil computations. *SIGPLAN Not.* 42, 6 (Jun. 2007), 235–244. DOI: https://doi.org/10.1145/1273442.1250761

[14] Naraig Manjikian and Tarek S. Abdelrahman. 1997. Fusion of loops for parallelism and locality. *IEEE Trans. Parallel Distrib. Syst.* 8, 2 (Feb. 1997), 193–209. DOI: https://doi.org/10.1109/71.577265

[15] 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, Article 83 (Jul. 2016), 11 pages. DOI: https://doi.org/10.1145/2897824.2925952

[16] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 429–443. DOI: https://doi.org/10.1145/2694344.2694364

[17] Catherine Olschanowsky, Michelle Mills Strout, Stephen Guzik, John Loffeld, and Jeffrey Hittinger. 2014. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Los Alamitos, CA, 793–804. DOI: https://doi.org/10.1109/SC.2014.70

[18] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. 2011. Local Laplacian filters: Edge-aware image processing with a Laplacian pyramid. In *ACM SIGGRAPH 2011 Papers (SIGGRAPH'11)*. ACM, New York, NY, Article 68, 12 pages. DOI: https://doi.org/10.1145/1964921.1964963

[19] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part Ii, multidimensional time. *SIGPLAN Not.* 43, 6 (Jun. 2008), 90–100. DOI: https://doi.org/10.1145/1379022.1375594

[20] PolyMage project. 2016. PolyMage Repository (Apache 2.0 License 2016). Retrieved from https://bitbucket.org/udayb/polymage (commit 0ff0b46456605a5579db09c6ef98cb247dd2131d).

[21] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 519–530. DOI: https://doi.org/10.1145/2491956.2462176

[22] C. Rhemann, A. Hosni, M. Bleyer, C. Rother, and M. Gelautz. 2011. Fast cost-volume filtering for visual correspondence and beyond. In *Proceedings of the Annual Conference on Computer Vision and Pattern Recognition (CVPR'11)*. 3017–3024. DOI: https://doi.org/10.1109/CVPR.2011.5995372

[23] 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'18)*. ACM, New York, NY, 254–264. DOI: https://doi.org/10.1145/3168823

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

[25] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI'91)*. ACM, New York, NY, 30–44. DOI: https://doi.org/10.1145/113445.113449

[26] Jingling Xue. 2005. Aggressive loop fusion for improving locality and parallelism. In *Proceedings of the 3rd International Conference on Parallel and Distributed Processing and Applications (ISPA'05)*. Springer-Verlag, Berlin, 224–238. DOI: https://doi.org/10.1007/11576235_28

[27] Qing Yi and Ken Kennedy. 2004. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *Int. J. High Perf. Comput. Appl.* 18, 2 (2004), 237–253. DOI: https://doi.org/10.1177/1094342004038956

[28] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical overlapped tiling. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 207–218. DOI: https://doi.org/10.1145/2259016.2259044

Received July 2018; revised December 2018; accepted January 2019