An Effective Fusion and Tile Size Model for PolyMage

ABHINAV JANGDA and UDAY BONDHUGULA, Indian Institute of Science, India

Effective models for fusion of loop nests continue to remain a challenge in both general-purpose and domain-specific language (DSL) compilers. The difficulty often arises from the combinatorial explosion of grouping choices and their interaction with parallelism and locality. This article presents a new fusion algorithm for high-performance domain-specific compilers for image processing pipelines. The fusion algorithm is driven by dynamic programming and explores spaces of fusion possibilities not covered by previous approaches, and it is also driven by a cost function more concrete and precise in capturing optimization criteria than prior approaches. The fusion model is particularly tailored to the transformation and optimization sequence applied by PolyMage and Halide, two recent DSLs for image processing pipelines. Our model-driven technique when implemented in PolyMage provides significant improvements (up to 4.32×) over PolyMage's approach (which uses auto-tuning to aid its model) and over Halide's automatic approach (by up to 2.46×) on two state-of-the-art shared-memory multicore architectures.

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

Additional Key Words and Phrases: Fusion, tiling, image processing pipelines, parallelism, locality

ACM Reference format:
Abhinav Jangda and Uday Bondhugula. 2020. An Effective Fusion and Tile Size Model for PolyMage. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 12 (November 2020), 27 pages.
https://doi.org/10.1145/3404846

1 INTRODUCTION

PolyMage [14, 16, 17] is an experimental domain-specific language and compiler for several domains of dense linear algebra, in particular, classes of image processing pipelines, geometric multigrid computations, and time-iterated stencils.

Computations in these domains can often be expressed as a directed acyclic graph of compute nodes where each node applies a simple operation, which is often data parallel, on all elements of a multi-dimensional array. A pipeline may involve a few simple point-wise functions or be as complex as comprising hundreds of functions ranging from point-wise to stencil operations and reductions. For example, in the case of image processing pipelines, domain-specific languages (DSLs) such as Halide [20] and PolyMage [14] provide high-level constructs to express such pipelines, and more importantly are able to do complex code transformations to enhance locality and parallelism.

The author was a project associate at the Indian Institute of Science when this work was carried out.
This work was supported in part by a grant (EMR/2016/008015) from the Science and Engineering Research Board (SERB), India through its Extramural Research funding program.
Authors' addresses: A. Jangda and U. Bondhugula, Indian Institute of Science, Dept of CSA, Bengaluru, 560012, India; emails: abhijangda@gmail.com, udayb@iisc.ac.in.

0164-0925/2020/11-ART12 $15.00
https://doi.org/10.1145/3404846

A key component of such DSL compilers that optimize image processing pipelines is their fusion algorithm. Fusion of different image processing stages is used to exploit producer-consumer locality across such stages. While the problem is viewed as that of loop fusion [9] in compiler optimization, other terms such as *kernel fusion* and *operator fusion* are used in specific contexts. Both Halide and PolyMage perform fusion and tiling across multiple stages through special techniques that are a realization of overlapped tiling [10, 14, 20]. The locality is important to exploit, since outside of the reuse available from the production and repeated consumption, there is no additional reuse unlike in the case of other algorithms such as BLAS level-3 or time-iterated stencil computations [26]. Dramatic speedups have been reported in recent works [13, 14, 20] as a result of such optimization. Similar optimizations have yielded performance improvements for the Geometric Multigrid Method [24].

The automatic fusion heuristics used in both PolyMage and Halide share a greedy aspect of an approach that excludes a large space of fusion possibilities. The heuristics start from a node and explore nodes around it in a particular way to merge and form larger groups while evaluating profitability. We propose an approach that explores a larger space of valid groupings than previously considered. In addition, fusion is tightly connected to tile size selection as well as other optimizations such as inlining or expression propagation [4] that may in cases add redundant computation.

In this article, we present a new fusion-cum-tile size determination model that uses a dynamic programming-based approach to address limitations of the state-of-the-art. In addition, PolyMage searches and auto-tunes over a range of tile sizes that are all a power of two and a fixed number of tile overlap thresholds (boundary redundant computation) to keep the auto-tuning space tractable. Our approach is completely model-driven—it alleviates the need for auto-tuning by directly incorporating tile sizes and relative tile overlap within the concrete cost function of the fusion heuristic. In addition, the tile sizes are not restricted to powers of two. In summary, the key contributions of this article are:

- a dynamic programming-based grouping algorithm to find a good grouping along with the tile sizes to be used for tiling each group, while considering a larger space of possibilities than prior approaches;
- a concrete cost function to be used in conjunction with the grouping algorithm that considers multiple aspects including locality, parallelism, and other optimization criteria much more precisely than prior approaches;
- an incremental variant of the grouping algorithm to control its running time by trading off certain choices;
- multi-level tiling to generate tiles that provides both L1 and L2 cache locality;
- automatic inlining of producer expressions into consumer expressions with no redundant computations;
- and an implementation of the proposed technique in PolyMage and its experimental evaluation on two state-of-the-art multicore architectures over image processing benchmarks and multigrid stencil benchmarks.

Experimental results demonstrate significant improvement over PolyMage's current auto-tuning approach, over Halide's auto scheduler, and over expert manual schedules provided with Halide where applicable. Our approach is applicable to DSLs where computations can be expressed through directed acyclic graphs where each node of the graph is a loop nest working on dense multi-dimensional arrays or tensors. The TensorFlow/XLA [6] project for compilation of dense linear algebra is another effort where our approach is applicable.

The rest of this article is organized as follows: In Section 2, we describe limitations of state-of-the-art approaches. Section 3 through Section 6 describe all our contributions. Experimental

```
1  R, C = Parameter(Int, "R"), Parameter(Int, "C")
2
3  # Vars
4  x = Variable(Int, "x")
5  y = Variable(Int, "y")
6  c = Variable(Int, "c")
7
8  # Input Image
9  img = Image(Float, "img", [3, R+2, C+2])
10
11 # Intervals
12 cr = Interval(Int, 0, 2)
13 xrow,xcol = Interval(Int, 1, R), Interval(Int, 0, C+1)
14 yrow,ycol = Interval(Int, 1, R), Interval(Int, 0, C)
15
16 cond = Condition(x,'>=', 1) & Condition(x,'<=', R) &
17        Condition(y,'<=', C) & Condition(y,'>=', 1)
18
19 blurx = Function(([c,x,y], [cr,xrow,xcol]), Float, "blurx")
20 blurx.defn = [Case(cond,(img(c,x-1,y) +
21                          img(c,x  ,y) +
22                          img(c,x+1,y))*1./3)]
23
24 blury = Function(([c,x,y], [cr,yrow,ycol]), Float, "blury")
25 blury.defn = [Case(cond,(blurx(c,x,y-1) +
26                          blurx(c,x,y  ) +
27                          blurx(c,x,y+1))*1./3)]
```

Fig. 1. PolyMage DSL specification for *blur*.

evaluation is presented in Section 7. Related work is discussed in Section 8, and conclusions are presented in Section 9.

2 BACKGROUND AND MOTIVATION

In this section, we provide an overview of PolyMage, then discuss the automatic grouping heuristics currently used by PolyMage [14] and Halide [13], and their limitations.

2.1 PolyMage

PolyMage is a domain-specific language (DSL) for writing image Processing Pipelines. It allows a user to intuitively express common computation patterns such as point-wise operations, stencils, upsampling, and downsampling. PolyMage DSL is embedded in Python. Figure 1 shows the specification of *blur*, a simple image processing pipeline with two stages, *blurx* and *blury*. Parameters to the pipeline such as the number of rows and columns of the input image are declared using the `Parameter` construct at line 1. Input data to the pipeline is declared using the `Image` construct at line 9. `Function` is used to declare a stage of an image processing pipeline as a function mapping a multi-dimensional integer domain to values representing intensities of image pixels. The domain of the function is defined using the `Interval` construct at lines 12–14. The `Case` construct allows conditional execution of computation. Function *blurx* (lines 19–22) takes the image as input and blurs it in the x-direction. Function *blury* (lines 24–27) takes the output of *blurx* as input and blurs it in the y-direction. The output of *blury* is the output of this pipeline.

PolyMage's compiler constructs a polyhedral representation of the pipeline, with functions as the statements with polyhedral domains. The compiler then is able to perform fusion and tiling across these functions by constructing polyhedral schedules. Overlapped tiling for the *blur* pipeline leads to trapezoid shaped tiles along the y-direction, and this is illustrated in Figure 3 (the x dimension is not shown for convenience). Such a tiling leads to recomputation of function values in the overlapping region between two neighboring tiles. As a result, the dependence between neighboring tiles is broken, allowing parallel execution of all tiles without the need for synchronization

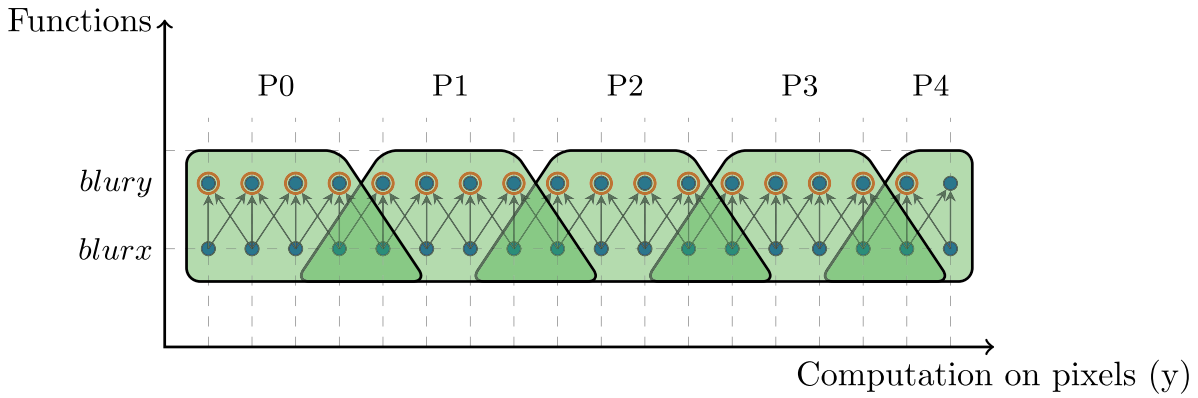

Fig. 2. Overlapped tiling of *blur* kernel in the y-direction.

or communication between them. In Figure 3, there is no dependence between tiles P1 and P2, as both recompute function values at their intersection. Hence, tiles P0 through P4 can be executed in parallel.

2.2 PolyMage's Fusion Heuristic

PolyMage's existing fusion heuristic takes as input a directed acyclic graph, (V, E), which represents the image processing pipeline, where V is the set of stages or functions and edges in E represent the producer-consumer relationship between two stages. We first explain what exactly fusion or grouping or merging means in this context, since all of these terms are used synonymously later in the article. When a set of nodes from the DAG are referred to as being grouped or merged or fused together, that group of statements (where a statement corresponds to a node) will also be tiled using overlapped tiling when necessary, and their tile space loops are fused. In PolyMage, the intra-tile loops of the individual statements are distributed inside, i.e., a single trapezoidal tile (illustrated in Figure 3) runs tiles of the individual statements one after another (*blurx* tile followed by *blury* tile). The generated code for the *blur* specification in Figure 1 is shown in Figure 2.

At the start of the PolyMage's Fusion algorithm each pipeline function is in a separate group. The algorithm iteratively merges groups in the following way until no further merging is possible: In each iteration, it finds all groups that have only a single child (successor) in the pipeline graph (so cycles are not formed due to merging). These candidate groups are then sorted in the decreasing order of their sizes determined from the parameter estimates.

The algorithm then iterates over the sorted groups to check for merging opportunities. A group is merged with its child only if the following two conditions are met: First, dependences between the group and its child have to be made constant (not dependent on problem sizes) by performing loop transformations (scaling and aligning) on the loops of the functions in the child and parent groups. Note that scaling loops of different functions by different factors allows inter function dependences that are originally non-constant (distances dependent on problem sizes) to be made constant. Such scaling is necessary when functions involve upsampling and downsampling operations. Aligning here refers to matching a specific dimension of one function with that of another function for loop fusion. For example, consider two functions S_1 and S_2 each defined on a two-dimensional domain: Now, a mapping $S_1(i, j) \rightarrow (2i, j)$; $S_2(x, y) \rightarrow (4y, x)$ implies that the first dimension of S_1 and the second dimension of S_2 have been scaled by factors of two and four, respectively; then $2i$ has been aligned with $4y$ and j with x (in the transformed space) for fusion. The scaling was necessary to make inter-function dependences constant. The second condition to be met for merging is that the size of the overlapping region (which represents redundant

```
1  float blury[3 * 2046 * 2048];
2
3  #pragma omp parallel for schedule(static) collapse(2)
4  for (int tx = 0; tx < 2048 / 64; tx++) {
5    for (int ty = 0; ty < 2048 / 64; ty++) {
6      float blurx[3][64][68];
7
8      /* Tile of function blurx. */
9      for (int c = 0; c <= 2; c++){
10       for (int x = 0; x < 64; x++){
11         #pragma ivdep
12         for (int y = -1; y < 66; y++){
13           blurx[c][x][y + 1] = (img[c][tx*64+x-1][ty*64+y] +
14                                 img[c][tx*64+x  ][ty*64+y] +
15                                 img[c][tx*64+x+1][ty*64+y])/3.0;
16         }
17       }
18     }
19
20     /* Tile of function blury. */
21     for (int c = 0; c <= 2; c++){
22       for (int x = 0; x < 64; x++){
23         #pragma ivdep
24         for (int y = 0; y < 64; y++){
25           blury[c][x][y] = (blurx[c][tx * 64 + x][ty * 64 + y - 1] +
26                             blurx[c][tx * 64 + x][ty * 64 + y] +
27                             blurx[c][tx * 64 + x][ty * 64 + y + 1]) / 3.0;
28         }
29       }
30     }
31   } // end of tile-space loop ty
32 } // end of tile-space loop tx
```

Fig. 3. C++ code generated by PolyMage for the *blur* pipeline. tx, ty are the tile-space loops (fused for both *blurx* and *blury*) corresponding to their intra-tile counterparts x and y, respectively. Tile size is set to 64×64, and tiles overlap along y. Output of *blurx* is stored in a small local buffer.

computation when the group is tiled), as a fraction of given tile size, is less than the overlap tolerance (provided as a parameter and is part of the auto-tuning phase).

The parameters for PolyMage's grouping algorithm, tile size and overlap tolerance, are tuned over by its auto-tuner. The auto-tuner search space typically contains seven tile sizes per dimension and three overlap tolerance values. For each combination of these values, grouping is performed and the generated code is executed, and fastest version is empirically determined.

2.3 Halide Fusion Heuristic

Halide was originally proposed as a language where the schedule was specified by the programmer [19]; the optimization approach was thus semi-automatic. Earlier attempts to automate scheduling through auto-tuning met with limited success due to the large space being searched. However, recent work by Mullapudi et al. [13] introduced model-driven automatic scheduling for Halide that is very effective; the fusion heuristic used is a significant improvement over PolyMage in the following two ways: (1) the heuristic determines grouping in conjunction with the best tile size for the group, (2) the auto-tuning is done on parameters that more directly capture the tradeoff between performing redundant computation and cache misses.

Halide's auto-scheduling fusion heuristic [13] first places each function in the program in its own group and tiles the function to maximize input data reuse, if such an opportunity exists. Then, the algorithm considers those pair-wise merging opportunities where there is a direct producer-consumer relationship. Halide enumerates these merging opportunities, and for each such merging opportunity, it estimates the performance benefit of merging the two groups and picks that merging that provides the highest benefit. This process is repeated until no profitable merging

opportunity remains. The performance benefit is calculated using a cost function that enumerates several tile size possibilities and assigns tile sizes that are determined to provide best performance analytically. Performance estimation for a given tile size is based on three factors: (i) the number of tiles should be greater than or equal to PARALLELISM_THRESHOLD, which is set to the number of cores; (ii) memory footprint, where a larger memory footprint is penalized more than a smaller one depending on the L2 cache size; and (iii) the innermost storage dimension should have at least VECTOR_WIDTH data points to iterate on. The cost of a group with a given tile size is estimated as the sum of the costs of arithmetic operations in the group and the cost of loading a group tile from main memory into cache (calculated using a LOAD_COST parameter). Two groups are merged only if the cost of merging them is the lowest among all grouping possibilities and that cost is less than that of not merging them. In addition, while evaluating a specific fusion choice, tile sizes that are powers of two are iteratively tried to determine the largest one that allows data to fit within the CACHE_SIZE setting.

2.4 Limitations of Prior Art

Both PolyMage's and Halide's algorithms are evidently greedy and fast. The algorithms tend to maximize reuse by grouping stages connected by producer-consumer relationships and prevents a merge if it is not profitable.

The downsides of PolyMage's fusion algorithm are as follows.

- Several valid and potentially profitable opportunities may be missed.
- The same tile size is used for each group in a particular grouping. However, clearly, each group can have its own best tile size. The combinatorial explosion that results from auto-tuning each group with an independent tile size prevents the auto-tuner from exploring this freedom. For example, instead of a typical 147 choices ($7^2 \times 3$ corresponding to several tile sizes along each dimension and three overlap tolerance values), it may lead to $(7^2)^g \times 3$ choices, where g is the number of groups. The resulting explosion in auto-tuning time is thus undesirable for an approach that is aimed at being mostly model-driven if not completely.
- The tile sizes considered by the auto-tuner are only powers of two to explore the entire typical range 32–2,048. However, there is no strong reason to discard tile sizes that are not powers of two, and the latter may often lead to higher returns.
- PolyMage does not support inlining of producer expressions into consumer expressions. Inlining of expressions helps in decreasing the total cache footprint and the amount of redundant computations.
- PolyMage generates only one level of tiling, hence, providing locality for only level of cache. However, we can provide locality for both L1 and L2 cache, thereby improving performance.

Halide's automatic scheduling algorithm solves three key limitations of PolyMage's: (a) it computes tile sizes for each group independently, (b) it uses a cost model to determine tile sizes instead of auto-tuning, and (c) automatically inlines expressions even if that results in redundant computations. The auto-tuning is instead performed on another parameter that captures the ratio of computation costs to cache misses, which is expected to be specific to different microarchitectures. Halide's algorithm first runs a function inlining pass to determine all the inlining candidates. However, Halide's algorithm still has the remaining limitations, i.e., of not considering several potentially profitable grouping opportunities due to a local greedy choice, of not considering tile sizes that are not powers of two (the implementation considers only power of two sizes, since each candidate tile size has to be evaluated), of not exploiting locality for two levels of cache, and of not considering inlining in the grouping algorithm.

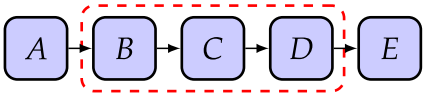

Fig. 4. Illustration to explain prior art limitations.

The first key limitation, the greedy nature of the algorithm, remains common to PolyMage and Halide. It allows both algorithms to evaluate a narrow space of valid groupings (and hence tiling opportunities). We show now in what way the greedy nature of both of these algorithms excludes a large number of groupings.

With the fusion algorithms of Halide and PolyMage, the exclusion of a significant number of valid and potentially profitable grouping can be classified as occurring due to the following two reasons: (1) Grouping along a path stops completely if the connected nodes make the merge non-profitable. Hence, a single node that makes the grouping non-profitable can prevent the consideration of a large number of groupings that may be profitable as per yet unevaluated costs. (2) More importantly, several different merging possibilities may exist and a choice of one over the other (irrespective of which among them is profitable) precludes a large number of other possibilities. We explain this limitation through a simple example, the DAG of Figure 4.

For any linear pipeline graph with n stages, there are 2^{n-1} valid grouping possibilities. For the given example, there are $2^{5-1} = 16$ valid groupings. In this case, invalid groupings are those that comprise stages that are not adjacent to each other in the graph. For the Figure 4 DAG, Halide's grouping algorithm starts by first considering {A,B}, {B,C}, {C,D}, {D,E} as the merging candidates and picks the one that provides the best benefit, say, {D,E}. In the next iteration, the grouping candidates will be: {A, B}, {B, C}, and {C,{D, E}}. It will stop merging when no pair-wise producer-consumer merging provides benefit. For a linear pipeline graph with n stages, we notice that Halide's grouping algorithm will evaluate at most $n(n-1)/2$ pair-wise groupings and ultimately, $1 + n(n-1)/2$ final grouping candidates. However, grouping considerations are influenced by local greedy choices based on lowest cost function values at a particular step. For example, in the above case, either of the groupings {{A,B,C,D},E} and {{A},{B,C,D},{E}} might have been ultimately better, and these are not evaluated as a result of the initial choice to group D and E, which in turn may make grouping other nodes not profitable or less profitable. The limitation is thus fundamental, easy to see, and common to both PolyMage and Halide.

It is, however, challenging to develop an approach that is able to efficiently evaluate all valid or all interesting valid groupings for a general DAG while incorporating a concrete cost function, and we address this in our work. We show that the approach we develop will be able to evaluate a significantly larger number of valid groupings through its cost function than prior approaches do; in the specific case of linear DAGs like that of Figure 4, all 2^{n-1} valid groupings would be considered (evaluated in effect), but in only $O(n^2)$ time.

3 A NEW FUSION APPROACH

In this section, we describe our new fusion algorithm. The cost function that will be used by the fusion algorithm will be described in the next section.

Dynamic programming-based grouping:. In contrast to previous works that used a greedy algorithm, we evaluate valid merging opportunities using a dynamic programming (DP) approach. Given a pipeline graph (V, E), V being stages as vertices, and E edges, our algorithm returns the grouping that has the minimum cost over the space of possibilities considered, which we will present in Section 4. Since our focus here is on improving producer-consumer locality, our algorithm will evaluate only those merging opportunities for a group where at least one producer of

$$
\begin{aligned}
F(G) &= F(\{H_1, \ldots, H_n\}) \\
&= \begin{cases}
\displaystyle\min_{H_i \in G} \text{COST}(H_i), & \text{if } \displaystyle\min_{H_i \in G} SUCC_G(H_i) = \emptyset, \\
\min \left(\begin{array}{l}
\displaystyle\min_{H_i \in G} \left(\min_{s_j \in SUCC_G(H_i)} F\left(\{H_1, \ldots, H_i \cup \{s_j\}, \ldots, H_n\}\right) \right), \\
\displaystyle\min_{H_i \in G} \text{COST}(H_i) + \min_{P_i \in PARTS\left(\bigcup_{H_i \in G} SUCC_G(H_i)\right)} F(P_i)
\end{array} \right), & \text{if } \displaystyle\min_{H_i \in G} SUCC_G(H_i) \neq \emptyset.
\end{cases}
\end{aligned}
$$

Fig. 5. Dynamic programming formulation of fusion. $PARTS(S)$ generates all partitions of the set S. $SUCC_G(H)$ returns the set of direct successors of H that are not in any $H_i \in G$.

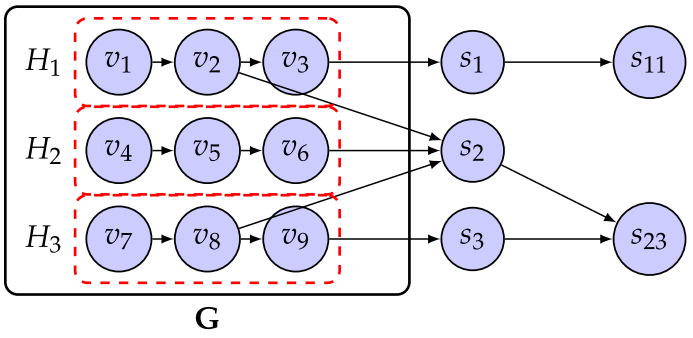

Fig. 6. Illustration for DP formulation.

the merge candidate is in the group. This restriction reduces the number of merging opportunities to be enumerated significantly. We now present the formulation of the dynamic programming recurrence.

3.1 Near-Optimal Sub-structure for Fusion

Let $S = (V, E)$ be the DAG of the image processing pipeline. Let G be the grouping for a portion of the DAG, i.e., it is a set of sets of individual graph nodes such that:

$$
\begin{aligned}
G &= \{H_1, H_2, \ldots, H_n\}, \\
H_i &\text{ is a sub-graph of } S, \\
\text{for any } H_i, H_j \in G,\ H_i \neq H_j,\ H_i \cap H_j &= \emptyset.
\end{aligned} \tag{1}
$$

The function $F(G)$ represents the minimum cost grouping for the portion of the DAG (V, E) that includes G and all nodes in V reachable from G under the condition that the grouping specified in G will not be decomposed further nor will those groups be merged with each other, i.e., only the descendants of G can be added to or merged into groups of G, and groups of G cannot be partitioned further. This formulation underlies our DP-based approach.

The DP recurrence is provided in Figure 5, and Figure 6 helps with illustration; it works as follows: If none of the groups H_i in G have any successors, then $F(G)$ is the cost of G; otherwise, the cost is the minimum between the choices of (Case I) grouping $H_i \in G$ with any of H_i's successors that are not in G, and (Case II) that of not grouping with any of H_i's successors that are not in G. Both Case I and Case II involve multiple possibilities over which the minimum is computed. $PARTS(S)$ represents all possible partitionings of a set S. A partitioning of S is a set of non-empty subsets of S such that every element $s \in S$ appears in exactly one of these subsets. $SUCC_G(H)$ represents all the immediate successor nodes of H in the DAG that are not in any $H_i \in G$. In case II, G is finalized (no further changes to it), and hence the recurrence subsequently explores the graph starting from all possible partitionings of successors of G.

ALGORITHM 1: Dynamic programming-based grouping

```
1:  function DP-GROUPING(G, T)
2:      if G ∈ T then return T[G]
3:      if ⋃_{H_i ∈ G} SUCC_G(H_i) = ∅ then
4:          T[G] ← ⟨∑_{H_i ∈ G} COST(H_i), G⟩
5:          return T[G]
6:      ⟨minCost, minGroup⟩ ← ⟨∞, ∅⟩
7:      for all H_i ∈ G do
8:          for all s_j ∈ SUCC_G(H_i) do
9:              isCycle ← false
10:             for all t ∈ SUCC(H_i) do
11:                 if t ≠ s_j and s_j.isReachableFrom(t) then
12:                     isCycle ← true
13:                     break
14:             if isCycle = false then
15:                 C ← {H_i ∪ {s_j}}
16:                 ⟨cost_1, g⟩ ← DP-GROUPING((G ∪ C) − {H_i},T)
17:                 if cost_1 < minCost then
18:                     ⟨minCost, minGroup⟩ ← ⟨cost_1, g⟩
19:     minPartCost ← ∞
20:     for all P_j ∈ PARTS(∪_{H_i ∈ G} SUCC_G(H_i)) do
21:         ⟨cost, G'⟩ ← DP-GROUPING(P_j, T)
22:         if cost < minPartCost then
23:             ⟨minPartCost, minPartGrouping⟩ ← ⟨cost, G'⟩
24:     cost_2 ← ∑_{H_i ∈ G} COST(H_i) + minPartCost
25:     if cost_2 < minCost then
26:         ⟨minCost, minGroup⟩ ← ⟨cost_2,G∪ minPartGrouping⟩
27:     T[G] ← ⟨minCost, minGroup⟩
28: return ⟨minCost, minGroup⟩
```

The algorithm starts with the source vertex of the pipeline graph. If there are multiple source vertices, then a dummy source vertex is created with edges into the original source vertices. Grouping then starts from this dummy source vertex, which has a cost of zero. This recurrence can then use memoization by using a table T to store the cost of $F(G)$ along with the grouping information. By construction, we notice that the elements of G will always be disjoint sub-graphs of (V, E). The objective of the DP is to minimize the sum total of the costs of all groups as computed by the *COST* function that will be described in the next section.

Algorithm 1 is driven by the DP recurrence of Figure 5 and uses memoization. It takes two parameters: a pre-fused set G as described earlier, and the DP memo (table). DP-GROUPING first checks if the minimum cost grouping had already been calculated and stored in T. Line 5 returns the cost of G corresponding to the base case, i.e., when G has no successors. Lines 8–18 find the grouping corresponding to the best possible way H_i could be merged with its successors that are not already in G (Case I of Figure 5). Lines 15–16 evaluate merging H_i with each of its candidate successors as long as grouping is the valid. Lines 20–24 evaluate the cost of not merging G with any of its successors (Case II of Figure 5). This is done by separating out (or finalizing) G and starting again from all partitionings of successors of G. Line 27 determines the final minimum cost and the corresponding merging as the minimum between the two cases ($cost_1$ and $cost_2$). The minimum cost and the grouping corresponding to it are returned.

3.2 Validity

A grouping is valid if it does not create a cycle between groups in the pipeline graph. Our algorithm does not merge nodes into a group if it creates a cycle. For simplicity, we excluded this check from Case I of Figure 5. For any s_j to be a valid merge candidate, it should not form a cycle with the respective H_i. Lines 9–13 check if grouping s_j with H_i would create a cycle. Only if no cycles are formed, s_j is grouped with H_i. Hence, Algorithm 1 always generates valid groupings.

3.3 Complexity

For a linear DAG, $|SUCC(G)|$ (where G is the argument to the DP function as we defined it in Section 3.1) is always one, since G will always have a single set H, i.e., $G = \{H_1\}$. The complexity of the algorithm is thus $O(|V|^2)$, since the number of possible configurations for G is $|V| * (|V| + 1)/2$. In other cases, the complexity of the algorithm is related to the number of sub-graphs of (V, E). Note that we do not consider configurations that create cycles. Coming up with an upper bound on the complexity is beyond the scope of this work, but it is clearly at least exponential. Intuitively, the number of choices depends on $|SUCC(G)|$ at any point. In practice, we see that $|SUCC(G)|$ is often a small value (Section 7). This combined with the bounded incremental variant of this DP algorithm that we will present in Section 6 allows us to always keep the running time within desired limits, no matter what the size of the graph is.

4 COST FUNCTION

In this section, we present the cost function that is used in conjunction with the dynamic programming algorithm presented in the previous section.

4.1 Locality, Parallelism, and Prefetching

The cost function COST(H) in Algorithm 2 is expected to provide the cost of grouping the nodes (image processing stages) of H with the best possible tile sizes that it estimates based on locality and parallelism considerations. It takes into consideration four criteria: (1) cache reuse, (2) number of cores available to run in parallel, (3) amount of redundant computation performed as a fraction of tile volume, and (4) the difference between the extents of the corresponding dimensions of the stages being fused. Each of these parameters is multiplied by a weight and summed up to form the cost as follows:

$$\begin{aligned} cost \quad = \quad & w_1 \times \text{ratio of live-in/live-out data to computation} \\ & -w_2 \times ((\text{num\_tiles} + \text{num\_cores} - 1)\% \text{ num\_cores}) \\ & +w_3 \times \text{fraction of overlap} \\ & +w_4 \times \text{relative difference between sizes of dimensions.} \end{aligned} \tag{2}$$

The lower the cost above, the more profitable the specific grouping being evaluated is. The first factor takes into account the ratio of live-in and live-out data to the computation; it is thus one way to calculate the inverse of the arithmetic intensity or the amount of cache reuse for a tile. The second factor represents the number of "cleanup" tiles, i.e., the remainder tiles whenever the number of tiles is not a multiple of the number of cores. This factor has to be minimized as well. The third factor captures the fraction of redundant computation introduced. The ratio of the first and the third factors capture the tradeoff between redundant computation and the improvement in locality as a result of it. The fourth factor considers the difference between the extents of the corresponding domain dimensions of different functions of the fused group; this is to avoid fusion if the difference in trip counts are too high.

ALGORITHM 2: Cost Function: Cost(H)

```
1:  function COST(H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores)
2:      if not constantDependenceVectors(H) then return ∞
3:      ⟨cost, tileSizes, overlapSize⟩ ←CostForCacheSize(H, L1CacheSize, NCores,
4:                                                                  InnerMostTileSize)
5:      if overlapSize > TileVolume(H, tileSizes) then
6:          ⟨cost, tileSizes, overlapSize⟩ ←CostForCacheSize(H, L2CacheSize, NCores,
7:                                                                  InnerMostTileSize)
8:      return ⟨cost, tileSizes⟩
9:
10: function CostForCacheSize(H, cacheSize, NCores, innerMostTileSize)
11:     liveout_size ← liveOutsSize(H)
12:     totalFootprint ← intermediateBuffersSize(H) + liveout_size
13:     tileFootprint ← min(totalFootprint÷NCores, cacheSize)
14:     tileSizes ← ComputeTileSizes(H, tileFootprint, innerMostTileSize)
15:     livein_tile_size ← liveInTileSize(H, tileSizes)
16:     liveout_tile_size ← liveOutTileSize(H, tileSizes)
17:     comp_vol ← ComputeTileVolume(H, tileSizes)
18:     n_tiles ← totalFootprint ÷ tileFootprint
19:     overlapSize ← OverlapSize(H, tileSizes)
20:     relative_overlap ← overlapSize ÷ tileFootprint
21:     dim_diff ← dimSizeStandardDeviation (H)
22:     cost ← w1×(livein_tile_size + liveout_tile_size)÷comp_vol− w2 × ((n_tiles + NCores−1) % NCores)
        + w3 × relative_overlap + w4 × dim_diff
23:     return ⟨cost, tileSizes, overlapSize⟩
24:
25: function ComputeTileSizes(H, tileFootprint, innerMostTileSize)
26:     tileVol ← tileFootprint ÷ numBuffers(H)
27:     nDims ← numDims(H)
28:     dimReuse[1 . . . nDims] ← getDimensionalReuse(H)
29:     dimSizes[1 . . . nDims] ← getDimensionalSizes(H)
30:     tileSizes[nDims] ← min(dimSizes[nDims], innerMostTileSize)
31:     τ ← tileVol ÷ tileSizes[nDims]
32:     maxDimReuse ← max(dimReuse[1:nDims − 1])
33:     for i = 1 → nDims − 1 do
34:         τ ← τ÷ (dimReuse[i] ÷ maxDimReuse)
35:     τ ← τ^(1/(nDims−1))
36:     for i = 1 → nDims − 1 do
37:         tileSizes[i] ← min(dimSizes[i], τ× dimReuse[i] ÷ maxDimReuse)
38:     return tileSizes
```

In Algorithm 2 function COST takes five arguments: group, L1 and L2 cache sizes, innermost dimension tile size, and the number of cores. It returns the cost of the group while also determining the associated tile sizes. COST returns ∞ if the dependence vectors in the given group are not constant (line 2); note that PolyMage is able to perform overlapped tiling (Figure 3) only if the scaling and alignment of all group functions' dimensions is possible in a way that makes the dependence vectors constant. Then, CostForCacheSize is called to first compute the cost and the corresponding best tile sizes for L1 cache at line 3. If the overlap size is not too high (less than the total volume of computation), then L1 tiling is chosen. Otherwise, line 6 calculates the

cost and the tile sizes for the L2 cache. Function CostForCacheSize calculates the cost and tile sizes of a group for a given cache size and number of cores. Line 12 calculates the total memory footprint, which is the sum of the size of intermediate outputs and live-outs. Lines 15–16 calculate the live-in and live-out data sizes associated with a tile. OverlapSize computes the overlap size at line 19 for the given tile sizes, which is the total volume of redundant computations in a tile due to its overlap with its neighboring tiles along all dimensions. A local buffer is not used for live-in data. ComputeTileSizes computes the actual tile sizes and is described in the next subsection (Section 4.2). We now explain how each of the desired optimization objectives is achieved.

Locality: Locality is captured in the cost function by considering the ratio of the number of loads and stores performed into/out of local buffers from main memory (or the L3 cache) to the volume of computation in a tile. Note that a tile comprises multiple stages successively processing a subset of the data. The local buffers are constrained (through tile sizes) to fit in L2 cache (or the L1 cache, as we will see later). Using such a ratio captures the amount of reuse in a tile for the fused group. This includes the producer-consumer reuse between stages, input reuse between stages, and reuse within a stage (self-temporal and group-temporal) [25]. We will see later that the relative degree of reuse across different dimensions is used to determine the ratio of tile sizes.

Prefetching: Since all intermediate data is stored in local buffers (which are contiguous and proportional to the tile sizes), spatial reuse is always exploited. In addition, the number of prefetch streams needed is greatly reduced as a result of contiguity. The cost function determines the best tile sizes for the group while ensuring a particular minimum tile size along the innermost dimension. This is to ensure that the subsequent prefetching and auto-vectorization performed by the compiler are effective.

Parallelism: A loss of parallelism while merging is also captured through the cost function. If the set of nodes being grouped cannot be aligned and scaled in a way that prevents overlapped tiling, an infinite cost is returned (line 2) and such a grouping is thus prevented. Algorithm 2 always ensures that we have at least as many tiles as the number of cores we intend to run on. In addition, the part of the expression that is weighted with w_2 also minimizes the number of "cleanup" tiles whenever the number of tiles is not a multiple of the number of cores. This minimizes load imbalance.

4.2 Tile Size Determination

We now describe how tile size selection works in Algorithm 2.

Our tile size determination algorithm is based on three parameters: (i) the L1 cache size, (ii) the L2 cache size, and (iii) reuse of data along each dimension. We first assign tile sizes such that the volume of data accessed by the tile fits in L1 cache. If this leads to an overlap size more than the tile size (for L1 cache), we determine tile sizes based on the L2 cache size. In addition, we ensure that the tile size for the innermost dimension is large enough for a profitable prefetching and auto-vectorization. As a result, we set the tile size of the innermost dimension to the minimum of size of the dimension and a fixed value that we call innerMostTileSize; this is typically either 128 or 256. For all other dimensions, tile sizes are set in a ratio based on reuse along the dimensions in a way we describe further below.

Reuse along a particular dimension (both temporal and spatial, group and self) is determined by inspecting data accesses using well-known techniques [25]. A *reuse score* is determined for each dimension. To keep the discussion simple here, we skip explaining an additional metric we consider that captures the degree of overlap a particular dimension incurs at the boundary, which is a function of dependence components along that dimension. Then, the tile size assigned for a dimension is proportional to reuse along it—the dimension with larger reuse will have larger tile

sizes. Such a strategy leads to tiles that are longer along dimensions with higher reuse. Let the ith dimension have a tile size of τ_i. Since there is a one-to-one correspondence between iterations and accessed data in the image processing pipelines that we currently support, for m dimensions, the product of tile sizes along all dimensions times the number of local buffers has to be equal to the allowable tile footprint, T (tileFootprint in Algorithm 2), i.e., $\tau_1\tau_2\cdots\tau_m = T$.

For the ith dimension, let γ_i (dimReuse[i] in Algorithm 2) be the ratio of the reuse score of the ith dimension to the maximum amount of reuse across all dimensions. Let τ be the tile size for the dimension with maximum reuse. We then set: $\tau_i = \gamma_i\tau$. Hence, we have: $\tau^m\gamma_1\gamma_2\cdots\gamma_m = T$, and τ can be determined.

Function ComputeTileSizes in Algorithm 2 is our tile size determination algorithm. ComputeTileSizes takes group, cache size, and innermost dimension tile size as parameters. The tile volume is then determined by dividing the allowable tile memory footprint with the number of intermediate buffers and live-outs (line 26). Function ComputeTileSizes is first called by CostForCacheSize to determine tile sizes that would fit in L1 cache. Lines 28–29 get the dimensional reuse scores and lengths for each dimension. Lines 30–37 assign tile sizes for all dimensions. Line 30 sets the tile size for the last dimension as the minimum of the dimension size and InnerMostTileSize. Lines 36–37 set tile sizes for the remaining dimensions weighted by per-dimension reuse, as described earlier.

5 OPTIMIZATIONS

This section describes additional optimizations, beyond the tiling and fusion described earlier, that we incorporated.

5.1 Expression Inlining

Inlining of producer expressions into consumer decreases the number of scratchpads required for storing intermediate. Inlining arbitrary expressions can increase the amount of redundant computations but inlining pointwise expressions does not add any redundant computations. We inline a producer expression to its consumer expression (i) if the consumer expression is a pointwise expression, (ii) if the producer expression is a pointwise expression, or (iii) a producer expression with branches, with each branch producing values for one pointwise consumer expression such that each condition of the branch has *non-overlapping references* to the input of producer expression. We make sure we do not inline the live-outs of a group.

Algorithm 3 presents our expression inlining algorithm. Our algorithm takes a DAG starting at node G and modifies that graph. Cost function calls the inlining algorithm for the given group to modify the graph after inlining expressions. Our algorithm executes while there are producers eligible for inlining into consumers (line 3). Line 5 iterates over all groups g in the DAG. If g is live-out, then we do not inline it at line 6. Line 8 inlines g in its consumers if g is a pointwise expression. Line 11 inlines g if there is only one consumer and that consumer is pointwise. Line 14 checks if all the consumers of g has non-overlapping references to input data.

5.2 Multi-level Tiling

Tiling for L2 cache only ensures that the generated code exploits reuse in the L2 cache, but not along all the dimensions in the L1 cache. While rectangular tiling for multiple levels for typical dense linear algebra is well-studied, we show how the specific tiling scheme used here for image processing pipelines is turned into a multi-level one.

Our multi-level tiling scheme can be described as follows: We create overlapped tiles for the L2 cache; these overlapped tiles are parallelized across CPU cores. For each overlapped tile in the L2 cache, our technique generates parallelogram tiles to exploit reuse in the L1 cache, i.e.,

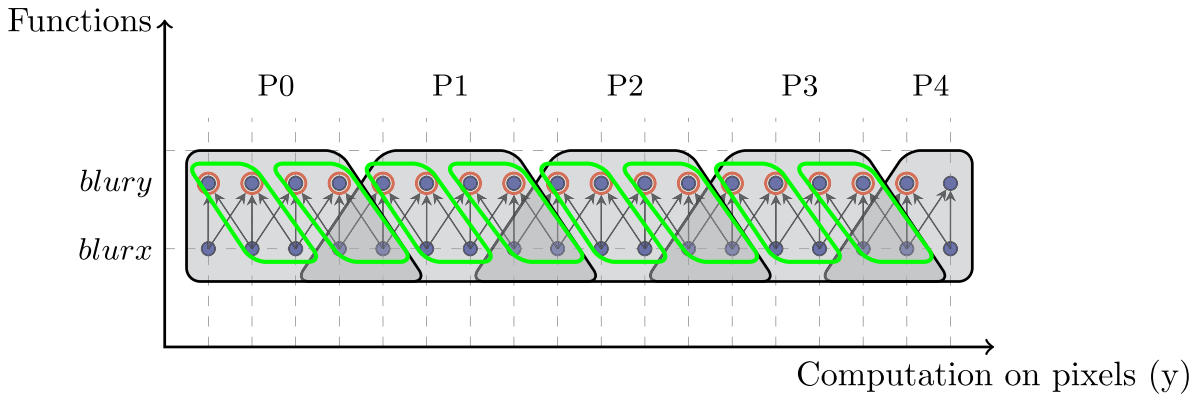

Fig. 7. Multi-level tiling for *blur*. Parallelogram tiles with green boundaries are inside overlap tiles in gray.

ALGORITHM 3: Expression Inlining Algorithm

```
1:  function EXPRESSION INLINING(G)
2:      workToDo ← true
3:      while workToDo do
4:          workToDo ← false
5:          for node ∈ G do
6:              if node.isLiveout() then
7:                  continue
8:              if node.isPointwise() then
9:                  node.inlineInAllConsumers()
10:                 workToDo ← true
11:             else if |node.consumers| = 1 and node.consumers[0].isPointwise() then
12:                 node.inlineInAllConsumers()
13:                 workToDo ← true
14:             else if |node.consumers| > 1 and node.nonOverlappingConsumers() then
15:                 node.inlineInAllConsumers()
16:                 workToDo ← true
```

each parallelogram tile's resident data set fits in the L1 cache. All parallelogram tiles in a larger L2 tile execute sequentially. We choose parallelogram tiles here for the inner level as opposed to overlapped tiles within overlapped tiles, since the dependences between the inner tiles are not an issue. Tile sizes at the outer level would have been chosen to provide sufficient parallelism from the overlapped tiles already. If the cost function we presented in Algorithm 2 chose to generate an overlapped tile for the L1 cache in the first place (instead of the L2 cache), then we choose not to perform any multi-level tiling, because the tiling for the L1 cache is used for the parallelization across cores.

The inner parallelogram tiles are generated with their slope parallel to the right hyperplane of the outer overlapped tiles, as shown in Figure 7. Note that if parallelogram tiles are generated with their slope parallel to the left hyperplane of the overlapped tiles, then there would be a cyclic dependence between successive tiles. However, generating tiles with the right hyperplane as the slope leads to no cyclic dependence among the tiles. Figure 7 illustrates multi-level tiling for the *blur* example. The parallelogram tiles are shown in green. Each of these parallelogram tiles is dependent on the left parallelogram tile. Hence, these tiles are executed from left to right sequentially, with each tile fitting in L1 cache.

Let ϕ_r be the right hyperplane of overlapped tiles and τ be the tile size for inner tiling, i.e., of parallelogram tiles. Let the scaled and aligned schedule for a function f_k in the group be $(\vec{i_k}) \rightarrow (\vec{s_k})$

ALGORITHM 4: Cost function for multi-level tiling

```
1:  function MultiLevelCost(H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores)
2:      if not constantDependenceVectors(H) then
3:          return ∞
4:      ⟨cost, L1tileSizes, overlapSize⟩ ←CostForCacheSize(H, L1CacheSize, NCores,
5:                                                                  InnerMostTileSize)
6:      if overlapSize < TileVolume(H, tileSizes) then
7:          return ⟨cost, L1tileSizes⟩
8:      ⟨cost, L2tileSizes, overlapSize⟩ ← CostForCacheSize(H, L2CacheSize, NCores,
9:                                                                  InnerMostTileSize)
10:     return ⟨cost, ⟨ L1tileSizes, L2tileSizes⟩⟩
```

after the overlapped tiling has been applied. The newly added dimension t corresponding to the iterator on the tile space will be given by:

$$\tau \cdot t \leq \phi_r(\vec{s_k}) \leq \tau \cdot (t+1) - 1.$$

The schedule for f_k can then be updated to $(\vec{i_k}) \rightarrow (t, \vec{s_k})$.

Algorithm 4 presents our cost function for multi-level tiling. Similar to COST in Algorithm 2, MultiLevelCost takes a group H, L1 cache size, L2 cache size, innermost dimension tile size, and the number of cores as parameters and returns the cost of the group with single-level or multi-level tile sizes. When "multi-level" tiling is enabled, MultiLevelCost is called instead of COST. Similar to COST, MultiLevelCost proceeds only if dependence vectors are constant at line 2. Line 5 calls CostForCacheSize to get the tile sizes for L1 cache. If overlap size is less than or equal to the tile sizes, then we return the L1 tile sizes and the cost at line 7. Otherwise, we determine the cost and tile sizes for L2 cache at line 9 and return the concatenated tile sizes at line 10.

5.3 Intra-tile Loop Fusion for Register Reuse

In addition to performing fusion across loops to create overlapped tiles, we can fuse loops inside a tile to increase register reuse. In a group, more than one consumer could have the same producer function and the set of references belonging to the producers' live-outs could overlap across consumers. In this case, we can fuse loops of all consumers to enable register reuse of the output from the producer function, thereby decreasing the number of loads from memory. We fuse loops of two or more consumer functions when (i) all consumers have the same nesting depth, (ii) the loops bounds of all consumer functions at all nesting depths are equal, and (iii) the intersection between the references of elements in producer's live-out for all consumers is not empty.

This schedule transformation is performed for each group after the grouping has been performed and before code generation. For each group, we enumerate all combinations of two functions f, g in the group and fuse the loops of f and g if (i) f and g have common inputs and the intersection of the references to the input are common in each iteration, and (ii) f and g have the same loop bounds.

Note that the fusion-cum-tile size determination approach described in previous sections models cache locality as one of the criteria. This fusion of intra-tile loops, however, enhances register reuse over an already cache locality optimized choice. As such, register reuse is only indirectly modeled as part of the DP-based approach, and we do not add an additional term to our weighted cost formula (2) in Section 2 to capture the fact that intra-tile loop fusion may be possible and profitable for a particular grouping choice.

ALGORITHM 5: Bounded Incremental Dynamic Programming Grouping

```
1:  function INC-GROUPING(G, initialLimit, step)
2:      groupLimit ← initialLimit
3:      max_size ← initialLimit
4:      T ← Dictionary⟨G, float⟩()
5:      G′ ← G
6:      while true do
7:          ⟨cost, G′⟩ ← DP-GROUPING-BOUNDED({{source(G′)}}, T, groupLimit)
8:          if max_size ≥numVertices(G) then
9:              break
10:         groupLimit ← step
11:         max_size ← step × max_size
12:     return ⟨cost, G′⟩
```

6 BOUNDED INCREMENTAL GROUPING

The DP-based grouping algorithm described in Section 3 can lead to significant execution time as described in Section 3.3. To reduce the number of choices explored and thereby the compilation time, we introduce a variant of the DP grouping algorithm.

The key idea is to produce a grouping with a size constraint, which we call the *group limit*, and then perform subsequent grouping on top of the groups from the previously grouped graph where groups have already been coalesced into single vertices. This process can be performed iteratively, either with a specific group limit each time or with the group limit being increased multiplicatively by a factor.

Algorithm 5 presents our incremental dynamic programming algorithm. The *groupLimit*, l, is increased until it becomes larger than the number of nodes in the pipeline graph. INC-GROUPING is the driver function that calls DP-GROUPING-BOUNDED to find the best grouping subject to the size of groups being bounded by l. DP-GROUPING-BOUNDED is a slight variant of DP-GROUPING (Algorithm 1) that does not allow any groups H_i with size greater than l; DP-GROUPING-BOUNDED thus takes *groupLimit* as an additional parameter. INC-GROUPING takes three parameters: the source node of pipeline graph, initial group size limit, and the factor by which the group size limit should be increased every iteration. The iterative part of the algorithm (lines 6–11) runs for a sufficient number of iterations to allow a grouping where all nodes of the original pipeline graph could be grouped. In each iteration of the loop (lines 6–11), a grouping is performed on the DAG returned at line 7 with group limit set to *step*. The grouping obtained from the previous iteration is used for the next iteration to explore further grouping opportunities over groups already formed from the previous iteration.

This algorithm allows us to collapse arbitrarily large graphs into smaller ones efficiently (by choosing small l), while using DP-GROUPING in an incremental and iterative manner.

7 EXPERIMENTAL EVALUATION

The proposed fusion-cum-tile size algorithm and optimizations have been integrated into PolyMage [16]. In the rest of this section, we refer to our implementation as **PolyMage+**, i.e., PolyMage with our DP-based fusion-cum-tile size determination and the additional optimizations presented. All presented optimizations including the incremental dynamic programming-based grouping (Section 6) were implemented.

Table 1. Benchmark Summary, Problem Size, Maximum Number of Successors in Graph, Number of Fusion Choices Evaluated for Different l Values, and Time Taken by Grouping Algorithm

Benchmark	Stages	Image size (W×H×c)	max(succ(G))	Groupings enumerated				Time(s)			
				$l = \infty$	32	16	8	$l = \infty$	32	16	8
Unsharp Mask	4	4,256 × 2,832 × 3	2	10	–	–	–	0.05	–	–	–
Harris Corner	11	4,256 × 2,832	2	104	–	–	–	0.15	–	–	–
Bilateral Grid	7	1,536 × 2,560	1	16	–	–	–	0.02	–	–	–
Multi. Interp.	49	1,536 × 2,560 × 3	2	741	–	–	–	3.00	–	–	–
Camera Pipe.	32	2,592 × 1,968	5	12,227	12,227	3,825	1,631	13.7	13.7	5.10	1.0
Pyramid Blend	44	3,840 × 2,160 × 3	3	27,108	26,952	7,809	923	25.7	25.0	10.3	0.3

"–" for certain l values indicates that the algorithm took reasonable time with $l = \infty$ itself; hence, lower l values were not required.

7.1 Experimental Setup

Evaluation was performed on the following two state-of-the-art multicore systems:

Intel Xeon (Haswell): A dual-socket NUMA server with the 8-core Intel Xeon E5-2630 v3 processor based on the Intel Haswell microarchitecture running at 2.40 GHz with 64 GB DDR4 2400 MHz RAM, 32 KB L1 cache, 256 KB L2 cache, and a 20 MB shared L3 cache. The OS is 64-bit Linux kernel 3.10.0. The Haswell architecture supports 256-bit AVX2 vector instruction set. All experiments were conducted with hyperthreading disabled.

AMD Opteron: This is a server with a 16-core AMD Opteron 6386 SE processor running at 1.4 GHz with 128 GB of DDR3 800 MHz RAM, running 64-bit Linux kernel 3.10.0 version. The AMD Opteron 6386 SE processor uses a 16 KB L1 cache, 2 MB of L2 cache shared between two cores, and an 8-core shared 12 MB L3 cache.

Code generated by PolyMage and PolyMage+ were compiled with Intel C/C++ compilers (icc/icpc) 16.0.0 with flags "-O3 -xhost -openmp" on the Intel Haswell server and with GCC/G++ 4.8.2 with flags "-O3 -march=native -fopenmp -ftree-vectorize" on the AMD Opteron. All experiments were conducted with five sample runs with each sample using 500 runs. We report the minimum of the average of each sample. We used six image processing benchmarks, which were also used earlier for evaluation in papers on Halide and PolyMage [13, 14, 19, 20], and eight multigrid benchmarks used earlier in Reference [24]. The parameter InnerMostTileSize (Algorithm 2) was set to 256 for the Xeon system and to 128 for the Opteron.

7.2 Image Processing Benchmarks

Unsharp Mask is a simple pipeline used to sharpen image edges, and comprises a series of image blur operations. *Harris Corner Detection* is an implementation of theHarris corner detection algorithm combining several stencils and point-wise operations. *Multiscale Interpolation* interpolates image pixel values using an image pyramid of 10 pyramid levels. *Bilateral Grid* is used for computing a fast approximation of the bilateral filter. *Camera Pipeline* processes raw images captured by the camera into a color image. The pipeline stages have stencil-like, interleaved, and data-dependent access patterns. *Pyramid Blending* blends two images into one using a mask and constructing a Laplacian pyramid of four levels. The number of stages in each benchmark and the image size used are reported in Table 1.

Bounded Incremental Grouping. Table 1 shows how the time taken by the bounded incremental DP algorithm (Section 6) changes when l, the group limit, is changed. It shows that our fusion heuristic always runs in an acceptable amount of time when the group limit is $l \leq 32$, through

Table 2. Weights for Intel Xeon and AMD Opteron

System	w_1	w_2	w_3	w_4
Intel Xeon	1.0	100.0	46,875	1.5
AMD Opteron	0.3	100.0	46,875	2.0

Table 3. Normalized Weights for the Cost Function

System	w_1	w_2	w_3	w_4
Intel Xeon	47.04	1	13.18	33.57
AMD Opteron	14.11	1	13.18	44.76

which a grouping is obtained, and the subsequent iteration is run again without any group limit, i.e., with $l = \infty$.

7.3 Cost Function Weights

We obtained the values for weights w_1, w_2, w_3, w_4 in Section 4 using three image processing benchmarks: Harris Corner, Multiscale Interpolation, and Camera Pipeline. We choose these benchmarks, because each of these benchmarks exercise all parts of the cost function. The weights were obtained manually through an empirical trial of these benchmarks in two steps. We first obtained the performance of the three training benchmarks using an initial guess of the weights. Then, we manually adjusted the weights by increasing (or decreasing) the weight of a component, depending on whether the grouping obtained had a high (or low) value for that component. For example, if the grouping had a high overlapping computation, we increased w_3. However, if there is a low overlap, w_3 could have been higher than necessary.

Table 2 shows the values of the weights we used on each of the two systems. These weights were used in the evaluation of all image processing and multigrid benchmarks.

Relative importance of each cost component. To better understand the relative importance of each metric involved in the cost function, we performed a scaling of the weights. For this, we first collected the value of each component for all the groupings explored by our algorithm in all six image processing benchmarks. Then, we filtered the outliers by selecting values that were not in between two standard deviations around the mean for each component. This process filtered out less than 5% of the collected values of each component. After filtering, we modified the cost function so the value of each component in the filtered range was normalized to a 0 to 1 range, and the weights were scaled based on that such that the final cost remained the same as with the unscaled weights in Table 2. The normalized weights are given in Table 3.

7.4 Performance Analysis for Image Processing Benchmarks

In this section, we report and analyze the performance improvements obtained.

7.4.1 Comparison Baselines. We compare the performance of PolyMage+ against Halide with expert-tuned schedules, Halide with automatically generated schedules [13], and PolyMage with auto-tuning. The version of PolyMage compared against was the same base version [16] that we used for implementing our approach. For Halide, the latest version as of this writing (git commit[1]) was used, both for auto-scheduling as well as expert-tuned Halide schedules in its repository. We found this current version to be providing significantly better performance than that reported in

[1]https://github.com/halide/Halide/commit/89679918b42eb14d358a8e6214755de1e42ff046.

Table 4. Execution Times (in ms) of Image Processing Benchmarks on Intel Xeon Haswell on 1 and 16 Cores

Benchmark	H-manual		H-auto		PolyMage-A		PolyMage+		Speedup of PolyMage+ on 16 cores over		
	1	16	1	16	1	16	1	16	H-manual	H-auto	PolyMage-A
Unsharp Mask	159	20.4	76.4	17.1	105	19.7	89.3	8.71	2.34	1.96	2.26
Harris Corner	257	33.0	111	10.7	94.5	19.8	31.4	3.10	10.64	3.45	6.39
Bilateral Grid	66.1	6.47	78.3	6.13	84.9	7.66	84.7	7.3	0.88	0.84	1.05
Mult. Interp.	108	35.3	141	18.3	101	14.2	77.1	11.2	3.15	1.63	1.27
Camera Pipe.	34.2	3.60	36.8	5.10	52.7	4.40	51.4	4.05	0.89	1.26	1.09
Pyramid Blend	195	67.5	175	33.7	196	20.2	184	18.0	3.75	1.87	1.12

Table 5. Execution Times (in ms) of Image Processing Benchmarks on AMD Opteron on 1 and 16 Cores

Benchmark	H-manual		H-auto		PolyMage-A		PolyMage+		Speedup of PolyMage+ on 16 cores over		
	1	16	1	16	1	16	1	16	H-manual	H-auto	PolyMage-A
Unsharp Mask	270	74.7	135	60.04	298	83.87	260	32.31	2.31	1.86	2.59
Harris Corner	432	57.8	142	46.68	266	87.80	194	20.32	2.85	2.30	4.32
Bilateral Grid	167	17.1	121	13.16	491	47.31	480	46.12	0.37	0.26	1.03
Multi. Interp.	266	153	157	37.91	245	58.11	234	51.40	2.98	0.74	1.13
Camera Pipe.	39.0	5.80	58.0	14.31	190	19.20	210	21.30	0.27	0.67	0.90
Pyramid Blend	443	366	234	169.1	325	73.44	343	68.70	5.33	2.46	1.07

its publication [13]; it has thus significantly improved and is a strong baseline. LLVM 3.9 was the backend used with Halide. PolyMage+ is compared in the following ways with the state-of-the-art on all benchmarks.

H-manual: We compare with the manually tuned schedules for these benchmarks available in the Halide repository. These schedules were also used earlier to evaluate Halide and PolyMage.

H-auto: We compare with the tuned schedules generated by Halide's auto scheduler [13]. The parameters of Halide Auto Scheduler (mentioned in Reference [13]) were set in the following way: For both Intel Xeon and AMD Opteron systems, `VECTOR_WIDTH` was set to 16, twice the native vector width of AVX and AVX2 vector instructions for 32-bit floating point data; `PARALLELISM_THRESHOLD` is set equal to the core count, i.e., 16. For the Intel system, `CACHE_SIZE` is set to 256 KB (the per core L2 cache size), and for the AMD Opteron, it is set to 1 MB, which is half the size of 2-core shared L2 cache. Note that Halide only uses the size of one cache level. `LOAD_COST` is set to 40, which is a rough estimate of the relative cost of DRAM load vs. compute on modern multi-core machines.

PolyMage-A: We compare with the generated PolyMage code tuned by the PolyMage's auto-tuner, which represents state-of-the-art with respect to PolyMage [14, 24]. For auto-tuning in PolyMage, we used three threshold values, 0.2, 0.4, and 0.5, and seven tile sizes: 8, 16, 24, 128, and 256, for tiling of only two dimensions. This is the parameter space that was also explored in Reference [14].

7.4.2 Performance Improvement and Analysis. Table 4 shows the absolute execution times on the Haswell system for implementations generated by all configurations and the speedup of implementations generated by PolyMage+ over H-manual, H-auto, and PolyMage-A. Similarly, Table 5 shows the execution times and speedup on the Opteron system.

PolyMage+ provides a significant speedup over H-manual, H-auto, and PolyMage-A on both the Intel Xeon and the AMD Opteron system as a result of better tile sizes and groupings. Since both Halide and PolyMage/PolyMage+ use different backend compilers, we conducted additional experiments to isolate performance benefits to grouping and tile sizes. For *Harris Corner*, we found that if the grouping generated by PolyMage+ is used in H-Manual (without the storage optimizations performed by PolyMage+, since there is no way to specify storage mappings explicitly with Halide), the latter runs in 12.6 ms instead of 33.0 ms (on the Intel Xeon). Furthermore, if we use PolyMage+'s tile sizes in H-manual in addition to the former's grouping, it runs in 8.8 ms (better than H-auto).

Groupings generated by PolyMage+ are different from the ones generated by PolyMage-A for all benchmarks. Except for *Unsharp Mask* and *Harris Corner*, the groupings generated by PolyMage+ are different from the ones used by H-auto. Due to space constraints and the large number of stages involved in the benchmarks, we are unable to list the grouping configuration and the corresponding tile sizes. Note that the auto-tuning in PolyMage-A typically takes from a few minutes to up to 27 minutes for these benchmarks. However, PolyMage+ is completely model-driven.

For *Bilateral Grid*, PolyMage+ and PolyMage-A perform slightly worse than H-manual and H-auto, because both H-manual and H-auto group the *histogram* function, which is a reduction operation, with other computations. However, PolyMage-A and PolyMage+ do not yet group or optimize reductions in any way. For *Camera Pipeline*, H-manual performs slightly better than PolyMage-A and PolyMage+ because H-manual has aggressive inlining of several functions, which PolyMage-A and PolyMage+ currently do not support.

We also investigated the low performance of PolyMage+ in three cases on the AMD Opteron (Table 5: *Camera Pipeline*, *Bilateral Grid*, and *Multiscale Interpolation*) and found that to be an issue with compiler auto-vectorization. Note that PolyMage-A and PolyMage+ generated code rely on icpc or g++'s auto-vectorization, while Halide generated code uses vector intrinsics and was not affected by this issue. The cases here involved more challenging patterns, including integer operations, for which the evaluated Opteron did not have AVX2. In these cases, we confirmed that compiler auto-vectorization on the Opteron did not provide any improvement, while a significant improvement close to 2× was obtained on the Intel Xeon for the same code running on 16 cores. In these benchmarks, although PolyMage+ and PolyMage-A perform worse than H-manual and H-auto due to vectorization, PolyMage+ performs better than PolyMage-A in nearly all these benchmarks, supporting our claim that the DP-based approach performs better than or is competitive with an auto-tuned approach. For *Pyramid Blend*, g++ was not able to perform any vectorization; hence, if auto-vectorization support is improved, we expect even better results for *Pyramid Blend*.

7.4.3 Analysis and Impact of Individual Optimizations. Figure 8 shows the performance of comparing various configurations of PolyMage+ over all six image processing benchmarks on different threads to provide insights into the benefits of dynamic programming with cost function, multi-level tiling, expression inlining, and register reuse fusion. The configurations of PolyMage+ are:

- *PolyMageDP* represents PolyMage with our dynamic programming-based cost function.
- *PolyMage+G+TC* represents PolyMage with Halide's greedy fusion algorithm but with PolyMageDP's cost function with inlining and multi-level tiling.
- *+ F* represents the addition of fusion for register reuse to PolyMageDP.
- *+ M* represents the addition of multi-level tiling to PolyMageDP.
- *+ I* represents the addition of expression inlining to PolyMageDP.

In this case, the implementations were generated for 16 threads and then executed on 1, 4, and 16 cores on the Intel Xeon. The baseline is the single threaded (sequential) version of the respective

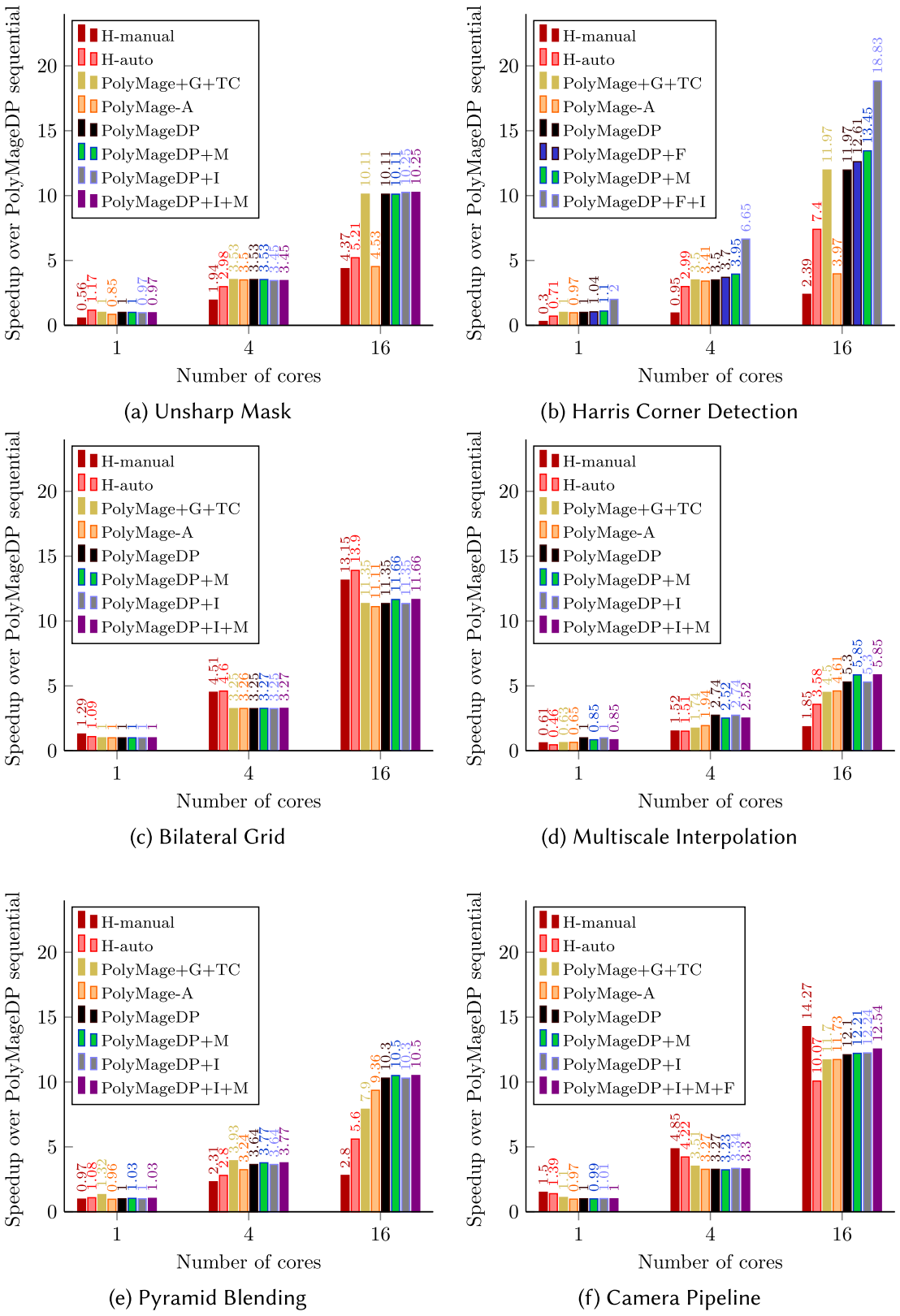

Fig. 8. Speedups of H-auto, H-manual, PolyMage-A, and several PolyMage+ configurations relative to PolyMageDP single-threaded version. Section 7.4.3 explains each PolyMage+ configuration in detail. Absolute execution times can be determined in conjunction with Table 4.

Table 6. Fraction of Cache Hits/misses (in %) over Total Cache Accesses on Intel Haswell for Unsharp Mask

Tile size	L1_HIT	L2_HIT	L2_MISS	Runtime (ms)
128 × 256	83.43	5.04	11.52	10.7
16 × 256	82.05	12.36	5.59	10.3
8 × 416	83.34	11.2	5.46	9.3
5 × 256	95.55	1.50	2.85	8.8

Table 7. Fraction Cache Hits/misses (in %) over Total Cache Accesses on Intel Haswell for Harris Corner for L2 tile Sizes and Multi-level Tile Sizes

Tile size	L1_HIT	L2_HIT	L2_MISS	Runtime (ms)
42 × 256	88.08	8.10	3.82	6.4
32 × 256	85.45	10.84	3.70	6.59
42×5×256	96.85	1.53	1.61	5.8

benchmark generated by PolyMageDP. We explain below the results of each optimization on each benchmark:

Tile size selection schemes. To provide greater insight on the benefits of the contributions, we separately also compare the benefits of the tile size selection algorithms of PolyMageDP and H-auto. Figure 8 also provides results for PolyMage+G+TC, which is PolyMage using H-auto's greedy fusion algorithm but with PolyMageDP's cost function. This could be compared to H-auto to understand improvements without the dynamic programming-based approach. For *Unsharp Mask* and *Harris Corner Detection*, both PolyMage+G+TC and H-auto generate the same grouping and perform expression inlining. For *Unsharp Mask*, PolyMage+G+TC's generated schedule performs 1.96× better, because the tile size generated by PolyMage+G+TC is for the L1 cache and is not a power of two; however, H-auto generates tile size for the L2 cache that is a power of two. Table 6 shows the tile size generated by PolyMage+G+TC leads to a lower cache miss ratio than any other tile size. The L1 tile 5 × 256 has a smaller fraction of L2 misses than the best performing L2 tile 8 × 416; 128 × 256 and 16 × 256 are two sub-optimal L2 tiles, with the former "spilling" the L2 cache and the latter underutilizing its capacity. This shows that it is better to perform L1 tiling whenever L1 tile sizes are not constrained to be too small to adversely affect prefetching and redundant computation. Our heuristic automatically takes this into account. For *Harris Corner Detection*, PolyMage+G+TC's generated schedule performs 3.45× faster than H-auto's generated schedule, because the tile size generated by PolyMage+G+TC for the L2 cache is not a power of two. The tile sizes generated by H-auto are again powers of two here. Table 7 shows that the tile sizes generated by PolyMage+G+TC lead to a lower L2 cache miss rate. For all other benchmarks, PolyMage+G+TC provides improvements from 16% for *Camera Pipeline* to 41% for *Pyramid Blending*. These results show that PolyMageDP's cost function on its own produces better tile sizes than H-auto.

Impact of fusion approach. To compare the effectiveness of dynamic programming-based fusion and previous greedy fusion heuristics, we compare the performance of PolyMageDP and PolyMage+G+TC (which is PolyMage using H-auto's greedy fusion algorithm but with PolyMageDP's cost function). Figure 8 shows the results of PolyMageDP's and PolyMage+G+TC. For *Unsharp Mask*, *Harris Corner Detection*, and *Bilateral Grid*, there is no difference in performance, because

Table 8. Geomean Speedups of all Optimizations on Image Processing Benchmarks

Optimization	Cost function	Dynamic programming fusion	Intra-tile loop fusion	Multi-level tiling	Inlining
Geomean speedup	1.30× (1.30×)	1.08 × (1.17×)	1.04× (1.02×)	1.04× (1.05 ×)	1.07× (1.08 ×)

Each column is represented as XX× (YY×). XX× is the geomean speedup on all six benchmarks and (YY×) is the geomean speedup on benchmarks where the optimization generated a different grouping and tile sizes.

both PolyMage+G+TC and PolyMageDP produce the same grouping with the same tile sizes. However, for all other benchmarks, PolyMageDP provides improvements from 4% for *Camera Pipeline* to 30% for *Pyramid Blending*. These results support our claim that a dynamic programming-based fusion could provide better performance than greedy fusion algorithm even with the same powerful cost function. This is again because the dynamic programming-based approach explores more fusion possibilities than the greedy one; in fact, the former explores nearly all interesting valid fusion possibilities as described earlier.

Intra-tile loop fusion. The fusion of intra-tile loops to enable register reuse improves performance in *Unsharp Mask, Harris Corner,* and *Camera Pipeline*. Since no such opportunities are present in other benchmarks, there is no improvement in performance.

Multi-level tiling. Multi-level tiling improves performance for each benchmark, except for *Unsharp Mask,* where there is no improvement. Without multi-level tiling an L1 overlap tile is generated for *Unsharp Mask,* hence, multi-level tiling is not performed. Table 7 shows the performance for two L2 tiles and a multi-level L1/L2 tile. Tile size 42 × 256 is the best performing L2 tile, because it perfectly fits in the L2 cache and hence has a better L1 and L2 hit rate than a sub-optimal fitting 32 × 256 L2 tile. A multi-level L1/L2 tile 42×5×256 gives best performance, because it provides locality for both L1 and L2 caches. This tile gives best L1 hit rate and lowest L2 Miss rate as compared to other tile sizes, because the overlapped tile of size 42 × 256 perfectly fits in L2 cache, while smaller parallelogram tiles of size 32 × 256 underfit L2 cache.

Inlining. Expression inlining improves performance in *Unsharp Mask, Camera Pipeline,* and gives significant improvement in *Harris Corner. Harris Corner* contains several pointwise computations and computations with pointwise consumers. Our inlining algorithm inlines all these computations to reduce the number of groups from 11 to 3. Since there are no inlining opportunities in *Multiscale Interpolation, Bilateral Grid,* and *Pyramid Blend,* there is no improvement in performance for these benchmarks.

Summary. Table 8 presents the geomean speedups of each of the optimizations.

7.5 Multigrid Benchmarks

We also performed evaluation on Multigrid benchmarks that solve the Poisson's equation:

$$\nabla^2 u = f, \tag{3}$$

where ∇ is the vector differential operator, and u and f are real functions. The Poisson's equation is a second-order elliptic partial differential equation of fundamental importance to electrostatics, mechanical engineering, and physics. We solve the Poisson's equation for two-dimensional and three-dimensional data grids using V-cycle and W-cycle. These benchmarks are also used for evaluation by Vasista et al. [24]. We use two smoothing configurations, 4-4-4 and 10-0-0, for each cycle; hence, there are eight benchmark configurations. We also evaluate Minifluxdiv benchmark from Davis et al. [3]. Table 9 shows the multigrid benchmarks with their problem sizes, number of stages, and time taken by the grouping algorithm.

Table 9. Multigrid Benchmarks: Problem Size, Number of Fusion Choices Evaluated for Different l values, and Time Taken by the Grouping Algorithm

Benchmark	Stages	Grid size	Groupings enumerated				Time(s)			
		(W × H × c)	$l = \infty$	32	16	8	$l = \infty$	32	16	8
Minifluxdiv	4	128 × 128	10	–	–	–	0.05	–	–	–
Jacobi2D-V-4-4-4	34	8,192 × 8,192	2,926	1,936	1,096	580	76	33.6	9.2	3.5
Jacobi2D-V-10-0-0	7	8,192 × 8,192	595	–	–	–	0.07	–	–	–
Jacobi2D-W-4-4-4	102	8,192 × 8,192	415	306	305	195	30	28	25	20
Jacobi2D-W-10-0-0	76	8,192 × 8,192	2,926	1,936	1,096	580	75.7	35.9	8.7	3.6
Jacobi3D-V-4-4-4	37	256 × 256	703	688	472	268	0.06	–	–	–
Jacobi3D-V-10-0-0	36	256 × 256	666	656	456	260	0.06	–	–	–
Jacobi3D-W-4-4-4	109	256 × 256	435	383	199	47	80	65	45	30
Jacobi3D-W-10-0-0	79	256 × 256	3,160	2,032	1,144	604	139	59	16.2	5.6

"–" for certain l values indicates that the algorithm took reasonable time with $l = \infty$ itself; hence, lower l values were not required.

Table 10. Execution times (in ms) of T-stencils on Intel Xeon Haswell on 1 and 16 Cores

Benchmark	PolyMG-A		PolyMage+		Speedup of PolyMage+ on 16 cores over PolyMG-A
	1	16	1	16	
Minifluxdiv	121	28.7	118	20.7	1.39
Jacobi-2D-V-4-4-4	17.3	3.52	18.5	4.2	0.84
Jacobi-2D-V-10-0-0	12.9	3.15	14.68	3.54	0.89
Jacobi-2D-W-4-4-4	61.8	12.82	61.8	12.1	1.06
Jacobi-2D-W-10-0-0	23	5.25	18.74	5.23	1.0
Jacobi-3D-V-4-4-4	5065	490	3760	489	1.0
Jacobi-3D-V-10-0-0	5643	651	4413	625	1.04
Jacobi-3D-W-4-4-4	4153	824	3750	521	1.58
Jacobi-3D-W-10-0-0	5710	843	4196	536	1.57

We compare PolyMage+ with PolyMG [24]—the latter represents the state-of-the-art for multigrid stencils. PolyMG is based on PolyMage and adds several optimizations over PolyMage for multigrid benchmarks. We compare with code generated by PolyMG and auto-tuned with PolyMG's auto-tuner. For auto-tuning in PolyMG, we used three threshold values: 0.2, 0.4, and 0.5, and seven tile sizes, 8, 16, 32, 64, 128, and 256, for tiling of only two dimensions. This is the parameter space that was also explored by Vasista et al. [24].

Table 10 shows the performance on Multigrid benchmarks on the Intel Xeon. PolyMage+ is considerably faster than PolyMage-A on four benchmarks, and on the other five benchmarks, PolyMage+ is competitive to PolyMage-A. For these benchmarks, we found that there are no inlining and inner fusion opportunities. Moreover, PolyMage+ prefers tile sizes that fit in L1 cache, and thus providing L2 locality already without multi-level tiling. Table 11 shows the results for multigrid benchmarks on AMD Opteron. Even on the AMD Opteron, PolyMage+'s code is competitive to PolyMG's auto-tuned code. Note that auto-tuning in PolyMage-A takes from about 10 minutes to 30 minutes for these benchmarks. However, PolyMage+ takes a few seconds to about 2 minutes to generate code competitive with PolyMage-A.

Table 11. Execution Times (in ms) of T-Stencils on AMD Opteron on 16 Cores

Benchmark	PolyMG-A	PolyMage+	Speedup of PolyMage+ on 16 cores over PolyMG-A
Minifluxdiv	200	199	1.00
Jacobi-2D-V-4-4-4	8.09	8.45	0.96
Jacobi-2D-V-10-0-0	8.83	8.79	1.00
Jacobi-2D-W-4-4-4	25.3	25.0	1.01
Jacobi-2D-W-10-0-0	12.6	12.5	1.00
Jacobi-3D-V-4-4-4	1,029	1,010	1.02
Jacobi-3D-V-10-0-0	1,630	1,900	0.86
Jacobi-3D-W-4-4-4	2,544	2,530	1.00
Jacobi-3D-W-10-0-0	2,171	2,400	0.90

8 RELATED WORK

As discussed in detail and through examples in Section 2, our fusion approach addresses limitations in recently proposed state-of-the-art fusion heuristics in domain-specific compilers of image processing pipelines—those of Halide [13] and PolyMage [14]. Other than for image processing pipelines, prior work has shown that fusing across sequences of loop nests within a single time iteration in stencil codes (for example, in iterative numerical solvers), demonstrates significant performance improvement [1, 15, 21, 28].

Fusion approaches used in general-purpose optimizing compilation [2, 5, 8, 9, 18, 22, 27] have extensively dealt with aspects such as preserving parallelism while trying to maximize fusion for locality. In contrast to what such approaches addressed, being domain-specific allows us to use a large amount of information on what transformations are going to be subsequently applied on the fused components; this in turn allowed us to integrate it well with the tiling technique—by evaluating the cost of a grouping while considering the ratio of computation to loads/stores, the amount of redundant computation, and other factors—to determine tile sizes in conjunction with fusion. To the best of our knowledge, this is also the first dynamic programming-based approach to perform fusion and tile size selection for a class of multi-dimensional loop nests. There has also been prior work on determination of tile sizes for affine loop nests via analytical as well as empirical approaches [12, 23]. For the subset of affine loop nests that are relevant in the context of image processing pipelines, tile size determination for a specific grouping itself becomes quite simple: Our cost function does not evaluate tile sizes in isolation after having determined a fusion, but only indirectly as part of the cost function of the outer dynamic programming-based algorithm. While determining the tile sizes to use for a specific grouping, we simply ensure that the associated footprint of the tile maximally utilizes the effective cache capacity (Algorithm 2).

Megiddo and Sarkar [11] developed an integer programming-based approach for weighted loop fusion while (a) ensuring that parallelism was not lost and (b) the sum total of inter-cluster edge weights were minimized. Their cost function evaluates the benefits of having less edges (or interaction) across groups, thus maximizing locality inside a group. Our cost function is much more concrete and tackles the problem in the opposite direction—by evaluating what is inside the group; it looks at the ratio of memory load/store to compute, is aware of the specific tiling scheme used, and captures the tradeoff between redundant computation and locality. Formulating the cost function based on the group's composition thus allows us to incorporate more precise optimization criteria.

Davis et al. [3] recently presented an approach for automatic fusion of data nodes in Partial Differential Equation (PDE) solvers using a modified macro dataflow graph and a cost model to decide which nodes to fuse based on the amount of data read and the number of streams being accessed simultaneously. The particular execution order employed therein in conjunction with tiling is slightly different from that of PolyMage. Their technique achieves better performance than PolyMage due to better vectorization and better reduction in temporary storage, both of which are complementary to our work. However, their approach still requires auto-tuning, while our is fully model-driven. In addition, our approach captures additional criteria in its cost model and includes additional optimizations (multi-level tiling and inlining).

The recent polyhedral expression propagation work of Doerfert et al. [4] is related to the inlining optimization that we incorporated. Their approach takes a general view of expression propagation that is applicable in other contexts beyond inlining. However, their expression propagation technique is not integrated with other aspects of optimization for locality and parallelism. Such expression propagation by itself was reported to provide significant performance improvements, but in absolute terms lower than the improvements we obtain here over a naive parallelization. As such, it is important to be able to reason about and model multiple optimization criteria.

This work is a significantly extended version of our previous work [7]. This extension has primarily included in addition a multi-level tiling scheme, an expression inlining optimization, an evaluation on the Multigrid benchmarks, and a clear characterization of the additional benefit of each optimization for all the benchmarks.

9 CONCLUSIONS

We presented a new model for fusion and tile size determination in a high-performance domain-specific compiler for image processing pipelines. Our approach was driven by dynamic programming with concrete cost evaluation criteria. The approach is fully analytical and is able to consider spaces of valid fusion possibilities and tile sizes not covered by previous approaches. Its implementation in PolyMage obtains a significant improvement of up to 4.32× over PolyMage's current approach (even when auto-tuning was used with the latter) and up to 2.46× over Halide's automatic approach on two state-of-the-art multicore architectures. Our implementation and the benchmarks are available as part of PolyMage's public source code repository.

ACKNOWLEDGMENTS

We are grateful to Google Research for a Google Faculty Research Award in support of this line of work in 2015. We would also like to thank all *TOPLAS* reviewers for their extremely useful feedback.

REFERENCES

[1] Protonu Basu, Anand Venkat, Mary W. Hall, Samuel W. Williams, Brian van Straalen, and Leonid Oliker. 2013. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *Proceedings of the 20th International Conference on High Performance Computing (HiPC'13)*. 452–461.

[2] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 343–352.

[3] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. 2018. Transforming loop chains via macro dataflow graphs. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. ACM, New York, NY, 265–277. DOI: https://doi.org/10.1145/3168832

[4] Johannes Doerfert, Shrey Sharma, and Sebastian Hack. 2018. Polyhedral expression propagation. In *Proceedings of the 27th International Conference on Compiler Construction (CC'18)*. ACM, New York, NY, 25–36.

[5] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. 1992. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. 281–295.
[6] Google Inc. 2017. XLA (Accelerated Linear Algebra) for TensorFlow. Retrieved from https://www.tensorflow.org/performance/xla/.
[7] Abhinav Jangda and Uday Bondhugula. 2018. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 261–275.
[8] Ken Kennedy. 2001. Fast greedy weighted fusion. *Int. J. Parallel Prog.* 29, 5 (2001), 463–491.
[9] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. 301–320.
[10] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'07)*.
[11] Nimrod Megiddo and Vivek Sarkar. 1997. Optimal weighted loop fusion for parallel programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. 282–291.
[12] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. 2013. Tile size selection revisited. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013).
[13] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (July 2016), 83:1–83:11.
[14] Ravi Teja Mullapudi, Vinay Vasista, and Uday. Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 429–443.
[15] 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*. 793–804.
[16] PolyMage project, Apache 2.0 license 2017. PolyMage. Retrieved from https://bitbucket.org/udayb/polymage.
[17] PolyMagePage 2015. PolyMage: A DSL and compiler for automatic optimization of image processing pipelines. Retrieved from http://mcl.csa.iisc.ernet.in/polymage.html.
[18] Apan Qasem and Ken Kennedy. 2006. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the International Conference on Supercomputing (ICS'06)*. 249–258.
[19] 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 (2012), 32:1–32:12.
[20] 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 ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 519–530.
[21] István Z. Reguly, Gihan R. Mudalige, and Mike B. Giles. 2017. Loop tiling in large-scale stencil codes at run-time with OPS. *CoRR* abs/1704.00693 (2017).
[22] Gerald Roth and Ken Kennedy. 1998. Loop fusion in high performance Fortran. In *Proceedings of the International Conference on Supercomputing (ICS'98)*. 125–132.
[23] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. 2012. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction*. 101–121.
[24] Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, and Uday Bondhugula. 2017. Optimizing geometric multigrid method computation using a DSL approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*.
[25] M. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN Symposium on Programming Languages Design and Implementation*. 30–44.
[26] David Wonnacott. 1999. Time skewing for parallel computers. In *Proceedings of the 12th Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 477–480.
[27] Qing Yi and Ken Kennedy. 2004. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *Int. J. High Perf. Comput. Applic.* 18, 2 (2004), 237–253.
[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 International Symposium on Code Generation and Optimization (CGO'12)*. 207–218.

Received December 2018; revised March 2020; accepted June 2020