Optimizing Geometric Multigrid Method Computation using a DSL Approach

Vinay Vasista
Dept of CSA
Indian Institute of Science
Bangalore 560012 India
vinayv@iisc.ac.in

Kumudha Narasimhan
Dept of CSA
Indian Institute of Science
Bangalore 560012 India
kumudha@iisc.ac.in

Siddharth Bhat[*]
International Institute of Information Technology,
Hyderabad 500032 India
siddharth.bhat@research.iiit.ac.in

Uday Bondhugula
Dept of CSA
Indian Institute of Science
Bangalore 560012 India
udayb@iisc.ac.in

ABSTRACT

The Geometric Multigrid (GMG) method is widely used in numerical analysis to accelerate the convergence of partial differential equations solvers using a hierarchy of grid discretizations. Multiple grid sizes and recursive expression of multigrid cycles make the task of program optimization tedious. A high-level language that aids domain experts for GMG with effective optimization and parallelization support is thus valuable.

We demonstrate how high performance can be achieved along with enhanced programmability for GMG, with new language/optimization support in the PolyMage DSL framework. We compare our approach with (a) hand-optimized code, (b) hand-optimized code in conjunction with polyhedral optimization techniques, and (c) the existing PolyMage optimizer adapted to multigrid. We use benchmarks varying in multigrid cycle structure and smoothing steps for evaluation. On a 24-core Intel Xeon Haswell multicore system, our automatically optimized codes achieve a mean improvement of 3.2x over straightforward parallelization, and 1.31x over the PolyMage optimizer.

CCS CONCEPTS

• **Software and its engineering → Compilers**;

KEYWORDS

Multigrid, Tiling, Parallelization

ACM Reference Format:
Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, and Uday Bondhugula. 2017. Optimizing Geometric Multigrid Method Computation using a DSL Approach. In *Proceedings of SC17*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3126908.3126968

[*]This work was done when the author was an intern at the Indian Institute of Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126968

1 INTRODUCTION

The stencil approach using a finite difference method for discretization is currently the dominant approach for the numerical solution of partial differential equations (PDEs) [11]. Stencil computation can be viewed as data parallel operations repeatedly applied to a structured grid: computation at each grid point involves values only from neighboring grid points by application of a geometric pattern of weights (hence *stencil*); the computation thus exhibits spatial and temporal locality. One of the key reasons that a stencil approach is attractive is its suitability for a parallel and high-performance implementation in a variety of settings: on shared-memory multicore processors, distributed-memory clusters of multicore processors, and with accelerators like GPUs.

Solving partial differential equations using a stencil approach requires good iterative solvers that quickly converge to the solution, i.e., in fewer iterations. Multigrid methods [5] solve the problem at multiple grid resolutions (Figure 1), and are widely used to accelerate this convergence. Improved convergence is achieved by solving the problem on a coarser grid, and approximating the solution to the finer grid's solution with necessary corrections. The process is carried out at multiple levels of coarsening, to arrive at the solution for the finest level from the coarsest one — this forms a multigrid cycle. The number of iterations of such a cycle depends on (but not limited to) the cycle type, quality of convergence, and steps used within the cycle. Multigrid algorithms can be used either as direct solvers or as pre-conditioners for Krylov solvers. Algorithm 1 is the algorithm for a typical multigrid V-cycle.

As Geometric multigrid methods (GMG) [5] use a hierarchy of grids to evaluate the discretized equations at each level, GMG applications are memory-intensive, and demand more space for finer grids, which need to be live until the end of each multigrid cycle. Few of the multigrid cycle types are shown in Figure 2. The underlying computations in an MG cycle are mostly data parallel, and its constituents have near-neighbor data dependences. All of these characteristics provide ample opportunity to optimize performance on modern parallel architectures.

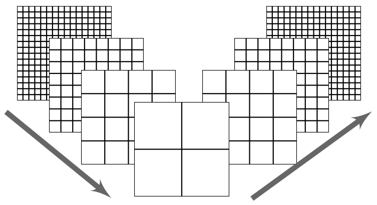


Figure 1: Hierarchical mesh structure of multigrid levels

A domain-specific language (DSL) for high performance is one that exploits computational domain information to deliver productivity, high performance, and portability to programmers. Geometric multigrid method computations constitute a well-defined domain perfect for the development of a high-performance DSL with its optimizing code generator. We observe that the individual operations or kernels used in GMG are similar in nature to those used in image processing pipelines. The latter is expressed as composition of typically data parallel filters such as convolutions, downsampling, upsampling, and other point-wise ones on image pixels, which are typically two-dimensional data. The restrict operator in GMG (Algorithm 1 line 5) in similar to an image downsampling, interpolation (line 8) is similar to upsampling, each individual pre-smoothing (and post-smoothing) iterations (lines 1, 10) is a convolution. Correction and residual are simple point-wise operations. Halide [17] and PolyMage [14] are two recent high-performance DSL compilation efforts for image processing pipelines that apply complex loop and data reordering transformations.

In this paper, we develop an approach to optimize geometric multigrid method computations using a DSL approach. The key contributions of the paper are:

- Language constructs to allow compact code to be written productively for GMG,
- Optimizations in the DSL compiler to deliver high performance on shared-memory multicore systems by optimizing for parallelism, locality, and memory usage,
- A detailed experimental evaluation on two-dimensional and three-dimensional grids with various configurations and problem classes, and on the Multigrid benchmark from the NAS parallel benchmarks.

Our optimization approach, which we refer to as PolyMG, is implemented in the open-source PolyMage infrastructure [16], while reusing the latter's language and optimization infrastructure for image processing pipelines. In particular, we add several memory usage optimizations to PolyMage's existing optimization infrastructure. In our experimental evaluation, we also provide a comparison to a reference hand-optimized version for each evaluated benchmark. All evaluation was carried out on a high-end state-of-the-art shared-memory multicore server. Experimental results demonstrate a significant productivity-cum-performance gain through our approach.

The rest of this paper is organized as follows. Section 2 describes how multigrid programs are described in PolyMage DSL. Section 3 describes our optimization process in detail. Experimental results are presented in Section 4. Related work is presented in Section 5 and conclusions in Section 6.

Algorithm 1: *V-cycle*^{h}

```
Input  : $v^h, f^h$
1  Relax $v^h$ for $n_1$ iterations                    // pre-smoothing
2  if coarsest level then
3      Relax $v^h$ for $n_2$ iterations                // coarse smoothing
4  $r^h \leftarrow f^h - A^h v^h$                      // residual
5  $r^{2h} \leftarrow I_h^{2h} r^h$                    // restriction
6  $e^{2h} \leftarrow 0$
7  $e^{2h} \leftarrow V-cycle^{2h}(e^{2h}, r^{2h})$
8  $e^h \leftarrow I_{2h}^{h} e^{2h}$                  // interpolation
9  $v^h \leftarrow v^h + e^h$                          // correction
10 Relax $v^h$ for $n_3$ iterations                    // post smoothing
11 return $v^h$
```

2 LANGUAGE

Multigrid applications consist mostly of stencil computations or other point-wise data parallel operations. In addition, grid accesses in interpolation or restriction phases involve stencils with a scaling factor, usually of 2 or 1/2. This means, the producer and consumer grids' access variables should be of the same scale. The PolyMage language captures these types of accesses for image processing applications, often used in upsampling or downsampling an image. Execution reordering transformations such as tiling and fusion are also supported on these classes of computations [14]. This makes PolyMage a good fit as a language to extend, to provide a high-level abstraction to domain experts for writing multigrid methods. We additionally introduce a small set of constructs to further reduce verbosity in expressing multigrid steps, which we describe below.

PolyMage treats an operation on a structured grid or an image as a function of a variable defined with a domain or as a composition of other functions. The language allows the programmer to define such operations using the `Function` construct. Each step in a multigrid cycle can be expressed using the `Function`. Another construct is `Stencil`, using which the weights of a stencil kernel for a 2-d grid can be expressed as a list of lists in Python. We extended this construct to 3D grids, by letting the programmer use a list of list of lists. This can be used to express smoothers, error corrections and residual computation used in multigrid. The center of a stencil of size $m \times m$, is by default assumed to be $(m/2, m/2)$. A stencil with its center off the default value, can also be expressed by passing its value to the construct. An example usage is as follows.

Stencil`(f, (x,y), `$[[0,1],[-1,2]]$`, 1.0/16)`

translates to:

```
1.0/16 * ( f(x,y+1) - f(x+1,y) + 2*f(x+1,y+1) )
```

In order to make the programming for multigrid more convenient, we introduced the constructs Restrict and `Interp`, derived from the Function construct to express restriction and interpolation, respectively. These constructs are associated with default sampling factors (1/2 in case of Restrict and 2 in case of `Interp`). The sampling factor decides the grid access index coefficients as well as the total size of the output grid. For example, the output of the interpolation function will be 2^d times larger than its input function, where d is the grid dimensionality. Due to this, the function definition for interpolation will be of the form f(`input`(x/2, y/2)), and not all input

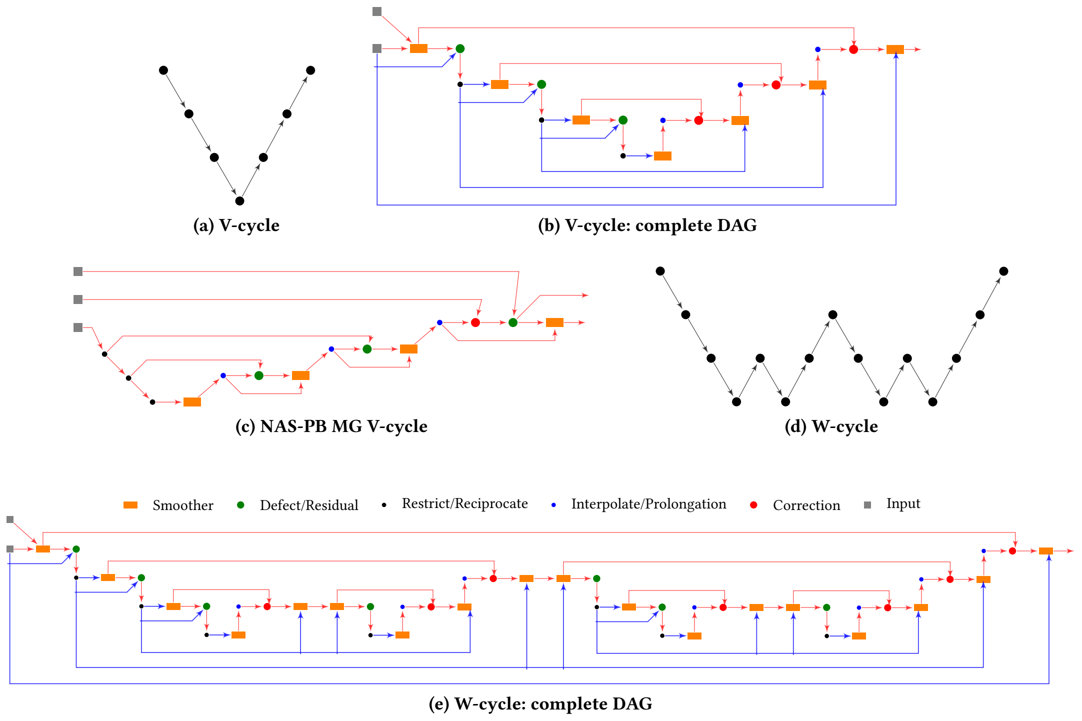


Figure 2: DAG of operators corresponding to a (a) V-cycle and (b) W-cycle iteration of Multigrid method; each operator is data parallel and applies on a multi-dimensional grid

points surrounding a corresponding output grid point are used for the approximation. A natural way of selecting input grid points is based on evenness of the output point's index, for which programmers prefer to use modulo operators. Such overhead in indexing for sampling is prone to human error, and the specialized constructs for restriction and interpolation help in mitigating them.

The boundary values of a function can be set with the help of `Case` construct, by creating a set of piecewise definitions. Other basic language support including `Variable`, `Parameter`, `Interval`, etc., are retained from PolyMage while extending the support for multigrid. Additional detail on the language rules and usage can be found in the original PolyMage paper by Mullapudi et al. [14].

The PolyMG specification for the V-cycle Algorithm 1 is given in Figure 3. This program operates on the initial guess grid `V` and the RHS grid `F`, for the parametric problem size `N` (value of h has to be derived using `N`). n1, n2, and n3 represent the pre-smoothing, post-smoothing and coarse-smoothing steps respectively. This PolyMG specification, like the original V-cycle Algorithm 1, is expressed in a recursive fashion.

The Python function smoother applies the smoothing operator for the specified number of steps, using the TStencil construct, introduced in PolyMG. The function can also be expressed using the basic Stencil construct (which does not require the time-step parameter) available in PolyMage, using a python loop. But, this results in more number of PolyMage Functions, and duplication of the common dependence information passed to the compiler - thereby increasing the analysis time. The TStencil also allows initialization of the parameter T at runtime, thus making the algorithm generic to variable number of smoothing steps – this is not supported by the basic Stencil construct.

The PolyMG compiler processes the specification embedded in Python as a collection of functions defined over polyhedral domains along with a directed acyclic graph with additional information specifying instance-wise dependences, which provides the producer-consumer relationships exactly. Thus, the specification is a feed-forward pipeline. The loop iterating over an entire V-cycle or W-cycle is thus external to PolyMG. A polyhedral representation is then constructed for this specification, and schedule and storage transformations are determined and specified to improve locality and parallelization; code generation is then performed from the polyhedral specification of the domains and schedules using ISL [18, 19]. Figure 4 shows the different phases of PolyMG's optimizing code generator.

3 OPTIMIZATIONS

In this section, we describe in detail optimizations performed by PolyMG's code generator.

3.1 Grouping for Fusion and Tiling

Several past works have focused on improving performance of multigrid codes [4, 7, 8, 12, 20]. Among these, [7, 8, 20] considered and evaluated tiling across multiple smoothing steps; [3, 4, 20] considered fusion of operators and optimizing the fused stencils together. All of these approaches were manual or semi-automatic.

```
N = Parameter(Int, 'n')
T = Parameter(Int, 'T')
V = Grid(Double, "V", [N+2, N+2])
F = Grid(Double, "F", [N+2, N+2])
...
def rec_v_cycle(v, f, l):
    # coarsest level
    if l == 0:
        smooth_p1[l] = smoother(v, f, l, n_3)
        return smooth_p1[l][n_3]
    # finer levels
    else:
        smooth_p1[l] = smoother(v, f, l, n_1)
        r_h[l] = defect(smooth_p1[l][n_1], f, l)
        r_2h[l] = restrict(r_h[l], l)
        e_2h[l] = rec_v_cycle(None, r_2h[l], l-1)
        e_h[l] = interpolate(e_2h[l], l)
        v_c[l] = correct(smooth_p1[l][n_1], e_h[l], l)
        smooth_p2[l] = smoother(v_c[l], f, l, n_2)
        return smooth_p2[l][n_2]

def smoother(v, f, l, n):
  ...
  W = TStencil(([y, x], [extent[l], extent[l]]),
               Double, T)
  W.defn = [ v(y, x) - weight *
    (Stencil(v, [y, x],
      [[ 0, -1,  0],
       [-1,  4, -1],
       [ 0, -1,  0]], 1.0/h[l]**2) - f(y, x)) ]
  ...
  return W

def restrict(v, l):
    R = Restrict(([y, x], [extent[l], extent[l]]),
                 Double)
    R.defn = [ Stencil(v, (y, x),
                 [[1, 2, 1],
                  [2, 4, 2],
                  [1, 2, 1]]) * 1.0/16 ]
    return R

def interpolate(v, l):
    ...
    expr = [{}, {}]
    expr[0][0] = Stencil(v, (y, x), [1])
    expr[0][1] = Stencil(v, (y, x), [1, 1]) * 0.5
    expr[1][0] = Stencil(v, (y, x), [[1], [1]]) * 0.5
    expr[1][1] = Stencil(v, (y, x), [[1, 1], [1, 1]]) * 0.25
    P = Interp(([y, x], [extent[l], extent[l]]),
               Double)
    P.defn = [ expr ]
    return P
```

Figure 3: PolyMG specification for Multigrid V-cycle

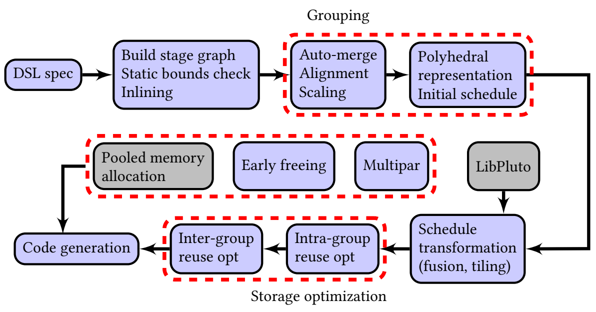


Figure 4: The PolyMG optimizer: various phases

We will show that we can leverage more optimization opportunity using the PolyMage approach – both in execution re-ordering by application of tiling and fusion, and in storage optimizations: hence, our contributions are not purely limited to automating existing techniques in a DSL compiler.

Optimizing smoothing iterations of a multigrid algorithm using state-of-the-art tiling techniques based on the polyhedral framework was recently studied by Ghysels and Vanroose [8]. Pluto's diamond tiling technique was used by them to block the smoothing iterations. Note that the smoothing iterations are no different than time-iterated stencils typically used for experimentation by a large body of work on loop transformations. The number of smoothing iterations may be a small number, typically from a few iterations to few tens. However, the smoothing iterations at the finest level constitute the bulk of the execution time [8]; thus optimizing them to enhance data reuse and improve the arithmetic intensity is useful. Moreover, a higher number of smoothing steps can be accommodated, to improve the quality of the rate of cycle convergence, which in turn makes fewer cycle iterations sufficient to arrive at the solution [8].

Ghysels and Vanroose [8] as well as most past work [12, 20] however do not consider optimizing locality across different types of stages involved in a multigrid cycle iteration. There is an opportunity to further enhance locality through grouping and fusion of stages that include not just smoothing steps. In addition, use of local buffers or *scratchpads* for tiles can realize benefits of tiling better due to reduced conflict misses and TLB misses, and better prefetching benefits. Although the semi-automatic compiler approach of Basu et al. [3, 4] considers fusion of different multigrid operators, it makes a fixed fusion choice, and storage optimizations are not considered. Related work (Section 5) includes a more detailed discussion. Our approach addresses all of these limitations, and in addition, is fully automatic.

Time skewing or parallelogram tiling [21] may not yield any or enough tiles to parallelize on the wavefront, and incurs pipelined startup. Overlapped tiling [13], diamond tiling [2], or split tiling [10] allow concurrent start and are more suitable. We employ the same overlapped tiling strategy as the one used in PolyMage for image processing pipelines — it leads to overlapping tiles. However, for multigrid computations, the shapes are symmetric on both sides, i.e., they are hyper-trapezoidal. This is not the case for image processing pipelines where the heterogeneity in the dependences could lead to asymmetrically shaped overlapped tiles. For a multigrid computation on 2-d grids, our tile would be shaped like a square or rectangular pyramid. The notion can be extended to three or higher dimensions. When compared to diamond or split tiling, although overlapped tiling involves redundant computation, it simplifies usage of local buffers due to the lack of any dependence between neighboring adjacent tiles.

Figure 5 illustrates both tiling schemes – only one dimension of space/grid is shown for simplicity. The data points encircled in red must be communicated to the next tiles since they at least one use outside the group. These are said to be live-out from the tile where they are computed, and live-in to the tiles where they are read. In this text, we extend this terminology to the functions that define such points.

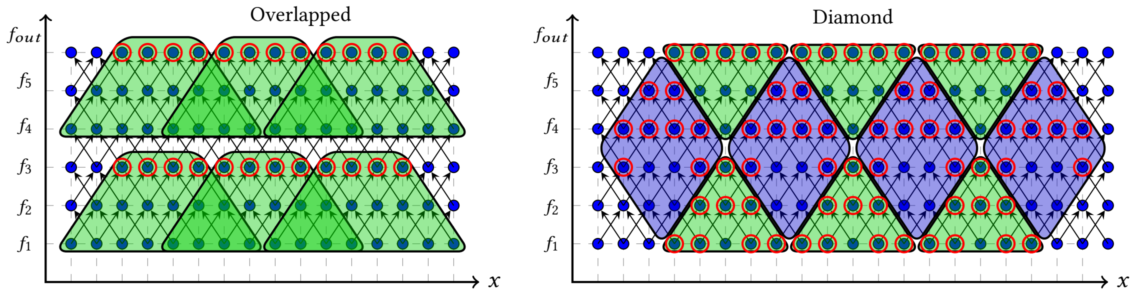


Figure 5: Tiling: data live out of a tile is encircled. Overlapped tiling has live-out data only at its top face. While the overlapped tiling techniques suffer from the overhead of redundant computations, diamond tiling incurs more synchronization overhead

Finding a good grouping of functions for tiling is practically hard, given the complexity of the function computations, number of parameters affecting performance on modern parallel architectures, and number of valid grouping choices. PolyMage uses a greedy heuristic to generate a set of function groups for overlapped tiling, based on tile size and redundant computation thresholds. We use the same auto-grouping algorithm to group multigrid functions. Figure 6 shows the grouping for a 2-D V-cycle configuration. In this figure, the scratchpad nodes represent stages with no use outside the tiled group, with a memory requirement of the order of tile sizes. The live-out nodes require full array allocations (i.e., with size in the order of function interval size) for communications across groups. The colors of nodes indicate reuse of storage between those of the same color and node type, which can either be scratchpad or *full array*. In summary, no changes were needed to the fusion and tiling transformations already employed in PolyMage.

3.2 Memory Optimizations

PolyMage abstracts away memory allocation, management, and indexing from the programmer. This gives complete freedom in utilization and reuse of local buffers, allocation of multi-dimensional arrays that are live-out of one fused group and consumed at another. Allowing the programmer to allocate and index them often limits the ability of a compiler to perform data layout transformations: since techniques like alias analysis, escape analysis, and delinearization are often necessary to perform data reordering in a safe way. Using a DSL avoids these difficulties.

The set of functions within a group, except for the live-ins and live-outs, do not produce any values used outside the group. Also, when these functions or groups of them are tiled, there are no dependences between the tiles, since overlapped tiling is communication-avoiding. This fact is exploited to minimize storage requirements of a tile, by using small scratchpads for such functions. Scratchpads sizes are of the order of tile sizes along all dimensions, which actually are compile-time constants here. Such constant-sized buffers (one per each thread), declared within the scope of the tiled loop nest, are on thread local stack. Full array allocations for the live-in and live-out functions are made using ‘malloc’.

In multigrid cycles, the outputs of many functions like the intermediate steps of smoothers, have a short lifetime. But the allocations currently made by PolyMage are one-to-one, i.e., one buffer is used for each individual function in the pipeline. This leads to more than necessary storage to hold the computed intermediate data. We address this issue by introducing an additional pass, to extract reuse opportunities using a best effort approach. This is done at two levels – reusing scratchpads within a group (intra-group tiling), and reusing full arrays across groups (inter-group tiling). Both these passes require the groups of functions and the function inside an overlapping tile to have been already scheduled, with a total order specified.

In order to perform the storage allocation, we first create a storage class, whose objects represent an abstract buffer for the pipeline functions. Functions are then categorized into these storage classes, such that buffer reuse is allowed only among functions of the same storage class. This classification is based on the dimensionality, data type and buffer size requirements of functions. Also, this is carried out separately for functions within a group (scratchpads buffers) and functions that are group live-outs (full-arrays).

3.2.1 Intra-Group Buffer Reuse. Although it might appear that the amount of reduction in the total memory consumption of the program may not be substantial, the gain is not just on the total byte count. A tiled code will be able to use larger tile sizes if the memory consumption for a given tile size is reduced. Without an optimized or the optimal scratchpad buffer size, maximum reuse and the potential performance of a tiling scheme is not realized. Effects of this optimization will be quantified in detail in the experimental evaluation section.

The constant size of tile scratchpads (known at compile time) makes the search for reuse candidates simpler, when compared to sizes with parametric bounds. Scratchpads that have equal sizes fall under the same class. However, this size requirement can be relaxed by adding a small ±constant threshold to increase the reuse opportunities. We use a simple greedy approach to color the classified functions within a group for storage reuse.

First, the DAG of the entire group is scanned for the last use of each function and its scheduled time within the group, as described in Algorithm 2. Using this information, a second pass, Algorithm 3 is performed in schedule-order (earliest to the latest), to map each function to it’s logical storage. This algorithm creates a separate logical pool of storage objects for every class of storage. Given a function, the Algorithm determines if a new storage should be

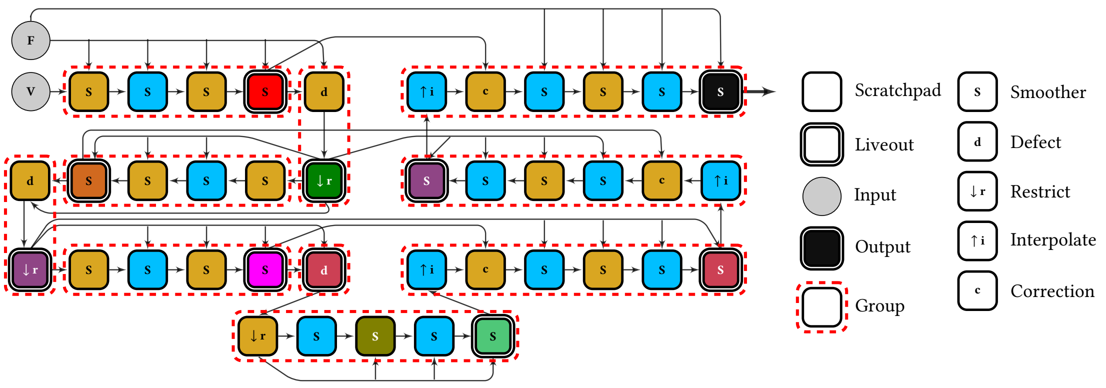


Figure 6: Grouping (fusion of operators): the dashed boxes correspond to a fused group

Algorithm 2: Procedure to find last use of functions in a pipeline: *getLastUseMap*

```
Input : set of functions − F,
        timestamp map − T
// timestamp map: function -> time
// last-use map: time -> function
1 lastUseMap ← ∅
2 for each func ∈ F do
3     lastUseTime ← −1
      // get timestamp of child that uses func the last
4     for each child ∈ children(func) do
5         t ← T(child)
6         lastUseTime ← max(lastUseTime, t)
      // add func to the set of functions with the same last-
      // use timestamp
7     lastUseMap(lastUseTime) ← lastUseMap(lastUseTime) ∪ func
8 return lastUseMap
```

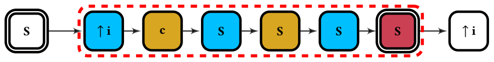


Figure 7: Scratchpad reuse in one of the pipeline groups

allocated or an already existing unused storage should be remapped. If there are any functions that are used after the timestamp of the function being currently evaluated, their storage is collected into the corresponding pools. These storage objects are available for use by other functions, of the same class, scheduled at a later time point.

An illustration of intra-group reuse (for scratchpads) enabled in one of the groups in Figure 6, is shown in Figure 7, using a group-local colouring scheme. In this example, an interpolation and a correction step are fused with four post-smoothing steps, shown in the bounded region. Except the last smoothing step, which is live-out of the group, all other nodes are allocated as scratchpads to store just the data computed within a tile. Since none of the nodes in the group produce data that is consumed by more than one node, the allocations made to each node can be reused immediately after their first use. Hence, in this case, our intra-group reuse algorithm uses just two colours to assign unique buffers to the nodes. Nodes with the same color get to use the same scratchpad buffer, which is allocated privately for each thread executing a tile of this group. In

Algorithm 3: *remapStorage* Algorithm to remap PolyMage functions to arrays

```
Input : set of functions − F,
        timestamp map − T,
        storage class map − storageClass
// timestamp map: function -> time
// storageClass map: function -> storage-class
1  lastUseMap ← getLastUse(F, T)
// Sort the functions in ascending order of timestamp
2  $F_{sorted}$ ← sort(F, key = T(f))
// arraypool map: storage-class -> set of integers (array ids)
// initialize the array pool of all storage classes to empty
3  arrayPool(c) ← {}, ∀ c ∈ $\bigcup_{f \in F}$ storageClass(f)
// storage map: function -> integer
4  storage(f) ← ∅, ∀f ∈ F
5  arrayID ← 0
6  for each func ∈ $F_{sorted}$ do
7      c ← storageClass(func)
       // If array pool is empty, return a new array
8      if arrayPool(c) = {} then
9          arrayID ← arrayID + 1
10         storage(func) ← arrayID
       // Else return an unused array from the pool
11     else
12         storage(func) ← pop(arrayPool(c))
       // If some function has no use after this timestamp
13     t ← T(func)
14     if t ∈ lastUseMap then
15         for each deadFunc ∈ lastUseMap(t) do
               // Return the arrays to array pool
16             c′ ← storageClass(deadFunc)
17             arrayID ← storage(deadFunc)
18             arrayPool(c′) ← arrayPool(c′) ∪ arrayID
19 return storage
```

the given example, only two buffers are sufficient to execute the tile computation, in contrast to using five buffers without reuse.

3.2.2 Inter-Group Array Reuse. Compaction of the set of full arrays plays a major role in minimizing the resident memory of a running program. For smaller problem sizes, this optimization can yield performance gains by letting the entire data fit in the last level cache. Similarly, for very large problem sizes, not optimizing for storage might cause the allocated area exceed the available DRAM space, either resulting in movement of data in and out of swap memory, or requiring more nodes for weak scaling. This situation can be helped by reusing arrays to serve multiple functions. The problem is particularly important in a DSL compiler that abstracts away memory.

Full arrays in PolyMage DSL can have parametric sizes in each dimension, which makes it tricky to classify them. If the intermediate arrays contain ghost zones (boundary paddings), they typically differ from each other by constant offsets. Such arrays are collected under one class, if the parameters for the corresponding (matching) dimensions are the same. The size of the storage class is then calculated using the maximum offset value in each dimension of the class' arrays. This ensures that all arrays in a class have sufficient size to avoid out-of-bound accesses. Constant sized full arrays are flagged under one class, that does not include any parametric sized array. Program input and output arrays are not considered to be available to serve as reuse buffers.

Since the storage reuse algorithm requires a schedule of the functions, a live-out function's schedule is set to the schedule of the group it belongs to. In case there are multiple live-outs from a group, and more than one of them happens to be eligible to reuse an array, only one of these is allowed to reuse it. This constraint is taken care of, by the remapping algorithm since such live-outs will all have the schedule time, equal to that of their group. Allocation and deallocation of full-arrays happen at the granularity of a group.

3.2.3 Pooled Memory Allocation. The optimization of the multigrid programs, using PolyMage, is limited to one multigrid cycle. As a result, any opportunity to extract performance across cycles has to be done outside PolyMage. One such optimization is of employing a pool of memory that can serve allocation and deallocation requests. We use a pooled memory allocator with appropriate interface calls to it generated along with the output code. Thus, all intermediate buffer allocation requests have to go through this allocator, which scans over a list of already allocated buffers to decide whether to create one or return a pointer to an available array. Freeing an array will be simply a table entry update, to keep track of reusable arrays, using `pool_deallocate`. With this setting, arrays are actually allocated at the entry of the first multigrid cycle, and are all freed after the last call to it.

This strategy not only prevents frequent `malloc` calls across the invocation of a multigrid cycle, but also enables reuse within a cycle that might have been missed by the inter-group reuse pass. (For this we insert calls to `pool_deallocate` during code generation, as soon as all uses of an array's user are finished. This ensures that free arrays are returned to the pool in time before any new request is made).

3.2.4 Auto-tuning. We use PolyMage's auto-tuner to search for the best configuration over a reasonably small space for both 2D and 3D benchmarks. For 2D benchmarks, tile sizes of the outermost dimension are allowed within a range of 8:64, and that of innermost dimension in range 64:512, in powers of two. For 3D benchmarks, this space will be larger by a multiplier due to their dimensionality. The tile size ranges for the two outermost and the innermost dimensions are 8:32 and 64:256, respectively, in powers of two. Five different values of grouping limit, which controls the size of a group during the automerging phase, are used. This limit affects the amount of overlap that can be tolerated and how much temporal locality can be exploited. In total, 2D benchmarks are tuned for 80 configurations and 3D benchmarks are tuned for 135 configurations.

3.2.5 Other practical considerations. As most of the computations in multigrid programs are data parallel, there is abundant parallelism along each dimension of the loop nests generated in the output code. Also, the loop-nest for an overlapped tile is typically parallel in all dimensions. However, if boundary conditions are set in the DSL specification, the intra tile loops will possibly be generated as ISL [18] AST node with multiple boundary conditions and inner loops. Hence, we separate out such imperfect parallel loop nests, and add the OpenMP clause '`parallel for collapse(`d`)`' for perfect loop nests, where d is the depth of perfect loop-nest. Iteration space of parallel loops annotated with this clause, are flattened by the OpenMP runtime so that iteration chunking for threads happen on a much longer loop, which can potentially improve the load balance for threads at loop boundaries.

In order to generate collapsed loops, the scratchpad buffer allocation code for the tile is moved to the innermost loop of the nest. We add a module that scans the polyhedral AST generated by ISL for parallel perfect loop nests, and determines the depth for both the collapse clause and scratchpad allocation. This information is later used during code generation. Figure 8 shows a snippet of code generated by PolyMG after all its optimizations.

4 EXPERIMENTAL EVALUATION

Experimental setup. All experiments were run on a dual socket NUMA multicore system with Intel Xeon v3 processors (based on the Haswell microarchitecture). Table 1 provides details of the hardware and software. OpenMP thread affinity was set to 'scatter' to evenly balance threads across cores of different processors. The minimum execution time from five runs was taken in all cases.

4.1 Benchmarks and Comparison

Experimental evaluation is performed on Multigrid benchmarks that solve the Poisson's equation, which is given by:

$$\nabla^2 u = f, \qquad (1)$$

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 in general. We solve the Poisson's equation for 2-dimensional and 3-dimensional data grids (with a finite difference discretization), using V-cycle and W-cycle; we thus have four benchmarks resulting from these choices. These

```
void pipeline_Vcycle(int N, double * F,
                     double * V, double *& W)
{
  /* Live out allocation */
  /* users : ['T9_pre_L3'] */
  double * _arr_10_2;
  _arr_10_2 = (double *) (pool_allocate(sizeof(double) *
                          (2+N)*(2+N)));
#pragma omp parallel for schedule(static) collapse(2)
  for (int T_i = -1; T_i <= N/32; T_i+=1) {
    for (int T_j = -1; T_j <= N/512; T_j+=1) {
      /* Scratchpads */
      /* users : ['T8_pre_L3', 'T6_pre_L3', 'T4_pre_L3',
                  'T2_pre_L3', 'T0_pre_L3'] */
      double _buf_2_0[(50 * 530)];
      /* users : ['T7_pre_L3', 'T5_pre_L3', 'T3_pre_L3',
                  'T1_pre_L3'] */
      double _buf_2_1[(50 * 530)];

      int ub_i = min(N, 32*T_i + 49);
      int lb_i = max(1, 32*T_i);
      for (int i = lb_i; i <= ub_i; i+=1) {
        int ub_j = min(N, 512*T_j + 529);
        int lb_j = max(1, 512*T_j);
#pragma ivdep
        for (int j = lb_j; (j <= ub_j); j+=1) {
          _buf_2_0[(-32*T_i+i)*530 + -512*T_j+j] = ...;
        }}

      int ub_i = min(N, 32*T_i + 48);
      int lb_i = max(1, 32*T_i);
      for (int i = lb_i; i <= ub_i; i+=1) {
        int ub_j = min(N, 512*T_j + 528);
        int lb_j = max(1, 512*T_j);
#pragma ivdep
        for (int j = lb_j; (j <= ub_j); j+=1) {
          _buf_2_1[(-32*T_i+i)*530 + -512*T_j+j] = ...;
        }}
      ...
  }}
  ...
  pool_deallocate(_arr_10_2);
  ...
}
```

Figure 8: Optimized code generated by PolyMG

Table 1: System details

Processors	2-socket Intel Xeon E5-2690 v3
Clock	2.60 GHz
Cores	24 (12 per socket)
Hyperthreading	disabled
L1 cache / core	64 KB
L2 cache / core	512 KB
L3 cache / socket	30,720 KB
Memory	96 GB DDR4 ECC 2133 MHz
Compiler	Intel C/C++ and Fortran compiler (icc/icpc and ifort) 16.0.0
Compiler flags	-O3 -xhost -openmp -ipo
OS	Linux kernel 3.10.0 (64-bit) (Cent OS 7.1)

benchmarks are based on code used for evaluation by Ghysels and Wanroose [8] and made available on BitBucket [9]. Our techniques are also applicable to a finite volume discretization, which was used for benchmarks in some past work [4, 20].

We use two smoothing configurations, namely 4-4-4 and 10-0-0, for each of the four benchmarks. The smoothing configuration 4-4-4 refers to four pre-smoothing iterations, four coarsest level smoothing iterations, and four post-smoothing iterations (corresponding to $n_1 = n_2 = n_3 = 4$ in Algorithm 1). In all cases, Jacobi smoothing steps are used. Although other smoothing techniques like successive over-relaxation and Gauss-Seidel Red Black (GSRB) exist, we only focus on Jacobi smoothing for this paper. GSRB for example presents additional considerations for vectorization [20], all optimization presented in this paper apply to it if the red and black points are abstracted as two grids. In addition to the four benchmarks above, we also evaluated our system on the NAS Multigrid benchmark (MG) from NAS Parallel Benchmarks (v3.2) and compare it with the reference version with non-periodic boundary setting. NAS MG uses a V-cycle with no pre-smoothing steps.

Problem sizes used for all benchmarks are listed in Table 2. We define problem size classes B and C for the benchmarks we evaluate in the same way they exist for the NAS Multigrid benchmark. We exclude classes W and A given the amount of parallelism and compute power we have on the experimental system.

Table 2: Problem size configurations: the same problem sizes were used for V-cycle and W-cycle and for 4-4-4 and 10-0-0.

Benchmark	Grid size, cycle #iters	
	Class B	Class C
2D	8192^2, 10	16384^2, 10
3D	256^3, 25	512^3, 10
NAS-MG	256^3, 20	512^3, 20

PolyMG is used in general to refer to our implementation of our GMG DSL compilation system. We use *polymg-opt+* to refer to the optimized code generated with all optimizations and considerations that are the contributions of this work. For the first four benchmarks, we compare with (a) manually optimized versions of the benchmarks that we obtained from Ghysels and Vanroose [8], referred to as *handopt*, that involves explicit loop parallelization, storage reuse (two modulo buffers per level) and pooled memory allocations, (b) *handopt* further optimized by time tiling the smoothing steps with Pluto's diamond tiling approach (version 0.11.4-229-gceac3ae) [15], which we refer to as *handopt+pluto* (also provided by Ghysels and Vanroose [8], and (c) with a version that does not have the storage optimizations and other improvements we described in Section 3 (*polymg-opt*). polymg-opt thus applies the same set of optimizations as PolyMage does for image processing pipelines. *polymg-naive* corresponds to a simple parallel code generation using PolyMage with loops generated in a straightforward manner with no tiling, fusion, or storage optimization, but with OpenMP pragmas on the outermost among parallel loops for each loop nest (loop iterating the outermost among space/grid dimensions). For *handopt+pluto*, tile sizes were tuned empirically around optimized ones that shipped with its release. In addition, to also evaluate diamond tiling for the TStencil construct in PolyMG, we integrated libPluto into PolyMG (Figure 4); we use *polymg-dtile-opt+* to refer to polymg-opt+ with the choice to applying diamond tiling instead of overlapped tiling to pre- and post-smoothing steps.

The four benchmarks capture a range of complexity. The W-cycle is a large and complex pipeline, with nearly 100 stages for smoother and level settings (as seen in the DAG of Figure 2e). Domain experts consider the W-cycle as a heavyweight method

Table 3: Benchmark characteristics: lines of code and baseline execution times

Benchmark	Stages	Lines of code			Lines of generated code		Execution time of polymg-naive (s)			
	(# DAG nodes)	PolyMG (DSL)	handopt (C)	handopt+pluto (C)	polymg-opt (C/C++)	polymg-opt+ (C/C++)	class B 1	class B 24	class C 1	class C 24
V-2D-4-4-4	40	160	140	150	2324	2496	51.36	9.61	141.43	25.8
V-2D-10-0-0	42				2155	2059	60.11	11.41	169.74	30.96
W-2D-4-4-4	100	165	145	155	6156	6768	95.39	13.19	268.15	37.19
W-2D-10-0-0	98				4306	4711	78.23	14.75	241.14	44.79
V-3D-4-4-4	40	220	185	200	4889	4457	20.89	4.1	67.35	15.05
V-3D-10-0-0	42				4593	4179	24.21	5.3	78.15	18.09
W-3D-4-4-4	100	225	190	205	12184	11535	40.69	6.16	132.95	17.74
W-3D-10-0-0	98				9237	7897	42.18	6.79	133.44	21.26
NAS-MG	34	180	500	-	2010	2013	6.72	0.95	60.34	7.84

due to its computational cost and the number of steps involved in its single cycle. The miniGMG and HPGMG benchmarks use F-cycle, which is in between V- and W- cycles in complexity.

Table 3 provides an indication on the programmer effort involved with developing the various versions.

4.2 Analysis

Figures 9, 10 show performance obtained when running on all 24 cores of the system. When compared to *handopt* and *handopt+pluto*, the difference in performance is obtained due to the combined benefit of fusion, tiling, and better storage management. The improvement of *polymg-opt* over *handopt+pluto* is due to fusion as well as use of local buffers (scratchpads). The difference between *polymg-opt* and *polymg-opt+* isolates the improvement due to reuse of scratchpads within a group, across groups, and more efficient allocation.

Improvement summary. polymg-opt+ achieves a mean (geometric) improvement of 3.2× over polymg-naive across all 2D and 3D benchmarks (4.73x for 2D and 2.18x for 3D). The mean improvement over polymg-opt is 1.31x. We even achieve an improvement of 1.23x over handopt+pluto on average (1.67x for 2-D cases). The improvements are higher for 2-d than for 3-d — this is an expected trend since multigrid on higher-dimensional grids is expected to have a higher memory bandwidth requirement for the same amount of computation.

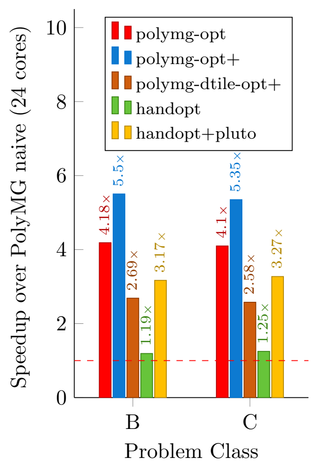
The histogram in Figure 11b shows the speedup breakdown for intra group reuse, pooled memory allocation, and inter group storage optimizations for V-10-0-0 for both 2D and 3D benchmarks, over polymg-naive. Figure 11a compares performance of PolyMG-opt+ and Pluto for just the Jacobi smoothing steps used in 3D class C benchmark. Comparison of just the smoothing iterations for both overlapped and diamond tiling for 512^3 problem size, tuned on 24 cores, is shown in Figure 11a. While for smaller smoothing steps (4), overlapped tiling using local buffers in polymg-opt+ performs slightly better, diamond tiling in Pluto is more effective for 10 smoother iterations. This is unlike the 2-d case, where we found the tiling in polymg-opt+ is always better than Pluto's diamond tiling.

We observed that handopt+pluto represented close to the best that could be done with respect to storage optimization (allocation and use of buffers). Hence, we consider automatically obtained performance that comes even close to it as significant. Unlike in the 2-d case, we observe that polymg-opt+ is not able to outperform handopt+pluto in the 10-0-0 3-d cases. The difference is in the amount of benefits overlapped tiling brings for the 3-d case (especially 10-0-0) as is evident from Figure 11a.

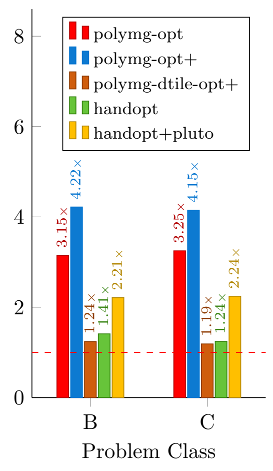
We also observe that polymg-dtile-opt+ outperforms polymg-opt+ in only one scenario 3D-W-10-0-0, and its performance is always lower than handopt+pluto. In fact, polymg-dopt+ outperforms polymg-dtile-opt+ by a big margin for the 2D cases. The gap is narrower for 3D, and along expected lines more for 10-0-0 than for 4-4-4 configurations (as explained by Figure 11a). However, this does not still explain the large gap between polymg-dtile-opt+ and polymg-opt+. Due to an implementation issue arising from conservative assumptions in reusing input/output arrays, we note that polymg-dtile-opt+ performs more memory copies; such copies are not present in handopt+pluto or polymg-opt+. We confirmed that this reduces polymg-dtile-opt+'s performance by up to 60% in 3D cases. We expect this issue to be resolved soon, but in either case, it is clear that for 2D grids, and for fewer pre-smoothing steps (whether for 2D or 3D), overlapped tiling with storage optimization has an edge over diamond tiling for such Multigrid computations. The fraction of redundant computation with overlapped tiling increases with dimensionality. It is straightforward to choose polymg-dtile-opt+ over polymg-opt+ for 3D grids when the number of pre-smoothing steps is high (order of ten or more).

Over the reference NAS MG implementation included for NAS parallel benchmarks (NPB), polymg-opt+ obtains an improvement of 32% for the Class C problem size (Figure 10e). It is important to mention here that the NAS MG implementation uses a hand-optimized loop body computation that computes a partial sum and reuses it multiple times through a line buffer in the innermost loop.

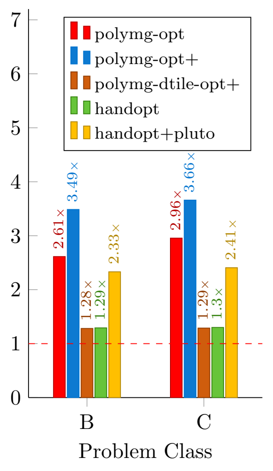
Scaling. Table 3 can be used in conjunction with Figures 9 and 10 to determine how the optimized code scales with core count. For example, for W-2D-10-0-0 class C, a naive parallelization (polymg-naive) yields a speedup of only 5.38× on 24 cores when compared



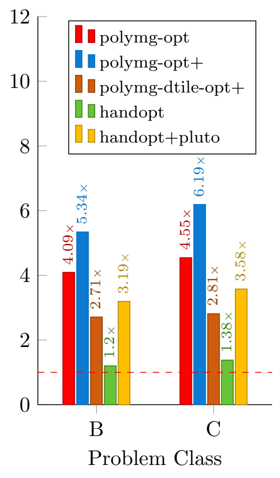
(a) 2D-V-10-0-0



(b) 2D-V-4-4-4

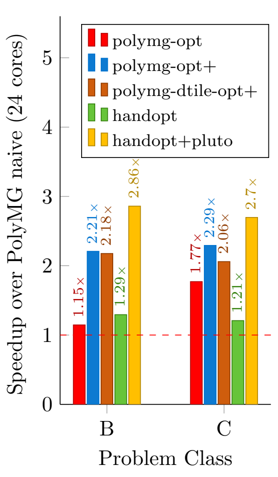


(c) 2D-W-4-4-4

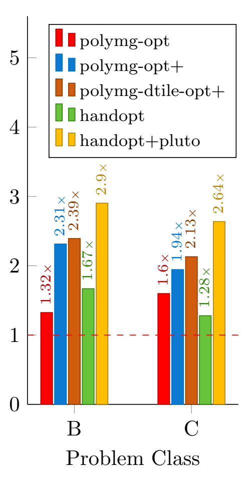


(d) 2D-W-10-0-0

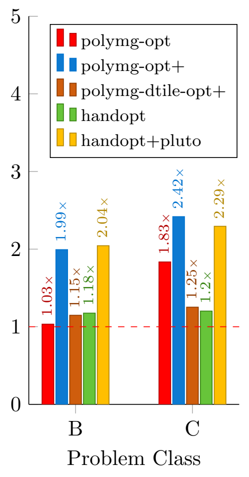
Figure 9: Performance: speedups and scaling for 2D benchmarks — absolute execution times can be determined using Table 3.



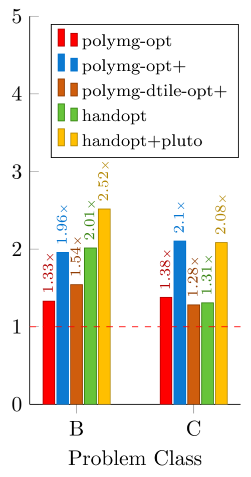
(a) 3D-V-10-0-0



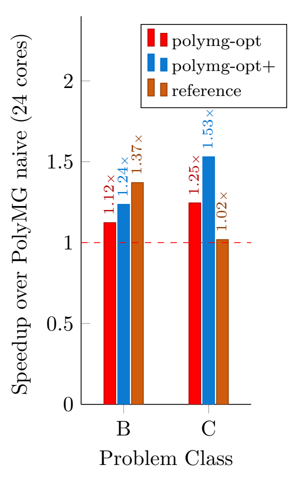
(b) 3D-W-10-0-0



(c) 3D-V-4-4-4



(d) 3D-W-4-4-4



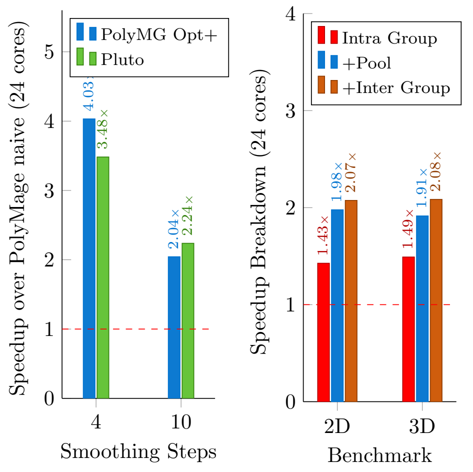
(e) NAS-MG

Figure 10: Performance: speedups and scaling for 3D benchmarks — absolute execution times can be determined using Table 3.

to running on a single core, while polymg-opt+ with all its optimizations for locality and parallelism provides a final speedup of 33.3× on 24 cores over the sequential polymg-naive. Similarly, for V-3D-4-4-4 class C, polymg-opt+ delivers a speedup of 10.8× when run in parallel on 24 cores over sequential polymg-naive, while the corresponding speedup for polymg-naive on 24 cores is only 4.47×.

Figure 11b shows the speedup breakdown for our storage optimizations over polymg-naive on 24 cores. The best performing Opt+ configurations of both 2D and 3D V-cycle 10-0-0 benchmarks were chosen for this analysis. The three bars in the plot refers to (a) enabling just the intra-group storage reuse, (b) enabling (a) and pooled memory allocation, and finally (c) enabling inter-group reuse with (a) and (b). It can be clearly seen that pooled memory allocation exploits inter-group reuse opportunities, even if the latter is not enabled. Generating code with inter-group reuse provides further performance.

Figure 6 shows the grouping and storage mapping for the best performing code for 2D V-cycle 4-4-4 benchmark. In this case, the code contains ten groups, all of which, except one, were overlap tiled (exception was the single defect node in a group, which does not require tiling for temporal reuse). The sizes of groups varied between a minimum of one to a maximum of six nodes. Among these, some contain smoothing steps fused with interpolation while others contain defect or smoothing steps fused with restrict. This shows that fusing computations across levels is indeed beneficial for performance. Figure 12 shows the execution times of variants generated with different tile size and group size configurations while auto-tuning for best performance. These measurements were made for the class C 2D-V-10-0-0 benchmark. We observe here that



(a) Smoothing Steps **(b) Inter Group Storage Opt**

Figure 11: Smoother performance comparison for 3D Class C and storage optimization for V-10-0-0 for 2D/3D

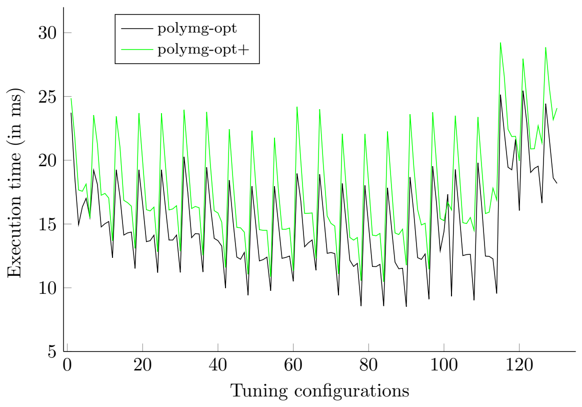


Figure 12: Execution times of various configurations from autotuning. These configurations vary in group size and tile size.

the polymg-opt+ variant always performs better than the polymg-opt one for the same configuration. Adjacent configurations in x-axis differ only in tile size within a range, for which the group size is constant. Thus, a clear repetitive pattern can be seen throughout the plot, as variants with the same tile size exhibit similar performance characteristics.

5 RELATED WORK

As the geometric multigrid algorithm has been popular due to its low computational complexity and amenability to parallelization, a number of past works have focused on improving its performance [3, 4, 7, 8, 12, 20]. All of these approaches were either manual or semi-automatic in the application of their optimizations. We discuss and compare with these in more detail below.

Williams et al. [20] explored optimization techniques for geometric multigrid on several multicore and many-core platforms; the optimizations appear to have been applied manually, and mainly included communication aggregation, wavefront-based technique to reduce off-chip memory accesses, and fusion of residual computation and restriction. All of these optimizations relate to optimizations we automated. The communication aggregation technique is equivalent to overlapped tiling, but applied in a distributed-memory parallelization setting: a deeper ghost zone is communicated and redundant computation at the boundaries is performed to reduce communication frequency. However, as the authors note, the tile size in play is not sufficient to exploit cache locality, and the authors use a wavefronting method [22] to keep a certain number of lower-dimensional planes in smaller working memory that fits in L3 cache, and stream through the larger working set on the node. This technique is equivalent to a transformation involving a loop skewing followed by a tiling, and is complex to automate with local buffers. The same wavefronting techniques is automated via a composition of loop transformations by Basu et al. [4]. In contrast to the wavefront method, our approach performs overlapped tiling for both locality and shared-memory parallelization (with a single level of tiling) — the trapezoidal tile fits in L2 cache. We find this strategy also suitable for easier automation. The grouping algorithm used in PolyMG encompasses residual-restriction fusion. In summary, in contrast to the approaches of [12, 20] (a) our approach has been towards building language and automatic compiler optimization support via a DSL tool, and (b) the optimization approach we have considered takes a holistic view of all steps involved in geometric multigrid.

Ghysels and Vanroose [8] used state-of-the-art stencil optimization techniques from compiler research to improve the arithmetic intensity of geometric multigrid algorithm, and while doing that, analyzed the trade-offs between convergence properties and the improved arithmetic intensity. The authors applied recent polyhedral tiling techniques to the smoothing iterations of the algorithm to improve data reuse. The impact of increasing the number of smoothing steps on arithmetic intensity, and on the time to solution was also studied through the roofline performance model. Our work has mainly been inspired by results obtained by Ghysels and Vanroose [8]. However, instead of applying tiling techniques to just the smoothing steps, we have taken a global view of optimizing the entire multigrid computation, and have explored opportunity to perform tiling and fusion across all steps. Enabling such optimization in an effective way is in fact one of the objectives of using a DSL. While [8] used Pluto's diamond tiling technique [2], we used overlapped tiling to allow storage optimization using local buffers (scratchpads), at the expense of redundant computation. Our experimental evaluation (Section 4) included a comparison with code obtained from [8], and the difference in performance was discussed therein.

Basu et al. [3, 4] studied optimization of geometric multigrid algorithm by application of certain compiler transformations in a semi-automatic way: the optimizations were specified using a script-driven system (using CHiLL), while the code generation was automatic given the specification. This work considered fusion across multigrid stages, hyper-trapezoidal tile shapes with redundant computation to avoid or reduce communication phases, and finally loop skewing to create a wavefront for multithreading. It thus automates most of the optimizations considered by Williams et al. [20] by composing the transformations using CHiLL. However, the fused groups are not tiled for locality: such tiling with tile sizes is necessary to improve L2 cache locality and reduce synchronization frequency; in addition, wavefronting suffers from pipelined startup and drain phases, which may be significant for certain problem sizes and a large number of cores. Like the implementation of Ghysels and Vanroose [8], ours does not involve pipelined startup.

Snowflake [23] is a recent DSL for stencils embedded in Python, developed independently and concurrently with our work. The language provides support for non-unit strided accesses, in-place updates, complex boundary conditions, and variable co-efficient, polynomial indexing. A greedy algorithm is used to form groups of stencils to compute all stencils within a group in parallel. Tiling and multi-color reordering is performed for OpenMP backend code. However, the approach does not perform any storage optimizations, which give significant performance improvement in case of Multigrid as we have shown (Figure 11).

As briefly mentioned in Section 1, image processing pipelines have similar properties as geometric multigrid algorithms. Hence, optimizations techniques studied for DSLs such as Halide [17] and PolyMage [14] were also applicable here. Unlike for image processing pipelines, both 2-d and 3-d grids are important for the Multigrid method. Three-dimensional grids are particularly more important here, whereas image processing filters or stages typically process two-dimensional data. Three dimensional grids have different implications on the choice of tiling strategy. In addition, we demonstrated benefits due to additional memory optimizations that included intra and inter group buffer reuse, and pooled memory allocation. On the language side, the addition of the TStencil construct allowed a compact specification for pre-smoothing and post-smoothing steps.

Adams [1] developed a low-memory, highly concurrent Multigrid algorithm that computes the coarse grid values as the fine grid is swept through, without storing the entire fine grid, and while maintaining convergence rate. The approach improves locality, reduces memory requirement, and increases the ratio of computation to memory. We plan to consider implementing such algorithmic advances in our work in the future.

6 CONCLUSIONS

In this paper, we demonstrated how high performance could be achieved along with enhanced programmability for GMG, through a domain-specific optimization system. We compared our approach with the existing optimization system of PolyMage, and codes hand-optimized in conjunction with Pluto, on benchmarks varying in multigrid cycle structure and smoothing steps among other aspects. Our automatically optimized codes obtained a mean performance improvement of 3.2× over a straightforward parallelization, 1.31× over an adapted state-of-the-art approach, while running on a 24-core shared-memory parallel system. We were also able to match or even outperform hand-optimized code in most cases. In the future, we plan to add a distributed-memory backend for our DSL leveraging existing distributed-memory code generation work [6], which is already available in the libPluto component we used. We also plan to integrate our approach into open community-driven efforts such as HPGMG.

ACKNOWLEDGMENTS

We would like to acknowledge Pieter Ghysels and Vim Wanroose for providing us their implementations of a 2-d and a 3-d Multigrid benchmark. We are grateful to the reviewers of SC 2017 for their feedback. We would like to thank Google for an unrestricted research gift under the Google Faculty Research Awards program in 2015.

REFERENCES

[1] M. F. Adams. 2012. A low memory, highly concurrent multigrid algorithm. *ArXiv e-prints* (July 2012). arXiv:math.NA/1207.6720

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC*. Article 40, 11 pages.

[3] Protonu Basu, Mary W. Hall, Samuel Williams, Brian van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *IEEE IPDPS*. 313–323.

[4] 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 *20th International Conference on High Performance Computing (HiPC)*. 452–461.

[5] William L. Briggs, Van Emden Henson, and Steve F. McCormick. 2000. *A Multigrid Tutorial (2Nd Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[6] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. 2013. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory. In *PACT*.

[7] Pieter Ghysels, Przemyslaw Klosiewicz, and Wim Vanroose. 2012. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Lin. Alg. with Applic.* 19, 2 (2012), 253–267.

[8] Pieter Ghysels and Wim Vanroose. 2015. Modeling the Performance of Geometric Multigrid Stencils on Multicore Computer Architectures. *SIAM J. Scientific Computing* 37, 2 (2015).

[9] GMG Tiled 2015. Pieter Ghysels, Geometric Multigrid Tiled. (2015). https://bitbucket.org/pghysels/geometric_multigrid_tiled.

[10] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. 2013. Split tiling for GPUs: automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 24–31.

[11] Christian Grossmann, Hans-GÃűrg Roos, and Martin Stynes. 2007. *Numerical treatment of partial differential equations*. Springer, Berlin, Heidelberg, New York.

[12] Markus Kowarschik, Ulrich Rüde, Christian Weiß, and Wolfgang Karl. 2000. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing* 64, 4 (2000), 381–399.

[13] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN PLDI*.

[14] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *ASPLOS*. 429–443.

[15] Pluto 2008. PLUTO: An automatic parallelizer and locality optimizer for affine loop nests. (2008). http://pluto-compiler.sourceforge.net.

[16] PolyMage GitHub project, Apache 2.0 license 2016. PolyMage. (2016). https://github.com/bondhugula/polymage.

[17] 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 *ACM SIGPLAN PLDI*. 519–530.

[18] Sven Verdoolaege. 2010. An integer set library for program analysis. (2010). http://isl.gforge.inria.fr/.

[19] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *International Congress Conference on Mathematical Software*. Vol. 6327. 299–302.

[20] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian van Straalen, Mikhail Smelyanskiy, Ann S. Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. 2012. Optimization of geometric multigrid for emerging multi- and manycore processors. In *SC*.

[21] D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IEEE IPDPS*. 171 –180.

[22] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U Rude, and G. Hager. 2008. Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method. *Progress in Computational Fluid Dynamics* 8 (2008), 179–188.

[23] N. Zhang, M. Driscoll, C. Markley, S. Williams, P. Basu, and A. Fox. 2017. Snowflake: A Lightweight Portable Stencil DSL. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 795–804.

A ARTIFACT DESCRIPTION APPENDIX

The entire PolyMage MG compiler, the benchmarks evaluated (including reference versions), and the PolyMG DSL versions of all benchmarks are available under a permissive open-source license at the URL below as part of the PolyMage project.
Repository web page: https://bitbucket.org/udayb/polymage

git clone URL: https://bitbucket.org/udayb/polymage.git

The entire code generation process can also be reproduced through the repository. The Multigrid benchmarks used in this paper can be found in the directory *sandbox/apps/python/multigrid/*. Detailed instructions on setting parameters for the benchmarks and on running them can be found in *sandbox/apps/python/multigrid/README.md*.