Optimizing Dense Matrix Computations with PolyMage

by

Kumudha KN

Department of Computer Science and Automation

Indian Institute of Science

BENGALURU – 560 012

January 2019

DEDICATED TO

My Parents

Acknowledgements

It is my extreme pleasure to have had this opportunity to be a research student at the Indian Institute of Science. Firstly, I would like to extend my gratitude to my advisor Dr. Uday Kumar Reddy B., for his invaluable guidance and support throughout my Masters degree. He has always helped me improve my research and engineering aptitude and help me step in the right direction during times of falter. My collaborations with Vinay Vasista, Aravind Acharya, Sumedh Arani, Abhinav Baid, and Kingshuk were a lot of fun and gave me plenty of insights into several topics. I would like to thank them for their patience and support. My special thanks to Vinay Vasista and Aravind Acharya for patiently helping me refine my understanding of intricate aspects in high performance computing and polyhedral model. My thanks also go out to my wonderful friends at IISc - Shilpa, Sridhar, Srinidhi, Pallavi, Arvind, Varun, Hari, Chaitra, Nishanth and several others who are not mentioned here. I have had some of the most fun days at the mess table.

I would like to thank Prof. R Govindarajan for the computer architecture course. The course taught not only the subject but also the basic art of reading and reviewing research works. I thank him for his lectures and amazing insights.

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

I would like to thank Adarsh Patil for always being there to help with technical or emotional stress. Also for giving great insight in writing the paper and for proofreading it multiple times. I also thank my family and friends for always being patient and supportive

in balancing my personal and academic life. I especially thank my parents for their guidance and love and my brother for never saying no to drop me when I had to reach IISc. I would also like to thank Adarsh's parents for being very supportive throughout this whole process.

Finally, I would like to thank the almighty for giving me the opportunity, energy and the honour of working at IISc.

Thanks a lot.

Publications based on this thesis

1. *Optimizing Dense Matrix Computations with PolyMage*. Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, Uday Bondhugula. [Under Preparation for submission in 2019]

Abstract

Linear algebra computations and other arbitrary affine accesses are ubiquitous in applications from domains like scientific computing, digital signal processing (DSP), and deep neural networks. Libraries such as OpenBLAS, MKL, and FFTW provide efficient hand-optimized implementations for matrix and vector primitives used in these domains for various architectures. Applications are then built upon these standard library routines to obtain high performance. These libraries do not perform well for all matrix sizes and obtain sub-optimal performance for small matrices. The interface of these libraries can also be fairly complex requiring several input parameters. Thus, an even higher level of abstraction is often desired to improve productivity. Further, by using these libraries the opportunity to optimize across different library calls is lost. Traditional programming to exploit this locality using library functions becomes complex.

The work in this thesis proposes (i) a tile size selection model which works for any arbitrary affine access and eschews auto-tuning, (ii) a simple heuristic to determine the profitability of library call mapping and falling back to generated code otherwise, (iii) an intra-tile optimization technique to expose inner-loop parallelism thus enabling general purpose compiler's vectorizer to generate vector instructions, (iv) a DSL approach with high level primitives and functions to allow expressing computations efficiently.

The optimizations proposed are implemented in the PolyMage DSL. PolyMage is a domain specific language (DSL) originally designed for image processing pipelines. Its optimizer is able to perform optimizations like fusion, tiling, and loop optimizations for image processing pipelines. Firstly, PolyMage's language specification is extended to support additional functions and primitives for matrix computations and perform domain inference

automatically. Secondly, PolyMage compiler's fusion and tile size selection heuristics are extended to work with arbitrary affine accesses. It is thus able to optimize computations from additional domains including dense linear algebra and certain DSP computations. The heuristic to map matrix operations to corresponding optimized library implementations are incorporated with the help of an idiom recognition algorithm.

The thesis finally experimentally evaluates these optimizations on modern multicore systems using representative benchmarks from PolyBench, digital signal processing, and image processing. The results are compared to state-of-the-art optimization approaches and frameworks in each domain. Experiments on DSP benchmarks show that our optimizations give a mean speed up of 7.7× over the existing PolyMage optimizer, 5.1× over the Python numpy package and 1.9× over MATLAB toolboxes with parallel computing support. Linear algebra computations from the PolyBench benchmark suite obtain an average a speedup of 21.7× over the existing PolyMage optimizer, 3.6× over Pluto and 4.1× over PPCG.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Several domains such as computer vision, image processing, machine learning, computational fluid dynamics, and digital signal processing are using computational capability to solve real-world problems. These domains are fast evolving and solve complex problems with increasingly large datasets, complex simulations, and tools for deeper analysis. A significant portion of the execution time of these modern applications is comprised of matrix computations and other arbitrary affine accesses. These computations often form the building blocks in these applications. Thus, providing high performant implementations of these computations allows us to accelerate and improve the performance of the applications. Further, with the introduction of increasingly complex hardware architectural features such as parallel processors, vector processors, hierarchical memory and systolic arrays, taking advantage of these modern systems is non-trivial.

Hence, optimization of applications is now a challenge for compiler and algorithm experts. These rapid technology evolution burdens the domain scientists to extract best performance for their applications. Providing an abstraction to domain scientists to express these intricate computations while hiding the complexity of extracting the best performance from the hardware is valuable. The backend of this compiler must provide performant code irrespective of the problem sizes, characteristics, and architectural features.

Section 1.1 introduces the computations that are considered for optimization in this work. Section 1.2 provides a primer of the optimizations proposed in this thesis to improve

the performance of these computations and Section 1.3 summarizes the contributions of this work.

1.1 Computation Primitives

This section introduces and describes the set of computation primitives and their applicability that are targeted to be optimized in this thesis.

1.1.1 Matrix Computations

Common matrix computations are synonymous with basic linear algebra computations. They include matrix-matrix multiplications/additions/subtractions, matrix-vector multiplications, matrix transpose and point-wise matrix multiplications where each index of the array in a loop nest is a function of only one dimension (or loop iterator). For example, array access of the form $mem[i], mem[i][j], mem[i+1][j+1]$. The computational complexity and architectural efficiency of these operations depends on matrix sizes, type of the operation and also the sequence in which these computations are performed. Optimizing the performance of these matrix computations remains an important problem. Matrix computations are already part of several computational science and engineering workloads such as computational biology and computational fluid dynamics. Another major example of these matrix operations can be seen in neural network computations. For example, a convolution neural network (CNN) can be reduced to perform simple $2D$ matrix-matrix multiplications, followed by addition of bias and then a normalization operation. Similarly, recurrent neural networks(RNN) perform many matrix vector multiplications.

1.1.2 Arbitrary Affine Accesses

Another kind of computation on two dimensional matrices which is very prevalent in digital signal processing is an operation where the array index in a loop nest is an affine function of two or more dimensions (or loop iterators). Some example of these memory access are

$mem[i+j+1], mem[i+j][j]$. These computations can be seen in low pass filters in digital signal processing. Exploiting locality in these kinds of computation is critical for better performance.

1.2 The PolyMage DSL Solution for High Performance and Productivity

The work in this thesis enhances PolyMage DSL to be able to support efficient authoring of the computational primitives and generate high-performance code for them. All the optimizations proposed are implemented as a part of the compiler of the PolyMage DSL. PolyMage [24] is a domain-specific language (DSL) that was originally designed for image processing pipelines. Its optimizer is able to perform optimizations such as fusion, tiling, and other loop optimizations for such computations.

PolyMage is enhanced to generate high-performance code for computations involving matrix computations and arbitrary affine accesses. Specifically, new constructs are added to support linear algebra operations thus allowing applications that leverage such computations to be expressed in our DSL. Further, to deliver high performance with this newly added support, a generic fusion and tile size selection heuristic is proposed for PolyMage to work for matrix computations. Its compiler is thus able to robustly optimize computations from additional domains including dense linear algebra and certain DSP computations over and above Image Processing Pipelines.

Simple heuristics are used to support mapping of matrix operations to corresponding BLAS library implementations when such a mapping is determined to be profitable. Thus, PolyMage is now able to recognize and substitute appropriate code fragments with calls to tuned libraries. If the mapping is deemed not profitable compared to the optimized code generated by PolyMage, the library mapping is eschewed. This allows getting the best of both worlds. Finally, an optimization to perform loop transformations at the outer level to work in conjunction with library calls to obtain complementary benefits from libraries. In effect, this allows to tile for locality at the outer loop level and exploit data reuse across

library calls.

1.3   Summary of the Contribution

To summarize, the contributions of this thesis include:

- Support for high-performance code generation for matrix computations and arbitrary affine accesses in PolyMage for improved programmability and productivity.

- A generic and robust model for tile size determination for PolyMage which eschews auto-tuning for best-performing tile size.

- An amiable heuristic for performance inflection point based on profiling BLAS functions i.e., the point after which BLAS call becomes profitable over PolyMage generated code. Using this heuristic in PolyMage along with idiom recognition allows to map linear algebra functions to pre-optimized BLAS library calls when the performance of such BLAS calls is determined to be superior.

- An intra-tile optimization algorithm to determine dimensions that allow for vectorization friendly loops to be selected as innermost dimensions.

- Extension of PolyMage to support optimizations like fusion and tiling for reduction operations which further enable high-performance code generation by improving data reuse in caches.

- Experimental evaluation of this approach and comparison with the state-of-the-art frameworks. The optimized PolyMage shows a mean speedup of 7.7× over the baseline naive PolyMage, 5.1× over numpy and 1.9× over MATLAB toolboxes with parallel computing support for DSP benchmarks. The optimized PolyMage also obtains a speedup of 3.6× over Pluto and 4.1× over PPCG on relevant PolyBench benchmarks.

1.4   Thesis Organization

The rest of the thesis is organized as follows: Section 2 motivates the work of this thesis and discusses limitations in state-of-the-art works. Section 3 provides the necessary background of PolyMage and its fusion and tile size selection model. The details of the optimizations proposed and the new compiler flow comprising of the proposed optimizations is described in Section 4. Section 5 presents the experimental setup, methodology, and benchmarks used for empirically evaluating the optimizations discussed. This is followed by the experimental results and quantitative comparison with the state-of-the-art works in the domain. Section 6 discusses the related work in this area and Section 7 concludes the thesis and notes a few potential future directions.

Chapter 2

Motivation

This chapter discusses the motivation for new techniques to optimize basic linear algebra computations and other arbitrary affine access and the need for a new DSL. The chapter describes the various techniques currently employed to obtain high-performance code for the computation primitives and the limitations of each method.

2.1 Library Implementation

Basic linear algebra computations are commonly specified using a fixed Basic Linear Algebra Subprograms (BLAS) interface. The BLAS abstraction allows code to be customized for high performance on various architectures. Several computer vendors such as Intel (MKL [23]), NVIDIA (CuBLAS [7]) and other open source community driven projects (OpenBLAS [25]) provide high-performant, machine-specific optimized implementations for linear algebra kernels. These libraries are optimized for architectural advancements such as vector processors, hierarchical memory with several levels of caches, hardware prefetching and shared-memory parallel processors.

There are several limitations in naively mapping linear algebra computations to use BLAS libraries. First, as observed by prior works, these libraries are well-optimized for large matrices that fit in memory and beyond by leveraging tiling and several other architecture-specific optimizations. However, for smaller matrices, these libraries do not provide the

best possible performance as observed in several prior works [35, 13]. Second, the general agreement on standard names and parameter lists for these BLAS operations allows for better portability and efficiency of these libraries at the expense of productivity. The generality of these libraries makes the interface complex comprising several tens of parameters to be specified by a programmer. This requires programmers to have some knowledge on using these BLAS libraries. We also observe that optimizations which exploit reuse across library calls cannot be expressed with the library interface itself. Furthermore, access to the library sources does not help solve the problem as the ability to extract performance with such a scheme still requires expertise and skillful programming.

2.2   Polyhedral Source-to-source tools

The other common approach to generate high-performance code from a sequential implementation of a program is by using an polyhedral source-to-source tools like Pluto [26]. Pluto accepts a sequential code and performs optimizations like loop permutation, scaling, skewing and tiling on imperfectly nested loops. The major advantage of using this approach is in the fact that the domain expert need not learn and understand any library specification. However, there are two primary disadvantages of using this approach - (i) the domain experts need to tune over a set of tile sizes to determine the optimal tile size for a given architecture. The tuning time is proportional to the application execution time and can be quite high for DSL applications with big datasets. Also, the tile size for a given loop nest depends on the number of memory accesses in it. In case there are multiple loop nests, a user cannot specify multiple tile sizes and can miss out on the performance of training the loop nests separately. (ii) BLAS libraries still perform better for large matrix sizes as compared to the code generated by Pluto.

2.3 Domain-Specific Languages

Domain-specific languages are languages designed for specific domains. They provide domain experts with the ability to express high level algorithms and computations in an intelligible manner, freeing them from low level implementation details. They also frequently provide several high level constructs like Matrix and Image as primitives. Thus, DSLs can make use of domain information to (i) abstract out tedious implementation details from the programmer hence improving productivity, (ii) extract domain information to apply optimizations which can improve the performance of the code. The optimizations applied by the DSLs are applicable to different computer architectures. FLAME [12] is an example of a DSL which internally maps the computations to BLAS calls. This approach overcomes the need for the domain expert to understand the specifics of interfacing with a library but still suffers the limitation of using a library implementation (discussed in Section 2.1). LGen [21], BTO [34], Tensor comprehension [36] are some of the other DSLs that can be used for matrix computations. These DSLs perform fusion and tiling optimizations on the specified code, hence improving its performance. However, for large matrix sizes, they still do not perform as well as the architecture specific library implementations (for e.g. Intel MKL). Currently, these DSLs use tuning of the tile sizes over a range of parameters to determine the best tile size for a given architecture.

2.4 DSL Approach

Each of the above implementations has its own advantages and disadvantages. This work proposes an approach which tries to overcome all the above limitations while still making sure to improve domain expert's productivity. This is achieved by using a DSL approach to hide the complexity of the optimizations from the domain expert. Note that one major disadvantage of all the approaches discussed above is their lack of a good tile size selection model. They either use default tile sizes or need tuning to decide on the best performing tile sizes. Thus, a model which analytically determines the tile sizes for a given loop nest with array computations having arbitrary affine access is required. Another important

optimization is to be able to determine when to replace a loop nest with an optimized library implementation, thereby not missing out on the performance benefits of the highly optimized routines. The compiler should also enable better auto-vectorization and fusion where possible for improved performance. Thus, a new DSL which automatically provides the above optimizations for domain experts is essential.

PolyMage as a DSL for Matrix Computations

Polyhedral model is a mathematical framework which provides an abstraction for representing loop information and dependencies between statements as sets and maps. Sets are used to represent the domain of the computation and maps are used to capture the dependencies and the execution order of the statements (schedule) in the program. Polyhedral model is applicable only when the array access is an affine function of the program parameters and the loop indices. The framework helps in performing loop optimizations as affine transformations on these maps. Tiling, fusion and the other optimizations are performed as a transformation on the program's schedule using the polyhedral representation of the input program. This fits perfectly for optimizing the computation primitives discussed in Section 1.1.

PolyMage [24] is a domain-specific language which was proposed for image processing applications. PolyMage implements an optimizing compiler, using the polyhedral model theory, which provides support for extracting the polyhedral representation of the input program using ISL [38] and hence generating the polyhedral schedules. Affine transformations for fusion and tiling can be applied on these schedules to obtain the desired optimizations. PolyMage also has support for polyhedral code generation and storage optimizations for the tiled computation which is an additional advantage of using this framework. Hence, PolyMage is a good candidate framework for implementing the proposed optimizations for computational primitives discussed in Section 1.1.

2.5 Summary

To summarize, this chapter described the motivation for the work in this thesis. The chapter shows how the current approaches for optimizing computational primitives using libraries, optimizing compilers and domain-specific languages suffer from various shortcomings. Thus, a comprehensive and wholistic approach to optimize the computational primitives for high-performance and productivity is thus invaluable. This chapter also discusses the advantages of using and extending PolyMage as the DSL for matrix Computations.

Chapter 3

Background

This chapter provides the necessary background on the current PolyMage DSL. The details of the existing optimizations performed by the backend compiler in PolyMage - its current fusion and tile size selection model is explained in Section 3.1. Section 3.2, highlights certain assumptions made in these models, which are ineffective when directly applied to basic matrix operations.

3.1 PolyMage

PolyMage [24] is a high-performance domain-specific language designed for image processing benchmarks and later extended for Multigrid methods [37]. It is embedded in python and has language constructs that allow domain experts to express common operations like stencils, point-wise, upsampling, downsampling, interpolate and restrict as function calls. Basic matrix-matrix or matrix-vector operations like multiplications, additions, and transpose can already be implemented using these constructs. The example in Figure 3.1 illustrates the PolyMage specification to compute the product of two matrices in a single pipeline stage. The input to the DSL is specified with the keyword `Image` (lines 10 and 11) – here the two matrices to be multiplied. A matrix can be interpreted as a function that maps a two dimensional integer domain (representing the row and column number) to a value representing the corresponding element of the matrix. Program parameters such

13

```
1   # Parameters
2   N = Parameter(Int, "N")
3
4   # variables
5   i = Variable(Int, "i")
6   j = Variable(Int, "j")
7   k = Variable(Int, "k")
8
9   # Input
10  A = Image(Double, "A", [N, N])
11  B = Image(Double, "B", [N, N])
12
13  # Domain/ Intervals
14  n_dom = Interval(Int, 0, N−1)
15
16  # Matrix multiplication operation
17  C = Reduction(([i, j], [n_dom, n_dom]),
18                ([i, j, k], [n_dom, n_dom, n_dom]),
19                Double, "C")
20  C.defn = [Reduce(C(i, j), A(i, k)   B(k, j), Op.Sum)]
```

Figure 3.1: Matrix multiplication in PolyMage

as the number of rows and columns of matrices are specified using the keyword `Parameter`. A `Function` declares each stage (or an operation) in the pipeline. The domain of the function is specified using an `Interval`. Note that the domain of the function represents the domain of its output. `Reduction` is a special type of function which takes two types of domains: (1) the domain of the output given as the first argument and (2) the domain of the computation (*reduction domain*) given as the second argument. In Figure 3.1, Lines 17-19 define the output matrix C as a two dimensional function that results from a three dimensional computation. The definition of the computation (function) is given in lines 20-22. The definition indicates that the reduction operator is a summation, and the values that are being reduced correspond to the pointwise multiplication of elements from i^{th} row of A and j^{th} column of B.

From a specification, PolyMage compiler constructs a DAG whose nodes correspond to functions (stages) and edges represent the dependences between these functions. Assuming each function as a statement, it constructs a polyhedral representation for each

function. Polyhedral schedules are computed to fuse and tile these functions. PolyMage performs *overlapped tiling*; a strategy that satisfies dependences between tiles by performing redundant computation. These tiles are executed in parallel. The set of functions within a tile form a *group*. The input to a group is called a *live-in* and output of a group is called a *live-out*. Functions within a group, excluding live-in and live-out, are allocated thread-local scratchpads – small buffers whose sizes are of the order of tile sizes. PolyMage performs storage optimizations [37] that reuse scratchpads and the memory allocated for live-in and live-out across groups, resulting in significant performance gains due to reduced memory footprint.

Grouping (fusion) and tile sizes determine the amount of overlap and hence the amount of redundant computation. The rest of this section discusses the current fusion and tile size selection model in PolyMage.

3.1.1   Dynamic Programming based Fusion Model

Fusion or grouping or merging of the various functions is an important class of optimizations as it enables reuse of the data brought from memory. PolyMage originally proposed a greedy fusion heuristic. A group is merged with its child if the following two conditions are satisfied (i) the dependence distance between the group and its child is a constant and (ii) the size of the overlapping region does not exceed the overlap threshold. This algorithm however, had many limitations as explained in Jangda and Bondhugula [19] and the fusion model was revised to use a Dynamic programming (DP) based heuristic which evaluates all possible fusion strategies to pick the best one using a cost function.

The DP recurrence is shown in Figure 3.2. Where G is the grouping for a portion of the DAG. G can be written as:

$$G = \{H_1, H_2, \ldots, H_n\},$$

$$H_i \text{ is a connected sub-graph of } S,$$

$$\text{for any } H_i, H_j \in G, \ H_i \cap H_j = \emptyset. \tag{3.1}$$

$$
\begin{aligned}
F(G) \;=\;& F(\{H_1, H_2, \ldots, H_n\}) \\[1em]
=\;& \begin{cases}
\min \left(\begin{aligned}
& \min_{H_i \in G} \left(\min_{s_j \in SUCC(H_i)} F\left(\{H_1, H_2, \ldots, H_i \cup \{s_j\}, \ldots, H_n\}\right) \right), \\
& \sum_{H_i \in G} \mathrm{COST}(H_i) + \min_{P_i \in PARTITIONS\left(\bigcup_{H_i \in G} SUCC(H_i)\right)} F(P_i)
\end{aligned} \right), & \text{if } \bigcup_{H_i \in G} SUCC(H_i) \neq \emptyset, \\[2em]
\sum_{H_i \in G} \mathrm{COST}(H_i), & \text{if } \bigcup_{H_i \in G} SUCC(H_i) = \emptyset.
\end{cases}
\end{aligned}
$$

Figure 3.2: Dynamic programming formulation of fusion
Courtesy: Jangda and Bondhugula [19]
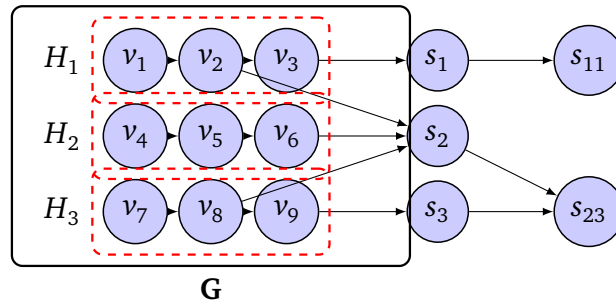
Figure 3.3: DP optimal sub-structure
Courtesy: Jangda and Bondhugula [19]

The function F(G) represents the optimal grouping with minimum cost. *PARTITIONS* returns all possible partitioning, and *SUCC* returns all successor nodes in the DAG.

If all groups H_i in G do not have successors, then $F(G)$ is the cost of G; otherwise, it is the minimum between grouping $H_i \in G$ with any of H_i's successors, and that of not grouping with any of H_i's successors. The algorithm starts with the source vertex of the pipeline graph. This recurrence uses memoization to store the grouping and cost of $F(G)$.

The cost function $F(G)$ is shown in Algorithm 1. The cost function takes in four criteria:

- amount of locality

- number of cores available to run in parallel

- amount of redundant computation

Algorithm 1: Existing cost function for DP based fusion

Courtesy: Jangda and Bondhugula [19]

Input: H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores
Output: Cost of the Grouping

1 **Function** *Cost(H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores)***:**
2   **if** *not constantDependenceVectors(H)* **then**
3   return ∞
4   **end**
5   ⟨cost, tileSizes, overlapSize⟩ ← CostForCacheSize(H, L1CacheSize, NCores, InnerMostTileSize)
6   **if** *overlapSize > TileVolume(H, tileSizes)* **then**
7   ⟨cost, tileSizes, overlapSize⟩ ← CostForCacheSize(H, L2CacheSize, NCores, InnerMostTileSize)
8   **end**
9   return ⟨cost, tileSizes⟩
10 **Function** *CostForCacheSizeReduction(H, cacheSize, NCores, innerMostTileSize)***:**
11   liveout_size ← liveOutsSize(H)
12   totalFootprint ← intermediateBuffersSize(H) + liveout_size
13   tileFootprint ← min(totalFootprint÷NCores, cacheSize)
14   tileSizes ← ComputeTileSizes (H, tileFootprint,innerMostTileSize)
15   livein_tile_size ← liveInTileSize(H, tileSizes)
16   liveout_tile_size ← liveOutTileSize(H, tileSizes)
17   comp_vol ← ComputeTileVolume(H, tileSizes)
18   n_tiles ← totalFootprint ÷ tileFootprint
19   overlapSize ← OverlapSize(H, tileSizes)
20   relative_overlap ← overlapSize ÷ tileFootprint
21   dim_diff ← dimSizeStandardDeviation(H)
22   cost ← w_1×(livein_tile_size + liveout_tile_size)÷comp_vol $-w_2$ × ((n_tiles + NCores-1) % NCores) + w_3 × relative_overlap + w_4 × dim_diff
23   return ⟨cost, tileSizes, overlapSize⟩

- difference between the extent of the dimensions

Algorithm 2: Existing tile size computation

Courtesy: Jangda and Bondhugula [19]

    Input: H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores
    Output: Tile sizes of each dimension of *G*

1   **Function** *ComputeTileSizes(H, tileFootprint, innerMostTileSize)***:**
2   tileVol ← tileFootprint ÷ numBuffers(H)
3   nDims ← numDims(H)
4   dimReuse[1 ... nDims] ← getDimensionalReuse(H)
5   dimSizes[1 ... nDims] ← getDimensionalSizes(H)
6   tileSizes[nDims] ← min(dimSizes[nDims], innerMostTileSize)
7   τ ← tileVol ÷ tileSizes[nDims]
8   maxDimReuse ← max(dimReuse[1:nDims − 1])
9   **for** $i \in 1 \rightarrow nDims - 1$ **do**
10   τ ← τ ÷ (dimReuse[i] ÷ maxDimReuse)
11   **end**
12      $\tau \leftarrow \tau^{1/(nDims-1)}$
13   **for** $i \in 1 \rightarrow nDims - 1$ **do**
14   tileSizes[i] ← min(dimSizes[i],
15   τ × dimReuse[i] ÷ maxDimReuse)
16   **end**
17   return tileSizes

Each parameter is multiplied by an experimentally determined weight. The cost of a group is calculated as follows:

$$cost = w_1 \times \text{ratio of data to computation}$$
$$- w_2 \times ((\text{num\_tiles} + \text{num\_cores - 1})\% \text{ num\_cores})$$
$$+ w_3 \times \text{fraction of overlap}$$
$$+ w_4 \times \text{relative difference between sizes of dimensions,}$$

where w1, w2, w3, and w4 were calculated experimentally. The DP recursion terminates when a group has no successors to fuse. An exhaustive search for all possible groupings is made and the one with the minimal cost is chosen. The cost function also returns the

tile size for the given partitioning. The details on tile size calculations are explained subsequently in Section 3.1.2.

3.1.2   Tile Size Selection Model

Jangda and Bondhugula [19] proposed a model-driven approach to find tile sizes for each group as shown in Algorithm 2. Tile size is obtained as a byproduct of the grouping algorithm's cost calculation. The inputs to the model are (i) L1 cache size, (ii) L2 cache size and (iii) dimensional reuse along a dimension. A reuse score for each dimension is calculated using the well-known technique described in [42]. The model proceeds to fit the data for L1 cache size, by computing a tile size proportional to the dimensional reuse. If the amount of overlap is greater than the overlap threshold, the model runs the same algorithm for L2 cache size. The model assumes that there exists a one-to-one correspondence between the iterations and the data accessed within a group. Thus, the product of the tile sizes along each dimension is equal to the number of distinct memory accesses within a tile (tile volume). If the innermost loops are parallel, they allocate a fixed tile size of (128 or 256) for the innermost dimension to enable profitable prefetching and auto-vectorization.

For image processing pipelines, there is a one-to-one mapping between the iteration space and the data accessed. Hence, $\tau_1 \tau_2 ... \tau_m = T$ where T represents the total allowable tile footprint.

Let γ_i be the ratio of the reuse score of the ith dimension with the maximum reuse. γ_i is represented as the dimReuse[i] in Algorithm 2. Let τ be the tile size for the dimension with maximum reuse. Set $\tau_i = \gamma_i \tau$. Hence: $\tau^m \gamma_1 \gamma_2 ... \gamma_m = T$, and τ can be determined and hence the tile size for each dimension.

3.2   Limitations of PolyMage Compiler

There are some limitations to using PolyMage for matrix computations. The downsides are:

- Representation of linear algebra computations in PolyMage using reduction operations (shown in Figure 3.1) is complex and unintuitive.

- Tile size selection model in PolyMage assumes that there exists a one-to-one correspondence between the iterations and the data accessed. However, this does not hold for matrix computations. For example, a naive matrix-matrix multiplication with the standard three dimensional loop nest accesses $\mathscr{O}(N^2)$ data in $\mathscr{O}(N^3)$ iterations.

- PolyMage's ability to enable auto-vectorization is limited in cases where the innermost loop carries a dependence, as in the case of matrix multiplication.

- Reduction functions in PolyMage are treated as separate groups. These are neither tiled nor parallelized, thereby significantly degrading performance.

- PolyMage lacks the ability to invoke routines provided by highly tuned libraries, like OpenBLAS or Intel MKL, for basic linear algebra computations.

3.3   Summary

To summarize, the chapter provided the background on PolyMage, its existing fusion and tile size selection model and also discussed the limitations for using it to optimize matrix computations and arbitrary affine accesses. In the subsequent chapters enhancements to the PolyMage compiler to provide support and generate optimized code for the computational primitives are discussed.

Chapter 4

Optimizing Matrix and Arbitrary Affine Computations in PolyMage

This section describes the optimizations that are performed to improve the performance of the computation primitives discussed in Section 1.1. The additional stages added to Poly-Mage compiler flow is explained in Section 4.1. Section 4.2 describes the extensions in PolyMage's language specification to allow expressing computational primitives efficiently. Following that, Section 4.3 describes the tile size selection model that works for any arbitrary affine access. The heuristic for mapping computations to a suitable library call is described in Section 4.4. Section 4.5 describes the algorithm used to permute parallel loops to inner-most level in multi-level loop nests to exploit vectorization. Finally, Section 4.6 describes the enhancements made in PolyMage to facilitate grouping of reduction operations (introduced in Chapter 3)

4.1   Compiler Stages

This section describes the enhancements that were made to the PolyMage compiler to support optimizations for matrix computations. It describes the newly added stages as well as enhancements done to certain other phases and their contribution to the optimization of the benchmarks.
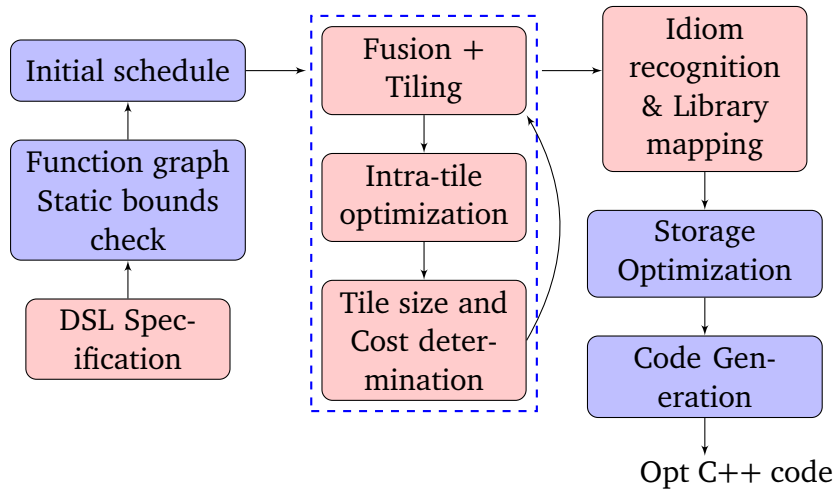
Figure 4.1: Enhanced compiler flow in PolyMage

Figure 4.1 shows the stages in the enhanced PolyMage compiler. The blocks in red indicate phases that have either been newly added or have been extended to include additional optimizations while the blue blocks indicate existing compiler flow. PolyMage accepts the specification from the programmer. For ease of use and to improve programmers productivity, PolyMage's specification has been extended to include language constructs that declare matrices and overload operators for basic matrix operations. The list of functions and operators introduced is described in Section 4.2. This is followed by the construction of the *function graph*. All computations in PolyMage are represented as functions (Figure 4.2). A function graph, represents these functions as nodes and the dependencies between them as edges. Thus, it captures the producer-consumer relationship between the operations. The bound checks for each of the functions defined is also performed at this stage. An out of bound error is thrown in case the computation exceeds the bounds specified in its domain. This is followed by extracting the polyhedral representation of the program and generating the initial schedule based on the order function graph.

The next stage tries to perform fusion and tiling optimizations on the initial schedule. Support to handle reduction operations in the fusion and tiling stage in the PolyMage compiler has been newly introduced. Details of these stages are presented in Section 4.6. After fusion and tiling, a new intra-tile optimization pass is introduced to bring parallel loops to the innermost level in order to enable auto-vectorization by a native compiler.

The algorithm used for finding an effective innermost loop is described in Section 4.5. Following this, the tile sizes for the new schedule is calculated using the approach presented in Section 4.3. This approach supports arbitrary affine accesses and hence can be effectively incorporated for basic matrix operations. The model proposed in Jangda and Bondhugula [19] is used, to determine the cost of the current grouping. The fusion and tiling + intra-tile optimization + tile selection phases are iterated for all possible groupings and the one with the lowest cost is selected as the final schedule as explained in Chapter 3.

Once the final schedule has been selected, a new idiom recognition stage is introduced. This stage maps basic matrix operations to optimized library calls when it is cost-effective. After this step, the storage optimizations of PolyMage described in Section 3 are performed and optimized C++ code is generated. This C++ code is then compiled with a native compiler like gcc. The rest of this chapter presents the details of each stage that performs fusion and tiling, tile size selection, intra-tile optimizations, idiom recognition, and language extensions to support new primitives.

4.2 Language Specification

This section describes the new language constructs and operations that were introduced in PolyMage. The introduction of matrix-specific constructs improves the readability and tractability of the code. Two new constructs `Matrix` and `Vector` for improved productivity have been introduced. PolyMage's specification is extended to include overloading of the basic operators for matrix/vector operations. Other commonly used operations are also supported as primitive functions. These overloaded operators along with their details are described in Table 4.1.

Table 4.1: Operators supported by PolyMage; where **A** and **B** are either matrix or vector

Operator	Usage	Description
$+$	$\mathbf{A} + \mathbf{B}$	Point-wise addition
$-$	$\mathbf{A} - \mathbf{B}$	Point-wise subtraction
$*$	$\mathbf{A} * \mathbf{B}$	Multiplication

Function name with usage	Description
$elementwise\_mul(\mathbf{A}, \mathbf{B})$	Element-wise multiplication
$scalar\_mul(\mathbf{A}, \alpha)$	Matrix/Vector Scalar multiplication
$transpose(\mathbf{A})$	Transpose
$symm(\mathbf{matA}, \mathbf{matB}, \mathbf{matC}, M, N, \alpha, \beta)$	Symmetric Matrix multiply
$syr2k(\mathbf{matA}, \mathbf{matB}, \mathbf{matC}, \alpha, \beta)$	Symmetric rank-2k operations
$syrk(\mathbf{matA}, \mathbf{matC}, \alpha, \beta)$	Symmetric rank-k operations
$trmm(\mathbf{matA}, \mathbf{matB}, \alpha, \beta)$	Triangular Matrix multiply

Table 4.2: New Functions supported in PolyMage
where $\mathbf{A}$ and $\mathbf{B}$ are either matrix or vector; α, β are scalars ; $\mathbf{matA}, \mathbf{matB}, \mathbf{matC}$ are matrices

These overloaded operators allow inferring the dimension of the output from the input dimensions and the computation performed, freeing the programmer from having to explicitly specify this. For example, in the case of matrix-matrix multiplication of dimensions, say (MxN) and (NxK), the dimensions of the resultant matrix dimensions (MxK) is automatically inferred from input dimensions and the multiplication operator(*). This avoids many out of bounds error which can arise by specifying the dimensions(domain) incorrectly while defining any matrix operation in PolyMage specification.

Several new functions for the commonly used matrix and vector operations are introduced as primitives. The list of these functions is described in Table 4.2.

To illustrate the power of these new functions primitives in PolyMage, consider the matrix-matrix multiplication specification using the old PolyMage specification as shown in Figure 3.1. This can be re-written more concisely as shown in Figure 4.2 using the constructs introduced in this section. Line $5-6$ represents the definition for the input matrices. Line 9 represents the matrix multiplication operation. Note that the dimensions for the matrix C are not specified. The compiler directly infers it to be a NxN matrix.

4.3   Tile Size Selection

This section describes the tile size selection model and illustrates its working with examples. Tiling or blocking is performed on loop nests to exploit locality in computations. This helps to improve performance by reusing elements brought into the cache from memory

```
1   # Parameters
2   N = Parameter(Int, "N")
3
4   # Input matrices
5   A = Matrix(Double, "A", [N, N])
6   B = Matrix(Double, "B", [N, N])
7
8   # Matrix multiplication
9   C = A * B
```

Figure 4.2: Matrix multiplication code represented in PolyMage

before being evicted. Further, a good tile size is one that is sized proportionately to fit the elements of the accessed computations perfectly in the cache. Choosing too small a tile size leaves cache space underutilized while too large a tile size may cause the computation to overflow the cache causing performance loss.

The existing tile size selection model in PolyMage, discussed in Chapter 3, makes an assumption that there exists one-to-one mapping between the iterations and the data accesses. This assumption makes the model inapplicable to matrix operations, as most of these operations involve reductions. The tile size model proposed here is generic and works for any arbitrary affine access.

The proposed tile size selection model assigns tile sizes to each dimension(loop) based on the amount of reuse available along that dimension. The reuse along a dimension also represents the number of data points that need to be received from the previous tile along that dimension. Thus, higher reuse implies higher data point movement. Therefore, increasing the tile size along that dimension, reduces the number of tiles which in turn reduces the total communication across tiles in the computation.

The reuse along a dimension is used to construct a *reuse expression*. This reuse expression is constructed for each group and represents the total memory accessed by a tile. This is equated to the tile capacity and the solution to this equation gives the tile sizes. The construction of the equation and the tile size calculation is explained with an example in the next paragraph.

Consider a naive matrix-matrix multiplication with the standard three dimensional loop

```
1    for(int  i  =  0;  i  <=  NI;   i=i+1)
2      for(int  j  =  0;  j  <=  NJ;   j=j+1)
3        for(int  k  =  0;  k  <=  NK;   k=k+1)
4          C[(i  *  NJ)  +  j]  =  C[(i  *  NJ)  +  j]  +  (A[(i  *  NK)  +  k]
5                                              *  B[(k  *  NJ)  +  j]);
```

Figure 4.3: Generated matrix-matrix multiplication code

nest for multiplying two matrices A and B to yield C as shown in Figure 4.3. Assume that the loop nests are tiled with tile sizes τ_i, τ_j and τ_k for loops i, j and k respectively. The memory accessed by each matrix for this tile is:

$$\tau_i * \tau_j \text{ elements of matrix C}$$
$$\tau_i * \tau_k \text{ elements of matrix A}$$
$$\tau_k * \tau_j \text{ elements of matrix B}$$

Hence the tile volume is given by

$$\tau_i * \tau_j + \tau_j * \tau_k + \tau_k * \tau_i = T. \tag{4.1}$$

In order to utilize the cache efficiently, the tile volume must be equal to the cache size. To exploit temporal reuse efficiently, the tile sizes of the dimensions with better temporal reuse should be higher. Keeping this in consideration, let us assume the tile size for each dimension i to be $\tau_i = \gamma_i * \tau$ where γ_i is the dimensional reuse along i. Therefore, Equation 4.1 can be re-written as:

$$(\gamma_i * \gamma_j + \gamma_j * \gamma_k + \gamma_k * \gamma_i) * \tau^2 = \mathbf{C}. \tag{4.2}$$

where $\mathbf{C}$ represents the cache size. Equation 4.2 represents the reuse expression for the matrix multiplication example. This equation can then be solved for τ and tile sizes can be computed.

The reuse along each dimension is calculated by counting the number of temporal and group temporal reuse along each dimension as proposed by Wolfe et. al [42]. For the matrix multiplication example, the dimensions i, j have temporal reuse of 1 and k has

temporal reuse of 2 (Considering both read and write operations for the access $C[i][j]$). These reuse values are normalized and written as :

$$\gamma_i = 0.5, \gamma_j = 0.5, \gamma_k = 1$$

In order to improve auto-vectorization, the tile size of the inner-most loop is fixed 256. In this case let us assume the inner-most loop is the j loop. The reason for choosing j loop as the inner-most loop is explained in detail in the next section. Assume an L1 cache size of 32KB and the data size as the size of double i.e. 8 bytes. Hence, Equation 4.2 can be re-written as:

$$(0.5 * \tau) * 256 + 256 * (0.5 * \tau) + (1.0 * 0.5) * \tau^2 = (32768/8).$$
$$\implies 0.5 * \tau^2 + 256 * \tau - 4096 = 0.$$

Solving the above equation yields the roots 15.52 and -527.52. Discarding the negative root, the tile size for the loops i, j and k can be obtained by multiplying the floor of the positive root with its corresponding dimension reuse. The final tile sizes obtained are $\tau_i = 7, \tau_j = 256$ and $\tau_k = 15$. For a matrix size of $NI = 2000$, $NJ = 2300$ and $NK = 2600$ the generated tiled code is shown in Figure 4.4.

Algorithm 3 describes our tile size selection model for an input group G. For each dimension d of the group, the algorithm computes the dimensional reuse (line 3) along d as the sum of temporal and group temporal reuse. The number of distinct memory accesses in G is calculated in line 6. The reuse expression for G can be obtained by first computing the tile volume, using the number of distinct memory access in a tile and further by expressing the tile size of each dimension as a factor of dimensional reuse. However, it is desirable that the innermost dimension has a large tile size when it is vectorizable. Therefore, the tile size corresponding to the innermost dimension (inner_dim) is set to the smallest of upper bound on the innermost dimension (obtained in Line 4), or inner_tile_size, an input to the algorithm (256). The algorithm to find the innermost dimension of a group is described subsequently in Section 4.5. Further, the algorithm computes the reuse expression (line 7) and the positive root of Equation 4.2 is used to determine the tile sizes (lines 8-10).

```
 1
 2    for (int _T_i = 0; (_T_i <= 76); _T_i = (_T_i + 1))
 3      for (int _T_j = 0; (_T_j <= 35); _T_j = (_T_j + 1))
 4        for (int _T_k = 0; (_T_k <= 49); _T_k = (_T_k + 1)) {
 5          int _ct0 = (1999 < ((26 * _T_i) + 25))? 1999: ((26 * _T_i) + 25);
 6          int _ct1 = (1999 < ((26 * _T_i) + 25))? 1999: ((26 * _T_i) + 25);
 7          for (int i = (26 * _T_i); (i <= _ct1); i = (i + 1)) {
 8            int _ct4 = (2299 < ((64 * _T_j) + 63))? 2299: ((64 * _T_j) + 63);
 9            int _ct5 = (2299 < ((64 * _T_j) + 63))? 2299: ((64 * _T_j) + 63);
10            for (int j = (64 * _T_j); (j <= _ct5); j = (j + 1)) {
11              int _ct2 = (2599 < ((53 * _T_k) + 52))? 2599: ((53 * _T_k) + 52);
12              int _ct3 = (2599 < ((53 * _T_k) + 52))? 2599: ((53 * _T_k) + 52);
13              for (int k = (53 * _T_k); (k <= _ct3); k = (k + 1))
14                C[(i * NJ) + j] = C[(i * NJ) + j] + (A[(i * NK) + k]
15                                                    * B[(k * NJ) + j]);
16            }
17          }
18        }
```

Figure 4.4: Generated tiled matrix-matrix multiplication code

Table 4.3 represents the reuse expression in Column 3 for accesses shown in Column 1. The number of distinct accesses per tile is given in Column 2. In cases of loop scaling transformations ($\alpha \geq 1$ in Row 2), the number of distinct accesses will still be equal to the tile size t_i.

Another interesting scenario is with accesses of the form a[i-j] where the tile volume is still the sum of the tile sizes along the dimensions i and j. This is explained with example code from the DSP domain shown in Figure 4.5. The code represents only the intra-tile loops of the tiled code. The tile size for dimension i is denoted as $\tau1$ and that of j is denoted as $\tau2$. Then the amount of memory accessed by the tiled loops ii and jj by array ybs is $\tau1$ and by array $window$ is $\tau2$. The memory required for yds is calculated as $(\tau1-0)-(0-\tau2)$ which is equal to $\tau1 + \tau2$. The reuse expression can be constructed as $(\tau1) + (\tau1 + \tau2) + (\tau2) = T$, where $\tau1$ and $\tau2$ can be replaced with the corresponding dimensional reuse scores. Reuse expressions can be constructed for multidimensional array access as shown in the last row of Table 4.3.

Algorithm 4 shows how the new tile size selection model fits in the Cost function used by the dynamic programming based fusion model. Lines 2-4, checks if the group contains any *Reduction* operations. In case of reduction operations, the *relative_overlap* calculation

---

Algorithm 3: Proposed tile size computation

Input: group *G*, cache_size, inner_tile_size, inner_dim
Output: Tile sizes of each dimension of *G*

1 **Function** ComputeTileSizes(*G, cache_size, inner_tile_size, inner_dim*):
2   *nDims* ← numDim(G)
3   *dim_reuse* [1...*nDims*] ← getDimReuse(*G*)
4   *inner_dim_size* ← getInnerDimSize(*G*)
5   *tile_sizes* [*inner_dim*] ← min (*inner_dim_size*,
                       inner_tile_size)
6   *mem_access* ← distinct memory references in *G*
7   *reuse_eqn* ← getReuseEquation (*mem_access, dim_reuse, inner_dim,*
              tile_size)
8   *root* ← floor(positive_root(*reuse_eqn*))
9   **for each** *i* ∈ *nDims* **do**
10   │ *tile_sizes* [i] ← *dim_reuse*[i] * *root*
11   **endfor**
12   return *tile_sizes*

```
1    for (int ii = 0; (ii <= t1); ii = (ii + 1))
2      for (int jj = 0; (jj <= t2); jj = (jj + 1))
3         ybs[ii] += (yds[((M + ii) − jj)] * window[jj]);
```

Figure 4.5: Generated DSP code from vuvuzela filter

is not required. This is because rectangular tiling is performed for these groups. The algorithm, calls the ComputeTileSize function described in Algorithm 3 (shown in Line 18). The tile size selection model requires the preferred innermost dimension which is calculated in Line 17. The model to find a good inner-most dimension is described in Section 4.5. Lines 29-31 represent the *cost* calculation of rectangular tiled groups. Thus, the parameters with relative overlap size are dropped.

To summarize, the tile size selection method is sufficiently generic to be applicable for arbitrary affine accesses. The tile size selection model is robust enough to be extended for multilevel tiling with minor modifications to Algorithm 3.

Algorithm 4: Updated cost function for DP based fusion

Input: group H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores
Output: Cost of group H and its tileSizes

1 **Function** *Cost(H, L1CacheSize, L2CacheSize, InnerMostTileSize, NCores)*:
2   **if** *has ReductionFunctions(H)* **then**
3   reduction ← *True*
4   ⟨cost, tileSizes⟩ ← CostForCacheSize(H, L1CacheSize, NCores, InnerMostTileSize, reduction)
5   **end**
6   **else**
7   **if** *not constantDependenceVectors(H)* **then**
8   return ∞ /* Overlap tiling not feasible */
9   **end**
10   reduction ← *False*
11   ⟨cost, tileSizes, overlapSize⟩ ← CostForCacheSize(H, L1CacheSize, NCores, InnerMostTileSize, reduction)
      if *overlapSize > TileVolume(H, tileSizes)* **then**
12   ⟨cost, tileSizes, overlapSize⟩ ← CostForCacheSize(H, L2CacheSize, NCores, InnerMostTileSize, reduction)
13   **end**
14   **end**
15   return ⟨cost, tileSizes⟩
16 **Function** *CostForCacheSize(H, cacheSize, NCores, innerMostTileSize, reduction)*:
17   liveout_size ← liveOutsSize(H)
18   totalFootprint ← intermediateBuffersSize(H) + liveout_size
19   tileFootprint ← min(totalFootprint÷NCores,cacheSize)
20   innerMostDim ← IntratileOptimize(H)
21   tileSizes ← ComputeTileSizes(H, tileFootprint, innerMostTileSize, innerMostDim)
22   livein_tile_size ← liveInTileSize(H, tileSizes)
23   liveout_tile_size ← liveOutTileSize(H, tileSizes)
24   comp_vol ← ComputeTileVolume(H, tileSizes)
25   n_tiles ← totalFootprint ÷ tileFootprint
26   dim_diff ← dimSizeStandardDeviation (H)
27   **if** *reduction = False* **then**
28   overlapSize ← OverlapSize(H, tileSizes)
29   relative_overlap ← overlapSize ÷ tileFootprint
30   cost ← w_1×(livein_tile_size + liveout_tile_size)÷comp_vol − w_2 × ((n_tiles + NCores−1) % NCores) + w_3 × relative_overlap + w_4 × dim_diff
31   **end**
32   **else**
33   cost ← w_1×(livein_tile_size + liveout_tile_size)÷comp_vol − w_2 × ((n_tiles + NCores−1) % NCores) + w_4 × dim_diff
34   **end**
35   return ⟨cost, tileSizes⟩

Table 4.3: Reuse expression

Access	Number of distinct accesses	Reuse expression
$a[i]$	τ_i	$\gamma_i * \tau$
$a[\alpha * i]$	τ_i	$\gamma_i * \tau$
$a[i+j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i-j]$	$\tau_i + \tau_j$	$(\gamma_i + \gamma_j) * \tau$
$a[i][j]$	$\tau_i * \tau_j$	$(\gamma_i * \gamma_j) * \tau$

4.4 Mapping to Function Calls

This section describes the heuristic used to map PolyMage functions into highly tuned BLAS implementations from OpenBLAS or Intel MKL and FFTW subroutine calls for computing Fourier transforms. As alluded to in Section 1, there exist highly optimized routines for linear algebra computations. Hence, exploiting the performance provided by these libraries is essential for generating optimal performance.

In order to map basic matrix operations to optimized BLAS routines, an *idiom recognition* stage is incorporated. This approach allows to maintains backward compatibility. This allows the optimization to be performed even in applications where the old constructs of PolyMage specification are used. BLAS routines are invoked only when they are profitable. It is well known that BLAS calls are not profitable for small matrix sizes and this has been extensively discussed in the past many works [35, 13].

The idiom recognition algorithm, traverse the polyhedral representation (PolyAST) of the function's expression tree and attempt to match it with the set of library routines. If a match is found and the mapping is profitable, the schedule is modified to generate a library call instead of generating code for the actual computation. BLAS routines provide sub-par performance for small matrix sizes and hence operations involving matrix-matrix (vector) multiplications are mapped to BLAS routines only if $M * N * K \geq (256)^3$ [†], where M, N, and K are the upper bounds of the loops representing the computation. This simple heuristic is used to determine the profitability of invoking the library call. Fourier transforms are

---

[†]This value was obtained experimentally on our setup by measuring BLAS performance while sweeping the design space. This value also correlates with prior observations in [13]

```
1      cblas_dgemm(CblasRowMajor, CblasNoTrans,CblasNoTrans,
2                     NI, NJ, NK, 1, A, NK, B, NJ, 1, C, NJ);
```

Figure 4.6: Generated matrix-matrix multiplication code with mapping to BLAS call

```
1   fftw_init_threads();
2   fftw_plan_with_nthreads(omp_get_max_threads());
3   fftw_plan_var = fftw_plan_dft_1d(220861,
4                        reinterpret_cast<fftw_complex >(_arr_6_18),
5                        reinterpret_cast<fftw_complex >(_arr_6_19),
6                     FFTW_FORWARD, FFTW_ESTIMATE);
7   fftw_execute((fftw_plan) fftw_plan_var);
8   fftw_destroy_plan((fftw_plan) fftw_plan_var);
```

Figure 4.7: Generated FFTW call for upsampling

always mapped to FFTW library call when a suitable computation is found since FFTW library calls reduce the complexity of computing discrete fourier transform from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$.

Further, our approach is modular and can be extended to map functions calls from any new library with appropriate heuristics (for example libxsmm [13], MKL DNN [15]). Our approach is architecture and library agnostic and hence, can benefit from any performance advancements in linear algebra libraries. In other words, the modular approach in the backend and abstraction in PolyMage specification, allows libraries to be swapped seamlessly.

Our approach also allows us to exploit locality across multiple BLAS calls in cases where these functions have input data reuse. These functions are tiled at outer levels and computations within the tile are mapped to BLAS routines. However, for the scope of the benchmarks investigated in this work no significant performance benefits were seen. Figure 4.6 shows the code generated when the idiom recognition phase recognizes the matrix multiplication computations as shown in Figure 4.2 and maps the computation to the BLAS dgemm(double precision matrix multiplication) call. Note that due to the backward compatibility of our approach, the optimization of mapping to a BLAS call when profitable is successful for the code written both in the form given in Figure 3.1 and Figure 4.2. Similarly, Figure 4.7 shows the sample code when a computation is mapped to an FFTW call.

4.5   Intra-tile Optimization

An optimization which helps improve the performance by providing code which exploits auto-vectorization by the native compiler is explained in this section. Exploiting vectorization derives rich performance benefits for the code on modern wide-SIMD machines. PolyMage relies on host compilers like gcc/icc for auto-vectorization. These compilers generate efficient vector code when

- the innermost dimension is parallel and

- the innermost dimension has stride-0 or stride-1 accesses.

PolyMage's intra-tile optimization mechanism is responsible to find the best vectorizable innermost dimension for a given *group*. A *group* in PolyMage represents a set of functions that can be tiled.   Intra-tile optimization phase employs Algorithm 5 to try to determine a

---

Algorithm 5: Intra-tile Optimization

Input: group (G)
Output: Innermost dimension for each function in G
1 **Function** IntratileOptimize(G):
2   **for each** $function(f) \in group(G)$ **do**
3   **for each** $dimension(d) \in function(f)$ **do**
4   s $\leftarrow$ getNumSpatialReuse(d, f)
5   t $\leftarrow$ getNumTemporalReuse(d, f)
6   a $\leftarrow$ getTotalAccess(d, f)
7   v $\leftarrow$ isVectorizable(d, f)
8   score[d] $\leftarrow$ score[d] +
                       $(2*s)+(4*t)+(8*v)-(16*(a-s-t))$
9   **endfor**
10   **endfor**
11   inner_dim $\leftarrow$ getDimWithMaxScore(score)
12   return *inner_dim*

"vectorization friendly" dimension to be at the innermost level. Assuming each dimension d of the input group G is the innermost dimension, the algorithm finds, per iteration of the innermost dimension:

- the number of memory accesses which have spatial reuse (s),

- the number of memory accesses which have temporal reuse (t) and

- the total number of distinct memory accesses (a).

This corresponds to lines 4-6 in Algorithm 5. A dimension is considered vectorizable (v=1) if it is parallel and does not result in non-contiguous accesses in the innermost iteration (line 7). The score for the dimension is calculated using the heuristic in line 8 which favours a parallel dimension that has stride-0 access (temporal reuse), stride-1 access (spatial reuse) and smaller number of accesses with high scatter-gather distances. The intra-tile iterator corresponding to the dimension with the highest score is permuted to the innermost level of the loop nest. This loop is marked vectorizable if it is parallel.

For example, consider a naive matrix-matrix multiplication with three dimensional loop nest as in Figure 4.3. The scores for each dimension (loop) is shown in Table 4.4. The loop k carries a dependence while the loop i, when permuted to the innermost level, results in non-contiguous accesses for arrays C and A. Hence both i and k loops are not considered vectorizable. The algorithm finds the highest score for the j loop as it is parallel and all arrays have stride-0 and stride-1 accesses only. The intra-tile iterator corresponding to loop j is permuted to the innermost level. The final tiled code after applying intra-tile optimization which permutes the j loop is shown in Figure 4.8.

Table 4.4: Intra-tile optimization score for matmul

dim	s	t	v	a	score
i	0	1	false	4	-44
j	3	1	true	4	18
k	1	2	false	4	-6

4.6   Fusion for Reductions

The existing Dynamic Programming (DP) based fusion model [19] is extended to support fusion and tiling of reduction operations. These operations are tiled with rectangular tiles only when dependences along all the dimensions are non-negative. Dimensions of two

```
1
2   for (int _T_i = 0; (_T_i <= 76); _T_i = (_T_i + 1))
3     for (int _T_j = 0; (_T_j <= 35); _T_j = (_T_j + 1))
4       for (int _T_k = 0; (_T_k <= 49); _T_k = (_T_k + 1))
5       {
6           int _ct0 = (1999 < ((26 * _T_i) + 25))? 1999: ((26 * _T_i) + 25);
7           int _ct1 = (1999 < ((26 * _T_i) + 25))? 1999: ((26 * _T_i) + 25);
8           for (int i = (26 * _T_i); (i <= _ct1); i = (i + 1))
9           {
10            int _ct2 = (2599 < ((53 * _T_k) + 52))? 2599: ((53 * _T_k) + 52);
11            int _ct3 = (2599 < ((53 * _T_k) + 52))? 2599: ((53 * _T_k) + 52);
12            for (int k = (53 * _T_k); (k <= _ct3); k = (k + 1))
13            {
14              int _ct4 = (2299 < ((64 * _T_j) + 63))? 2299: ((64 * _T_j) + 63);
15              int _ct5 = (2299 < ((64 * _T_j) + 63))? 2299: ((64 * _T_j) + 63);
16              for (int j = (64 * _T_j); (j <= _ct5); j = (j + 1))
17                C[(i * NJ) + j] = C[(i * NJ) + j] + (A[(i * NK) + k]
18                                                * B[(k * NJ) + j]);
19            }
20          }
21        }
```

Figure 4.8: Generated tiled matrix-matrix multiplication code with intra tile optimization

functions are fused only if a dependence is not satisfied at that level, i.e., the corresponding loop does not carry a dependence. This check is performed by comparing the memory accesses of the uses with the definition of the function in the specification [24].

For example, consider the code in Figure 4.9. The code corresponds to a matrix multiplication followed by adding a bias to all the elements of the resultant matrix. When dynamic programming based fusion model is run, the algorithm first finds all the loops which carry the dependence. The dependences which exist include:

- RAW dependence on line 4 by tmp[i1][j1] and

- RAW dependence between write of tmp[i1][j1] on Line4 and read of tmp[i2][k2] on line7.

The first dependence is carried by loop $k1$ (Line 3). The second dependence is satisfied because the two loop nests are distributed. Note that loops $i1$, $j1$, $i2$, and $j2$ do not carry any dependence. Next, the model performs dimension matching, also called as *alignment*, to figure out the loops which can be fused together. This algorithm is similar to that of

```
1 for (i1 = 0; i < NI; i++)
2   for (j1 = 0; j < NJ; j++)
3     for (k1 = 0; k < NK; ++k)
4       tmp[i1][j1] += A[i1][k1] * B[k1][j1]; \\S1
5 for (i2 = 0; i < NI; i++)
6   for (j2 = 0; j < NJ; j++)
7       out[i2][j2] = tmp[i2][j2] + b[j2]; \\S2
```

Figure 4.9: Matrix-Matrix Multiplication followed by bias addition

```
1 for (i1 = 0; i < NI; i++)
2   for (j1 = 0; j < NJ; j++) {
3     for (k1 = 0; k < NK; ++k) {
4       tmp[i1][j1] += A[i1][k1] * B[k1][j1]; \\S1
5     }
6     out[i1][j1] = tmp[i1][j1] + b[j1]; \\S2
7   }
```

Figure 4.10: Fused matrix-matrix multiplication followed by bias addition

the dimension matching algorithm discussed in Mullapudi et. al. [24]. The dimension matching algorithm tries to match each dimension of the statements that are considered for fusion. Fusion is possible only if the matching is successful. Dimension matching fails if any of the dimension (considered for matching) carries a dependence. For the example, the dimension matching for Figure 4.9 is:

$$S1: [i1, j1, k1] \rightarrow [1, 2, 3]$$
$$S2: [i2, j2, -] \rightarrow [1, 2, -]$$

This means that the loop $i1$ is matched with $i2$ and similarly loop $j1$ is matched with loop $j2$. This satisfies the second dependence even when the loops are fused and also does not violate any other dependence. Hence, this is chosen as the final schedule. The final fused code is shown in Figure 4.10.

Another important optimization performed during this phase is that, when a group does not have communication free parallel loop, the heuristic finds a parallel loop at some inner level (if it exists) and permutes it to the outermost level. Performing this optimization does not violate any dependences because the optimization is applied to each group. These

groups are tilable and hence permutable. This optimization is useful in case of gemver benchmark. The benefits of this optimization is discussed in Chapter 5.

4.7 Summary

To summarize, the newly added constructs to PolyMage DSL enables programmers to concisely represent basic matrix operations. With enhancements to the fusion and tiling model to handle reduction operations, basic matrix operations like matrix-vector and matrix-matrix applications can be tiled and parallelized. Further, implementation of an intra-tile optimization pass enhanced PolyMage's ability to find vector friendly loops. PolyMage's tile size selection model is enabled to handle arbitrary affine accesses. The idiom recognition phase is now able to map matrix multiplications to optimized BLAS routines, whenever profitable. Finally, storage optimizations of PolyMage are performed and optimized C++ code is generated. The impact of these optimizations on performance is discussed in Chapter 5.

Chapter 5

Experimental Evaluation

This chapter describes the details of our experiments. Section 5.1 describes the experimental setup used to evaluate the benchmarks. Section 5.2 describes the benchmarks used for evaluation. Section 5.3 presents the experimental results and analysis of performance.

The baseline for comparison is PolyMage, with tile sizes obtained using the approach of Jangda and Bondhugula [19]. The baseline is compared against PolyMage with all the optimizations proposed in this thesis implemented (optimizations discussed in Chapter 4). The optimizations presented in this thesis for linear algebra computations are referred to as *PolyMage-opt-BLAS* or *PolyMage-opt-FFT* for arbitrary affine computations involving Fourier transforms in the rest of this section.

5.1 Experimental Setup

The experiments are run on a NUMA based, dual socket multicore system with Intel Xeon v3 (Haswell) processors. The complete details are provided in Table 5.1. Experiments are also run on a 64 core, 2.8GHz AMD Opteron X86 system with 4 sockets to show the architecture agnostic nature of the optimizations proposed in this work. The AMD Opteron system has 16KB L1d cache, 64KB L1i cache, 2MB L2 cache, and 6MB L3 cache.

For all experiments, five runs for each of the benchmarks were taken. The first two runs were discarded and the average of the next three was taken as the execution time

of the benchmark. PolyMage maps matrix operations to BLAS implementations provided

Table 5.1: System details

Processors	2-socket Intel	4-socket AMD
	Xeon E5-2630 v3	Opteron 6386 SE
Clock	2.40 GHz	2.80 GHz
Cores	16 (8 per socket)	32 (8 per socket)
Hyperthreading	disabled	disabled
Private caches	64 KB L1 cache	80 KB L1 cache
	512 KB L2 cache	2048 KB L2 cache
Shared cache	20,480 KB L3 cache	2,8 MB L3 cache
Memory	64 GB DDR4	128 GB DDR4
Matlab version	9.3.0.713579 (R2017b)	
Scipy version	1.0.0	
Compiler	Intel C/C++ (icc/icpc) 18.0.1	gcc 4.8.5
Compiler flags	-O3 -xhost -qopenmp	-O3 -fopenmp -march=native
	-fma -ipo	-ftree-vectorize -fPIC
OS	Linux kernel 3.10.0	Linux kernel 3.10.0
	64-bit CentOS 7.3	64-bit CentOS 7

by Intel's Math Kernel Library *(MKL, version 18.0.1) [23] and OpenBLAS for the AMD system.

Comparison of performance with Pluto [26] (version 0.11.4) and PPCG [28](version 0.07) which are state-of-the-art polyhedral auto-parallelizers is also performed. The flags `-tile` and `-parallel` were used with Pluto. PPCG was used to generate tiled, openmp-parallelized, C code using the flags `-target=c`, `-openmp` and `-tile`. Loop nests were tiled with default tile sizes (32 for Pluto and 16 for PPCG).

Matlab and Python's scipy libraries provide optimized implementations of some routines functions in the digital signal processing domain. Vendor-specific python 3.6 packages from Intel's distribution was used for python [14], which optimizes numpy and scipy routines using MKL. These routines were faster by a factor of 2× over default scipy versions that are available with standard Linux distributions.

---

*For 2 benchmarks (symm and trmm) code generated by PPCG failed to compile with the Intel compilers. Thus to keep the comparison fair gcc compiler v4.8.5 and OpenBLAS v0.2.20 is used for all configurations.

5.2 Benchmarks

The approach presented in this thesis is evaluated on benchmarks from a wide range of domains consisting of linear algebra computations (13 benchmarks from PolyBench [27] suite), Digital Signal Processing (2 filters) and image processing benchmarks (6 benchmarks from PolyMage [24]).

5.2.1 PolyBench

Table 5.2: PolyBench benchmarks used in evaluation

Benchmark	Description
2mm	2 Matrix Multiplications
3mm	3 Matrix Multiplications
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
doitgen	Multi-resolution analysis kernel (MADNESS)
gemm	Matrix-multiply
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
mvt	Matrix Vector Product and Transpose
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trmm	Triangular matrix-multiply

The PolyBench [27] is a benchmark suite of numerical computations which have been extracted from applications of various domains. For evaluation, the basic linear algebra computations are used. The computation names of the benchmarks and their descriptions are shown in Table 5.2.

5.2.2 Digital Signal Processing

Two filters - *unwanted spectral* and *vuvuzela* are used as benchmarks from the Digital Signal Processing(DSP) domain. The *unwanted spectral* filter uses a low pass filter to remove noise in the input signal. The *vuvuzela filter* is used to filter out the *vuvuzela* noise from the

input signal. These filters have reduction operators with arbitrary affine accesses in their loop nests as shown in Figure 4.5. It is hard to represent these routines as basic matrix operations and hence domain experts rely on optimized implementations from vendors like Matlab (with parallel computing support) and Intel's Scipy for high performant implementations. Performance of PolyMage-opt-FFT with these optimized libraries is compared for DSP benchmarks.

5.2.3   Image Processing

For image processing applications, Jangda and Bondhugula [19] propose several optimizations which are implemented in PolyMage. Hence for applications in image processing domain their approach serves as a strong baseline for comparison with the tile size selection model in our approach. Six image processing pipelines are picked, namely Unsharp Mask, Harris Corner, Bilateral Grid Multiscale Interpolate, Camera Pipeline, and Pyramid Blend. Details regarding these benchmarks are available at [19].

5.3   Performance Analysis

This section describes the performance of the benchmarks written in PolyMage. It also analyzes the benefits of optimizations discussed in Chapter 4. Comparison with state-of-the-art approaches/libraries in respective domains is also discussed.

5.3.1   Linear Algebra Benchmarks

The execution times for linear algebra benchmarks from PolyBench suite for our complete approach - PolyMage-opt-BLAS (discussed in Chapter 4), and that of Pluto, PPCG, baseline PolyMage, and BLAS configurations are listed in Table 5.3 and Table 5.4 for the *small* and *extralarge* datasets respectively. The tables present the execution times for 1 thread and 16 threads for each configuration. For BLAS configuration (Columns 7 and 8) all basic matrix operations are mapped to optimized BLAS routines in PolyMage, ignoring the

Table 5.3: Execution times (in ms) with *small* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	0.558	0.095	0.922	0.441	0.914	0.913	0.219	0.081	0.259	0.043
gemver	0.122	0.034	0.178	0.062	0.053	0.022	0.048	0.049	0.059	0.109
gesummv	0.104	0.040	0.094	0.042	0.011	0.010	0.012	0.012	0.015	0.021
symm	0.431	0.391	1.165	0.984	1.109	1.120	0.063	0.069	0.262	0.212
syr2k	0.600	0.116	0.616	0.230	0.682	0.675	0.236	0.349	0.427	0.153
syrk	0.413	0.076	0.411	0.147	0.340	0.333	0.267	0.201	0.393	0.099
trmm	0.324	0.106	0.746	0.367	0.797	0.814	0.054	0.050	0.129	0.142
2mm	0.532	0.140	0.823	0.609	0.724	0.735	0.220	0.122	0.205	0.094
3mm	1.126	0.267	1.293	0.974	1.286	1.317	0.379	0.168	0.365	0.148
atax	0.104	0.033	0.109	0.046	0.020	0.015	0.015	0.019	0.027	0.022
bicg	0.113	0.039	0.111	0.046	0.023	0.016	0.013	0.017	0.024	0.021
doitgen	1.322	4.088	1.300	3.857	0.930	0.948	-	-	0.224	0.070
mvt	0.136	0.026	0.106	0.043	0.018	0.015	0.017	0.016	0.021	0.022

heuristic mentioned in Section 4.4. In the specific case of the doitgen benchmark, we do not map to BLAS calls as it involves multiplications of three dimensional matrices and the corresponding entries for this are marked with a '-'.

Figure 5.2 also plots the speedups for the extralarge dataset for each of these configurations over the baseline PolyMage.

All linear algebra benchmarks from PolyBench suite involve reduction operations and hence the baseline PolyMage does not perform any optimizations that are described in Chapter 4. Hence, with PolyMage-opt-BLAS observes a geometric mean speedup of 2.6× and 21.7× over PolyMage for *small* as well as *extralarge* datasets respectively for 16 threads
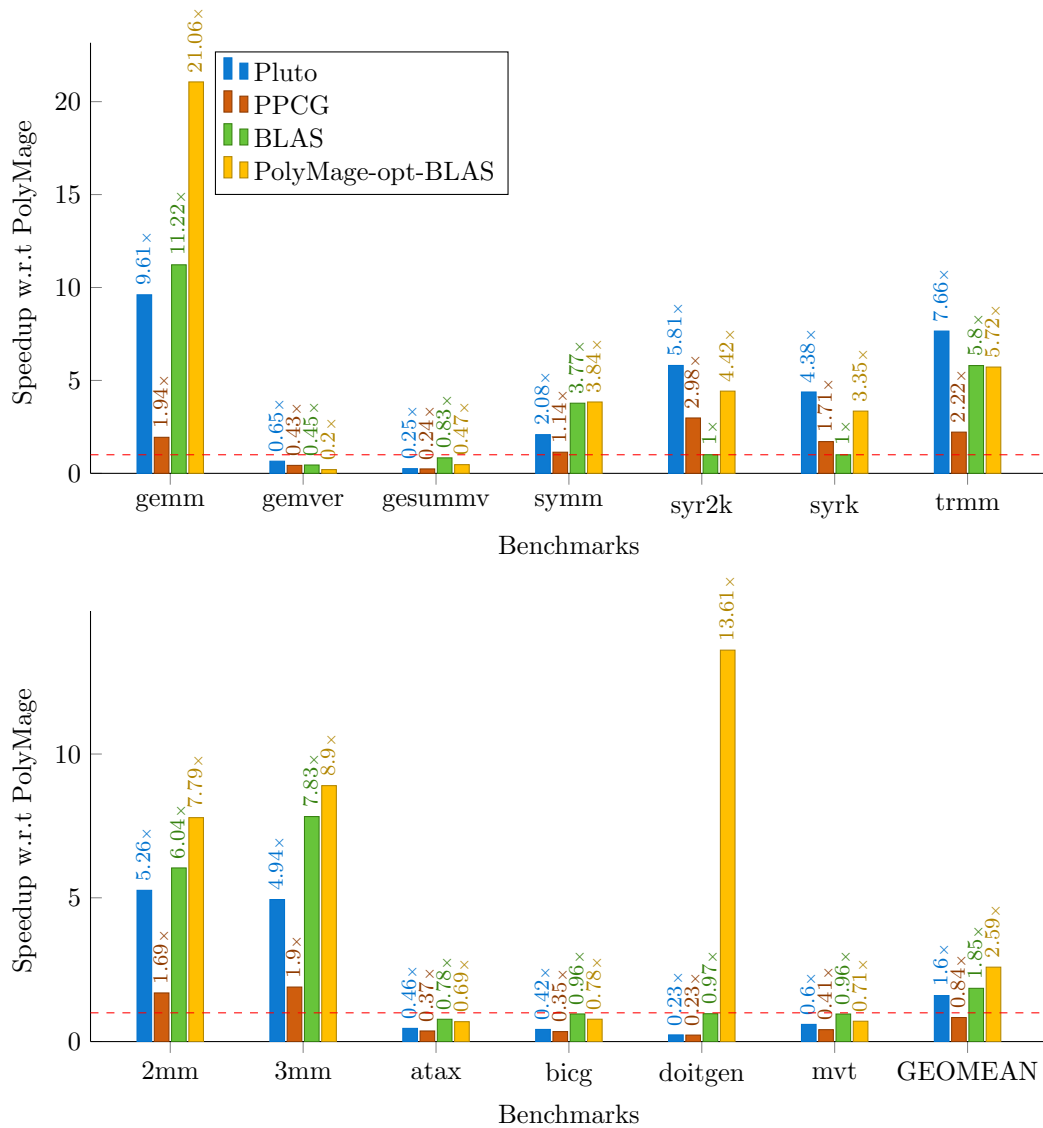
Figure 5.1: Speedup over PolyMage for LA benchmarks for small dataset with 16 threads
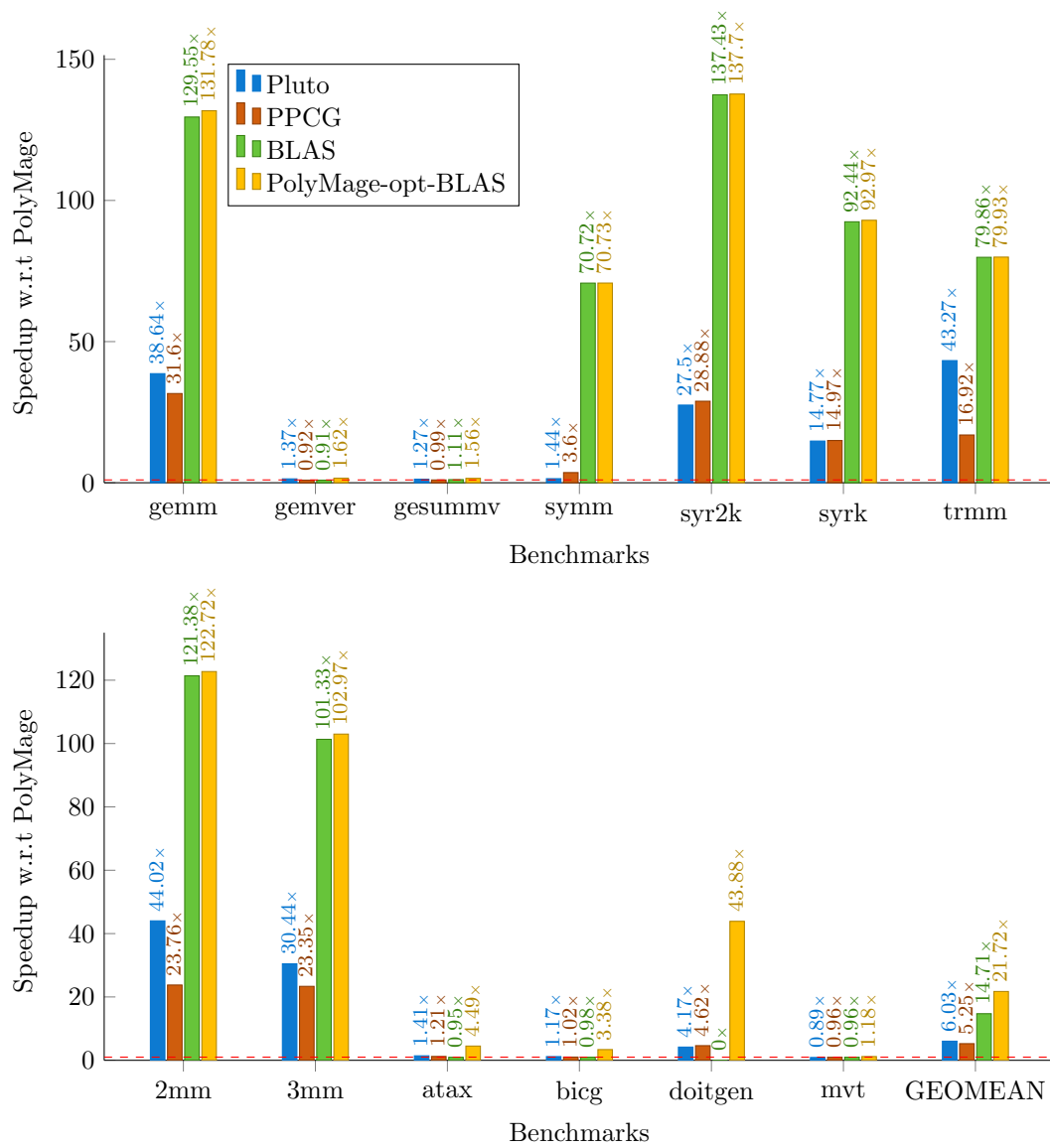
Figure 5.2: Speedup over PolyMage for LA benchmarks for extralarge dataset with 16 threads

Table 5.4: Execution times (in s) with *extralarge* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	9.541	0.784	12.09	0.958	30.10	30.29	2.459	0.234	2.462	0.230
gemver	0.165	0.031	0.356	0.046	0.053	0.043	0.051	0.047	0.054	0.026
gesummv	0.048	0.011	0.102	0.014	0.014	0.014	0.013	0.013	0.017	0.009
symm	24.36	24.13	22.96	9.686	34.93	34.86	0.503	0.493	0.503	0.493
syr2k	9.101	1.342	9.051	1.278	36.78	36.90	1.618	0.269	1.618	0.268
syrk	6.150	0.907	6.308	0.895	13.35	13.39	2.973	0.145	2.973	0.144
trmm	5.455	0.491	14.00	1.255	21.12	21.24	0.311	0.266	0.311	0.266
2mm	9.503	0.751	13.96	1.391	32.91	33.06	2.740	0.272	2.741	0.269
3mm	18.43	1.458	20.34	1.901	44.30	44.39	4.457	0.438	4.460	0.431
atax	0.031	0.005	0.052	0.006	0.007	0.007	0.008	0.008	0.009	0.002
bicg	0.031	0.005	0.052	0.006	0.007	0.006	0.007	0.006	0.007	0.002
doitgen	3.124	1.030	3.981	0.931	4.366	4.299	-	-	0.894	0.098
mvt	0.229	0.026	0.171	0.024	0.028	0.023	0.027	0.024	0.030	0.020

(maximum of 21× and 137× for small and extralarge datasets). For small datasets (Table 5.3), PolyMage-opt-BLAS does not invoke routines from BLAS libraries. Instead, matrix operations are tiled and parallelized using openmp. Even though both Pluto and PPCG perform the similar optimizations, both of them lack a tile size selection model. Apart from good tile sizes, when the loop bounds for a particular dimension are very small, PolyMage-opt-BLAS chooses not to tile the loop nest. Hence, there is no loop tiling overhead in such small cases. Therefore, an improvement of 60% over Pluto and 3.17× over PPCG for small datasets with 16 threads (Table 5.3) is observed. For the small dataset, in the case of gemver a slowdown of 70% over Pluto for 16 threads is observed. This is because (i) Pluto finds a different loop permutation and hence a better fusion that exploits register reuse

(ii) The execution times of PolyMage-opt-BLAS includes the time spent in allocation and initialization of matrices which are not measured in Pluto. This overhead appears to be significant in cases where the execution times are very small. For the same reason, in the case of syr2k, syrk, and trmm a slowdown of about 25% over Pluto for small datasets is observed.

Our heuristic described in Section 4.4 maps matrix-matrix multiplications for large and extralarge sizes to Intel MKL's implementation of BLAS. Hence in case of extralarge large datasets with 16 threads (Table 5.4), PolyMage-opt-BLAS provides a geomean improvement of 3.6× and 4.1× over Pluto and PPCG respectively.

Further, for extralarge datasets with 16 threads, a geomean performance improvement of 39% over Intel MKL is observed. This improvement is observed in case of gemver, gesummv, atax, bicg, and mvt, significant performance improvements ranging from 77% (gemver) to 4.7× (atax) over MKL is observed. This is due to two reasons: (i) Despite the extralarge dataset, the computations involved are matrix-vector multiplications and the dataset is still small to be profitably mapped to BLAS routines. Hence, our heuristic in Polymage-opt-BLAS does not call routines from Intel MKL; instead it performs rectangular tiling and parallelization of the loop nest. (ii) In case of gemver, there are other matrix-vector operations, that can not be mapped to BLAS routines. However, PolyMage-opt-BLAS performs tiling and parallelization of those routines as well, thus giving significant performance improvements over Intel MKL.

For symm and doitgen, loop carried WAR dependences on temporary variables prevents Pluto from tiling and parallelizing the loop nests. However, in our approach these dependences do not exist as these temporary variables inside a loop nest are represented using arrays whose dimensionality is equal to the nesting depth of the statement containing the definition of the scalar variable. Hence for these benchmarks, significant performance improvement of 49× and 10.9× is noticed, for symm and doitgen respectively, over Pluto. In case of gemver, for extralarge datasets an improvement of 18% over Pluto is noticed. Here, the improvement is due to better tile sizes found for each group by PolyMage-opt-BLAS.

Tables 5.5, 5.6 and 5.7 show the execution times of *mini, medium* and *large* datasets

Table 5.5: Execution times (in ms) with *mini* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	0.037	0.023	0.048	0.052	0.037	0.037	0.022	0.021	0.012	0.012
gemver	0.022	0.016	0.029	0.020	0.007	0.006	0.006	0.006	0.018	0.089
gesummv	0.021	0.024	0.017	0.015	0.001	0.002	0.002	0.003	0.006	0.017
symm	0.022	0.018	0.042	0.039	0.036	0.036	0.007	0.009	0.015	0.029
syr2k	0.037	0.019	0.028	0.033	0.038	0.034	0.023	0.037	0.036	0.015
syrk	0.024	0.020	0.015	0.027	0.022	0.023	0.022	0.031	0.032	0.015
trmm	0.027	0.053	0.032	0.028	0.023	0.023	0.003	0.008	0.009	0.021
2mm	0.046	0.046	0.044	0.030	0.034	0.039	0.019	0.048	0.012	0.021
3mm	0.074	0.058	0.060	0.045	0.044	0.049	0.029	0.060	0.018	0.038
atax	0.020	0.019	0.018	0.013	0.003	0.001	0.002	0.001	0.014	0.062
bicg	0.024	0.021	0.018	0.020	0.002	0.002	0.003	0.002	0.015	0.086
doitgen	0.118	0.670	0.084	0.540	0.011	0.022	-	-	0.019	0.009
mvt	0.020	0.009	0.018	0.016	0.002	0.002	0.003	0.002	0.024	0.135

respectively. PolyMage-opt-BLAS performs better than state-of-the-art consistently across all dataset sizes. Thus the optimizations approaches presented in this work are extremely robust and show good weak scaling properties as well.

The results of the experiments run on the AMD Opteron machine with *extralarge* dataset are presented in Table 5.8. PolyMage-opt-BLAS is architecture agnostic and shows good improvement on this system as well for the *extralarge* dataset over state-of-the-art. Similar performance improvement trends were obtained on the *mini, small, medium* and *large* datasets.

The ability of our approach to utilize optimized BLAS routines when the size of the matrices are large, or perform polyhedral optimizations for smaller problem sizes allows

Table 5.6: Execution times (in ms) with *medium* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	14.17	1.283	19.71	4.028	20.38	20.68	5.318	0.552	6.089	0.481
gemver	1.915	0.235	1.402	0.326	0.427	0.448	0.448	0.448	0.504	0.506
gesummv	0.652	0.123	0.789	0.185	0.089	0.086	0.091	0.099	0.081	0.069
symm	31.03	18.82	41.68	16.05	25.51	26.25	1.284	1.299	6.475	2.947
syr2k	14.55	1.945	12.08	3.622	16.16	16.28	4.161	1.128	9.894	1.545
syrk	10.42	1.399	10.00	2.777	8.161	7.576	5.971	0.648	8.166	1.346
trmm	10.05	1.152	27.45	3.943	22.07	21.45	1.005	1.049	3.799	3.816
2mm	21.98	2.235	27.76	6.090	25.17	24.09	7.100	1.140	7.015	0.749
3mm	38.01	3.692	30.58	8.463	33.12	33.38	10.06	1.473	9.705	1.113
atax	0.950	0.235	0.895	0.209	0.208	0.208	0.200	0.211	0.236	0.138
bicg	0.927	0.235	0.879	0.255	0.201	0.206	0.201	0.206	0.239	0.159
doitgen	11.38	15.19	12.65	15.92	11.36	12.25	-	-	4.268	0.522
mvt	2.065	0.222	1.192	0.203	0.186	0.191	0.215	0.227	0.244	0.135

us to incorporate the best of both world.

Comparison with Polly [18]

Polly [18] is a loop and data-locality optimization framework for LLVM. It uses a mathematical representation based on integer polyhedra to analyze and optimize the memory access pattern of a program with the help of ISL [38].

Table 5.9 and Table 5.10 present execution times for Polly and PolyMage-opt-BLAS on Intel and AMD processors respectively. Polymage-opt-BLAS provides an average improvement of 4.61× on the Intel Xeon machine and 1.59× on the AMD Opteron machine for a 16-threads execution over Polly. This performance benefit of PolyMage-opt-BLAS over

Table 5.7: Execution times (in s) with *large* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	0.812	0.094	1.303	0.132	1.618	1.618	0.275	0.030	0.274	0.030
gemver	0.050	0.005	0.094	0.008	0.014	0.010	0.014	0.012	0.016	0.007
gesummv	0.010	0.003	0.034	0.003	0.003	0.003	0.003	0.003	0.003	0.000
symm	1.991	1.940	2.819	1.259	3.262	3.267	0.058	0.058	0.058	0.058
syr2k	0.922	0.143	0.913	0.119	1.544	1.516	0.190	0.033	0.190	0.033
syrk	0.649	0.101	0.660	0.088	0.676	0.713	0.313	0.022	0.313	0.022
trmm	0.776	0.062	1.798	0.174	2.020	2.015	0.045	0.046	0.045	0.046
2mm	1.137	0.127	1.640	0.195	1.921	2.004	0.348	0.038	0.347	0.038
3mm	2.041	0.215	2.433	0.266	2.883	2.984	0.563	0.062	0.563	0.062
atax	0.031	0.005	0.053	0.006	0.008	0.007	0.007	0.007	0.009	0.002
bicg	0.031	0.005	0.053	0.006	0.007	0.006	0.006	0.007	0.008	0.002
doitgen	0.406	0.228	0.607	0.262	0.513	0.557	-	-	0.146	0.016
mvt	0.052	0.006	0.044	0.006	0.008	0.007	0.007	0.005	0.007	0.002

Polly can be attributed to the tile size selection model. Polly uses default tile sizes for all the loop nests while PolyMage-opt-BLAS empirically chooses an optimal tile size as discussed in Chapter 4. Secondly, while Polymage-opt-BLAS maps some functions to BLAS routines, Polly does not perform this optimization. Benchmarks like syr2k and syrk show performance improvement on Intel machine whereas there is a slight slow down on AMD, in comparison with Polly. This is caused by using different BLAS libraries on Intel (Intel MKL) and AMD machines (OpenBLAS).

Table 5.8: Execution times (in s) with *extralarge* input dataset for linear algebra benchmarks from PolyBench suite on AMD Opteron

Benchmark	Pluto		PPCG		PolyMage		BLAS		PolyMage-opt-BLAS	
	1	16	1	16	1	16	1	16	1	16
gemm	22.28	1.536	78.71	5.118	94.82	94.66	1.300	1.360	1.315	1.351
gemver	0.215	0.055	0.157	0.047	0.115	0.114	0.110	0.082	0.109	0.082
gesummv	0.105	0.021	0.073	0.012	0.045	0.044	0.042	0.021	0.042	0.021
symm	67.77	84.52	44.10	12.30	63.13	63.56	7.996	4.783	8.032	4.755
syr2k	45.81	5.255	45.31	5.319	104.0	104.0	20.24	2.858	20.27	2.875
syrk	43.45	4.955	43.50	4.962	49.76	49.68	19.47	2.400	19.47	2.401
trmm	11.27	0.874	33.93	2.527	44.93	53.70	3.947	0.629	3.960	0.630
2mm	50.28	4.116	87.06	7.006	100.1	102.3	1.489	1.456	1.511	1.493
3mm	107.7	8.193	139.3	10.72	136.4	136.2	2.346	2.371	2.380	2.444
atax	0.042	0.009	0.025	0.006	0.025	0.026	0.023	0.009	0.023	0.009
bicg	0.042	0.009	0.026	0.006	0.025	0.025	0.023	0.005	0.023	0.009
doitgen	6.391	2.766	12.04	2.556	10.63	10.62	-	-	2.832	0.381
mvt	0.213	0.033	0.106	0.026	0.079	0.080	0.074	0.047	0.074	0.047

Comparison with Mehta et al. [22]

Mehta et al.[22] proposes an analytical model for tile size selection which leverages the associativity available in modern caches. The model chooses tile sizes which benefits vectorization and reuse along multiple levels of the cache.

Table 5.9 and Table 5.10 present execution times for this model on Intel Xeon and AMD Opteron machines respectively. Our approach, Polymage-opt-BLAS provides an average improvement of 4.39× on Intel Xeon machine and 2.1× on AMD machine for 16-thread execution. This performance gain can be attributed to the fact that the model proposed in Mehta et al. generates one set of tile sizes for the entire benchmark, thereby missing on the performance gain of having different tile sizes for different loop nests.

Table 5.9: Execution times (in s) with *extralarge* input dataset for linear algebra benchmarks from PolyBench suite on Intel Xeon

Benchmark	PolyMage-opt-BLAS		Polly		Mehta et al.	
	1	16	1	16	1	16
gemm	2.462	0.230	3.016	3.002	2.536	0.404
gemver	0.054	0.026	0.131	0.026	0.090	0.037
gesummv	0.017	0.009	0.049	0.008	0.042	0.013
symm	0.503	0.493	13.57	9.289	24.29	25.65
syr2k	1.618	0.268	8.636	0.811	8.922	1.564
syrk	2.973	0.144	5.265	0.538	6.168	1.107
trmm	0.311	0.266	10.78	7.862	3.544	0.481
2mm	2.741	0.269	2.844	2.942	8.157	1.416
3mm	4.460	0.431	3.971	3.830	15.39	2.480
atax	0.009	0.002	0.018	0.008	0.017	0.008
bicg	0.007	0.002	0.017	0.004	0.018	0.007
doitgen	0.894	0.098	3.494	0.784	1.302	1.616
mvt	0.030	0.020	0.080	0.015	0.144	0.030

5.3.2 DSP Filters Benchmarks

Figure 5.3 provides the execution times for vuvuzela (Figure 5.3a) and unwanted spectral filters (Figure 5.3b). Both these filters have reduction operations for which PolyMage does not do any tiling or parallelization. PolyMage is modified to map up-sampling operations in these filters to optimized FFT routines in the `fftw3` library (referred to as PolyMage-FFT). These filters involve significant number of reductions and hence just optimizing upsampling operations with fftw library does not scale with the increase in number of cores. PolyMage-opt-FFT has the ability to optimize reduction operations thus providing an improvement of 7.68× over PolyMage-FFT. The scipy library from Intel's python distribution

Table 5.10: Execution times (in s) with *extralarge* input dataset for linear algebra benchmarks from PolyBench suite on AMD Opteron

Benchmark	PolyMage-opt-BLAS		Polly		Mehta et al.	
	1	16	1	16	1	16
gemm	1.315	1.351	4.622	4.618	16.67	2.931
gemver	0.109	0.082	0.127	0.039	0.175	0.038
gesummv	0.042	0.021	0.024	0.011	0.099	0.016
symm	8.032	4.755	26.98	16.28	36.75	36.86
syr2k	20.27	2.875	15.84	1.406	46.57	7.628
syrk	19.47	2.401	11.41	0.874	44.37	7.166
trmm	3.960	0.630	24.18	26.68	6.663	0.861
2mm	1.511	1.493	5.014	5.015	50.87	7.339
3mm	2.380	2.444	8.170	8.159	108.7	14.24
atax	0.023	0.009	0.021	0.012	0.033	0.009
bicg	0.023	0.009	0.017	0.006	0.034	0.007
doitgen	2.832	0.381	7.325	1.699	5.226	6.345
mvt	0.074	0.047	0.096	0.021	0.188	0.032

provides optimized routines required to build these filters. We observe that, among the implementations of the filters, Intel's scipy provides best sequential performance. However, as shown in Figures 5.3a and 5.3b, the implementation of filters using Intel's scipy does not scale well with the number of cores. Matlab with parallel computing support, provides an improvement of 7.2× over PolyMage-FFT. PolyMage-opt-FFT provides an improvement of 5.1× over scipy and 1.8× over Matlab. Overall, our approach PolyMage-opt-FFT provides better performance than state-of-art along with good thread scalability.
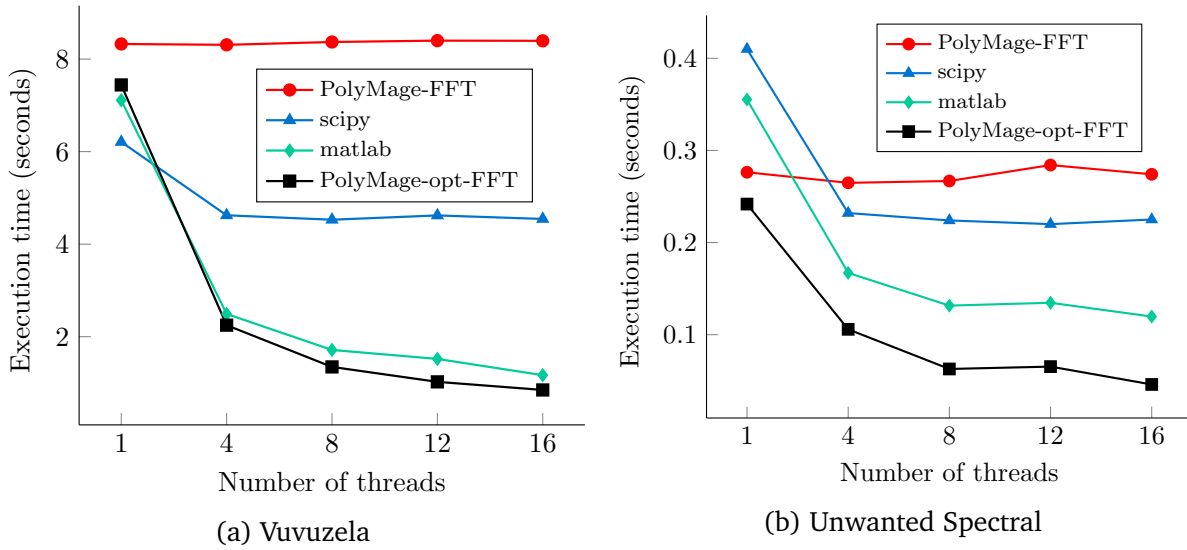
Figure 5.3: Execution time for DSP benchmarks and scaling across cores

5.3.3 Image Processing benchmarks

To recall, the baseline PolyMage's tile size selection for image processing applications is based on a dynamic programming model [19]. Since both PolyMage and PolyMage-opt perform same optimizations on these benchmarks, the objective here is to compare the tile size selection routines. In most of the cases PolyMage-opt finds the same tile sizes as PolyMage. In cases where different tile sizes were found, the difference of tile sizes (per dimension) were very small (±2). Therefore, for image processing benchmarks (Unsharp Mask, Harris Corner, Bilateral Grid, Multiscale Interpolate, Camera pipeline and pyramid blend) PolyMage-opt performs within ±4% of baseline PolyMage performance which is within experimental bounds. Thus, the proposed tile size selection model (along with additional optimizations for reduction operations) does not introduce any performance regression for image processing pipelines.

5.4 Summary

To summarize, our optimizations described in Chapter 4, provides an aggregate performance improvement of 8.1× over PolyMage (16 threads) across dataset sizes small and

extralarge in PolyBench. Similarly, an aggregate speedup of 2.5× over Pluto as the proposed approach finds better tile sizes and also map matrix operations to BLAS library calls. An overall speedup of 3.8× over PPCG is obtained. For DSP benchmarks, the proposed approach performs better than Matlab and Intel's scipy by 89% and 5.12× respectively. The results also prove that the optimization approach is comprehensive as it accomplishes to get good performance across all dataset sizes and is able to robustly out perform state-of-the-art approaches. Further, the chapter also showed that the architecture agnostic approach of this work provides good performance on any modern X86 system. Lastly, the proposed tile size selection model is iso-performant with the tile size selection models proposed by Jangda and Bondhugula [19] on benchmarks from the image processing domain.

Chapter 6

Related Work

There has been a stream of work in the area of optimizing linear algebra computations. This section discusses the relevant state-of-art works and how our approach differs from these and builds on their shortcomings.

6.1   LGen

LGen [21, 35] is a recent domain-specific language that produces performance-optimized basic linear algebra computations on small matrices of fixed sizes. LGen accepts inputs in a language called Linear Algebra Language (LL) along with an optional parameter v (vector length of the ISA). Firstly, the input is parsed into an expression graph and the compiler chooses a tiling strategy and annotates the computations with tile sizes. Next, the LL expression is converted into another language called $\sum$-LL, that makes access patterns and loops explicit. $\sum$-LL is based on $\sum$-SPL that is used in Spiral[30, 29]. The compiler performs tiling, loop merging, loop exchange, loop unrolling, scalar replacement and conversion to SSA by generating a C-intermediate representation. Finally, a C function is generated and its performance is used to auto tune. In case vectorized code is required, a small set of pre-defined codelets is used to generate vector intrinsics.

LGen performs well for small matrix sizes since it performs register tiling and generates

vector intrinsic code. It is also easy for LGen to port to a new vector machine by implementing only the basic codelets for that machine. However, LGen was designed very specifically for small matrices and does not scale well for large matrices. The work in this thesis is not limited to linear algebra computations but can also generate efficient code for other arbitrary affine accesses.

6.2   High Performance Libraries

OpenBLAS [40] and MKL [23] provide high performance linear algebra computations by hand-optimizing them for specific hardware. Open source libraries like Eigen [9], in addition to their optimizations, rely on these hand optimized implementations for high performance. There have also been many attempts to automate these optimizations, which include LAPACK [1], PHiPAC [3] and ATLAS [41]. They iteratively tune for performance by varying, block sizes, loop orders and also use runtime feedback mechanism for autotuning.

All these libraries provide excellent performance for large problem sizes but their performance can be sub-optimal for small matrices as shown by the results in the work of Hall et. al [32]. Some linear algebra computations may not match with any of the BLAS interfaces, and hence, Intel released the Integrated Performance Primitives (IPP) [17] which includes optimizations for small size matrices. Similar OpenBLAS or MKL, these libraries also miss out on the opportunity of optimizing across sub-routine calls to improve input data reuse. The work in this thesis targets to exploit this observation to obtain improved performance.

6.3   FLAME

FLAME [12] is a DSL to formally express complex linear algebra applications (Eg: LU, Cholesky decomposition). The language is based on the assumption that most complex linear algebra computations can be performed using algorithms that view matrices as a collection of sub-matrices (or blocks). The operations on these blocks can be mapped to

BLAS libraries. The Flame compiler selects the data layout for these blocks and relies on BLAS libraries like Intel MKL or OpenBLAS for high performance. *Libflame* [44, 5] overcomes the limitations of FLAME for small matrices by providing a runtime that supports task parallelism. Dependences between blocks are computed at runtime and the blocks that are ready to be executed are assigned to threads in a two dimensional block-cyclic manner. A thread then performs all tasks that write to a particular block. Basic operations on these blocks are mapped to BLAS libraries which are executed serially.

All these libraries [12, 44, 5] require the programmer to specify the decomposition. Secondly, these libraries rely on auto-tuning to determine the block sizes i.e. sizes of the sub-matrices. Incorporating these matrix decompositions in PolyMage is deferred to future work. In the case of matrix decompositions, like LU decomposition, addressing load balancing issues is extremely essential to obtain high performance. This requires compiler assisted runtime techniques like [8], to be incorporated in PolyMage which is also deferred to future work.

6.4   Build to Order BLAS (BTO)

Build to Order BLAS (BTO) [34] is a domain-specific compiler which optimizes for loop fusion, data partitioning, and parallelism. The compiler is divided into four phases - Analysis, Refinement, Optimization ,and Code generation. During the analysis phase, the compiler generates a data-flow graph based on the operation and data type specified in the computation. The type of storage and implementation for each node in the graph is decided at this stage of the compiler. BTO stores information on how to implement basic linear algebra computation in a separate database called the linear algebra database. The compiler then chooses a matching implementation for a node from this database and adds this information to the dataflow graph. The refinement phase of the compiler further breaks down the nodes into sub-graphs, based on the implementation chosen in the Analysis phase. Next, the Optimization phase performs merging of the sub-graphs when they have a common operand but do not depend on one another or if the iteration strategies of the two graphs

are compatible. These optimizations improve the cache locality of the computation. BTO generates C code after performing these optimizations. It is up to the general purpose compilers to perform vectorization for the code. Currently, BTO supports sparse and dense computation for both single and double precision matrices. The matrices can be stored in row-major or column-major format. BTO also supports for both floating point operations and complex numbers.

BTO also tries to perform optimizations across different operations but they do not map to library calls for large matrices. Hence, losing the performance provided by these optimized libraries. The work in this thesis not only takes advantage of the optimized linear algebra libraries but also is able to able to map to these libraries calls automatically (if profitable) using Idiom recognition.

6.5   Diesel

Diesel [10] is a domain specific language for Linear Algebra and Neural Net computations on GPUs. This DSL accepts the input in an intuitive form and generates high performant CUDA code to be run on a GPU. They extract the polyhedral representation of the input and develop an initial schedule using ISL [38]. This schedule is then used to tile the program. The first set of parallel-tile loops are distributed among thread blocks and the subsequent set of parallel tile loops are distributed among warps and lanes. The DSL also performs memory optimizations to overlap memory and compute optimizations, thus hiding the data-transfer latency.

Diesel performs many GPU specific optimizations which may not be applicable in the case of multi-core CPU architectures, which is the primary target of this thesis. The tiling strategy also uses default tile sizes which are then tuned to obtain the best performing code. The work in this thesis eschews tile size tuning and employs a heuristic to decide best tile sizes. Secondly, Diesel always generates polyhedral schedules and does not map to any optimized library calls when there is no reuse across functions, thereby losing out on some performance benefits. The thesis proposes optimizations to map to optimized linear

algebra libraries and target to exploit cache locality when there is reuse across functions.

6.6 Tensor Comprehensions

Tensor Comprehensions [36] is another recent domain specific language for optimizing matrix operations on GPUs, specifically for Deep Neural Network domains. The framework uses a just-in-time polyhedral compiler to perform optimizations like loop fusion and tiling. It uses compilation caches to determine tile sizes and loop fusion scheduling strategy. Compilation caches store the best performant code for all combinations of input size, optimization options, and architectures. The best performing code is obtained by an auto-tuner which uses a genetic algorithm. The auto-tuner runs for a given period of time and updates the cache with better versions of the CUDA program.

Tensor Comprehension also relies on an auto tuning based mechanism to find the best performing code and hence the best performing tile size. However, the tile size selection model proposed in this Section 4.3 can be used to empirically determine the tile size for a given architecture.

6.7 Polyhedral Source-to-source tools

Another approach to obtaining fast linear algebra code is to use polyhedral source-to-source tools like Pluto [4], PPCG [39] and ISL scheduler [45]. These compilers perform automatic parallelization and locality optimization for affine loop nests. They perform tiling, vectorization and various loop optimizations automatically for regular programs with imperfectly nested loops. However, the scope of these compilers is broader and a domain-specific linear algebra compiler allows for more specific optimizations as well as improves productivity. Section 5.3 compared the experimental results against optimized code from Pluto and PPCG and the approach proposed in this thesis is able to obtain better performance than them.

6.8   Detection of Linear Algebra Operations in Polyhedral Programs

Another relevant work is the work of Iooss et.al. [16]. They recognize dense linear algebra computation by partitioning the computations into blocks. They use a method called "mono-parametric tiling" to create sub-blocks, and use a template recognition algorithm, similar to Barthou et al. [2] to find if the program fits a pattern.

This work uses C-code as input and uses a template recognition algorithm to find linear algebra computations and replace them with BLAS calls. The work presented in this thesis focuses on mapping to BLAS library calls only when profitable. In addition, to the improved productivity of expressing computations in the PolyMage DSL, the idiom recognition phase is simpler as the specification comes from the DSL.

6.9   Tile Size Selection Models

The closest approach to ours on tile size selection models was recently proposed by Jangda and Bondhugula [19]. The limitations of extending their approach to basic matrix computations were discussed in Section 3.2. The works on tile size selection models can be broadly classified as works which use an analytical model [22, 20, 6, 11, 31] to select tile sizes and ones which perform an extensive search to determine either good tile sizes or tile size models [43, 33].

Mehta et al.[22] proposes an analytical model for tile size selection which leverages the associativity available in modern caches. The model chooses tile sizes which benefits vectorization and reuse along multiple levels of the cache. However, the tile size selection is performed only after the final fused schedule is provided by the Pluto[26] compiler whereas, in our approach, fusion and tiling are tightly coupled. Also, the model generates one set of tile sizes for the entire program, thereby missing on the performance gain of having different tile sizes for different loop nests inside with an input program. Section 5.3 compared the experimental results against this model.

The other analytical models proposed are less powerful because of the following draw-backs:

- they assume a direct mapped cache while predicting the tile sizes, whereas modern architectures have moved towards associative caches [6],

- their model does not consider some important parameters like dataset sizes and reuse [11] and

- they do not account for modern architectural changes like prefetching and vectorization as a part of the model [20, 31]

These models also assume that the fusion of the computation is already available before the tile size calculation. These models do not ensure that the threads have enough work to do and the number of threads is at least equal to the number of cores available.

Yuki et al.[43] proposes a model which generates tile size selection models using program features, synthetic programs, and machine learning techniques. They use neural networks to predict a tile size selection model. They also generate synthetic programs to feed as input to this neural network. They show performance benefits on different architectures and compiler. The proposed tile size selection model is for one level of tiling which when extended to many dimensions increases the search space, and the scalability of this approach is yet to be investigated. They also assume that loop fusion has already been performed.

Shirako et al. [33], identifies bounds on the search space of tile sizes. They rely on an auto-tuning approach for finding the best tile size within these bounds. However, the search space although smaller than many other cases is still significantly large and hence the process of finding best tile sizes is time-consuming. Again, they tune for tile size for the entire computation and hence cannot predict different tile sizes for two different loop nests in a program.

Chapter 7

Conclusions and Future Work

This chapter summarizes the thesis and discusses potential avenues for future work.

7.1   Conclusions

Linear algebra computations and arbitrary affine accesses are prevalent in many domains including scientific computing, digital signal processing, and deep learning. These *computation primitives* have been proposed to be optimized in several prior works. The work in this thesis examined these approaches and is motivated by several limitations posed by them. Though there exist several *high performance libraries* tuned for a specific architecture, they are hard to program due to large parameter lists and do not perform well on all dataset sizes. While *polyhedral source-to-source tools* are unduly generic, certain *DSLs* overcome this but they place restrictions on what can be specified. This thesis identified specific limitations of these approaches and proposes novel solutions to overcome these drawbacks. Several aspects such as extracting parallelism, vector hardware performance, exploiting locality in caches and using existing high performance library implementations where applicable was looked at comprehensively to study limitations of state-of-the-art works. The thesis proposed to improve programmer productivity for expressing the computation primitives by using *the DSL approach*. The optimizations are thus chosen to be implemented in the compiler of the PolyMage DSL.

The DSL approach required enhancing the PolyMage's specification by adding new constructs for matrices and supporting basic matrix operations like addition, multiplication and transposes. Along with this newly added *language specification*, PolyMage's *fusion and tiling strategy* was enhanced to support reduction operations. The new *tile size selection model* which works for all affine accesses was implemented in PolyMage. This technique provided significant performance gains for application in digital signal processing domain. *Automatic idiom recognition* to map basic matrix operations to optimized library calls whenever profitable was added in PolyMage. An *intra-tile optimization* technique to improve auto-vectorization was added. Finally, for improving the performance of reduction operations the dynamic programming based model was extended to *fusion and tiling for reduction operations* as well.

The thesis experimentally evaluates the proposed approach with representative benchmarks from linear algebra, digital signal processing, and image processing. Experiments are carried out on the latest multi-core x86 hardware from Intel and AMD and evaluated over a range of dataset sizes to demonstrate the robustness of the proposed approach. The code generated by the proposed approach performs better than state-of-the-art, showing a mean performance improvement of 3.6× over Pluto, 4.1× over PPCG and 21.7× over the existing PolyMage compiler for linear algebra benchmarks from PolyBench suite. In the case of digital signal processing benchmarks, a mean speedup of 5.1× over Intel's Scipy and 1.9× over MATLAB for the two filters unwanted spectral and vuvuzela was observed. The proposed approach performed 7.7× better as compared to mapping some of the operations with FFTW calls. Thus, the proposed approach shows scalable speedups for various problem sizes in several domains.

7.2   Future Directions

Some of the enhancements to the proposed model are listed in this section.

- This work introduced support for mapping to a function call for two dimensional basic matrix operations. Extending this approach to map matrix operations on higher

dimensional matrices (tensors) to optimized library routines can be explored in the future. An example of this would include the *doitgen* benchmark in PolyBench.

- Extension of the tile size selection model to support multilevel tiling. This would require some minor modifications to the algorithm proposed in this work.

- Extending language specification and optimizations to add support for Recurrent Neural Networks.

- Extension of PolyMage code generation for multi-node support using MPI.

- The current tile size selection model is evaluated and tested for CPU applications. In the future, extensions to support tile size selection model for GPUs and other accelerators can be added.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[2] Denis Barthou, Paul Feautrier, and Xavier Redon. On the equivalence of two systems of affine recurrence equations (research note). pages 309–313, 01 2002.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 253–260, New York, NY, USA, 2014. ACM.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN conference on Programming Languages Design and Implementation*, pages 101–113, 2008.

[5] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multicore architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.

[6] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[7] Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[8] Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. Compiling affine loop nests for a dynamic scheduling runtime on shared and distributed memory. *ACM Trans. Parallel Comput.*, 3(2):12:1–12:28, July 2016.

[9] A C++ template library for linear algebra. http://eigen.tuxfamily.org/.

[10] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 42–51, New York, NY, USA, 2018. ACM.

[11] Karim Esseghir. *Improving data locality for caches*. Theses, 1993.

[12] John A. Gunnels and Robert A. van de Geijn. Formal linear algebra methods environment (flame) overview. Technical report, Austin, TX, USA, 2000.

[13] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. Libxsmm: Accelerating small matrix multiplications by runtime code generation. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 981–991, Nov 2016.

[14] Accelerate python perfromance, 2018. https://software.intel.com/en-us/distribution-for-python.

[15] Intel(R) Math Kernel Library for Deep Neural Networks. http://intel.github.io/mkl-dnn/.

[16] Guillaume Iooss. *Detection of linear algebra operations in polyhedral programs*. Theses, Université de Lyon, July 2016.

[17] Intel Integrated Performance Primitives (IPP). https://software.intel.com/en-us/intel-ipp.

[18] Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations. https://polly.llvm.org/.

[19] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 261–275, New York, NY, USA, 2018. ACM.

[20] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[21] LGen: A Basic Linear Algebra Compiler. http://www.spiral.net/software/lgen.html.

[22] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. Tile size selection revisited. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):35:1–35:27, December 2013.

[23] Intel math kernel library (MKL). http://software.intel.com/en-us/intel-mkl.

[24] Ravi Teja Mullapudi, Vinay Vasista, and Uday. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[25] An optimized BLAS library. https://www.openblas.net/.

[26] Pluto: An automatic polyhedral parallelizer and locality optimzer, 2008. https://github.com/bondhugula/pluto.

[27] Polybench suite, 2016. http://polybench.sourceforge.net.

[28] Polyhedral Parallel Code Generation, 2013. https://github.com/Meinersbur/ppcg.

[29] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Spiral*, pages 1920–1933. Springer US, Boston, MA, 2011.

[30] 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.

[31] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *2000 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS (Cat. No.00EX422)*, pages 146–153, 2000.

[32] Jaewook Shin, Mary Hall, Jacqueline Chame, Chun Chen, and Paul D. Hovland. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. 01 2009.

[33] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 101–121, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] J. G. Siek, I. Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

[35] Daniele G. Spampinato and Markus Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 23:23–23:32, New York, NY, USA, 2014. ACM.

[36] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[37] Vinay Vasista, Kumudha Narasimhan, Siddharth Bhat, and Uday Bondhugula. Optimizing geometric multigrid method computation using a dsl approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 15:1–15:13, New York, NY, USA, 2017. ACM.

[38] Sven Verdoolaege. Integer Set Library. an integer set library for program analysis.

[39] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[40] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013. ACM.

[41] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27:2001, 2000.

[42] M. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 30–44, 1991.

[43] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM.

[44] F. G. V. Zee, E. Chan, R. A. v. d. Geijn, E. S. Quintana-OrtÃ, and G. Quintana-OrtÃ. The libflame library for dense matrix computations. *Computing in Science Engineering*, 11(6):56–63, Nov 2009.

[45] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. Unified Polyhedral Modeling of Temporal and Spatial Locality. Research Report RR-9110, Inria Paris, November 2017.