Flextended Tiles: a Flexible Extension of Overlapped Tiles for Polyhedral Compilation

JIE ZHAO, INRIA & DI, École Normale Supérieure, France
ALBERT COHEN, Google, France

Loop tiling to exploit data locality and parallelism plays an essential role in a variety of general-purpose and domain-specific compilers. Affine transformations in polyhedral frameworks implement classical forms of rectangular and parallelogram tiling, but these lead to pipelined start with rather inefficient wavefront parallelism. Multiple extensions to polyhedral compilers evaluated sophisticated shapes such as trapezoid or diamond tiles, enabling concurrent start along the axes of the iteration space; yet these resort to custom schedulers and code generators insufficiently integrated within the general framework. One of these modified shapes referred to as overlapped tiling also lacks a unifying framework to reason about its composition with affine transformations; this prevents its application in general-purpose loop-nest optimizers and the fair comparison with other techniques. We revisit overlapped tiling, recasting it as an affine transformation on schedule trees composable with any affine scheduling algorithm. We demonstrate how to derive tighter tile shapes with less redundant computations. Our method models the traditional "scalene trapezoid" shapes as well as novel "right-rectangle" variants. It goes beyond the state of the art by avoiding the restriction to a domain-specific language or introducing post-pass rescheduling and custom code generation. We conduct experiments on the PolyMage benchmarks and iterated stencils, validating the effectiveness and applicability of our technique on both general-purpose multicores and GPU accelerators.

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

Additional Key Words and Phrases: polyhedral compilation, automatic parallelization, stencil computations, loop tiling

ACM Reference Format:
Jie Zhao and Albert Cohen. 2019. Flextended Tiles: a Flexible Extension of Overlapped Tiles for Polyhedral Compilation. *ACM Trans. Arch. Code Optim.* 0, 0, Article 0 (January 2019), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There has been a long thread of research and development tackling data locality and parallelism in computationally intensive stencil applications. Multiple loop tiling strategies have been proposed, starting from simple rectangular and parallelogram shapes [7, 35], further specialized into more complex shapes like overlapped [10, 37], split [21, 32], diamond [6], and hexagonal tiles [14]. The polyhedral framework has been brought to the forefront for its ability to analyze and optimize general-purpose loop nests. Its main scheduling and code generation algorithm remain limited to classical tile shapes however, leading to load-imbalanced wavefront inter-tile parallelism, while custom solutions with more complex tile shapes exist to exploit inter-tile parallelism along the axes

Authors' addresses: Jie Zhao, INRIA & DI, École Normale Supérieure, 45 rue d'Ulm, Paris, 75005, France, jie.zhao@inria.fr; Albert Cohen, Google, 8 rue de Londres, Paris, 75009, France, albertcohen@google.com.

XXXX-XXXX/2019/1-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of the iteration domain while improving data locality. Some of these custom solutions specialize on time-iterated stencils with uniform dependence vectors, such as solvers for the heat equation or more complex iterated stencil benchmarks such GemsFDTD in SPEC CPU2006 [21]. They may not directly apply to more irregular stencil patterns occuring in image processing pipelines, where the dependence vector varies from one stage to the next [6, 14]. The latter constitute a class of computations arising in computer vision and computational photography. Image processing pipelines can be specified as a directed acyclic graph of filtering stages and edges representing dependences between stages. The stages in a pipeline usually exhibit abundant data parallelism but locality optimization across stages is needed to achieve high performance, making manually optimizing such applications a difficult task.

PolyMage [24] is the state-of-the-art polyhedral compilation framework automatically generating high-performance schedules for such image processing pipelines, benefiting from the full inter-tile parallelism enabled by overlapped tiling [21]. It takes as input a domain-specific language (DSL) inspired by Halide [31], computes schedules competitive with manually-written ones for image processing pipelines, and generates high-performance imperative code. The PolyMage framework implements overlapped tiling by looking for bounding hyperplanes of all stages in a fused group of stencil operations in the pipeline; the slope of these bounding hyperplanes has to be gentle enough to encompass all dependence vectors in the fused group, leading to a wider trapezoid than the tighter shape one may manually expect.

This work aims at enhancing the performance of both image processing pipelines and iterated stencils. It does so by optimizing overlapped tile shapes, embedding overlapped tiling in a general-purpose polyhedral compilation framework to handle stencil computations homogeneously with other patterns of numerical computation, and taking advantage of this embedding to jointly optimize the affine schedule with the tiling parameters.

We leverage a well-defined general-purpose intermediate representation called *schedule trees* [16], an affine schedule and abstract loop nest hybrid implemented with systems of affine inequalities, offering all the expressiveness of traditional affine schedules while facilitating program transformations such as hierarchical strip-mining and tiling, inserting data transfers for software-controlled memories, and annotating dimensions for parallel execution or GPU acceleration [35, 38]. We implement overlapped tiling on top of schedule trees, and expose it as a source-to-source polyhedral compiler. This compiler processes image processing pipelines and time-iterated stencils written in a general-purpose language (C). Our work is neither restricted to a domain-specific language nor does it introduce domain-specific polyhedral rescheduling and code generation. Aside from the conceptual and software engineering benefits, this allows us to construct tighter tile shapes than the state of the art, minimizing the memory footprint of overlapped tiles, and improving performance.

To construct a tighter overlapped shape, we first let a general polyhedral framework perform rectangular/parallelogram tiling and then expand the bounding faces of a tile by taking into consideration the constraints caused by inter-tile dependences, followed by expressing these dependences with specific nodes of the schedule tree representation of a general polyhedral framework, and finally applying a classical algorithm for automatic code generation with minor adaptation for overlapped tiling. Our technique also goes beyond the state of the art by generating code for both CPU and GPU architectures, and by considering new and alternative overlapping tile shapes. We validate its general applicability by conducting experiments on image processing pipelines as well as time-iterated stencils, providing a comparison between overlapped tiling and other state-of-the-art techniques.

The paper is organized as follows. The next section presents the technical background and further explains our motivation. Section 3 describes the construction of two overlapped tile shapes and

introduces complementary optimizations. Section 4 presents experimental results on both image processing pipelines and iterated stencils on different architectures, followed by a discussion of related work in Section 5 and our conclusions in Section 6.

2 BACKGROUND AND MOTIVATION

The polyhedral model is a powerful mathematical abstraction of loop nests. It defines iteration domains consisting of all statement instances, access relations mapping statement instances to the memory locations they access, schedules defining partial or total execution order on statement instances, and dependences capturing producer-consumer relations between statement instances. A polyhedral compiler represents these sets and maps as systems of affine inequalities, before constructing an affine schedule respecting all dependences carried by statement instances. Loop tiling is often implemented as a post-pass affine transformation to exploit data locality and parallelism; it embeds the computation into a higher dimensional space of tile and point dimensions, but is generally limited to classical, rectangular or parallelogram tile shapes.

2.1 Trapezoid Tiles

Such decoupling of loop tiling from affine scheduling may complicate tile-level concurrent start, leading parallelization algorithms to sacrifice tile-level parallelism or to introduce load imbalance by implementing skewed schedules with pipelined startup and drain [6]. An alternative way allowing full inter-tile parallelism involves a tighter coupling of loop tiling and affine scheduling; diamond tiling was introduced into the Pluto compiler for this purpose [6]. Tile-level concurrent start along a face of the iteration space is possible if there are no inter-tile dependences parallel to this face; diamond tiling thus forces such a face to be evicted or linearly independent with the candidate linear forms composing the multi-dimensional schedule. The schedule found through this modified Pluto algorithm may actually degrade some of the locality and fine-grained parallelism objectives compared to the standard algorithm; this implies a different schedule may be needed for intra-tile parallelism and/or vectorization, complicating the process and follow-up code generation.

An alternative approach to eliminate pipelined startup and drain is to modify the tile shape [21]. As an illustrative example, Figure 2 shows the comparison between different tile shapes for the time-iterated stencil code in Figure 1. *Overlapped* tiling is constructed by adding an additional "shaded" region to the parallelogram tile resulting from a classical affine scheduler. This results in a trapezoid shape holding all the iterations needed to compute the parallelogram tile, breaking all dependences to any neighboring tile at the same time step or pipeline stage. Breaking such inter-tile dependences exposes inter-tile parallelism. The caveat is that the shaded regions in consecutive tiles overlap and have to be recomputed. *Split tiling* can be seen as a refinement of overlapped tiling, obtained by splitting overlapped tiles into two kinds of sub-tiles, one being the shaded region and the other consisting of all the remaining iterations. The two sub-tiles cannot be executed concurrently, but split tiling exposes inter-tile parallelism among sub-tiles of the same kind.

```
for (t=0; t<T; t++)
  for (i=1; i<N-1; i++)
    A[t+1][i] = 0.25 * (A[t][i+1] + 2.0 * A[t][i] + A[t][i-1]);
```

Fig. 1. An iterated stencil from *heat-1d*

Image processing pipelines are an important class of computations arising in computer vision, computational photography, medical imaging, etc. The PolyMage framework implements an overlapped tiling technique for exploiting data locality for this class of computations, generating high-performance imperative code competitive to those written by experts. Figure 3 shows a simple

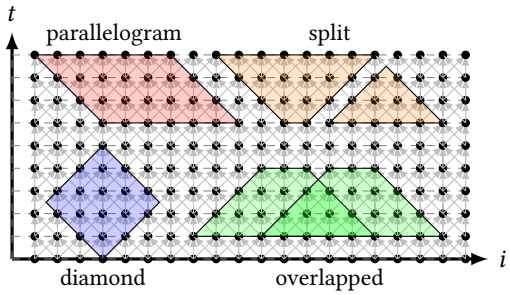

Fig. 2. Comparison of different tile shapes

image processing pipeline, and the overlapped tiles constructed by PolyMage on this example is shown in Figure 4 with the solid bounds between two tiles representing the bounding faces.

Among the tile shapes depicted in Figure 2, overlapped tiling is often selected for optimizing image processing applications. The reason is as follows. Parallelogram tiling can improve locality but fails to exploit tile-level parallelism. In other words, parallelogram tiling may only allow for wavefront parallelism since one parallelogram tile depends on the one at its left. Split tiling also exhibits some tile-level dependences, effectively sequencing the execution of successive trapezoids and inverted trapezoids composing a given group of fused stages. This sequence is associated with synchronization overhead, and even worse, one trapezoid tile consumes the values produced by its horizontal neighbor tiles, forcing such values to be kept live for multiple stages and missing the opportunity for aggressive storage optimizations. Overlapped tiling does not induce such intra-fused-group synchronization and facilitates storage mapping optimization and promotion to software-managed memories, although it does it in exchange for redundant computations.

```
for (i=1; i<N; i++)
  A[i] = f(i);
for(i=2; i<N-1; i++)
  B[i] = 0.25 * (A[i-1] + 2 * A[i] + A[i+1]);
for(i=4; i<N-3; i++)
  C[i] = 0.25 * (B[i-2] + 2 * B[i] + B[i+2]);
```

Fig. 3. A simple image processing pipeline

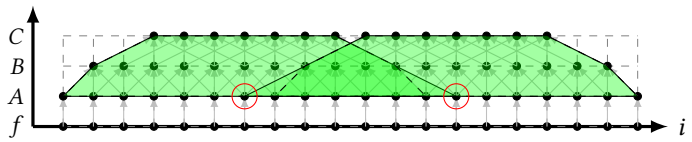

Fig. 4. Shaded regions of scalene trapezoid tiling

The PolyMage framework constructs schedules for overlapped tiling as follows. The shape of an overlapped tile is determined by checking the dependence vector of each level of the pipeline. To preserve all these producer-consumer relations, PolyMage may have to extend some of the slopes constructed according to these dependence vectors, enforcing a unique bounding face on each side of an overlapped tile. In this example, the (dashed) slopes caused by dependences between stages A and B are extended, resulting in a looser shaded region between tiles and greater memory footprint than expected.

To describe how overlapped tiling may be integrated into a general purpose polyhedral flow, we first need to introduce the schedule tree representation underlying our approach.

2.2 Schedule Trees

Affine schedules in the polyhedral model are used to define the execution order of programs, including both the original and those generated by scheduling algorithms like Pluto or its variants. An affine schedule often encodes a tree structure, implicitly. Making this structure explicit has the same expressiveness as the classical affine multidimensional schedules, but simplifying the composition with non-polyhedral transformations and the attachment of dimension-specific properties (e.g., levels of parallelism on the target hardware).

In a schedule tree, a statement instance is expressed by a named multi-dimensional vector with the name identifying the statement and the coordinates corresponding to iteration variables of the enclosing loops. The collection of all statement instances, i.e., the iteration domain, is expressed using Presburger formulas [29], retained in a *domain node*. For example, the iteration domain of the code shown in Figure 3 can be expressed as $\{S_A(i) : 1 \leq i < N; S_B(i) : 2 \leq i < N-1; S_C(i) : 4 \leq i < N-3\}$ with loop bounds expressed explicitly. A statement instance is also mapped to a multi-dimensional logical execution date [12] for defining its lexicographic execution order. Such mapping is referred to as a schedule, expressed by a piecewise multi-dimensional quasi-affine function over the iteration domain and contained in a *band node*. The band node concept is derived from tilable bands in the Pluto framework [7], defining permutability and/or parallelism properties on a group of statements. Rectangular tiling regardless of the correctness of the example in Figure 3 is written as $[\{S_A(i) \rightarrow (i/T); S_B(i) \rightarrow (i/T); S_C(i) \rightarrow (i/T)\}; \{S_A(i) \rightarrow (i); S_B(i) \rightarrow (i); S_C(i) \rightarrow (i)\}]$ with the former piece representing tile loops (iterating among tiles) and the latter for point loops (iterating within tiles).

A *filter node* selects a subset of statement instances introduced by an outer domain/filter node. Filter nodes usually appear as children of a *sequence/set node* expressing a given/arbitrary order on its children. As an illustrative example, Figure 5 is the original schedule tree of the example shown in Figure 3, indicating each filter node selects a subset (a stage) of the domain node. The sequence node defines the three stages should be executed in order, followed by a loop iterating over the iteration space of each stage.

There have been many other node types existing in schedule trees [16], among which we only discuss the *expansion node* in this paper. An expansion node can expand a statement instance to one or more instances, constituting a new set of statement instances to be scheduled by the schedule tree. Typical loop tiling would partition the iteration space into smaller blocks, each of which is disjoint with each other, making it difficult to construct an overlapped tile without an expansion node. With an expansion node, we are free to choose one statement instance of a stage in a tile and expand it to as many instances as we expect. These expanded statement instances would join with those of neighboring tiles, resulting in overlapped tiles over the whole iteration space.

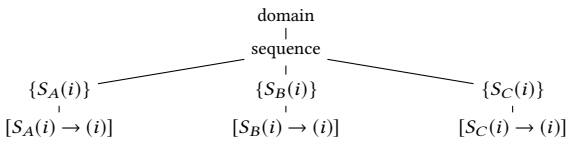

Fig. 5. The original schedule tree of the code in Figure 3

2.3 Overview of Overlapped Tiling on Schedule Trees

We propose an overlapped tiling technique eliminating the redundant (circled) iterations in shaded regions. To construct a tighter overlapped tile, we first fuse the stages in this example and let a general polyhedral framework perform rectangular tiling on the iteration space regardless of the correctness. We then expand the bounding faces of a tile by taking into consideration the inter-tile dependences, without further extending the slopes between stages A and B. This can be implemented as a transformation of the schedule tree intermediate representation [16] in a general polyhedral framework. In particular, one may construct a bounding face like the dashed slopes shown in Figure 4, eliminating the redundant iterations from shaded regions. Such overlapped shape is referred to as a *scalene trapezoid tile*.

One may also construct an overlapped tile by first applying the scheduling algorithm of a polyhedral compiler, transforming the iteration space into the form shown in Figure 6. In this case, a PolyMage-like technique may construct a wide overlapped tile, illustrated by the solid slopes on the figure. But we can still achieve a tighter shape by (1) resorting to a scheduler to shift the iteration space, (2) performing rectangular tiling and (3) extending the left bounding face of a tile by operating on the schedule tree representation, still eliminating the redundant iterations from shaded regions. This shape is referred to as *right-rectangle trapezoid*, or simply *rectangle trapezoid tile*.

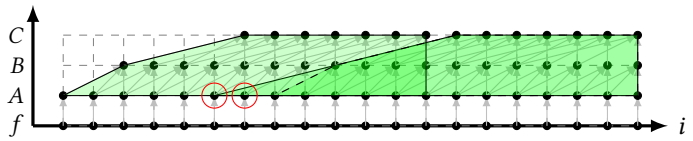

Fig. 6. Shaded regions of rectangle trapezoid tiling

3 OVERLAPPED TILING

Overlapped tiling is an efficient tiling technique allowing tile-level concurrent start. As we described above, by embedding overlapped tiling within a general-purpose affine transformation, one may choose to obtain a scalene or a rectangle trapezoid tile by fusing or shifting first, with both achievable by operating on the schedule tree representation.

3.1 Scalene Trapezoid Tiling

Let us focus on one-dimensional stencils for the sake of clarity. We may first consider implementing overlapped tiling by expanding both sides of a tile. Figure 7 shows the result of rectangular tiling on the iteration space of the example listed in Figure 3 regardless of the correctness. This rectangular tile can be obtained by first strip-mining [3] each stage and then fusing the tile loops of each stage. For the sake of simplicity, we call the tile on the left side the L-tile, and the tile on the right side the R-tile.

We may first handle the right tile boundary of the L-tile. According to the dependence vector between stages C and B, the last two iterations of stage C executed by the L-tile depend on the first two iterations of stage B executed by the R-tile. As a result, we need to expand the iterations of stage B executed by the L-tile to a set consisting of all the original iterations of the L-tile plus the first two iterations executed by the R-tile. This can be achieved by introducing an expansion node below the schedule of stage B.

In the same way, the last iteration of stage B executed by the L-tile depends on the first point of stage A executed by the R-tile. Note that the last iteration of stage B executed by the L-tile should be the second iteration executed by the R-tile due to the introduction of the expansion node in the schedule tree of stage B. The shifting of the last iteration in a tile induced by an expansion node is called *dependence propagation*. The images of the expansion node introduced to the schedule of stage A should therefore include all the iterations of stage A covered by the L-tile plus the first three iterations of the R-tile. Finally, we obtain an overlapped tile for the L-tile satisfying all dependences.

We can expand the left tile boundary of the R-tile for overlapped tiling in a similar way. The images of the expansion node of stage B consist of the original iterations plus the last two iterations of the L-tile, and these images of stage A should be extended with the last three iterations of the L-tile due to dependence propagation. Figure 8 shows the scalene trapezoid tiling generated by expanding both boundaries of a tile. For each tile other than the first and the last, one may have to expand both the left and right boundaries.

More generally, we insert expansion nodes into the schedule tree in a top-down manner. This is due to dependence propagation: the output stage referencing the live-out sets of a stage group (probably after fusion) should be considered first, followed by the stage it depends upon, and so on.

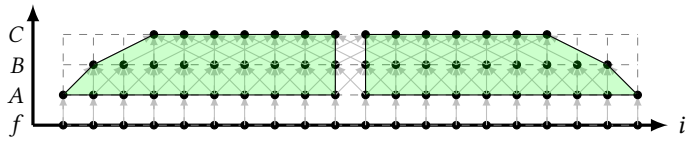

Fig. 7. Rectangular tiling regardless of correctness

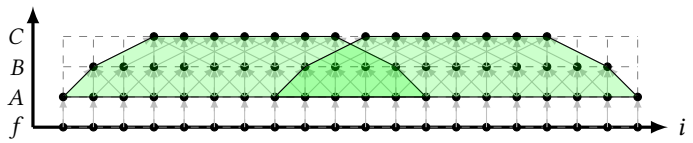

Fig. 8. Scalene trapezoid tiling

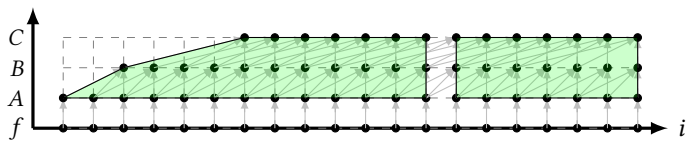

Fig. 9. Rectangular tiling after scheduling

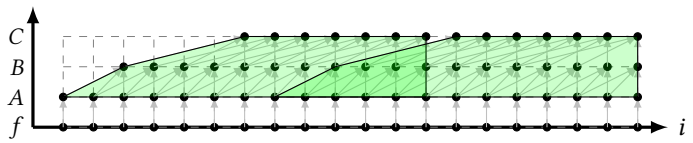

Fig. 10. Rectangle trapezoid tiling

3.2 Rectangle Trapezoid Tiling

Decoupling tiling from affine scheduling is the usual path taken by tools aiming at a greater coverage of benchmarks—such as PPCG or Pluto. A decoupled algorithm would generate a new schedule by shifting the iteration domain of stages B and C, followed by rectangular tiling on the transformed iteration space, as shown in Figure 9.

Again, the L-tile represents the tile on the left and the R-tile the one on the right. Unlike PolyMage, we only need to expand the left boundary of a tile. Considering the R-tile, the first iteration of stage C depends on two iterations (the last but one and last but three) of stage B in the L-tile, and the second iteration of stage C depends on another two iterations (the last and last but two) of stage B. The left boundary of the R-tile on stage B should be expanded to the left by four iterations. We then turn to stage B. Due to the dependence vector between stages B and A, the images of the expansion nodes introduced to the schedule of stage A should be expanded to left by two iterations, eventually expanding to left by a total of six iterations due to dependence propagation from the above level. Finally, we obtain a rectangle trapezoid tile as shown in Figure 10.

3.3 Schedule Generation

We can now generate schedules for both scalene and rectangle trapezoid tiling. Considering scalene trapezoid tiling, strip-mining is performed on each stage, followed by fusing the tile loops iterating among tiles. Correctness is enforced by introducing expansion nodes below the point loops of stages A and B iterating within a tile, with boundaries updated as explained in the last subsection. We finally get a schedule tree for scalene trapezoid tiling as shown in Figure 11.

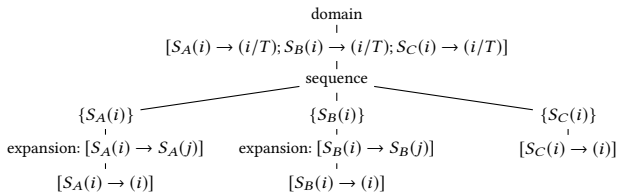

Fig. 11. Schedule tree of scalene trapezoid tiling

A scheduling algorithm may shift the iteration space of the pipeline, yielding a shifted schedule that is amenable to tiling across pipeline stages. We may then separate the point dimension from the band, in order to introduce expansion nodes to stages A and B, as shown on the generated schedule tree in Figure 12.

The code generator of a polyhedral compilation framework can take these schedule trees for code generation, with overlapped tiling enabled in the generated code.

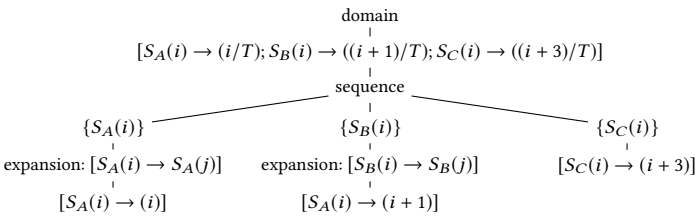

Fig. 12. Schedule tree of rectangle trapezoid tiling

3.4 Removal of Control Overhead

Let us still consider stage B in the code in Figure 7. An expansion node in schedule trees is used to map each iteration of a domain/filter node to one or more images, forming a wider set of iterations scheduled by the domain/filter node. Expanding the point loop may convert a rectangular/parallelogram tile obtained from existing affine schedulers into an overlapped tile, avoiding a separate rescheduling step. We may therefore insert an expansion node below a filter node representing the point loop as shown in Figure 11 and 12. An expansion node here is represented as a map expanding the original domain $S_B(i)$ to its image set $S_B(j)$.

Being introduced for iterating on the image of the expansion node, j is an unbounded parameter in the schedule tree, producing an unbounded domain for the sub-tree of stage B. One may take into consideration the (upper and lower) bounds of j by adding the original bounds on all iterations of stage B, guaranteeing the images of the expansion node would not exceed the original bounds. We can write it formally as

$$lb \leq j \leq ub \tag{1}$$

where lb and ub represent the lower and upper bounds.

This condition alone cannot produce an overlapped tile, as the boundaries of a tile remain unchanged. Let T be the tile size. $\lfloor i/T \rfloor$ denotes the tile iteration. The original iterations executed by the $(\lfloor i/T \rfloor + 1)$-th tile ($\lfloor i/T \rfloor$ is 0 for the first tile) can be expressed as

$$T \times \lfloor i/T \rfloor \leq j \leq T \times (\lfloor i/T \rfloor + 1) - 1 \tag{2}$$

To expand the boundaries of a rectangular/parallelogram tile as explained in previous subsections, we can change it into

$$T \times \lfloor i/T \rfloor - \ell \leq j \leq T \times (\lfloor i/T \rfloor + 1) - 1 + r \tag{3}$$

where ℓ represents the number of expanded iterations introduced by expansion nodes to the left boundary, and r to the right boundary. In practice, ℓ or r may be 0, e.g., rectangle trapezoid tiling.

Expanding the first point of a stage in a tile is straightforward, simplifying schedule transformations when changing a rectangular/parallelogram tile into an overlapped one. We find, however, that the selection of the point from which the images of an expansion node should be generated may have heavy impact on the control overhead. One may obtain a bounded map representing an expansion node through the conjunction of constraints (1) and (3), selecting iteration $T + 2$ in the L-tile and iteration T in the R-tile for expansion, isolating a partial tile (L-tile) from full tiles (the R-tile and all the remaining ones if they exist) and introducing more control overhead.[1] Such an isolation scheme may induce performance degradation when there are many more stages in a group.

A better solution to remove control overhead consists in integrating partial tiles with full tiles. Each starting point of stage B executed by a full tile (other than the first) can be expressed as $T \times \lfloor i/T \rfloor$, while $T \times \lfloor i/T \rfloor + 2$ ($T \times \lfloor i/T \rfloor$ is equal to 0) being used for the partial (the first) tile. One is free to choose any iteration of a stage covered by a tile for generating the image of an expansion node, implying that one may choose any iteration other than $T \times \lfloor i/T \rfloor$ in a full tile. We thus choose iteration $T \times \lfloor i/T \rfloor + 2$ as the starting point for full tiles, enforcing uniformity with the partial tile and removing control overhead.

Finally, one may write another condition constraining the selection of starting point of full tiles as

$$i = T \times \lfloor i/T \rfloor + s \tag{4}$$

[1] A full tile is completely contained in the iteration space while a partial one is not but has a non-empty intersection with the iteration space [20].

where s represents the shifted offset of the starting point in each tile due to the bounds on the whole domain. We may finally obtain an expansion node by constraining the map of an expansion node with a set of conditions consisting of (1), (3) and (4) as follows.

$$\{S_B(i) \rightarrow S_B(j) : (1) \wedge (3) \wedge (4)\} \tag{5}$$

Similarly, we may also obtain the map of expansion nodes like (5) and the schedule tree as shown in Figure 12, with r in (3) being equal to 0, removing control overhead for rectangle trapezoid tiling.

3.5 Comparing the Two Trapezoid Tile Shapes

Given a schedule tree with expansion nodes written as (5), one may obtain a scalene trapezoid tile by first fusing the stages of an image processing pipeline or a rectangle trapezoid tile by first shifting the iteration space. One may distinguish the difference between the two trapezoid tile shapes by comparing Figure 8 and Figure 10.

Apart from the shape, scalene trapezoid tiling and rectangle trapezoid tiling also differ with respect to data locality. Considering the example in Figure 3 for the sake of simplicity, let us compare the code associated with each tile shape in Figures 13 and 14.

In Figure 13, the point loops of individual stages are distributed. In other words, the individual stages in a given tile are executed one after another in a scalene trapezoid. On the contrary, the point loops of these stages are fused in Figure 14, minimizing the intra-tile producer-consumer distance. Maximizing intra-tile locality also distinguishes our technique from PolyMage in cases of uniform dependence slopes along "time" dimension.

```
for (c0=0; c0<N/T+1; c0++) {
  for (c1=max(T*c0-3, 1); c1<min(T*(c0+1)+3, N); c1++)
    A[c1] = f(c1);
  for (c1=max(T*c0-2, 2); c1<min(T*(c0+1)+2, N-1); c1++)
    B[c1-1] = 0.25 * (A[c1-1] + A[c1] + A[c1+1]);
  for (c1=max(T*c0, 4); c1<min(T*(c0+1), N-3); c1++)
    C[c1-1] = 0.25 * (B[c1-2] + A[c1] + A[c1+2]);
}
```

Fig. 13. Code of a scalene trapezoid tile

```
for (c0=0; c0<N/T+1; c0++)
  for (c1=max(T*c0-6, 1); c1<min(T*(c0+1), N); c1++) {
    A[c1] = f(c1);
    if (c1>=3) {
      B[c1-1] = 0.25 * (A[c1-2] + 2*A[c1-1] + A[c1]);
      if (c1>=7)
        C[c1-3] = 0.25 * (B[c1-5] + B[c1-3] + B[c1-1]);
    }
  }
```

Fig. 14. Code of a rectangle trapezoid tile

The rectangle trapezoid tile should be preferred as it holds a better intra-tile data locality than the scalene one. However, such data locality has very little impact on the performance of image processing pipelines since the tile height is usually not large enough to trigger any benefit. Iterated stencils may exploit such data locality when the tile size along the time dimension is large enough, as we will see in the experimental evaluation.

3.6 Handling Multi-statement/-dimensional Cases

The ability to handle multiple statements is important in practice, as we will show in the evaluation. Consider the schedule tree example in Figure 11 and let the filter node of stage B consist of two statements, S_{B_1} and S_{B_2}. For the sake of simplicity, we do not show the context in the expansion node of stage A.

Similar to the single statement case, we schedule S_{B_1} and S_{B_2} together, as if they formed a macro statement S_B, fusing the tile loop with those of stages A and C. This results in a schedule tree similar to Figure 11. An expansion node is allowed to insert right underneath the filter node consisting of S_{B_1} and S_{B_2}—regardless of the transformation correctness—followed by a band node representing the map of their point loops and its child sequence node defining their execution order. We finally obtain a schedule tree as shown in Figure 15. Rectangle trapezoid tiling can be handled in the same way.

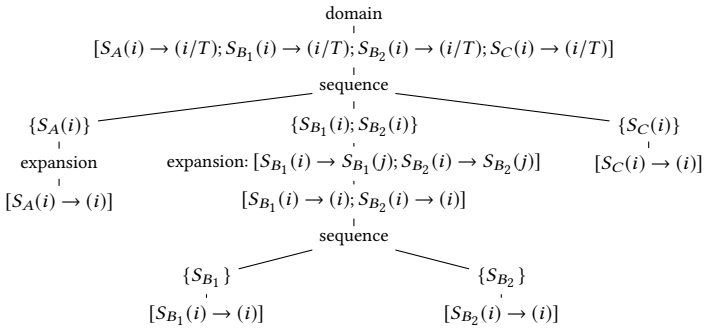

Fig. 15. Schedule tree of multiple statements

When considering multi-dimensional/nested fusion, we may transform the code a single dimension at a time without impacting other dimensions. As a result, the process for a multi-dimensional statement reduces to iteratively invoking the one-dimensional scheme, considering both scalene and rectangle trapezoid options at each dimension.

3.7 Other Transformations

Following the experience of the domain-specific code generator PolyMage, overlapped tiling alone does not generate competitive code for image processing pipelines: it needs to be composed with additional transformations and lowering steps. In addition, we also cover a wider range of applications, including iterated stencil codes with more than 2 spatial dimentions, and stencils of multiple statements. We survey the complementary transformations essential to optimize these more general computations and to lower them to efficient target-specific implementations.

Alignment and Scaling. Overlapped tiling applies to constant dependence vectors. When instructions have mismatching dimension or non-uniform dependence patterns, it may be possible to realign them via loop scaling. This is something the PolyMage framework does, leveraging domain-specific information about image processing pipelines. Alignment of non-uniform dependences into constant vectors can be achieved by up-sampling and down-sampling the stages involved, resulting in appropriately scaled schedules by multiplying the affine function for individual instructions by the appropriate image size ratio.

Fusion. Loop fusion is an important transformation implemented by polyhedral compilers for exploiting data locality. Benefiting from alignment and scaling, some stages of the pipeline may be fused together, creating opportunities for exploiting overlapped tiling across more stages.

One can make full use of the fusion heuristic adopted by a polyhedral compiler by setting compilation options, but it may not be good enough for image processing pipelines even with the aggressive fusion heuristic. The criteria the PolyMage code generator provides is fusing a successor stage with its only child when it has only one child by viewing the pipeline as a directed acyclic graph consisting of nodes representing stages and edges denoting dependences between stages, followed by iterative attempts for fusing opportunities until no fusion can be found. We reproduced this heuristic in our evaluation.

Reducing the Memory Footprint. Loop fusion transforms the pipeline into several groups, each of which consists of a set of intermediate stages and an output stage, requiring storage allocation optimization for improving performance.

Values produced by intermediate stages are only used within a tile: they can be discarded when they are not live after the computation of the tile. These intermediate values can therefore

be allocated in small scratchpad memory rather than full buffers, leading to better locality and improving the performance when integrating with overlapped tiling and the transformations mentioned above. Indexing expressions generated for such scratchpads can be determined according to the conditions defined in expansion nodes.

Hybrid Tiling. When targeting on multi-dimensional iterated stencils, we are allowed to restrict overlapped tiling only to the time dimension and a subset of space dimensions, leveraging existing polyhedral frameworks to perform rectangular/parallelogram tiling on the remaining space dimensions, just like diamond tiling [6] and hybrid hexagonal/classical tiling [14]. This lower dimensional overlapped tiling may change the tile shapes for a multi-dimensional case, as we shown in Figure 16 comparing the difference between a full and partial dimensional overlapped tile shapes for $2d$ stencil code. A full dimensional overlapped tiling would form a base of a pyramid, extending along both dimensions of the space, while a partial one only overlaps along one dimension.

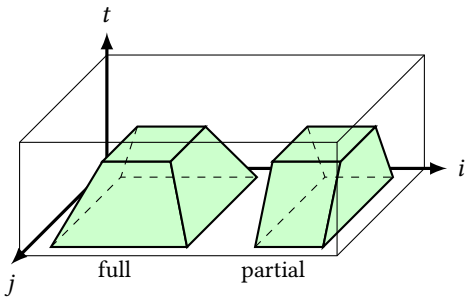

Fig. 16. Full and partial dimensional overlapped tiling

4 EXPERIMENTAL RESULTS

We conduct experiments on the PolyMage benchmarks and representative time-iterated stencils. The image processing pipelines covered by the PolyMage benchmarks are extracted from the Halide benchmarks, varying widely in structure and complexity.

4.1 Experimental Setup

We implement both trapezoid tiling techniques as well as the complementary transformations explained in subsection 3.7 in PPCG, a polyhedral compiler that exploits parallelism and data locality of programs automatically. All transformations are applied automatically by default when passing *--scalene* and *--rectangle* flags to PPCG for switching between scalene and rectangle trapezoid tiling. The PPCG version we use is `ppcg-0.07-26-g236d559`. It can take C programs as input, automatically generating OpenMP code on general-purpose multicores and CUDA code on heterogeneous accelerators.

The experiments are conducted on a 32-core, dual-socket workstation with an NVIDIA Quadro P6000 GPU. Each CPU is a 2.10GHz 16-core Intel Xeon(R) E5-2683 v4. We use the icc compiler (`18.0.1`) from Intel Parallel Studio XE 2018, with the flags *-fast -qopenmp*. CUDA code is compiled with the NVIDIA CUDA (`9.1.95`) toolkit with the *-O3* optimization flag. Each benchmark is executed 11 times, of which the first run is discarded and the average of the remaining is recorded.

4.2 Image Processing Pipelines

The PolyMage framework takes a DSL as input and generates both naïve and optimized OpenMP codes. A naïve version is generated by PolyMage without schedule transformations and overlapped

tiling, of which the sequential code is used as the baseline and also as the input of PPCG. The image processing benchmarks used in our experiments are listed in Table 1, together with the number of stages and the actual execution times of each baseline. Some of these benchmarks involve non-stencil computation patterns, such as reductions and non-affine accesses in histograms. This will help illustrating the benefits of embedding overlapped tiling in a general-purpose framework, not restricted to stencil computations. One may obtain the execution time of each case by multiplying the factors shown in Figure 17-23 and 25.

Table 1. Summary of the PolyMage Benchmark

Benchmark	Stages	CPU execution time (ms)			Speedup over PolyMage	GPU execution time (ms)		Speedup over PPCG
		naïve (1 core)	PolyMage (32 cores)	Our work (32 cores)	(32 cores)	PPCG	Our work	
Bilateral Grid (BG)	7	66.01	5.57	5.41	1.03	4.67	4.25	1.10
Camera Pipeline (CP)	32	116.32	5.95	5.87	1.01	4.29	2.18	1.97
Harris Corner Detection (HC)	11	246.88	5.10	5.10	1.00	1.89	1.07	1.77
Local Laplacian Filter (LF)	99	480.48	35.35	27.08	1.31	16.73	11.12	1.50
Multiscale Interpolation (MI)	49	209.10	20.07	16.44	1.22	12.86	7.76	1.66
Pyramid Blending (PB)	44	350.49	17.18	15.41	1.11	8.33	5.78	1.44
Unsharp Mask (UM)	4	142.16	5.01	3.68	1.36	2.17	1.29	1.68

We compare the performance of our generated code with both the naïve and optimized version generated by PolyMage. Table 1 also shows the speedup of our code over the optimized version of PolyMage; this comparison isolates the effect of the tile shape, the other code generation parameters being identical. Harris Corner Detection is interesting, as our technique generates the exact same tile shape as PolyMage in this case, and also yields the same performance; conversely, when tile shapes differ—all other benchmarks—significant speedup can be observed. This hints at the distinctive benefits of tightening the tile shape, independently of other possible transformation and code generation effects. To further validate this analysis, we evaluated the overlapped tile shape of PolyMage in our own PPCG-based flow for two additional benchmarks, Unsharp Mask and Local Laplacian Filter, and we could check that the generated code was identical to that of PolyMage up to insignificant syntactic variations. This allows us to conclude that the speedups are indeed caused by the tighter tile shapes enabled by our technique, rather than to unrelated transformation decisions or code generation improvements.

We also show the performance of the Halide manual schedule written by experts,[2] as well as the results of Halide's automatic scheduling algorithm inspired from PolyMage [23]. We use OpenCV version 2.4.9.1. Scalene trapezoid tiling is used in our experiment, as the data locality benefits of rectangle trapezoid tiling does not apply to this case, as explained earlier.

Finally, as our technique may generate tighter tile shapes, we compare the memory footprint with PolyMage in Tables 3 through 7. These numbers highlight the memory footprint benefits of our technique, in addition to mitigating the redundant computation overhead.

PolyMage resorts to auto-tuning for tile size selection. It considers 7 possible tile sizes (8, 16, 32, 64, 128, 256 and 512) for each dimension. We consider these auto-tuned tile sizes for a fair comparison and to guarantee that the performance improvement is due to tighter tile shapes. We refer to these auto-tuned tile sizes as standard sizes. They are shown in Table 2. In addition, we also consider tile sizes other than powers of 2. Searching all such sizes may be an impossible task in practice. Instead, leveraging the auto-tuning selection among all possible 49 candidates, we focus on such tile sizes close to the best sizes chosen by PolyMage. Let 2^m denote the size of the first

[2]Version: git commit 8c23a1970faba9b06bf7145d2653618fb978479e.

dimension and 2^n the size of the second dimension of a $2^m \times 2^n$ tile selected by the PolyMage auto-tuner; m should be less than n to mitigate the volume of redundant computations. Besides, the size for the second dimension should be a multiple of target machine vector width for vectorizing innermost loop, and the size of the second dimension should be slightly greater than the standard power of two in an attempt to maximize temporal locality. We therefore consider all possible tile sizes ranging from 2^{m-1} to 2^m for the first dimension and all values expressed as $2^n + 16k$ $(0 \leq k \leq 3)$ for the second dimension; the best sizes are also shown in Table 2, referred to as non-trivial sizes. Non-trivial tile sizes usually perform better than those sizes chosen by the auto-tuning strategy used in PolyMage; this is primarily due to higher cache hierarchy usage.

Table 2. Tile Sizes of the PolyMage Benchmarks

Benchmarks	BG	CP	HC	LF	MI	PB	UM
Standard sizes	8×128	64×256	32×256	8×256	32×128	16×256	8×512
Non-trivial sizes	6×144	32×288	24×288	6×272	24×160	12×272	4×544

Bilateral Grid. Bilateral Grid [9, 28] is a data structure enabling fast edge-aware image processing, localizing operations involved including bilateral filtering, edge-aware painting, local histogram equalization, etc. This benchmark smooths images while preserving their edges by first constructing a bilateral grid and then sampling the grid along each dimension, producing a pipeline consisting of a histogram operation and some stencil and sampling operations. The code we generate is composed of three groups, with the first consisting of the histogram operations, the second constructed by fusing all the stencil and sampling stages, and the final reduction. PolyMage and the automatic scheduling algorithm of Halide partially fuse the stencil and sampling stages. The manual schedule of Halide goes all the way to fusing stencil and sampling stages with the final reduction. Our technique is better than both the PolyMage framework and the automatic scheduling algorithm of Halide but falls behind the hand written schedule of Halide, as shown in Figure 17.

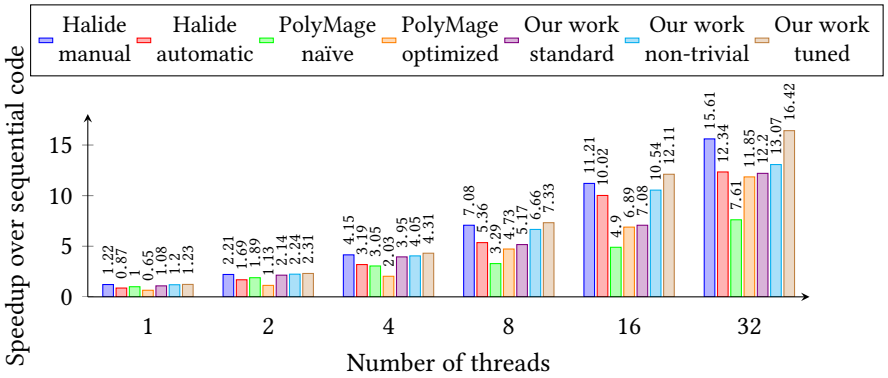

Fig. 17. Performance of Bilateral Grid on CPU

To validate the performance gap between our generated code and the manual schedule of Halide is not due to overlapped tiling, we manually tune our fusion strategy to align it to the latter, yielding the "tuned" version in the figure. In particular, we manually fuse the stencil and sampling stages with the final reduction, obtaining the same fusion structure as Halide's manual schedule. As expected, the resulting performance is competitive with the manual Halide schedule.

With regard to fusion, our heuristic follows the greedy algorithm of PolyMage, missing some fusion potentials that may be exploited by hand. This motivates further improvements of the fusion heuristic in the future. However, comparing the memory footprint of overlapped tiles with different fusion strategies makes less sense since the tile height differs, yielding shape variations at the lower stages in a tile.

Camera Pipeline. Targeting on transforming the raw data captured by a camera sensor into a color image, Camera Pipeline performs a sequence of steps including hot pixel suppression, demosaicing, color correction, and global tone mapping. These tasks are implemented by stencil-like stages consisting of multiple statements and table look-ups. The performance comparison of Camera Pipeline is shown in Figure 18. Our technique performs a similar fusion on the benchmark like PolyMage, grouping all stencil-like stages except the lookup table stages. PPCG also performs an automatic inlining of some of the stencil-like stages, outperforming the optimized version of PolyMage slightly. Our technique also beats the automatic scheduling algorithm of Halide, falling a little behind of the manually written schedule because the latter also benefits from unrolling optimization.

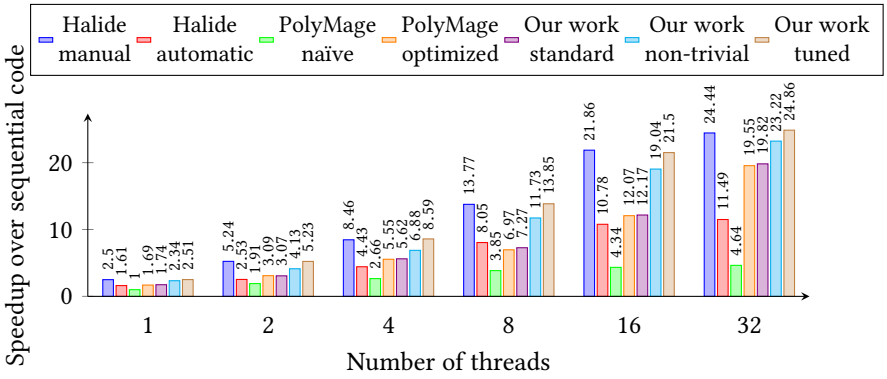

Fig. 18. Performance of Camera Pipeline on CPU

Like the Bilateral Grid benchmark, we may manually unroll the generated code, meeting the same result of the Halide manual schedule. Complementing our framework with a loop unrolling heuristic would further automate the process.

Table 3 shows the memory footprint of the overlapped tile shapes of the different frameworks. As can be seen from the table, our technique reduces the amount of redundant computation by shrinking the overlapped windows between consecutive tiles.

Table 3. The memory footprint Camera Pipeline (tile size 64×256)

	denoised	deinterleaved	r_r	g_gr	b_b	g_gb	g_b	greenX2	blueX2	redX1	g_r	greenX1
PolyMage	80×272	4×40×136	40×136	40×136	40×136	40×136	40×136	40×272	40×272	40×272	40×136	40×272
Our work	68×256	4×36×132	33×131	35×133	33×131	35×133	33×129	32×256	32×256	32×256	33×129	32×256
	b_gb	r_b	r_gr	b_r	r_gb	b_gr	redX2	green	blueX1	red	blue	corrected
PolyMage	40×136	40×136	40×136	40×136	40×136	40×136	40×272	80×272	40×272	80×272	80×272	3×80×272
Our work	32×128	32×128	32×128	32×128	32×128	32×128	32×256	64×256	32×256	64×256	64×256	3×64×256

Harris Corner Detection. Corner detection is an approach to extracting features and contents of interest from an image. Harris Corner Detection [18] is a widely used corner detection algorithm,

taking the differential of the corner score into account. The PolyMage framework fuses all stages of the pipeline into one group, benefiting from inlining all point-wise operations which we also implemented in our framework. A follow-up fusion is performed on the inlined version, grouping all stages of the pipeline together. The resulting performance matches that of PolyMage and beat the hand written schedule of Halide. The automatic scheduling algorithm also follows the PolyMage heuristic, performing similarly to PolyMage and our technique. Figure 19 shows the performance of Harris Corner Detection on CPU.

The group after fusion is composed of three stages, including *Iy*, *Ix* and *harris*. Both *Iy* and *Ix* are used to store intermediate values but they do not depend on each other, resulting in two 2-stage overlapped tile with the *harris* stage, and our technique therefore generates the same tile shape with PolyMage.

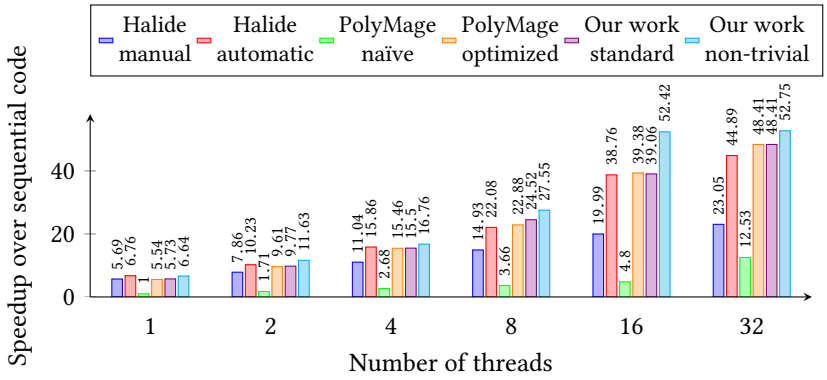

Fig. 19. Performance of Harris Corner Detection on CPU

Local Laplacian Filter. Local Laplacian Filter [2, 27] is an approach to producing edge-aware filters for manipulating images at multiple scales. The benchmark is the most complicated among all PolyMage benchmarks, consisting of 99 stages and involving both sampling and data-dependent operations. The performance comparison of Local Laplacian Filter is shown in Figure 20. Following the fusion criteria of the PolyMage framework, we fuse some of the stages of the pipeline and generate a much tighter overlapped tile shape than PolyMage, leading to better performance. Our technique also outperforms the manual schedule and Halide's automatic one, as they both miss important fusion opportunities.

The groups after fusion of Local Laplacian Filter could be divided into two categories, one with two stages or less, and the other with multiple stages. We do not show the memory footprint of 2-stage groups since the PolyMage algorithm does not over-approximate overlapped tile shapes in these cases (similar to Harris Corner Detection). There are 5 groups with multiple stages in this benchmark, whose memory footprints are shown in Table 4.

Multiscale Interpolation. Multiscale Interpolation is designed to preserve local shapes of an image at multiple scales by interpolating pixel values. The benchmark includes 49 stages consisting of sampling and stencil operations. Like Local Laplacian Filter, we apply the same loop fusion heuristic as PolyMage but minimize the redundant computation thanks to tighter overlapped tiles. This improves performance as shown in Figure 21. The Halide manual schedule is very similar to PolyMage's result. Unfortunately, we failed to obtain a version of the automatic scheduling algorithm that would successfully compile this benchmark, hence the missing comparison point.

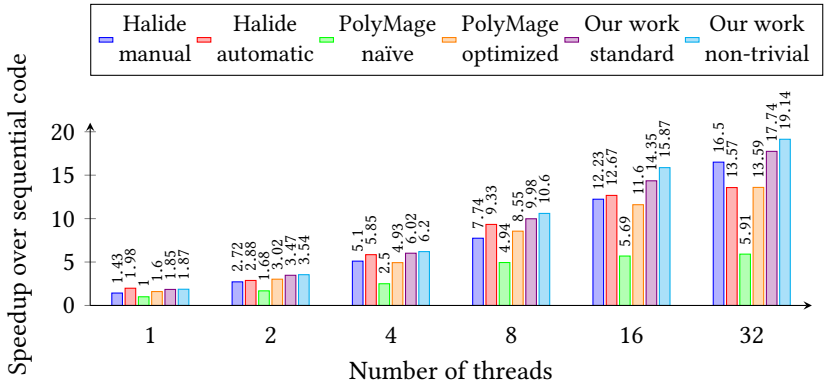

Fig. 20. Performance of Local Laplacian Filter on CPU

Table 4. The memory footprint of Local Laplacian Filter (tile size 8×256, P for Pyramid)

	U_lP_L4	outLP_L4	outGP_L4	Ux_lP_L3	U_lP_L3	outlP_L3	Ux_lP_L2	U_lP_L2	Ux_outGP_L2
PolyMage	8×10×100	10×100	10×100	8×8×100	8×8×196	8×196	8×8×131	8×8×262	8×131
Our work	8×8×130	8×8×256	8×130	8×6×100	6×100	6×100	8×8×100	8×8×196	8×196
	outLP_L2	Ux_lP_L1	U_lP_L1	Ux_outGP_L1	outLP_L1	Ux_lP_L0	U_lP_L0	Ux_result_ref_gray	outLP_L0
PolyMage	8×262	8×8×131	8×8×262	8×131	8×262	8×8×131	8×8×262	8×131	8×262
Our work	8×256	8×8×130	8×8×256	8×130	8×256	8×8×130	8×8×256	8×130	8×256

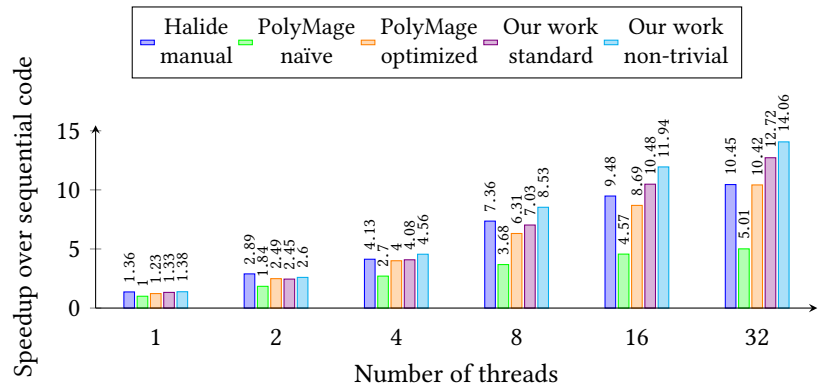

Fig. 21. Performance of Multiscale Interpolation on CPU

Like Local Laplacian Filter, one group has multiple stages and the memory footprint for each of its stages is shown in Table 5.

Table 5. The memory footprint of Multiscale Interpolation (tile size 32×128, i for interpolated)

	Ux_i_L3	i_L3	Ux_i_L2	i_L2	Ux_i_L1	i_L1	Ux_i_L0	i_L0
PolyMage	4×7×10	4×7×20	4×14×20	4×14×40	4×28×40	4×28×80	4×56×80	4×56×169
Our work	4×7×10	4×7×20	4×11×20	4×11×35	4×18×35	4×18×66	4×33×66	4×33×129

Pyramid Blending. Pyramid Blending [8] combines two images into a larger mosaic by constructing a Laplacian pyramid and performing filtering and splining operations. We show the performance of this benchmark in Figure 22. One may generate a complex grouping result by performing the fusion heuristic of the PolyMage framework, as we do for our technique. The pipeline comprises 44 stages, constituting several stage groups after fusion. As expected, our technique is effective at tightening the overlapped tile shapes for pipelines with a large number of stages, improving performance compared to PolyMage. As Pyramid Blending is not present in the Halide repository, we do not show the performance of Halide's automatic and manual versions for this benchmark.

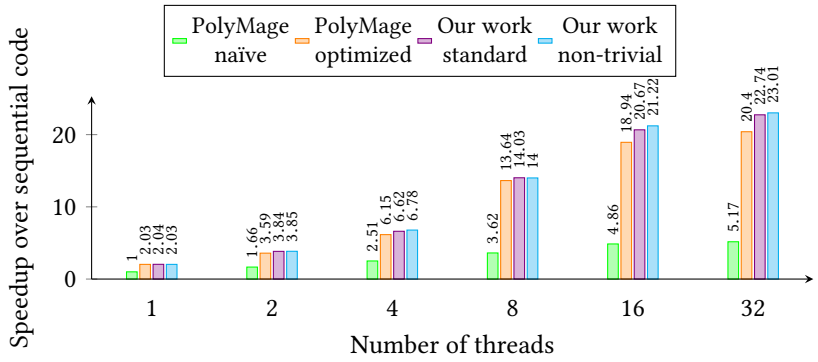

Fig. 22. Performance of Pyramid Blending on CPU

With respect to footprint evaluation, Pyramid Blending exhibits 3 multi-stage groups. Table 6 shows the comparison between our work and the PolyMage framework.

Table 6. The memory footprint of Pyramid Blending (tile size 16×256)

	Dx_3_img1	Dx_3_img2	Dy_3_img1	Dy_3_img2	Col_2_x	Dx_3_mask	Dy_3_mask	Ux_2_img1	Ux_2_img2
PolyMage	3×14×152	3×14×152	3×14×76	3×14×76	3×28×76	14×152	14×76	3×28×76	3×28×76
Our work	3×10×134	3×10×134	3×10×66	3×10×66	3×16×66	10×134	10×66	3×16×66	3×16×66
	Dx_2_mask	Dy_2_mask	Uy_2_img1	Uy_2_img2	Res_3	Res_2	Ux_1_img1	Ux_1_img2	Col_1_x
PolyMage	28×304	28×152	3×28×152	3×28×152	3×14×76	3×28×152	3×16×131	3×16×131	3×16×131
Our work	22×266	22×134	3×16×128	3×16×128	3×10×66	3×16×128	3×16×130	3×16×130	3×16×130
	Uy_1_img1	Uy_1_img2	Res_1	blend_x	Ux_0_img1	Ux_0_img2	Uy_0_img1	Uy_0_img2	Res_0
PolyMage	3×16×262	3×16×262	3×16×262	3×16×131	3×16×131	3×16×131	3×16×262	3×16×262	3×16×262
Our work	3×16×256	3×16×256	3×16×256	3×16×130	3×16×130	3×16×130	3×16×256	3×16×256	3×16×256

Unsharp Mask. Unsharp Mask is an image sharpening technique using an unsharp negative image to create a mask of the original image. This mask is then combined with the original image, constructing a less blurry image. The benchmark is a pipeline comprising a set of stencil operations, making it amenable for exploiting overlapped tiling using our technique. The stages of the pipeline can be fused into one group, benefiting from our tighter overlapped tile shape. The manually written schedule of Halide performs similar to the optimized version of PolyMage, outperforming the code generated from its automatic scheduling algorithm. Figure 23 shows the performance comparison among different approaches, and Table 7 compares the memory footprint of our technique with PolyMage.

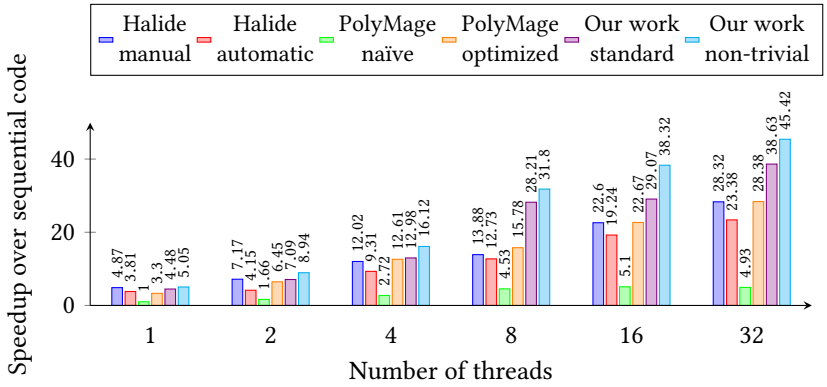

Fig. 23. Performance of Unsharp Mask on CPU

Table 7. The memory footprint of Unsharp Mask (tile size 8×512)

	blurx	blury	sharpen
PolyMage	3×8×518	3×8×518	3×8×518
Our work	3×8×516	3×8×512	3×8×512

Table 8. Problem sizes and tile sizes of the iterated stencils

	Problem sizes	Tile sizes			Baselinetime (s)
		standard	diamond	overlapped	
heat-1d	1000×160000	128×1024	2048^2	32×8192	6.99
heat-2d	1000×4000^2	16×64^2	64^3	4×64^2	7.17
heat-3d	100×150^3	$16^3 \times 256$	$16^3 \times 256$	$4 \times 16^2 \times 256$	1.21

4.3 Iterated Stencils

To validate the general applicability of our technique, we also conduct experiments on three representative iterated stencils. The detailed information about the examples and tile sizes we use in this subsection are listed in Table 8, with the execution times of each sequential code shown in the last column.[3] We use rectangle trapezoid tiling because it has a better locality for iterated stencils than scalene trapezoid tiling.

We first run the sequential code of the stencils and record the execution time as a baseline reference. We compare the performance with state-of-the-art diamond tiling, as enabled by the Pluto compiler, and parallelogram tiling enabled default in PPCG.

The *1/2/3d-heat* benchmarks evaluated in this subsection are iterated stencils solving the heat equation, iteratively updating data element using multi-point stencils. When selecting tile sizes, we follow the sizes chosen by diamond tiling [6].

One weakness of overlapped tiling is the redundant computation caused by shaded regions between neighboring tiles. One may thus have to select tile sizes for constructing a sharp overlapped tile, minimizing redundant computations as much as possible. We find 32×8192, $4 \times 64 \times 64$ and

[3]One may obtain the execution time of every optimized CPU and GPU version by crossing the baseline execution time with the speedups of Figures 24 and 26.

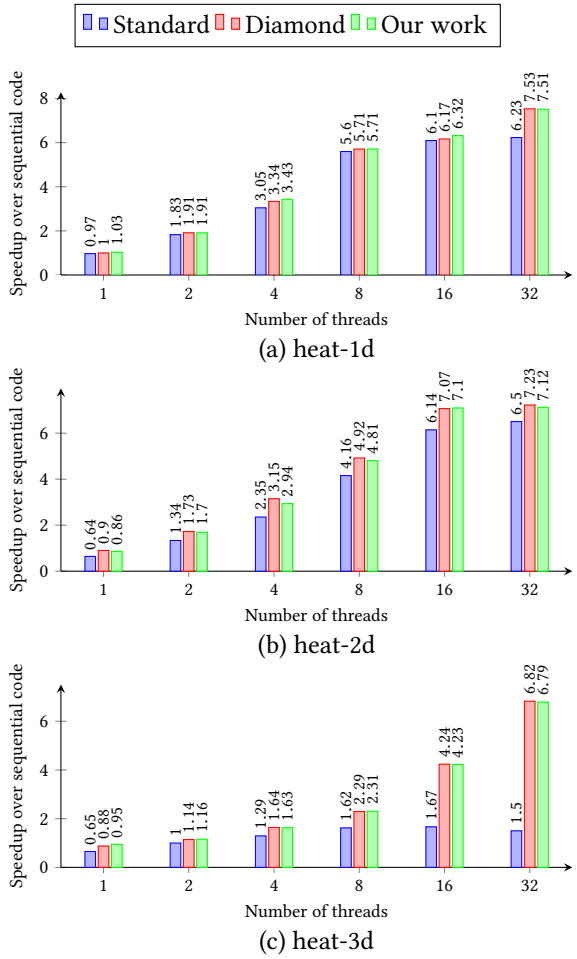

Fig. 24. Performance of the iterated stencils on CPU

$4 \times 16^2 \times 256$ in practice for these stencils as the best tile sizes. The performance results are shown in Figure 24, demonstrating overlapped tiling may achieve similar performance to diamond tiling by enabling inter-tile parallelization but without introducing a complex rescheduling step and the associated compilation overhead.

Note that the slight performance gap between overlapped tiling and diamond tiling is due to the re-computation induced by overlapped tiling. The purpose of our experiment by comparing with diamond tiling is not to prove overlapped tiling yields better performance, but instead to validate the general applicability of our technique on iterated stencils. This also provides additional comparison points across such tiling techniques, unavailable in previous publications.

4.4 Performance on GPU Architectures

We also evaluate our technique on GPU architectures—note that PolyMage does not target GPUs. We implement our technique in PPCG, allowing the generation of code for GPU architectures and thus extending the applicability of overlapped tiling on different architectures.

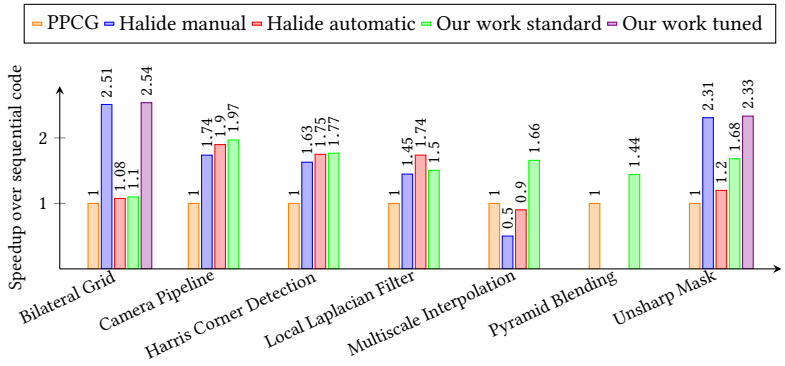

Fig. 25. Performance of the PolyMage benchmarks on GPU

The baseline PPCG performance results from a simple fusion heuristic over rectangular tiles, for image processing pipelines. Halide schedules may also be targeted to GPU devices, generating CUDA code by selecting approximate parameters. We therefore compare the performance with the manual schedule and the automatic scheduling algorithm of Halide. Figure 25 shows the performance comparison on GPU architectures.

Our technique yields steady performance improvements over the default setting of PPCG by enabling overlapped tiling on image pipelines. Yet it falls behind the manually written schedule of Halide for Bilateral Grid. The manual schedule fuses the pipeline into two groups, one combining the histograms with one stencil and the other grouping all the remaining stages. Our fusion heuristic is the same as the CPU case. The manual schedule of Unsharp Mask eliminates load operations by unrolling inner loops with invariant loads, improving performance over automatic techniques. The Halide performance on Pyramid Blending is missing because the benchmark is not available in the repository.

Following the CPU case and the associated Halide schedules, we fine-tune the generated code by manually fusing Bilateral Grid and unrolling Unsharp Mask. The performance of these "tuned" versions is also shown in Figure 25. Our technique reaches competitive performance with Halide's manual schedule provided such additional fusion and unrolling.

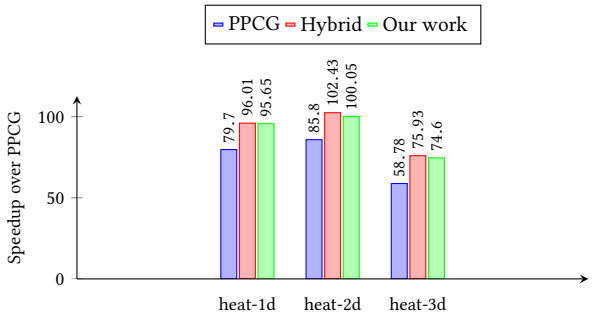

Fig. 26. Performance of the iterated stencils on GPU

We also apply our technique to the iterated stencils when targeting GPU architectures. We still use the sequential code of these stencils as a reference, and report the speedups of different tiling techniques in Figure 26. We first let PPCG perform a standard tiling on the stencils and compute the

speedup over sequential code. We also compare the performance with the state-of-the-art hybrid hexagonal/classical tiling [14], by setting *--hybrid* option to PPCG. It combines a hexagonal shape with classical tiling on time and space dimensions of stencils. We observed that varying the tile size had little performance impact as long as each tile loop executes within a single thread. For this reason, we reuse the tile sizes of the CPU case for overlapped tiling, and we reuse the tile sizes of diamond tiling for hybrid hexagonal/classical tiling. The thread and block dimensions of the GPU grid are set to the Halide ones for the image processing pipelines; Table 9 lists the auto-tuned thread and block dimensions for iterated stencils.

Table 9. Block and thread dimensions for iterated stencils

	Block dimensions			Thread dimensions		
	Standard	Hybrid	Overlapped	Standard	Hybrid	Overlapped
heat-1d	157	79	20	512	512	512
heat-2d	3907	3907	3907	64×8	32×16	64×8
heat-3d	88	88	88	16^2	16^2	16^2

5 RELATED WORK

Loop tiling was long considered as foreign to polyhedral techniques since it could not be easily expressed using an affine function. The Pluto scheduler [7] is driven by a practical cost function, integrating loop tiling with the polyhedral framework. The standard tile shape enabled by the Pluto scheduling algorithm or its variants is missing concurrent start, limiting the performance of the generated code especially those stencil-based applications.

Overlapped tiling and split tiling [21] were proposed to enable concurrent start by modifying the tile shape obtained by a Pluto-like scheduler. The latter may be implemented by either splitting the shaded region of overlapped tiles or introducing more advanced scheduling constraints on the bounding faces of a tile. No implementations of overlapped tiling are known to exist in general-purpose compilers, although the technique was implemented in domain-specific compilers for image processing pipelines [24, 31] or stencil code generator for GPU [19, 37]. There was also an implementation of split tiling for iterated stencils on GPU architectures [15]. Comparing with these approaches, our technique covers a wider application domain, improving performance over the state of the art by constructing tighter overlapped tiles.

Davis et al. [10] proposed fusion- and shifting-based overlapped tiles for optimizing applications involving stencil computations. Their image processing pipeline is limited to stencil operations but not sampling nor histogram operations. Our work, involving a general-purpose polyhedral flow, covers all the basic operations involved in image processing pipelines. The work of Davis et al. did not address the minimization of the memory footprint of overlapped tiling.

Bondhugula et al. [4, 6] proposed a general formalism for diamond tiling in the polyhedral model by introducing a rescheduling step in the Pluto compiler. There has been a great amount of work [11, 13, 22, 25, 33, 34] reported on the evaluation of diamond tiling. It was also generalized to handle iterated stencils defined over periodic data domains with index set splitting [5] and the Lattice-Boltzmann method [26]. Unlike overlapped tiling and split tiling, diamond tiling may work with arbitrary affine dependences. The introduction of scheduling to find tiling dimensions does not only complicate affine scheduling, but also increase code generation time in practice. Our experiments show that our technique achieves competitive performance with diamond tiling on iterated stencils while remaining applicable to image processing pipelines.

Hybrid hexagonal/classical tiling was proposed by Grosser et al. [14] to exploit full inter-tile parallelism of iterated stencils on GPU architectures. It can be seen as a generalization of diamond

tiling, allowing partial concurrent start by constructing a hexagonal tile shape along the time and first space dimensions, and classical tiling along the other space dimensions. Grosser et al. [17] also compare diamond tiling and hexagonal tiling. We compared our technique with hexagonal tiling in our GPU experiments.

Halide [30, 31] is a domain-specific language for image processing pipelines, decoupling algorithms from schedules for easy optimizations for such benchmarks, allowing users to experiment with schedules without touching the algorithms. Manually or auto-tuning approaches [1] to finding schedules usually takes a long time to reach competitive performance. The polyhedral framework is a promising solution to automatically search schedules for image processing pipelines by integrating with transformations like overlapped tiling, fusion, scratchpad allocation, etc. By revisiting overlapped tiling in polyhedral compilation frameworks, we demonstrated much tighter overlapped tile shapes for image processing pipelines, resulting in significant performance improvements.

Overlapped tiling is generally considered the best strategy for enabling tile-level concurrency for fused image processing pipelines, where the dependence chains are rather short and the degree of parallelism along the pipeline stages is limited. In addition, since trapezoid tile shapes allow for the allocation of intermediate values on scratchpad memory rather than full buffers, overlapped tiling is ideally suited to GPUs and accelerators with software-managed memories. However, split, diamond or hexagonal shapes may perform better on time-iterated stencils, or when targeting shared-memory architectures. We do not believe that parallelogram tiles perform better than any of these in general, and the diamond tiling evaluation provides some evidence of this [6]. Our results motivate further work in surveying all known tile shapes, revisiting the Zhou et al. shape selection results [36]; we are still missing a comprehensive understanding of the relative merits of each shape, depending on the application, data set and target architecture.

6 CONCLUSION

We revisited overlapped tiling in a general-purpose polyhedral compilation framework, validating our technique on image processing pipelines and iterated stencils. These classes of computations exhibit abundant data parallelism but require locality optimizations to achieve high performance. Our technique is implemented in the PPCG source-to-source compiler. It allows for tighter overlapped tile shapes than the state of the art, improving the performance on both general-purpose multicores and GPU accelerators. It integrates with affine transformations including alignment and scaling of image processing stages and stencil iterations, loop fusion, scratchpad allocation, hybrid tiling, etc. We evaluate the generation of both scalene and rectangle trapezoid tile shapes and the applicability and benefits of the approach over state-of-the-art domain-specific frameworks. Further automation and performance improvements could be achieved through combinations with better unrolling and loop fusion heuristics, inspired by manual schedules in Halide and machine learning approaches to the generation of Halide schedules.

ACKNOWLEDGMENTS

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61702546, and the European Commission through the MNEMOSENE project id. 780215. We would like to thank Michael Kruse, Chandan Reddy, Sven Verdoolaege and Oleksandr Zinenko for their tremendous support and valuable suggestions.

REFERENCES

[1] 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 *Proc. of the*

23rd Intl. Conf. on Parallel Architectures and Compilation (PACT '14). ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[2] Mathieu Aubry, Sylvain Paris, Samuel W. Hasinoff, Jan Kautz, and Frédo Durand. 2014. Fast Local Laplacian Filters: Theory and Applications. *ACM Trans. Graph.* 33, 5, Article 167 (Sept. 2014), 14 pages. https://doi.org/10.1145/2629645

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. https://doi.org/10.1145/197405.197406

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE CS, Los Alamitos, CA, USA, Article 40, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389051

[5] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. 2014. Tiling and Optimizing Time-iterated Computations on Periodic Domains. In *Proc. of the 23rd Intl. Conf. on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2628071.2628106

[6] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2017. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (Oct. 2017), 1285–1298. https://doi.org/10.1109/TPDS.2016.2615094

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[8] Peter J. Burt and Edward H. Adelson. 1983. A Multiresolution Spline with Application to Image Mosaics. *ACM Trans. Graph.* 2, 4 (Oct. 1983), 217–236. https://doi.org/10.1145/245.247

[9] Jiawen Chen, Sylvain Paris, and Frédo Durand. 2007. Real-time Edge-aware Image Processing with the Bilateral Grid. In *ACM SIGGRAPH 2007 Papers (SIGGRAPH '07)*. ACM, New York, NY, USA, Article 103. https://doi.org/10.1145/1275808.1276506

[10] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. 2018. Transforming Loop Chains via Macro Dataflow Graphs. In *Proc. of the 2018 Intl. Symp. on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 265–277. https://doi.org/10.1145/3168832

[11] H Eissfeller and S. M Muller. 1990. The Triangle Method for Saving Startup Time in Parallel Computers. In *Distributed Memory Computing Conf., 1990., Proc. of the Fifth*. 568–572.

[12] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Intl. Journal of Parallel Programming* 21, 6 (01 Dec 1992), 389–420. https://doi.org/10.1007/BF01379404

[13] Pieter Ghysels and Wim Vanroose. 2015. Modeling the performance of geometric multigrid stencils on multicore computer architectures. *SIAM Journal on Scientific Computing* 37, 2 (2015), C194–C216.

[14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proc. of Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 66, 10 pages. https://doi.org/10.1145/2544137.2544160

[15] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proc. of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/2458523.2458526

[16] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. https://doi.org/10.1145/2743016

[17] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. 2014. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters* 24, 03 (2014), 1441002.

[18] Chris Harris and Mike Stephens. 1988. A combined corner and edge detector.. In *Alvey vision conference*, Vol. 15. 10–5244.

[19] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proc. of the 26th ACM Intl. Conf. on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 311–320. https://doi.org/10.1145/2304576.2304619

[20] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-level Tiling: M for the Price of One. In *Proc. of the 2007 ACM/IEEE Conf. on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 51, 12 pages. https://doi.org/10.1145/1362622.1362691

[21] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/1250734.1250761

[22] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. 2017. Multidimensional Intratile Parallelization for Memory-Starved Stencil Computations. *ACM Trans. Parallel Comput.* 4, 3, Article 12 (Dec. 2017), 32 pages. https://doi.org/10.1145/3155290

[23] 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 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[24] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proc. of the Twentieth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 429–443. https://doi.org/10.1145/2694344.2694364

[25] Daniel A. Orozco and Guang R. Gao. 2009. Mapping the FDTD Application to Many-Core Chip Architectures. In *Intl. Conf. on Parallel Processing*. 309–316.

[26] Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. 2015. An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations. *ACM Trans. Archit. Code Optim.* 12, 2, Article 14 (May 2015), 23 pages. https://doi.org/10.1145/2739047

[27] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. 2015. Local Laplacian Filters: Edge-aware Image Processing with a Laplacian Pyramid. *Commun. ACM* 58, 3 (Feb. 2015), 81–91. https://doi.org/10.1145/2723694

[28] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, Frédo Durand, et al. 2009. Bilateral filtering: Theory and applications. *Foundations and Trends® in Computer Graphics and Vision* 4, 1 (2009), 1–73.

[29] William Pugh and David Wonnacott. 1994. Static Analysis of Upper and Lower Bounds on Dependences and Parallelism. *ACM Trans. Program. Lang. Syst.* 16, 4 (July 1994), 1248–1278. https://doi.org/10.1145/183432.183525

[30] 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, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[31] 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 *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[32] Fabrice Rastello and Yves Robert. 2002. Automatic Partitioning of Parallel Loops with Parallelepiped-Shaped Tiles. *IEEE Trans. Parallel Distrib. Syst.* 13, 5 (2002), 460–470. https://doi.org/10.1109/TPDS.2002.1003856

[33] Sunil Shrestha, Guang R. Gao, Joseph Manzano, Andres Marquez, and John Feo. 2015. Locality Aware Concurrent Start for Stencil Applications. In *Proc. of the 13th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO '15)*. IEEE CS, Washington, DC, USA, 157–166. http://dl.acm.org/citation.cfm?id=2738600.2738620

[34] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2011. Cache Accurate Time Skewing in Iterative Stencil Computations. In *Proc. of the 2011 Intl. Conf. on Parallel Processing (ICPP '11)*. IEEE CS, Washington, DC, USA, 571–581. https://doi.org/10.1109/ICPP.2011.47

[35] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[36] Xing Zhou, María J. Garzarán, and David A. Padua. 2015. Optimal Parallelogram Selection for Hierarchical Tiling. *ACM Trans. Archit. Code Optim.* 11, 4, Article 58 (Jan. 2015), 23 pages. https://doi.org/10.1145/2687414

[37] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. 2012. Hierarchical Overlapped Tiling. In *Proc. of the Tenth Intl. Symp. on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/2259016.2259044

[38] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling. In *Proc. of the 27th Intl. Conf. on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 3–13. https://doi.org/10.1145/3178372.3179507

Received January 2019