Automatic Optimization of Geometric Multigrid Methods Using A DSL Approach

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Science (Engineering)

IN COMPUTER SCIENCE AND ENGINEERING

by

Vinay V Vasista

Department of Computer Science and Automation

Indian Institute of Science

BENGALURU – 560 012

JULY 2017

Acknowledgements

It is my extreme pleasure to have got the opportunity to be a research student at the Indian Intitute of Science. First and foremost, I would like to thank the visionary founders of this prestigious Institute. I thank my advisor Dr. Uday Kumar Reddy B., for his invaluable guidance and support throughout my Masters degree. With his advice, I have always been able to step in the right direction during times of falter. Working with him has greatly helped me improve my research quotient, as well as become a better engineer today.

My collaborations with Ravi Teja Mullapudi, Irshad Pananilath, Aravind Acharya, Kumudha Narasimhan, Nitin Chugh and Siddharth Bhat were filled with great enthusiasm and gave me plenty of insights into several topics. I would like to thank them for their patience and support. My special thanks to Ravi Teja for patiently helping me refine my understanding of intricate aspects in high performance computing. Working on the PolyMage project with him and Dr. Uday was full of fun and learning.

The course on computer architecture by Dr. R Govindarajan taught me well, not only the subject, but also the basic art of reading and reviewing research works. I thank him for his lectures and encouragement.

I would like to thank all the staff in the department, including Mrs. Suguna, Mrs. Lalitha, and Mrs. Meenakshi, for ensuring that my time at the department was hassle free.

I also thank my family and friends for always being patient and supportive in balancing my personal and academic life. I specially thank my parents for their guidance and love.

Lastly I thank Google, ACM SIGPLAN and IARCS for their timely financial grants.

Publications based on this Thesis

1. Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, Uday Bondhugula. *Optimizing Geometric Multigrid Method Computation using a DSL Approach,* 2017, accepted at Supercomputing (SC'17), yet to be published.

2. Ravi Teja Mullapudi, Vinay Vasista, Uday Bondhugula. *PolyMage: Automatic Optimization for Image Processing Pipelines*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar 2015, Istanbul, Turkey.

Abstract

Geometric Multigrid (GMG) methods are widely used in numerical analysis to accelerate the convergence of partial differential equations solvers using a hierarchy of grid discretizations. These solvers find plenty of applications in various fields in engineering and scientific domains, where solving PDEs is of fundamental importance. Using multigrid methods, the pace at which the solvers arrive at the solution can be improved at an algorithmic level. With the advance in modern computer architecture, solving problems with higher complexity and sizes is feasible — this is also the case with multigrid methods. However, since hardware support alone cannot achieve high performance in execution time, there is a need for good software that help programmers in doing so.

Multiple grid sizes and recursive expression of multigrid cycles make the task of manual program optimization tedious and error-prone. A high-level language that aids domain experts to quickly express complex algorithms in a compact way using dedicated constructs for multigrid methods and with good optimization support is thus valuable. Typical computation patterns in a GMG algorithm includes stencils, point-wise accesses, restriction and interpolation of a grid. These computations can be optimized for performance on modern architectures using standard parallelization and locality enhancement techniques.

Several past works have addressed the problem of automatic optimizations of computations in various scientific domains using a domain-specific language (DSL) approach. A DSL is a language with features to express domain-specific computations and compiler support to enable optimizations specific to these computations. Halide [34, 35] and PolyMage [28] are two of the recent works in this direction, that aim to optimize image processing pipelines. Many computations like upsampling and downsampling an image are similar to interpolation and restriction in geometric multigrid methods.

In this thesis, we demonstrate how high performance can be achieved on GMG algorithms written in the PolyMage domain-specific language with new optimizations we added to the compiler. We also discuss the implementation of non-trivial optimizations, on PolyMage compiler, necessary to achieve high parallel performance for multigrid methods on modern architectures. We realize these goals by:

- introducing multigrid domain-specific constructs to minimize the verbosity of the algorithm specification;
- storage remapping to reduce the memory footprint of the program and improve cache locality exploitation;
- mitigating execution time spent in data handling operations like memory allocation and freeing, using a pool of memory, across multiple multigrid cycles; and
- incorporating other well-known techniques to leverage performance, like exploiting multi-dimensional parallelism and minimizing the lifetime of storage buffers.

We evaluate our optimizations on a modern multicore system using five different benchmarks varying in multigrid cycle structure, complexity and size, for two- and three-dimensional data grids. Experimental results show that our optimizations:

- improve performance of existing PolyMage optimizer by 1.31×[1];
- are better than straight-forward parallel and vector implementations by 3.2×[1];
- are better than hand-optimized versions in conjunction with optimizations by Pluto, a state-of-the-art polyhedral source-to-source optimizer, by 1.23×[1]; and
- achieve up to 1.5× speedup over NAS MG benchmark from the NAS Parallel Benchmarks.

[1]Geometric mean over all benchmarks

Contents

List of Tables

Keywords

Domain-specific languages, Geometric multgrid methods, Polyhedral optimization, Locality, Parallelism, Tiling, Storage optimization, Multi-cores, Vectorization

List of Figures

List of Algorithms

Chapter 1

Introduction

This chapter introduces geometric multigrid methods, and the importance of optimizing its implementations for high performance and describes the motivation behind our work, which is to achieve this goal through automation.

1.1 Geometric Multigrid Methods

The stencil approach using a finite difference method for discretization is currently the dominant approach for the numerical solution of partial differential equations (PDEs) [16]. A visualization example of the heat transfer in a pump casing solved using heat equation is shown in figure 1.1 Stencil computation can be viewed as data parallel operations applied repeatedly on a structured grid, where computation at each grid point involves values only from neighbouring grid points by application of a geometric pattern of weights (hence *stencil*). This kind of computation, thus exhibits spatial and temporal locality. One of the key reasons why a stencil approach is attractive is its suitability for a parallel and high-performance implementation in a variety of settings, like 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

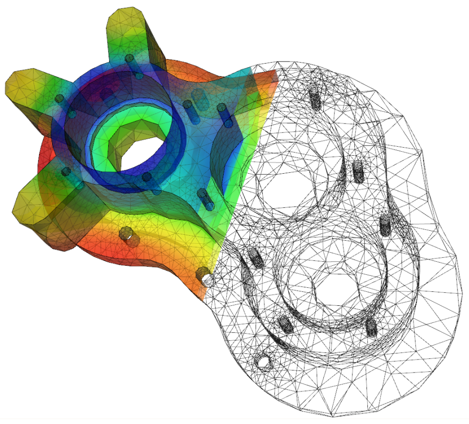


Figure 1.1: Visualization of heat transfer in a pump casing created by solving the heat equation [image credits:Wikimedia Commons CC BY-SA 3.0 [17]]

solvers that quickly converge to the solution, i.e., in fewer iterations. Multigrid methods [5] solve the problem at multiple grid resolutions, and are widely used to accelerate this convergence 1.2. Improved convergence is achieved by solving the problem on a coarser problem, and approximating the solution to the finer problem's solution with necessary corrections. The process is carried out at multiple levels of coarsening, to arrive at the solution to the finest level from the coarsest one – this forms a multigrid cycle. The number of iterations of such a cycle, required to converge, depends on (but not limited to) the cycle type (Figure 1.3), quality of convergence, and steps used within the cycle. Multigrid algorithms can be used either as direct solvers or as pre-conditioners for solvers (example Krylov Solvers [26]). Algorithm 1 shows a typical multigrid V-cycle algorithm.

In this work, we consider evaluation of optimized multigrid program implementations that solve the Poisson's equation [5], which is given by

$$\nabla^2 u = f, \tag{1.1}$$

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.

Geometric multigrid methods (GMG) [5] fall under this category, and use a hierarchy

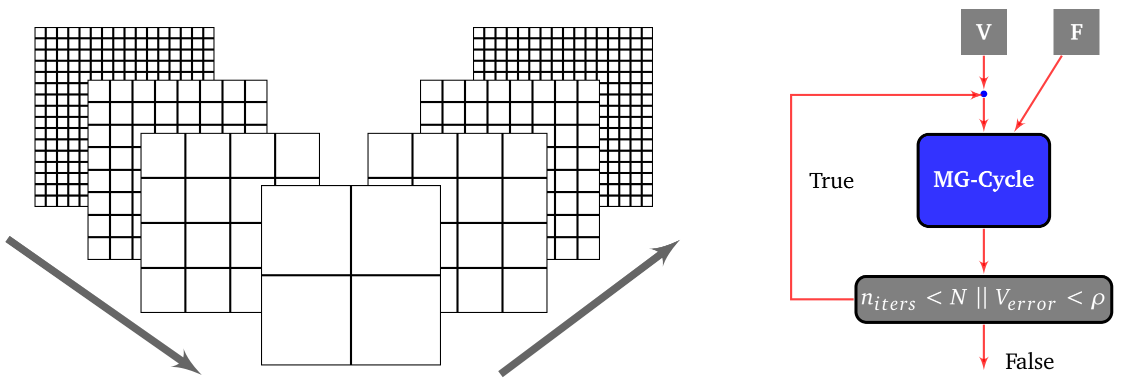


Figure 1.2: Illustration of multiple discretization levels in solving PDEs using a multigrid cycle

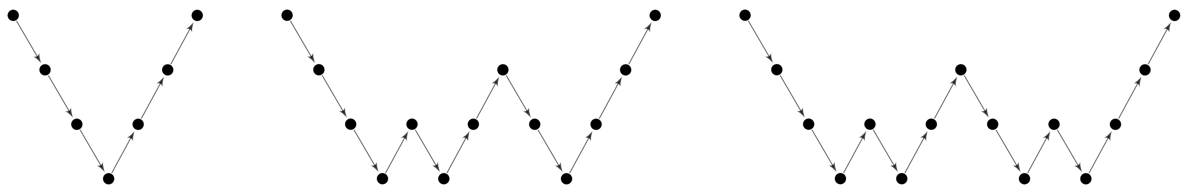


Figure 1.3: Different forms of multigrid cycles – V-cycle, F-cycle and W-cycle

Algorithm 1: A Multigird algorithm using V-cycle: *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 $v^h$
```

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. The underlying computations in a GMG 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.

1.2 Domain-Specific Languages and Optimizations

A domain-specific language (DSL) for high performance is one that exploits computational domain information to deliver productivity, high performance, and portability to programmers, through an optimizing compiler or code generator. With this facility, the domain-specific programmers can totally focus on expressing the algorithm in the DSL, without having to worry about the target architecture and the relevant set of optimizations needed to achieve high performance. Halide [34, 35] and PolyMage [28] are two such recently developed DSLs with an optimizing code generator for a class of image processing pipelines. Spiral [33], Green-Marl [20] and Liszt [9] are some notable domain-specific languages with optimization support for digital signal processing, graph analysis and mesh based PDE solvers, respectively.

The common computation patterns of image processing algorithms such as convolutions, downsampling, upsampling etc. can be used to define filters that can be composed to express a pipeline in these DSLs. These patterns exhibit one-to-one mapping to smoothing, restriction and interpolation respectively in the context of GMG algorithms. Thus, the very same constructs of the DSLs can seamlessly support programming for multigrid methods, and hence are good fit to express these algorithms in them. In this work, we will discuss how, with few extensions to the PolyMage [28] DSL and optimizer, we can effectively deal with expressing GMG algorithms and extraction of the latent high parallel performance.

1.3 Thesis Contributions and Organization

In this thesis, we present an extension to the PolyMage language to incorporate constructs specific to geometric multigrid methods to provide an easy programming environment to domain scientists and optimizing compiler/code generator. Chapter 2 describes the language features of PolyMage, its relevance to multigrid algorithms and the new constructs we added to the same. In Chapter 3 we discuss the implementation of non-trivial optimizations, on PolyMage compiler, necessary to achieve high parallel performance for multigrid methods on modern architectures. We achieve these goals by:

- introducing multigrid domain-specific constructs to minimize the verbosity of the algorithm specification in the language to capture all the basic computation patterns involved in it;
- storage remapping to reduce the memory footprint of the program and improve cache locality exploitation, which also leads to performance gains in execution time;
- mitigating execution time spent in data handling operations like memory allocation and freeing, using a pool of memory, that serves allocation requests across multiple multigrid cycles; and
- incorporating additional well-known techniques to leverage performance, like exploiting multi-dimensional parallelism inherent in the multigrid computations and minimizing the lifetime of storage buffers.

We evaluate our optimizations on a modern multicore system using five different benchmarks varying in multigrid cycle structure, complexity and size, for two and three dimensional data grids. The details of the experimental setup and analysis is given in Chapter 4. Experimental results show that our optimizations:

- improve performance of existing PolyMage optimizer by $1.31\times$[1];
- are better than straight-forward parallel and vector implementations by $3.2\times$ [1];
- are better than hand-optimized versions in conjunction with optimizations by Pluto, a state-of-the-art polyhedral source-to-source optimizer, by $1.23\times$[1]; and

[1]Geometric mean over all benchmarks

- achieve up to 1.5× speedup over NAS MG benchmark from the NAS Parallel Benchmarks.

We discuss related work and other studies in detail in Chapter 5, and present conclusions and future work for related research.

Chapter 2

Language

In this chapter, we will discuss the language support needed for GMG algorithms, about the PolyMage DSL and its constructs, and propose suitable extensions to the same so as to communicate multigrid specific computations to the PolyMage backend compiler minimally and more effectively.

2.1 GMG Algorithm in PolyMage DSL

Multigrid applications consist mostly of stencil computations or other point-wise data parallel operations. Grid accesses made during interpolation or restriction phases, however, involve stencils with a scaling factor, usually of 2 or 1/2. In other words, the output grid being written to will be half or twice the size of the input grid, respectively. Such computations are often common to the image processing domain and are used in upsampling or downsampling an image. PolyMage [28] is a domain-specific language embedded in python, that can capture these types of accesses for image processing applications. Execution reordering transformations such as tiling and fusion are also supported on these classes of computations. This makes PolyMage DSL well suited for providing a high-level abstraction to domain experts in writing GMG algorithms. We additionally introduce a small set of constructs to further reduce verbosity in expressing multigrid steps, as described in the following section 2.2. Typical computation patterns observed in GMG algorithms are enlisted in Table 2.1 and illustrated in Figure 2.1

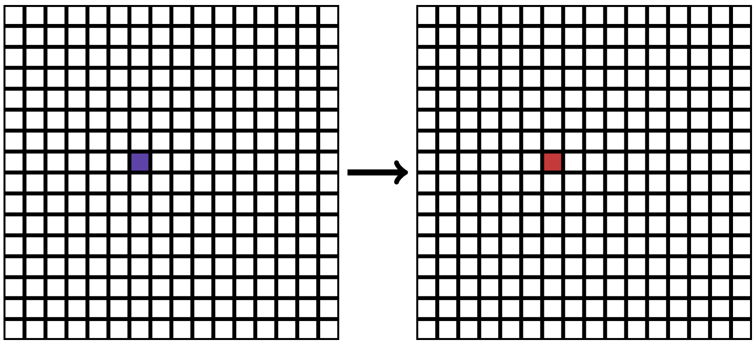
Operation	Example
Point-wise	$f(x,y) = g(x,y)$
Smoothing	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x+\sigma_x, y+\sigma_y)$
Interpolation	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g((x+\sigma_x)/2, (y+\sigma_y)/2)$
Restriction	$f(x,y) = \sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(2x+\sigma_x, 2y+\sigma_y)$
Time-iterated	$f(t,x,y) = g(f(t-1,x,y))$

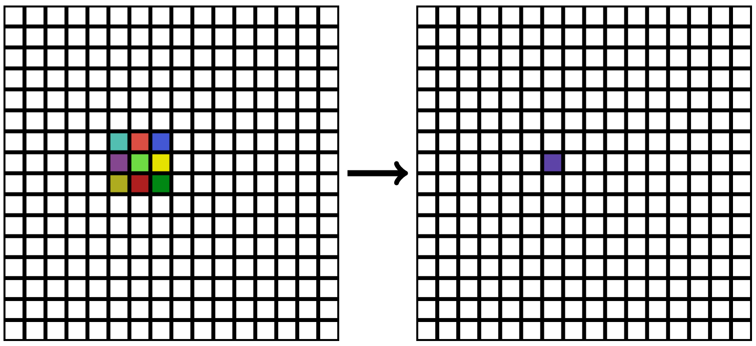
Table 2.1: Typical computation patterns in Geometric Multigrid algorithms

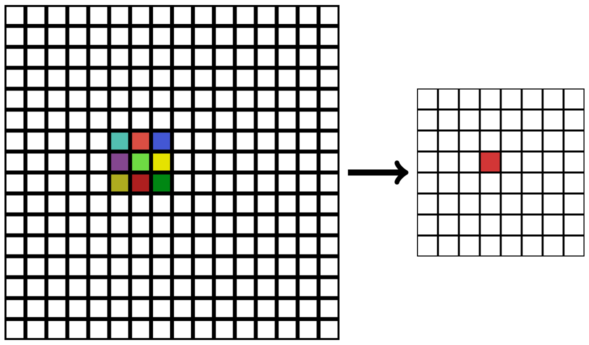
2.2 Language Constructs

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. Similarly the input structured grids can be declared using the `Grid` construct, The constructs `Parameter`, `Variables` and `Interval` are used to represent the grid size, variables to refer to the individual grid-elements, and the boundary in which each of the grid dimensions is defined, respectively. Using the `Condition` construct, one can express the logical conditional checks. With this and `Case` construct, piece-wise functions can be defined. The constructs supported by PolyMage language are enumerated in Table 2.2.

Another construct `Stencil`, can be used to express the weights of a stencil kernel for a 2-d grid 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 or origin 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.


(a) Point-wise


(b) Smoothing


(c) Restriction

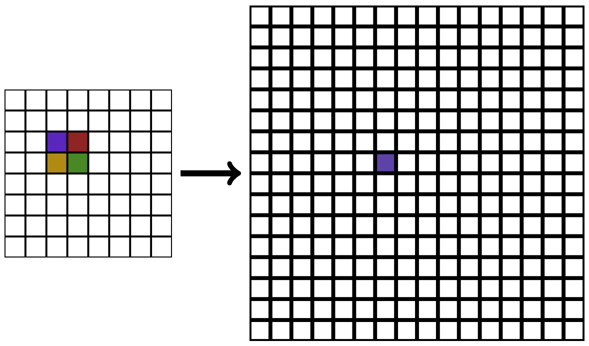

(d) Interpolation

Figure 2.1: Point-wise, Smoothing, Restriction and Interpolation operations

```
1.0/16 * Stencil(f, [x, y], [[0.0, 1.0], [-1.0,  2.0]])
```

translates to:

```
1.0/16 * ( f(x,y+1) - f(x+1,y) + 2.0*f(x+1,y+1) )
```

Language Construct	**Syntax**
Grid	*Name* = Grid(*type, list of dimensions*)
Parameter	*Name* = Parameter(*type*)
Interval	*Name* = Interval(*lower bound, upper bound, stride*)
Variable	*Name* = Variable()
Condition	*Name* = Condition(*expression, operator, expression*) where *operator* can be >, =, !=, <, >=, <=
Case	*Name* = Case(*condition, expression*)
Function	*Name* = Function(*(list of variables, list of intervals), type*) *Name*.defn = *<Case>*
Stencil	*Name* = Stencil(*Function, list of variables, kernel list, [origin]*) *where kernel list is a list of numbers or a list of kernel lists*
Restrict	*Name* = Restrict(*(list of variables, list of intervals), type*) *Name*.defn = *<Case>*
Interp	*Name* = Interp(*(list of variables, list of intervals), type*) *Name*.defn = *Dictionary of expressions for each interpolation Case*
TStencil	*Name* = TStencil(*(list of variables, list of intervals), type, T*) *Name*.defn = *Expression involving a* `Stencil`

Table 2.2: Syntax of key constructs in the PolyMage DSL; *Name*.defn represents the syntax to define the function *Name*

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 (i.e., 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 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 mitigate them.

Another important component of GMG cycles where `Stencil` construct finds its usage, is *smoothing*. This is repeatedly applied on the intermediate smoothened outputs for a particular number of iterations (say T), and this type of computations are commonly known as time-iterated stencils. In PolyMage, a programmer can express the same using python loops and one `Function` for each time step of the smoothing computation. This results in duplication of dependence information passed to the DSL. To overcome this, we introduced a special construct `TStencil`, similar to `Function`, that takes (T) as an extra parameter and a definition that involves `Stencil`. This not only makes the program description compact, but also simplifies the compiler analysis by eliminating the need to analyze similar redundant expressions. For this, we added the backend support to PolyMage that allows the initialization of the parameter T at runtime, thus making the algorithm generic to variable number of smoothing steps.

The PolyMage specification for a V-cycle algorithm is given in Figure 2.2. Figure 2.3 depicts the detailed pipeline DAG of V-cycle, W-cycle and the NAS-MG benchmark in the NAS Parallel Benchmarks (NPB) [29] suite, which is also used for experimental evaluation of this work. The NPB, developed and maintained by NASA Advanced Supercomputing Division, target performance evaluation of highly parallel supercomputers. The Multigrid benchmark (NAS-MG), available in this suite, which approximates the solution to a three-dimensional discrete Poisson equation, also uses a V-cycle algorithm, with no pre-smoothing steps.

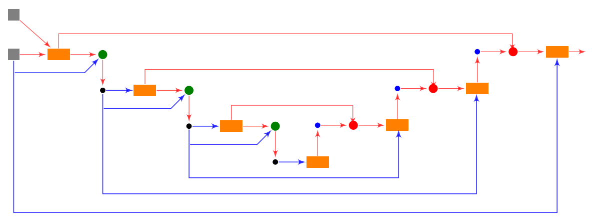
The boundary values of a function can be set with the help of `Case` construct, by creating a set of piece-wise definitions. Other basic language support including `Variable`, Parameter, Interval etc., are retained while extending the support for multigrid applications. More details on the language rules and usage can be referred to in [28].

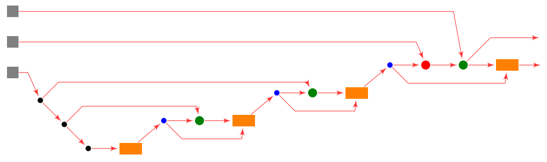
The PolyMage specification for the V-cycle Algorithm 1 is given in Figure 2.2 This program operates on initial guess grid V and the rhs grid F, for the parametric problem size N (value of h has to be derived using N). n1, n2 and n3 represent the pre-smoothing, post-smoothing and coarse-smoothing steps respectively. This PolyMage 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 TStencil construct. Similarly, restrict function uses Restrict and Stencil constructs. The interpolation function uses a python tuple of dictionaries to map multiple function definitions based on evenness of dimension variables, so that the Interp construct can use it to define the corresponding piece-wise function.

The PolyMage compiler sees the specification embedded in Python as a collection of functions defined over polyhedral domains and also as a directed acyclic graph with additional information specifying the instance-wise dependences, with the exact producer-consumer relationships. Thus, the specification is a feed-forward pipeline. The loop iterating over an entire V-cycle or W-cycle is thus external to PolyMage. A polyhedral representation is then constructed for this specification, followed by determining the right set of schedule and storage transformations to improve locality, parallelism, and storage compaction; code generation is then performed from the polyhedral specification of the domains and schedules, using cgen [6].

```
N = Parameter(Int, 'n')
V = Grid(Double, "V", [N+2, N+2])
F = Grid(Double, "F", [N+2, N+2])
...
def 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] = 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 = {}
  W = v
  W = TStencil(([y, x], [extent[l], extent[l]]),
               Double, n)
  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 2.2: PolyMage specification for Multigrid V-cycle


(a) V-cycle


(b) NAS-MG cycle

🟧 Smoother ● Defect/Residual • Restrict/Reciprocate • Interpolate/Prolongation ● Correction ■ Input

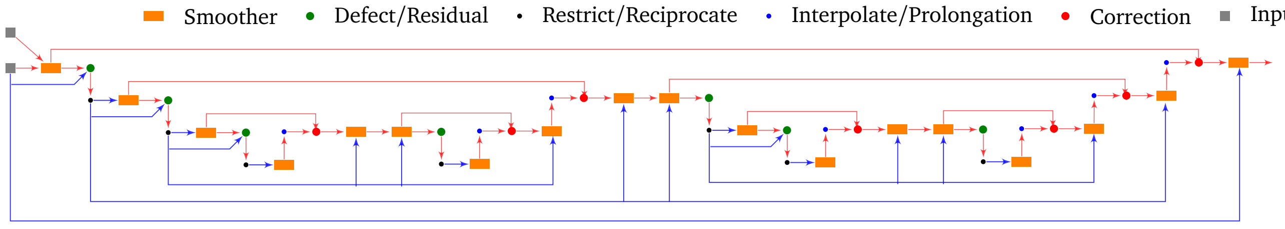

(c) W-cycle

Figure 2.3: Detailed pipeline DAG of different multigrid cycles

Chapter 3

Compiler Optimizations

In this chapter, we provide a brief overview of the existing PolyMage compiler phases, and describe in detail, the improvements and optimizations done on top of it.

3.1 PolyMage Compiler

The PolyMage compiler uses the program description written in the DSL to construct a directed acyclic graph (DAG) of nodes, that represent the functions and edges representing the producer-consumer relationship between two functions. It then statically analyses the function bounds and checks for out-of-bound accesses made to them, and asserts in case it found any, in order to avoid program crash during execution. Simple point-wise functions 2.1 and assignment operations are inlined into their consumers. Not doing so will lead to unnecessary array allocations to store computed values, for which register memory would have sufficed otherwise. In the next phase, the nodes are bundled together into groups, using a greedy grouping algorithm, on which many transformations like fusion, tiling and storage optimization etc. are applied later. It is in this phase, the compiler aligns and scales the dimensions of nodes within a group in order to minimize the dependence distances between the function points. Polyhedral representation of each node is obtained, with the help of Integer Set Library (ISL [22]) and its python wrapper islpy [23], before moving on to schedule transformations, using the interval and the constraint information

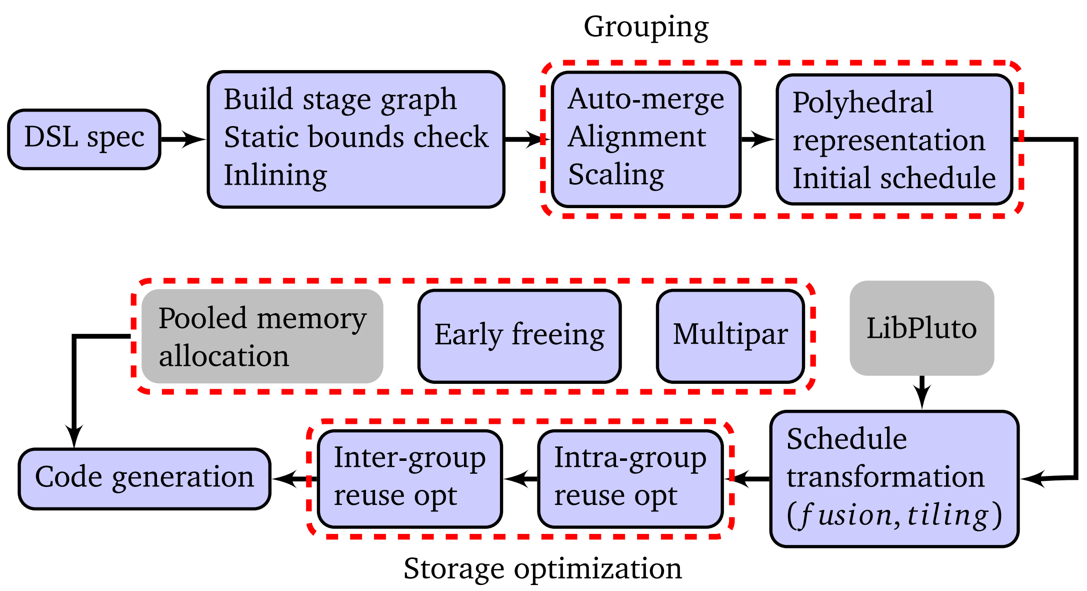


Figure 3.1: The PolyMage optimizer: various phases

provided by the programmer.

Each group is formed in such a way that the nodes within them can be fused and are amenable to overlap tiling, with a certain heuristic to maximize locality optimization enabled by it. Tile shape is determined based on the set of computations present in the group, and varies accordingly with different composition of patterns. PolyMage specializes this technique for multiscale accesses by inferring tighter bounds on the tile extents in each dimension, than those inferred by traditional overlap tiling techniques. Polyhedral transformations pertaining to this are then applied on the function domains. This is followed by a storage mapping pass which makes sure intermediate functions of a tile are allocated minimal scratchpad buffers just sufficient to store values computed within the tile. Since no communication across happens between any two individual tiles of a group, the scratchpads can be discarded once the tile execution is complete. Next, code generation phase takes over to produce a `C++` implementation that reflects all the transformations applied on the pipeline. PolyMage uses `cgen` [6], a `python` package, to crawl over the abstract syntax tree (AST) generated by `ISL`. Using `ctypes`, another python module that enables interoperability between `C++` and `python`, one can make calls to the generated implementation by passing the input and output grids in `numpy` [30] array format.

Figure 3.1 shows the different phases of PolyMage's optimizing code generator with the enhancements discussed in sections 3.2, 3.3, 3.4, and 3.5.

3.2 Grouping for Fusion and Tiling

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 [11]. Pluto's diamond tiling technique [2] 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 [18, 25, 19]. 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 [11]; 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 [11].

Ghysels and Vanroose [11] as well as most past work [24, 38], 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. [4, 3] considers fusion of different multigrid operators, it makes a fixed fusion choice, and storage optimizations are not considered. Related works Chapter 5 includes a more detailed discussion. Our approach addresses all of these limitations, and in addition, is fully automatic.

Execution re-ordering techniques like time skewing or parallelogram tiling [39] may not yield any or enough tiles to parallelize on the wavefront, and incurs pipelined startup.

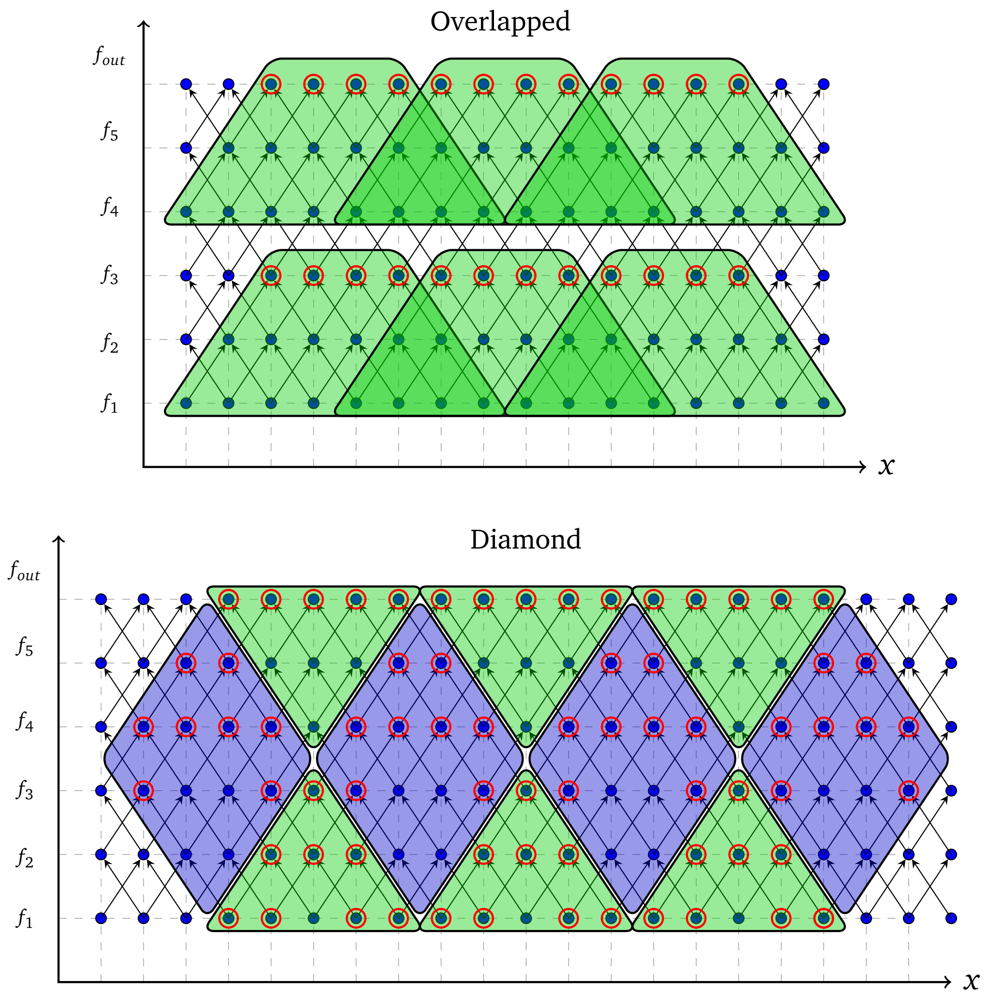


Figure 3.2: Overlapped tiling and diamond tiling. Data live out of a tile is encircled in red. Overlapped tiling has live-out data only at its top face. In diamond tiling, a green tile communicates the computed values at the boundary of all its faces (shown as live-outs) to its adjacent blue tiles, before the latter can start executing. While the overlapped tiling techniques suffer from the overhead of redundant computations, diamond tiling incurs more synchronization overhead

Stock et al. [36] explore the effects of associative reordering of computations in high order stencils to enhance register reuse. Array Sub-expression Elimination (ASE) is another technique that improves register reuse as well as the arithmetic intensity of the computations [8]. Our optimizations do not consider the low-level loop optimizations such as the above two. Overlapped tiling [25], diamond tiling [2], or split tiling [15] allow concurrent start and are more suitable to the set of GMG computations. In diamond tiling, a tile communicates the computed values at the boundary of all its faces to the tiles dependent on these values. While the overlapped tiling techniques suffer from the overhead of redundant computations, diamond tiling incurs more synchronization overhead. Hierarchical overlapped tiling [41], a variant of overlapped tiling, explores the efficiency of codes tiled using multiple levels of tiling, in order to balance synchronization overhead and redundant computation.

Another technique, known as hexagonal tiling, was proposed by Grosser et al. [14] for GPUs, which does not involve redundant computations like in overlapped tiling, and contains additional faces with adjustable width intended to aid compiler auto-vectorization. In this technique, tiles have to communicate boundary values at multiple faces, as in diamond tiling. A detailed analysis on the relation between diamond tiling and hexagonal tiling has been made by Grosser et al. [13]. The transformations applied by Pluto to schedule the iteration order for diamond tiling results in a tiled loop nest which cannot be flattened or linearized. This prevents the use of OpenMP's `collapse` directive to extract multi-dimensional parallelism, even though there exists no dependence between adjacent tiles in the same phase of diamond tiles. On the other hand, overlapped tiles neither have dependences among themselves nor is there a limitation in the PolyMage generated loop nest preventing the use of this directive.

We employ the same overlapped tiling strategy as the one used in PolyMage for image processing pipelines âĂŤ 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, the 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 adjacent tiles. Figure 3.2 illustrates both overlapped and diamond tiling schemes – only one dimension of space/grid is shown for simplicity.

Algorithm 2: Iterative grouping of stages

```
Input : DAG of stages, $(S, E)$; parameter estimates, $P$; group-size threshold, $g_{thresh}$
// Initially, each stage is in a separate group
1  $G \leftarrow \emptyset$
2  for $s \in S$ do
3      $G \leftarrow G \cup \{s\}$
4  repeat
5      $converge \leftarrow true$
       // Find all the groups which only have one child
6      $cand\_set \leftarrow$ getGroupsWithSingleChild($G$, $E$)
       // Sort the groups by size to give priority larger groups
7      $ord\_list \leftarrow$ sortGroupsBySize($cand\_set$, $P$)
8      for each $g$ in $ord\_list$ do
9          $child =$ getChildGroup($g$, $E$)
           // Check if the stages group $g$ and the $child$ group can be scaled and aligned
              to have constant dependence vectors
10         if hasConstantDependenceVectors($g$, $child$) then
               // Get the
11             $new\_size \leftarrow |g| + |child|$
12             if $new\_size < g_{thresh}$ then
13                 $merge \leftarrow g \cup child$
14                 $G \leftarrow G - g - child$
15                 $G \leftarrow G \cup merge$
16                 $converge \leftarrow false$
17                 break
18 until $converge = true$
19 return $G$
```

Halide [34, 35], a domain-specific language and compiler for image processing algorithms allows programmers to express both the computations and the schedule in their programs. The language offers automatic code generation for fusion and overlapped tiling, if specified by the programmer. Finding a good grouping of functions for tiling is practically hard, given the complexity of the function computations, number of parameters affecting the performance on modern parallel architectures, and number of valid grouping choices. Halide uses an external framework OpenTuner [1], to explore good schedules for a given pipeline. OpenTuner uses a genetic algorithm approach to combine different schedules while the tuning is being carried out, to find optimal ones. The search space of the schedules considered by the tuner, however, becomes exponentially large for pipelines with tens of functions.

PolyMage uses a greedy heuristic to generate a set of function groups for tiling, based on tile size and group size thresholds. We use the same auto-grouping algorithm (Algorithm 2) to group multigrid functions. Starting off with the default singleton groups for each function, the algorithm merges two groups in each iteration, until no further merge can be made. In each iteration, a list of groups with only one child is collected, sorted based on their sizes. For each group in the list, validity of merging the group and its child is evaluated. The validity is determined by two factors — whether the resultant merged group is amenable for overlapped tiling, and whether its size (number of functions contained) is less than the threshold for group size. The algorithm stops once the set of groups is saturated and no more merging can happen between any two groups.

Figure 3.3 shows the grouping for a 2-D V-cycle configuration that performed the best with our optimizations in an experimental setting explained in Chapter 4. In this figure, the scratchpad nodes represent the stages having no use outside the tiled group, with a memory requirement of the order of tile sizes. The liveout nodes are those which have at least one use outside the group, hence they require full array allocations. The colours of the nodes indicate the reuse of storage among similar colour and type of nodes. In summary, no changes were needed to the fusion and tiling approach employed already employed in PolyMage [28].

3.3 Memory Optimizations

PolyMage abstracts away memory allocation, management, and indexing from the programmer. This gives complete freedom to the compiler to decide on 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

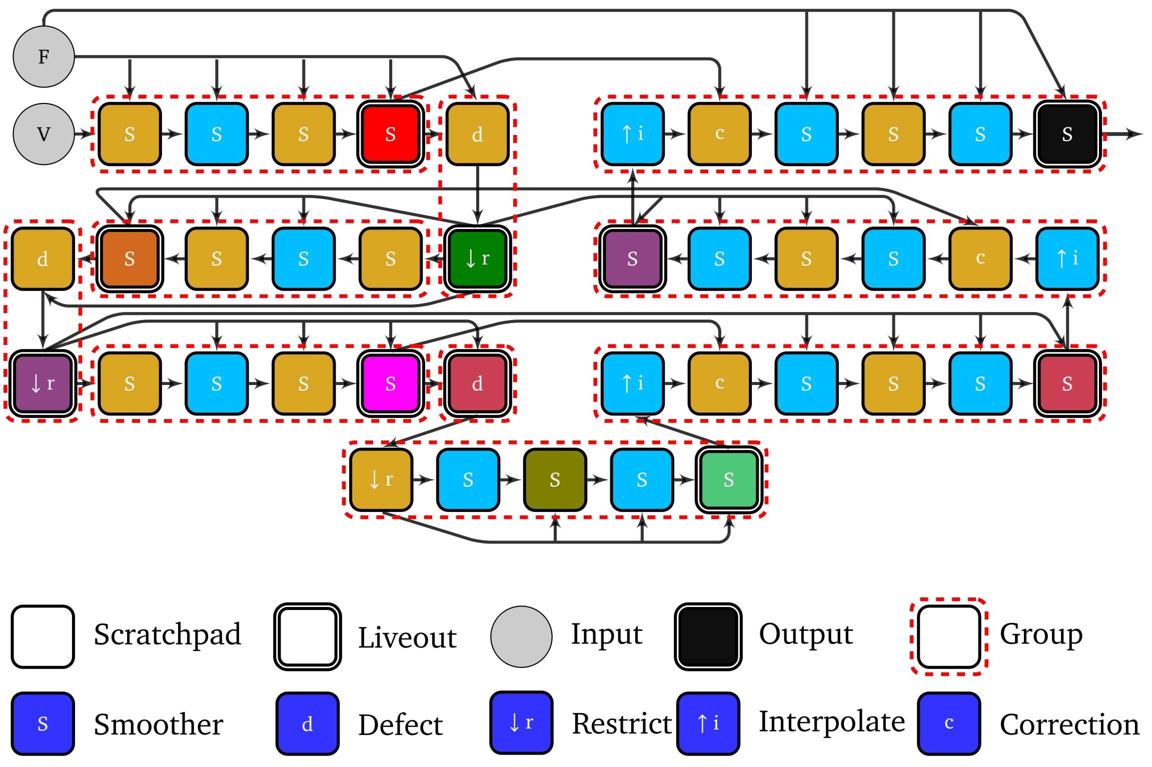


Figure 3.3: Grouping (fusion of operators): the dashed boxes correspond to a fused group

like alias analysis, escape analysis, and delinearization are often necessary to perform data reordering in a safe way. Using a DSL avoids these difficulties, and on the other hand it would increase the responsibility of the compiler in figuring out the best scheme for efficient memory allocation. By achieving an improved programming productivity, the DSL compiler lets the programmer be oblivious of the best performance practices for the target machine. Hence, the compiler would naturally require additional time for applying its own intelligence during compilation. This should be acceptable as long as the returns are favourable in terms of the performance of the generated executable code.

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 the 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. Such constant-sized buffers

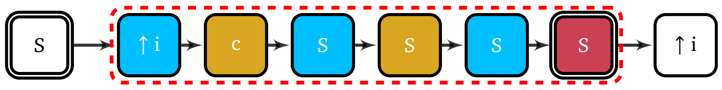


Figure 3.4: An illustration of scratchpad reuse in one of the pipeline groups

(one per each thread), declared within the scope of the tiled loop nest, use the stack space of threads. The live-in and live-out functions need to be live across groups, hence full array allocations (order of domain size) are made using calls to `malloc`.

In multigrid cycles, the output of many functions like the intermediate steps in smoother, 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. Thus, the total allocated storage space can often be more than what is necessary to hold the computed intermediate data. We address this issue by introducing an additional compiler pass, to extract reuse opportunities using a best effort approach. This is done at two levels – reusing the scratchpads within a group, and reusing full arrays across groups. Both these passes require the groups of functions and the function chunks inside an overlapping tile to have been already scheduled, with a total order specified.

Functions are initially classified into different storage classes, and 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 and functions that are group live-outs.

3.3.1 Intra-Group Buffer Reuse

Though, with scratchpad reuse, it might seem that we get too little a reduction in the total memory consumption of the program, the gain is not just about the bytes count. An overlapped tiled code gets maximum performance benefit when all its scratchpads fit in the cache closest to the processor. Without achieving this, it is not just the potential performance that will be ignored, but a more accurate behaviour of a tile size configuration that remains tampered. Effects of this particular optimization is discussed in detail in the Chapter 4.

Algorithm 3: Procedure to find last use of functions in a pipeline: *getLastUseMap*

```
   Input : set of functions : $F$, Timestamp map : $T$
1  $freeMap \leftarrow \emptyset$
2  for each $func \in F$ do
3      $lastUse \leftarrow -1$
       // get timestamp of child that uses $func$ the last
4      for each $child \in children(func)$ do
5          $t \leftarrow T(child)$
6          $lastUse \leftarrow max(lastUse, t)$
       // add $func$ to the list of functions having the same $lastUse$ timestamp
7      $freeMap(lastUse) \leftarrow freeMap(lastUse) \cup func$
8  return $freeMap$
```

The constant-sized nature of tile scratchpads makes the search for reuse candidates simpler, compared to full arrays with parametric bounds. Scratchpads that have exactly equal size in each dimension come under the same class, however this can be relaxed by a ±constant value. We use a simple greedy approach to colour the classified functions within a group for storage reuse. First, the entire group DAG is scanned for the last use of each function and it's scheduled time within the group. Using this information, a second pass is made to emulate a walk in the schedule order, to decide whether to allocate a new storage object to a function or to remap an already used object whose user function is no longer live. Algorithm 4 gives high level details of remapping algorithm, applicable to both intra- and inter-group storage optimizations. A subroutine, to find the last usages of all functions in the DAG, needed for Algorithm 4, is presented in Algorithm 3.

An illustration of intra-group reuse (for scratchpads) enabled in one of the groups in Figure 3.3, is shown in Figure 3.4, 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. Excluding the last smoothing step, which is live out of the group, all other nodes are allocated as scratchpads to store 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 its 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 colour get to use the same scratchpad buffer, which

Algorithm 4: *remapStorage* Algorithm to remap PolyMage functions to arrays

```
Input   : set of functions : $F$, Timestamp map : $T$
1  $lastUseMap \leftarrow getLastUse(F, T)$
   // Sort the functions with timestamp as the key
2  $F_{sorted} \leftarrow sort(F, key = T(f))$
3  $C \leftarrow \bigcup_{f \in F} storageClass(f)$
   // Initialize the array pool of all storage classes to empty
4  $arrayPool(c) \leftarrow \emptyset, \forall c \in C$
5  $storage(f) \leftarrow \emptyset, \forall f \in F$
6  $arrayID \leftarrow 0$
7  for each $func \in F_{sorted}$ do
8      $c \leftarrow storageClass(func)$
       // If array pool is empty, return a new array
9      if $arrayPool(c) = \emptyset$ then
10         $arrayID \leftarrow arrayID + 1$
11         $storage(func) \leftarrow arrayID$
       // Else return an unused array from the pool
12     else
13         $storage(func) \leftarrow pop(arrayPool(c))$
14     $t \leftarrow T(func)$
       // If some function has no use after this timestamp
15     if $t \in lastUseMap$ then
16         for each $freeFunc \in lastUseMap(t)$ do
               // Return the arrays to array pool
17             $c' \leftarrow storageClass(freeFunc)$
18             $arrayID \leftarrow storage_{freeFunc}$
19             $arrayPool(c') \leftarrow arrayPool(c') \cup arrayID$
20 return storage
```

is allocated privately for each thread executing a tile corresponding to this group. In the given example, only two buffers are sufficient to finish the tile computation, as opposed to using five buffers without reuse.

3.3.2 Inter-Group Array Reuse

Compaction of the set of full-arrays plays a major role in minimizing the resident memory of a running program. This will also reduce the number of page faults since new arrays would be touched fewer number of times. For smaller problem sizes, this optimization can yield performance gains by letting the entire data fit in the last level cache, which otherwise may not be possible. Similarly, for very large problem sizes, not optimizing for storage can cause the allocated area exceed the available space in main memory, resulting in movement of data in and out of swap memory. This situation can be avoided by reusing arrays to serve as storage for multiple functions.

Full-arrays can have parametric sizes in each dimension, which makes it quite tricky to classify them. If the intermediate arrays contain ghost zones (boundary paddings), they typically differ among each other by constant offsets. Such arrays are collected under the same class, if the set of parameters of all of their dimensions is the same. In addition, the coefficients of the dimension parameters also determine which storage class an array shall belong to. Our storage classifier makes sure that arrays of size, say `(M/2)×(N)`, come under the same class of arrays whose size is `(M)×(N/2)`. Finally, the representative size of the entire storage class is calculated using 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, which does not include any parametric sized array. Program input and output arrays are not considered 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 them is allowed to reuse the array. This constraint is taken care of by the remapping algorithm which assigns them array tokens taken from and returned to a common pool. Allocation and deallocation of full-arrays happen at the granularity of a group. The same Algorithm 4 is used to optimize reuse for liveout arrays. In Figure 3.3, the liveout nodes are coloured to show reuse among the same classes (in this case for multigrid levels) of arrays.

3.3.3 Pooled Memory Allocation

The optimization of the multigrid programs, using PolyMage, is limited to one multigrid cycle. Because of this, any opportunity to extract performance across cycles (Figure 1.2), have to be done outside PolyMage. One such optimization is to use a pool of memory, that can serve the allocation and deallocation requests of the PolyMage generated code. We use a pooled memory allocator to do this, with appropriate interface calls generated along with the PolyMage output code. C++ `malloc` function call is proxied through `pool_allocate`. Thus, all the intermediate buffer allocation requests have to go through this allocator, which scans over a list of already allocated buffers to decide whether to create a new one or to return a pointer to an unused allocated array. Reuse of arrays is permitted only if a request matches with available arrays in terms of total size. 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 freed after the last call to it, using `pool_init()` and `pool_destroy` respectively, operating on a memory pool.

This strategy not only prevents frequent and redundant `malloc` calls across the invocation of a multigrid cycle, but also enables reuse within a cycle, that could have been missed by the inter-group reuse pass. We insert calls to `pool_deallocate` during code generation, right after all the uses of an array's user is done. This ensures that free arrays are returned to the pool as early as possible and in time before any new request is made.

3.4 Auto-tuning

We use PolyMage's Autotuner to search for the best configuration over a reasonably small space for both 2D and 3D applications. 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 auto-merging phase, are used.

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            a[i][j][k] = b[i][j][k] * c[i][j][k];
        }
    }
}
```

Figure 3.5: Example code with OpenMP's collapse directive usage

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.

In our experiments 4, for comparison with Pluto, we tuned over 25 tile configurations for both 2D and 3D apps. We chose a smaller search space for Pluto, since tuning is semi-automatic in this case, given that we had to manually modify the source file generated in each case. This was because, Pluto's front-end accepts only multi-dimensional array accesses, while the hand-optimized code had flattened heap-allocated arrays.

3.5 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 PolyMage's output code. Also, the loop-nest for an overlapped tile is typically parallel in all dimensions. However, this might not be true if boundary conditions are set in the PolyMage specification – tile loop-nest generated as an *isl* AST node with `ifs` and other `for` loops. We separate out perfect parallel loop-nests from the tree, and add the OpenMP clause '`parallel for collapse(`*d*`)`' above the loop headers, where *d* is the depth of perfect loop-nest. Iteration space of parallel loops annotated with this clause, are flattened by OpenMP runtime so that iteration chunking for threads can be done at a finer level, and this can potentially improve the load-balance among threads. An example loop nest showing the usage of OpenMP `collapse` is given in Figure 3.5.

However, in order to be able to do this, the scratchpad buffer allocation for the tile has to be moved to the innermost loop of the nest so that they are not shared between threads. We added another module in PolyMage code generator that scans the polyhedral AST generated by *isl*, for such perfectly parallel loop-nests and decide the both the depths at which collapse clause and the scratchpad allocation code must be added.

Figure 3.6 shows a snippet of code generated by PolyMage after all optimizations. We implemented all the new optimizations and language features discussed in this Chapter in the publicly available PolyMage repository [32] available as open-source software under Apache License Version 2.0.

```
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 3.6: Optimized code generated by PolyMage

Chapter 4

Experimental Evaluation

In this chapter, we describe the experimental setup, the multigrid benchmarks we used for evaluation, discuss the results with a detailed analysis and summarize the outcomes.

4.1 Experimental Setup

All experiments were run on a dual socket NUMA multicore system with Intel Xeon v3 processors (based on the Haswell microarchitecture). Table 4.1 provides the details of the hardware and software. Hyper-threading was not used during the experimentation. OpenMP thread affinity was set to scatter to even balance threads across cores of different processors. The minimum execution time from five runs has been reported in all cases.

4.2 Benchmarks

Experimental evaluation is performed on Multigrid benchmarks that solve the Poisson's equation [5], which is given by

$$\nabla^2 u = f, \tag{4.1}$$

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

Table 4.1: Architecture details

	2-socket Intel Xeon E5-2690 v3
Clock	2.60 GHz
Cores / socket	12
Total cores	24
Hyperthreading	unused
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
Linux kernel	3.10.0 (64-bit) (Cent OS 7.1)

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 benchmarks are based on code used for evaluation by Ghysels and Vanroose [11] and made available at [12]. Our techniques are equally applicable to a finite volume discretization, which was used for benchmarks in past work [38, 4].

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). Similarly, 10-0-0 represents the use of only pre-smoothing steps, for 10 iterations. The 4-4-4 configuration was chosen to observe the behaviour of the optimized code in a uniform smoothing steps setting. 10-0-0 is a configuration on the other extreme, which is used to test the effectiveness of fusion when there are no smoothing steps between two-levels (i.e., between one interpolate and the next). Any other variant would more or less relate to one of these configurations, since neither other asymmetric smoothing-steps combination nor the coarse-smoothing steps are very different enough to stress-test the compiler.

In all cases, Jacobi smoothing steps and four discretization levels are used. Though, any number of levels can be used in PolyMage to generate optimized code, a higher number would not be of much significance for performance analysis, since the finer grid computations constitute the bulk of the execution time of a cycle. Although other smoothing techniques like successive over-relaxation and Gauss-Seidel Red Black (GSRB) exist, we focus only on Jacobi smoothing for this paper. GSRB for example presents additional considerations for vectorization [38], 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. It is also worth mentioning that the NAS-MG code incorporates a hand-optimized partial stencil accumulation technique, using a line buffer in the innermost loop of all stencil computations. This reduces the arithmetic intensity of the computations as the stencil kernel is evaluated as an accumulation of partially computed stencils. The hand-optimized code does not incorporate any tiling technique, and includes buffer reuse across multigrid cycle invocations, by using the same allocation pool for each cycle. The benchmark implementation is available in Fortran language, and uses static buffer allocations, resulting in the usage of stack space of the program – in contrast to this, our implementations use heap space for full-array allocations. Because of this, to accommodate larger arrays of higher benchmark classes (Table 4.2), the programs were compiled with an extra flag `-mcmodel=large` using `ifort`.

Problem sizes used for all benchmarks are listed in Table 4.2. We define problem size classes W, A, B, and C for all benchmarks we evaluate in the same way they exist for the NAS Multigrid benchmark.

Benchmark	Grid size, cycle #iters)			
	Class W	Class A	Class B	Class C
2D	2048^2 (50)	4096^2 (50)	8192^2 (10)	16384^2 (10)
3D	-	128^3 (25)	256^3 (25)	512^3 (10)
NAS-MG	128^3 (4)	256^3 (4)	256^3 (20)	512^3 (20)

Table 4.2: Problem size configurations: the same problem sizes were used for V-cycle and W-cycle and for 4-4-4 and 10-0-0

Table 4.3: Benchmark characteristics: lines of code and baseline execution times

Benchmark	Stages	Lines of code			Lines of generated code		Execution time of PolyMage-Naive (s)			
	(# DAG nodes)	PolyMage (DSL)	HandOpt (C)	HandOpt+Pluto (C)	PolyMage-Opt (C/C++)	PolyMage-Opt+ (C/C++)	class B		class C	
							1	24	1	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

4.3 Comparison

PolyMage-Opt+ refers to PolyMage generated code with all optimizations and considerations that are the contributions to the compiler in this work (Chapter 3). For the first four benchmarks, we compare with (a) manually optimized versions of the benchmarks that we obtained from Ghysels and Vanroose [11], referred to as *HandOpt*, (b) *HandOpt* further optimized by time tiling the smoothing steps with PlutoâĂŹs diamond tiling (version 0.11.4-229-gceac3ae)) approach [2, 31], which we refer to as *HandOpt+Pluto* (also provided by Ghysels and Vanroose [11], and (c) with a version that does not have the storage optimizations and other improvements we described in Chapter 3 (*PolyMage-Opt*). In addition, to also evaluate diamond tiling for the TStencil construct in PolyMage, we integrated libPluto into PolyMage (Figure 4); we use *PolyMage-Dtile-Opt+* to refer to *PolyMage-Opt+* with the choice to applying diamond tiling instead of overlapped tiling to pre-, coarse- and post-smoothing steps.

The manual optimizations in the *H*andOpt versions include explicit loop parallelization (using OpenMP pragmas), storage optimization within multigrid levels – i.e., to use two modulo buffers (precisely sufficient) instead of one buffer for each smoothing step, pooled memory allocation, etc. For *HandOpt+Pluto*, tile sizes were tuned empirically around optimized ones that shipped with its release. *PolyMage-Naive* corresponds to a simple parallel code generated using PolyMage, with loops generated in a straightforward manner with no tiling, fusion, or storage optimization, but OpenMP pragmas on the outermost among parallel loops for each loop nest (loop iterating the outermost among space/grid dimensions). This version is ideally what a programmer with a very basic proficiency in performance extraction would write, and is used as a baseline for comparison against the rest. Note that *PolyMage-Naive* is compiled with the best optimization flags of the vendor compiler, just like the other codes compared with.

The four benchmarks capture a wide range of complexity. The W-cycle is a large and complex pipeline, with nearly 100 stages for smoother and level settings mentioned in Table 4.2 (as seen in the DAG of Figure 2.3c). Domain experts consider the W-cycle as a heavyweight method due to its computational cost and number of steps involved in its single cycle [37]. Another inherent property of W-cycles, that restricts the amount of achievable speedup, is that the live-out arrays computed at the beginning of the cycle have to be live till the end for a relatively longer amount of time compared to V-cycles. The miniGMG [27] and HPGMG [21] benchmarks use F-cycle, which is in between V- and W- cycles in complexity. Figure 3.3 shows the grouping and storage mapping for the best performing code for 2D V-cycle 4-4-4 class W benchmark. In this case, the code contains ten groups all of which, except one, were overlapped tiled (exception is the single defect node in a group, which do not require tiling for temporal locality). 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 proves that fusing computations across levels is indeed beneficial for performance.

Table 4.3 provides an indication on the programmer effort involved with developing the various versions, size of the benchmark pipeline graphs and the lines of optimized code generated by PolyMage. The LOC count for PolyMage generated codes is relatively high because of the complex transformations applied on the original schedule, resulting in the extra tile loop nests and the conditional codes to handle boundary conditions in the presence of these loops.

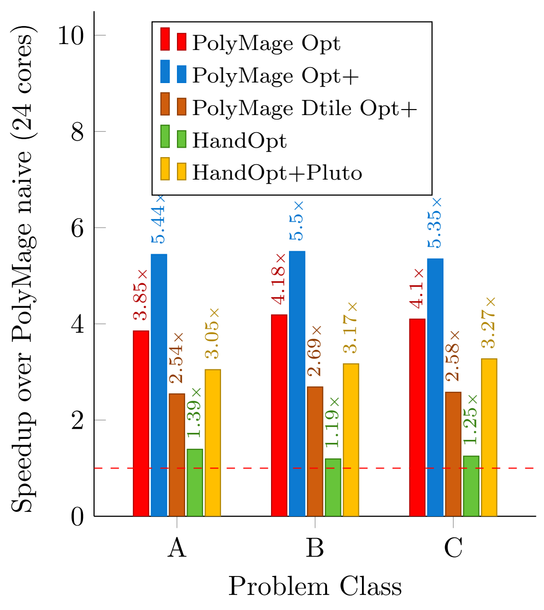
Benchmark	Absolute execution time (seconds)			
	Class W	Class A	Class B	Class C
V-2D-4-4-4	1.396	5.791	7.474	28.6
V-2D-10-0-0	1.875	9.748	14.434	35.49
W-2D-4-4-4	1.235	4.872	10.207	36.001
W-2D-10-0-0	1.245	5.962	15.19	48.357
V-3D-4-4-4	-	0.381	3.699	11.912
V-3D-10-0-0	-	0.513	4.374	18.891
W-3D-4-4-4	-	0.525	4.586	15.664
W-3D-10-0-0	-	0.543	5.597	23.385
NAS-MG	0.042	0.306	1.557	12.634

Table 4.4: Absolute performance of *PolyMage-naive*, baseline used in Figures 4.1, 4.2, 4.3, 4.4

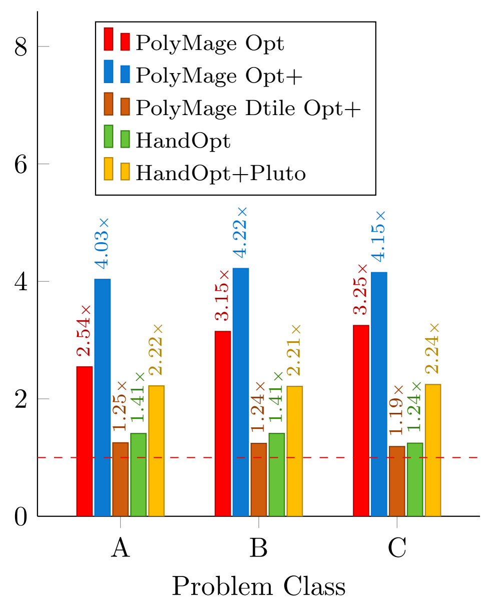
4.4 Analysis

Figures 4.1, 4.2, 4.3 show the 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 and tiling. The improvement of *PolyMage-Opt* over *HandOpt+Pluto*, is due to fusion as well as use of local buffers (scratchpads). The difference between *PolyMage-Opt* and *PolyMage-Opt+* isolates the improvement due to reuse of scratchpads within a group, across groups, more efficient allocation, and multidimensional parallelism.

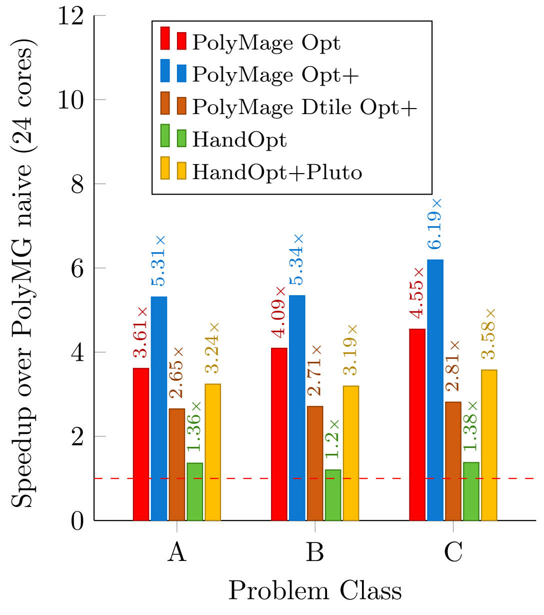
From the experimental results, we observe a general trend of *PolyMage-Opt+* performing better for higher classes than for lower classes, compared to *HandOpt+Pluto* in all benchmarks. One of the main reasons for this, is the communication avoiding nature and the shape of tiles in overlapped tiling, which appear to achieve good parallel performance due to fewer synchronizations. Though PolyMage codes make use of scratchpad buffers for intermediate functions, this is still an additional storage requirement which does not



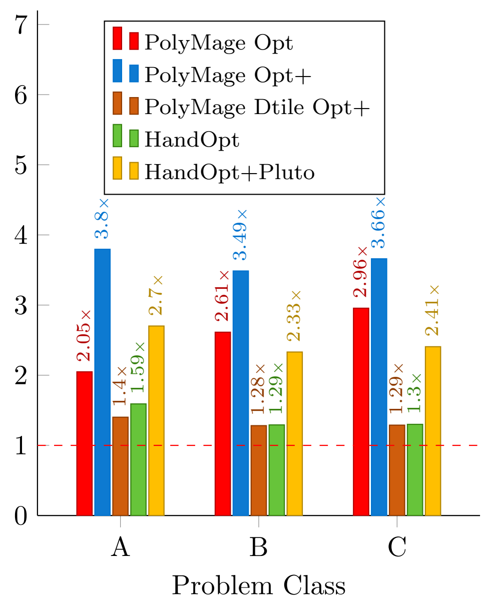
(a) 2D-V-10-0-0



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

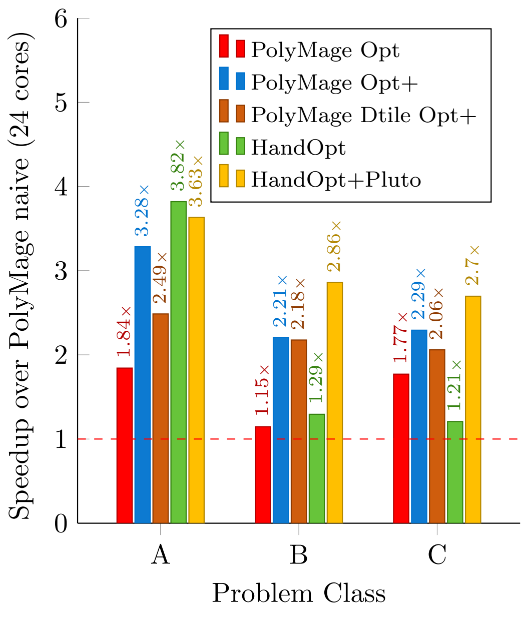


(c) 2D-W-10-0-0

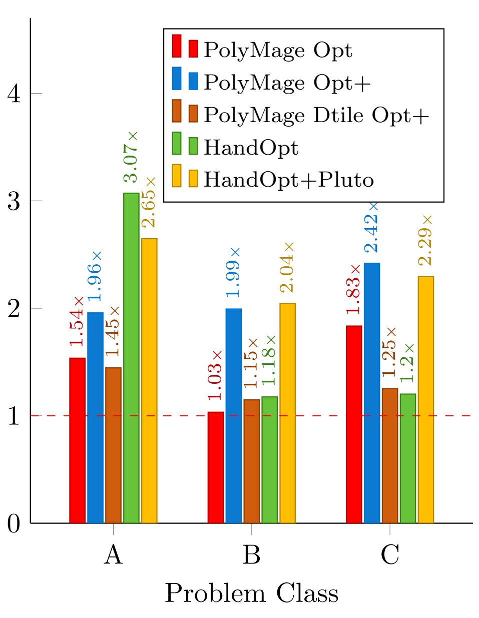


(d) 2D-W-4-4-4

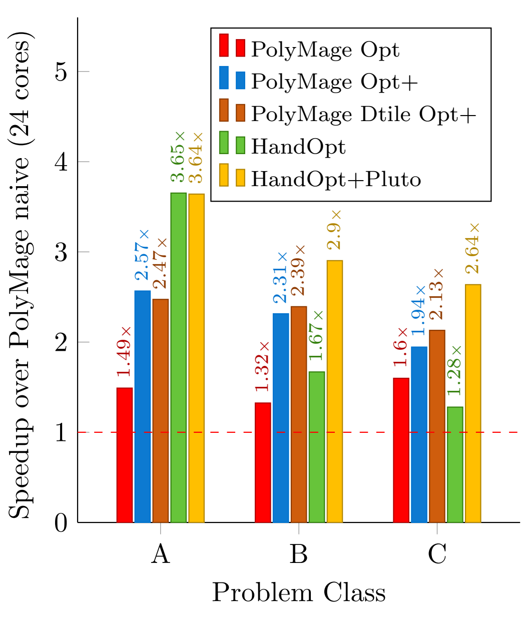
Figure 4.1: Performance: speedups for 2D benchmarks — absolute execution times can be determined using Table 4.4



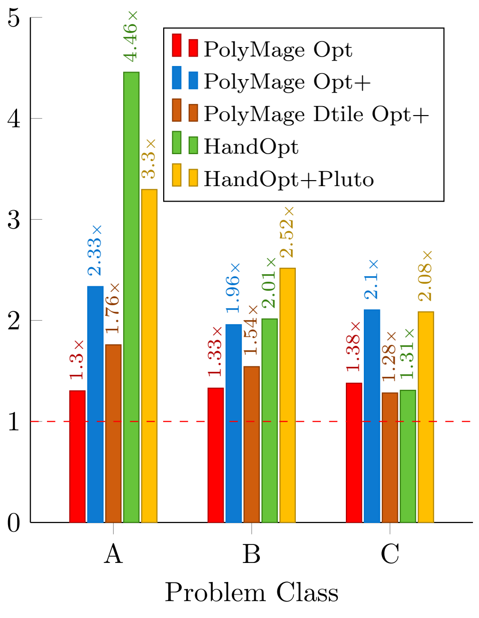
(a) 3D-V-10-0-0



(b) 3D-V-4-4-4



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



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

Figure 4.2: Performance: speedups for 3D benchmarks — absolute execution times can be determined using Table 4.4

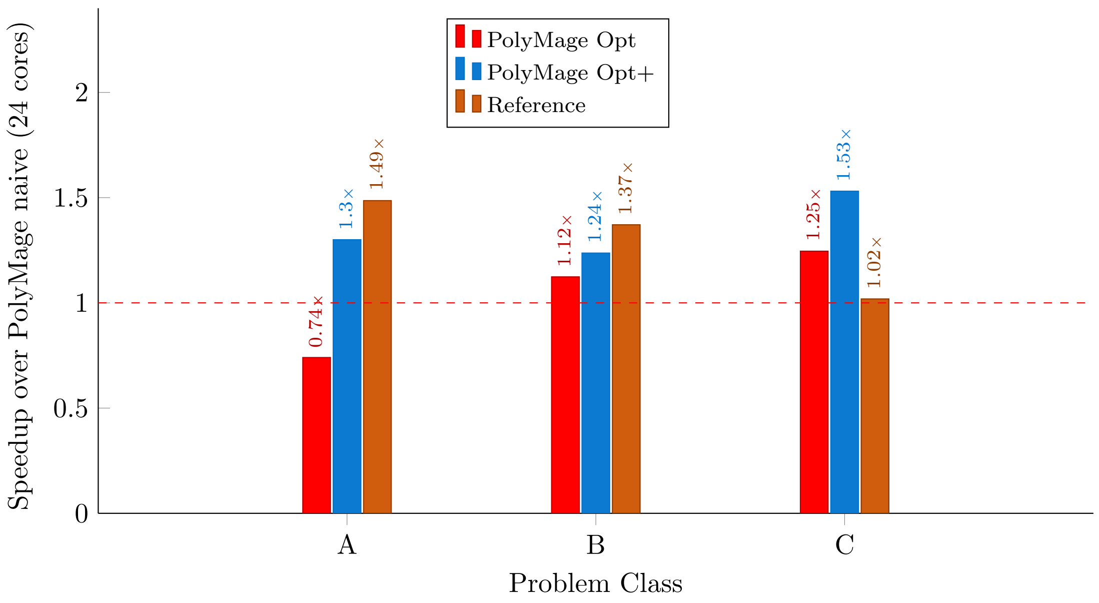
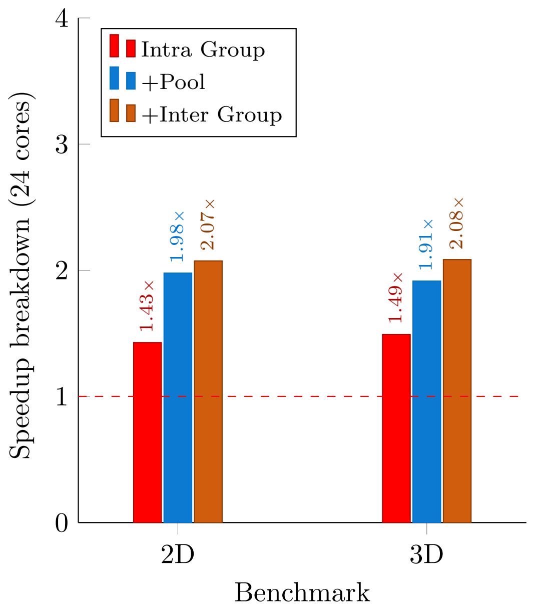
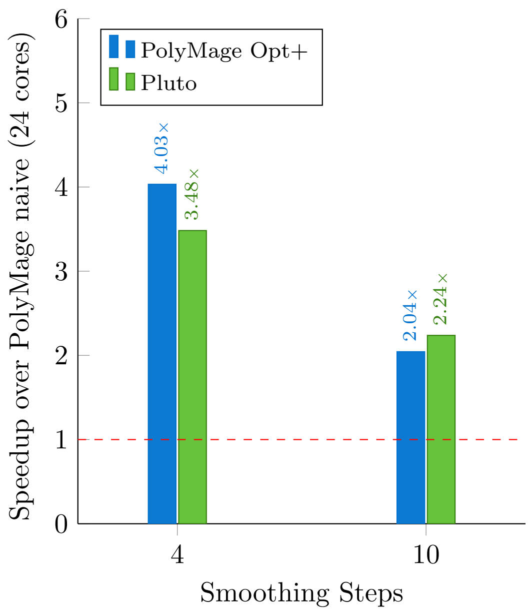


Figure 4.3: Performance: speedups and scaling for NAS-MG benchmark — absolute execution times can be determined using Table 4.4



(a) Inter Group Storage Opt



(b) Smoothing Steps

Figure 4.4: 4.4a 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 class W benchmarks, over PolyMage naive. 4.4b shows performance of PolyMage Opt+ and Pluto for just the Jacobi smoothing steps used in over benchmarks for 3D class C benchmark

exist for diamond tiled codes, since all operations take place between just two arrays in a ping-pong fashion alternating in time.

The reason for a larger performance gap between *PolyMage-Opt+* and *HandOpt+Pluto* in 3D benchmarks is that, the amount of overlap in the tiles is much higher than that in 2D benchmarks, due to the additional dimension. In other words, there is lot more redundant work for overlapped tile codes, which is a disadvantage that cannot be compensated by the locality benefits achieved. Because of the larger volume of scratchpads needed in 3D cases, the optimal codes had very small tile sizes for the outermost dimensions, which limits the group size to smaller values as redundancy cannot be tolerated. As the extents of the tile becomes smaller, the fraction of the tile volume that is subjected to redundant computation increases. In some cases, the optimal configurations had smaller tile sizes to fit scratchpads in cache in spite of a double overlap (a tile overlapping with the neighbour of its neighbour) resulted by fusing many functions.

Over the reference NAS MG implementation included for NAS parallel benchmarks (NPB), PolyMage-Opt+ obtains an improvement of 32% for the Class C problem size. Overall, the NAS implementation is better than *PolyMage-Opt+* by a mean of 17%. As mentioned earlier, the NAS MG implementation in NPB optimizes the core loop body computation by computing a partial sum and reusing it multiple times through a line buffer in the innermost loop, which turns out to be better for all classes, except C, even without tiling.

When compared to the manually optimized versions, we observe that the class A W-cycle 3-D cases are the only ones where *PolyMage-Opt+* does not obtain an improvement. As mentioned in Section 4.2, the W-cycle is the most complex of the evaluated benchmarks and serves as a stress test for our system. Code generated in this case is also long (Table 4.3). We believe that the lack of speedup is due to a combination of code quality, scratchpad usage, which can be further improved, and a greater fraction of redundant computation in 3-D — although the latter was tuned over a space of tile sizes and group size thresholds. Using a combination of polyhedral and syntactic code generation techniques is likely to improve code quality and should be explored in a DSL setting.

Performance comparison of overlapped and diamond tiling for a setting involving just the smoothing steps, with one level (finest grid) of 512^3 problem size, tuned on 24 cores, is shown in 4.4b. While for fewer smoothing steps (four in number), *PolyMage-opt+* performs slightly better than *HandOpt+Pluto*, the trend is quite the opposite for larger smoothing steps. This due to a good exploitation of temporal locality by fusing all 4 steps in one group for the former case. With ten smoothing steps, the amount of redundant computation in overlapped tiling suppresses the gains achieved from locality. In this case, the best performing code does not have all the steps fused, to avoid high amount of overlap.

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 PolyMage-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 4.4b.

We also observe that PolyMage-Dtile-Opt+ outperforms PolyMage-Opt+ in only one scenario 3D-W-10-0-0, and its performance is always lower than HandOpt+Pluto. In fact, PolyMage-Opt+ outperforms PolyMage-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 4.4b). However, this does not still explain the large gap between PolyMage-Dtile-Opt+ and PolyMage-Opt+. Due to an implementation issue arising from conservative assumptions in reusing input/output arrays, we note that PolyMage-Dtile-Opt+ performs more memory copies; such copies are not present in HandOpt+Pluto or PolyMage-Opt+. We confirmed that this reduces PolyMage-Dtile-Opt+‘s performance by up to 60% in 3D cases. 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 PolyMage-Dtile-Opt+ over PolyMage-Opt+ for 3D grids when the number of pre-smoothing steps is high (order of ten or more).

Scaling Table 4.3 can be used in conjunction with Figures 4.1 and 4.2 to determine how the optimized code scales with core count. For example, for W-2D-10-0-0 class C, a naive parallelization (PolyMage-Naive) yields a speedup of only 5.38× on 24 cores when compared to running on a single core, while PolyMage-Opt+ with all its optimizations for locality and parallelism provides a final speedup of 33.3× on 24 cores over the sequential PolyMage-Naive. Similarly, for V-3D-4-4-4 class C, PolyMage-Opt+ delivers a speedup of 10.8× when run in parallel on 24 cores over sequential PolyMage-Naive, while the corresponding speedup for PolyMage-Naive on 24 cores is only 4.47×.

Figure 4.4a shows the speedup breakdown for our storage optimizations, on 24 cores, over an implementation obtained by disabling all of them. 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). The arguments made in 3.3.1 and 3.3.2 on benefits offered by storage optimizations hold true, as observed from this experiment. It can be clearly seen that pooled memory allocation takes care of inter-group reuse opportunities, even when the latter is not enabled. Generating code with inter-group reuse further pushes up the performance, that was not captured by pooled allocation alone. On the Vcycle benchmark with configuration class C, 10-0-0, we disabled the multidimensional parallelism in the PolyMage-Opt+ version, to analyse the contribution of OpenMP collapse clause to the overall performance. The performance gains obtained with multidimensional parallelism, over the version without it, for 2D and 3D cases were 0.4% and 5.6% respectively. This clearly demonstrates that extracting parallelism in all dimensions is useful when there are fewer tasks, which is exactly the case with 3D benchmarks.

4.5 Summary

PolyMage-Opt+ achieves a mean (geometric) improvement of 3.2× over *PolyMage-Naive* across all 2D and 3D benchmarks (4.73× for 2D and 2.18× for 3D). The mean improvement over *PolyMage-Opt* is 1.31×. We even achieve an improvement of 1.23× over *HandOpt+Pluto* on average (1.67× 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.

Chapter 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 [24, 38, 10, 4, 3, 11]. Among these, [38, 10, 11] considered and evaluated tiling across multiple smoothing steps; [38, 4, 3] considered fusion of operators and optimizing the fused stencils together. All of these approaches were either manual or semi-automatic in the application of their optimizations. We discuss and compare our work with these in more detail below. Past works related to stencil loop optimizations, tiling strategies and their characteristics were discussed in detail in Chapter 3.

Williams et al. [38] 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 that 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 across compute nodes 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 hence use a wavefronting method [40] 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 this 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 PolyMage encompasses residual-restriction fusion. In summary, in contrast to the approaches of [24, 38] (a) our approach has been towards building language and automatic compiler optimization support via a DSL tool, and (b) the optimization approach that we consider takes a holistic view of all steps involved in geometric multigrid.

Ghysels and Vanroose [11] 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 [11]. 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 Ghysels and Vanroose[11] 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 (Chapter 4) included a comparison with code obtained from Ghysels and Vanroose[11], and the difference in performance was discussed therein. The performance modeling study made in Ghysels and Vanroose[11] linking arithmetic intensity to convergence rate is orthogonal to our experimentation, since we chose and restricted ourselves to two particular configurations for the smoothing steps.

Basu et al. [4, 3] 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 [7]), 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. [38] 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 approach of [11], ours does not involve pipelined startup.

As briefly mentioned in Chapter 1, image processing pipelines have similar properties as geometric multigrid algorithms. Hence, optimizations techniques studied for DSLs such as Halide [35] and PolyMage [28] are also applicable here. Halide's transformation of the input specification is semi-automatic, since the programmer specifies the schedule that determines how the algorithmic specification is to be computed. Significant effort and expertise on performance optimization and tuning is needed to determine good schedules. On the other hand, PolyMage is completely automatic in its scheduling; a meaningful comparison with semi-automatic approaches is thus infeasible unless pre-tuned schedules exist. Unlike for image processing pipelines, both 2-d and 3-d grids are important for multigrid, while image processing filters or stages typically process 2-dimensional data. The cases where three or higher-dimensional structures are used in image processing are typically those that involve reductions or histograms. Three dimensional grids have different implications on the choice of tiling strategy and tile size selection.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we demonstrated how high performance can be achieved along with enhanced programmability for GMG, through a domain-specific optimization system. We compared our approach with the existing PolyMage optimizer, and codes that were hand-optimized in conjunction with Pluto, on benchmarks varying in multigrid cycle structure, smoothing steps, and other aspects. Our automatically optimized codes obtained an improvement of 51% (geometric mean of all benchmarks) in execution time over existing PolyMage optimizer.

It was observed that the diamond tiling technique achieves parallel performance superior to that of overlapped tiling in higher dimensional computations. However, for large problem sizes in all benchmarks, the improved PolyMage optimizer performs better or nearly as good as hand-optimized programs, which make use of Pluto's diamond tiled code. Our optimizations-enabled PolyMage compiler achieves a speedup of up to 32% over the NAS-MG benchmark from the NAS-PB suite v3.2 [29]. From our experiments, it is clear that, for 3-D multigrid applications, the overlapped tiling technique suffers from high amount of redundant work, while it still appears to be a better choice for 2-D applications on larger problem sizes.

Another important conclusion that can be drawn from the experiments, is on the importance of storage optimizations in improving the execution time of a program. Though the motivating factor to enable these optimizations was to achieve minimality in the DRAM usage and cache footprints, they turned out to be some among crucial factors in extracting a higher performance. While intra-group storage reuse aids the programs to fit as much data as possible in lower level caches and reduce off-chip memory accesses, inter-group reuse minimizes the resident memory in DRAM, TLB misses and memory bandwidth contention. Such improvement helped in removing the latencies involved in data fetching and memory management, thus improving program performance.

6.2 Future Work

Some of the issues that can be addressed for future enhancements are listed below.

- Integration of diamond tiling technique in conjunction with the existing overlapped tiling in PolyMage optimizer. Such a combination can potentially yield a higher performance by capturing the best combination of both techniques.
- Deeper analysis is essential to make compile-time decisions on the scheduling strategy to be used. Individual behaviours of each strategy differ due to their sensitivity to many aspects like problem size and number of grid dimensions
- The current storage optimization algorithm contributed to the existing PolyMage compiler depends on the schedules of the functions in the pipeline graph. Currently, PolyMage's pipeline schedule uses level order, and storage reuse opportunities are not considered during scheduling. Finding schedules that result in optimal reuse is an interesting problem for future work.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International conference on Parallel Architectures and Compilation Techniques*, pages 303–316, 2014.

[2] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *International conference for High Performance Computing, Networking, Storage, and Analysis*, pages 40:1–40:11, 2012.

[3] Protonu Basu, Mary W. Hall, Samuel Williams, Brian van Straalen, Leonid Oliker, and Phillip Colella. Compiler-directed transformation for higher-order stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 313–323, 2015.

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

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

[6] cgen: C/C++ source generation from an AST. https://pypi.python.org/pypi/cgen.

[7] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, 2008.

[8] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 65–77, New York, NY, USA, 2001. ACM.

[9] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *International conference for High Performance Computing, Networking, Storage, and Analysis*, pages 9:1–9:12, 2011.

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

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

[12] Pieter Ghysels, Geometric Multigrid Tiled, 2015. `https://bitbucket.org/pghysels/geometric_multigrid_tiled`.

[13] T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. In *1st Int. Workshop on High-Performance Stencil Computations (HiStencils 2014)*, 2014.

[14] Tobias Grosser, Albert Cohen, Justin Holewinski, P Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *International symposium on Code Generation and Optimization*, page 66, 2014.

[15] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: automatic parallelization using trapezoidal tiles.

In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 24–31, 2013.

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

[17] Visualization of heat transfer in a pump casing, created by solving the heat equation, Wikimedia Commons CC BY-SA 3.0. https://en.wikipedia.org/wiki/Differential_equation#/media/File:Elmer-pump-heatequation.png.

[18] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *International conference on Supercomputing*, pages 13–24, 2013.

[19] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *International conference on Supercomputing*, pages 311–320, 2012.

[20] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362, 2012.

[21] High Performance Geometric Multigrid Methods. https://hpgmg.org/.

[22] Integer Set Library: A library for manipulating sets and relations of integer points bounded by linear constraints. http://isl.gforge.inria.fr/.

[23] islpy: Wrapper around ISL, an integer set library. https://pypi.python.org/pypi/islpy.

[24] Markus Kowarschik, Ulrich Rüde, Christian Weiß, and Wolfgang Karl. Cache-aware multigrid methods for solving poisson's equation in two dimensions. *Computing*, 64(4):381–399, 2000.

[25] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, 2007.

[26] A Brief Introduction to Krylov Space Methods for Solving Linear Systems. http://www.sam.math.ethz.ch/ mhg/pub/biksm.pdf.

[27] miniGMG. https://crd.lbl.gov/departments/computer-science/PAR/research/previous-projects/miniGMG/.

[28] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[29] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[30] numpy: python package for scientific computing with Python. http://www.numpy.org/.

[31] PLUTO: An automatic parallelizer and locality optimizer for affine loop nests. http://pluto-compiler.sourceforge.net.

[32] PolyMage open-source repository. https://bitbucket.org/udayb/polymage/.

[33] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[34] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, 2012.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 519–530, 2013.

[36] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM.

[37] Multigrid Methods, 2013. `https://www.wias-berlin.de/people/john/LEHRE/MULTIGRID/multigrid.pdf`.

[38] 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. Optimization of geometric multigrid for emerging multi- and many-core processors. In *SC Conference on High Performance Computing Networking, Storage and Analysis*, 2012.

[39] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium*, pages 171 –180, 2000.

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

[41] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.