Copyright
by
Tze Meng Low
2013

The Dissertation Committee for Tze Meng Low
certifies that this is the approved version of the following dissertation:

A Calculus of Loop Invariants for Dense Linear Algebra Optimization

Committee:

Robert van de Geijn, Supervisor

Don Batory

Calvin Lin

James Browne

Victor Eijkhout

A Calculus of Loop Invariants for Dense Linear Algebra Optimization

by

Tze Meng Low, B.A., B.S.C.S, M.S.C.S

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2013

Acknowledgments

The first step of the journey that cumulated in this dissertation[1] happened when Robert convinced me to apply to the PhD program, even though I was about to return to Singapore. Throughout this journey, Robert has been an exemplary advisor and mentor to me. He gave me the room to grow as an individual researcher, allowing me the space to pursue my own ideas, while still guiding me to refine my thoughts and ideas through the sharing of his experiences and the various discussions that we had. I sincerely thank him for giving me the time and space to mature so as to undertake the task of completing this dissertation, and for going out of his way to ensure that it remained possible for me to continue on this journey after my extended leave of absence.

I thank my dissertation committee for their advice and guidance in helping to shape this dissertation into its current form. Throughout this journey, including my seven-year hiatus, many nights were spent pondering how to address their concerns and incorporate their suggestions. Their comments

[1] This research has been funded by the following: NSF grants ACI-0850750:Collaborative Research: Mechanical Transformation of Knowledge to Libraries, CCF-0917167:SHF:Small:Transforming Linear Algebra Libraries, Award ACI-1148125/1340293 (supplement): Collaborative Research: SI2-SSI: A Linear Algebra Software Infrastructure for Sustained Innovation in Computational Chemistry and other Sciences, and a grant from Microsoft.

encouraged me to explore many issues related to this work in greater depth, and thus finding greater possibilities and interesting areas of research. I thank them for that.

A big thank you to my colleagues and friends from the FLAME project: Maggie Myers, Enrique Quintana-Ortís, Paolo Bientinesi, Field van Zee, Thierry Joffrain, Martin Schatz, Bryan Marker, Tyler Smith, Ardavan Pedram. From the noisy discussions at the whiteboard to the rowdy debates over lunch, they have been great sounding boards and sources of information and ideas. This journey was definitely interesting and fun with these great traveling companions.

Finally, I thank my loving wife for agreeing to pack our bags and kids to travel halfway around the world so that this journey that started almost a decade ago can be completed, and for being enthusiastic about the new journey that is about to begin.

A Calculus of Loop Invariants for Dense Linear Algebra Optimization

Publication No. ____________________

Tze Meng Low, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Robert van de Geijn

Loop invariants have traditionally been used in proofs of correctness (e.g. program verification) and program derivation. Given that a loop invariant is all that is required to derive a provably correct program, the loop invariant can be thought of as being the essence of a loop. Being the essence of a loop, we ask the question "What other information is embedded within a loop invariant?" This dissertation provides evidence that in the domain of dense linear algebra, loop invariants can be used to determine the behavior of the loops. This dissertation demonstrates that by understanding how the loop invariant describes the behavior of the loop, a goal-oriented approach can be used to derive loops that are not only provably correct, but also have the desired performance behavior.

Table of Contents

Acknowledgments **iv**

Abstract **vi**

List of Figures **xi**

Chapter 1. Introduction **1**

1.1 Motivation 1
1.2 Loop Invariant 6
1.3 Background 7
1.3.1 Mathematical approach to optimizing DLA algorithms . 8
1.3.2 Compiler approach to loop optimizations 11
1.4 A Calculus of Loop Invariants 14
1.5 Contributions 16
1.6 Outline 17

Chapter 2. Formal Linear Algebra Methods Environment **19**

2.1 FLAME Derivation Methodology 19
2.2 FLAME APIs 29
2.3 Relevance to this Work 29

Chapter 3. The Loop Invariant and its Remainder **32**

3.1 The Remainder 32
3.2 Generalization and Formalization 34
3.2.1 Prototypical operation 34
3.2.2 Other partitionings of operands 35
3.2.3 Expressing loop invariants in general 36
3.2.4 Expressing the remainder of a loop invariant 37
3.2.5 Status of Computation 38

3.3 Properties of the Remainder . . . 40
3.4 Summary . . . 43

Chapter 4. Analysis at a Higher Level of Abstraction 45
4.1 Types of Dependences . . . 45
4.1.1 True (Flow) dependence . . . 45
4.1.2 Anti-dependence . . . 46
4.1.3 Output dependence . . . 47
4.2 Loop-carried Dependence . . . 48
4.3 Analysis of $\mathcal{P}^{\mathcal{F}}$, $\mathcal{I}^{\mathcal{F}}$ and $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$ . . . 49
4.3.1 Dependencies in $\mathcal{P}^{\mathcal{F}}$ . . . 49
4.3.2 An illustration . . . 50
4.3.3 Loop-carried dependences in $\mathcal{I}^{\mathcal{F}}$ and $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$ . . . 51
4.3.4 An additional illustration . . . 54
4.4 Summary . . . 56

Chapter 5. Independent Iterations 58
5.1 Characteristics of Loop Invariants for Loops with Independent Iterations . . . 58
5.2 Application of Theory . . . 62
5.3 Summary . . . 63

Chapter 6. Merging DLA Loops 65
6.1 Motivation . . . 65
6.2 Loop Fusion . . . 66
6.2.1 Loop fusion for DLA in practice . . . 67
6.3 Characteristics of Desired Loop Invariants . . . 69
6.4 Generalization . . . 71
6.5 Merging More Than Two Operations . . . 74
6.6 Application of the Theory . . . 75
6.7 Summary . . . 77

Chapter 7. Domain-Specific Heuristics **78**
7.1 Unit Stride Data Accesses 78
7.1.1 Matrix storage 79
7.1.2 An example of access by columns 79
7.1.3 The theory 81
7.2 Matrix Multiplication 83
7.2.1 Examples of analysis with shapes of operands 85
7.2.2 The theory 88
7.3 Algorithms with Small Checkpoint Sizes 92
7.3.1 Checkpoint-optimal algorithms 93
7.3.2 Example of checkpoint-optimal algorithms 95
7.3.3 The theory 98
7.4 Summary 102

Chapter 8. A Goal-Oriented Approach To Loop Invariants Derivation **103**
8.1 Key Insights 103
8.2 Goal-Oriented Loop Invariant Derivation Algorithm 104
8.2.1 Initialization 105
8.2.2 Construction 106
8.2.3 Enumeration 107
8.3 Illustration 108
8.4 Summary 111

Chapter 9. Beyond Dense Linear Algebra **114**
9.1 Insights From Dense Linear Algebra Algorithms 114
9.2 Primitive Recursive Functions (PRF) 115
9.2.1 Important results regarding primitive recursive functions 117
9.3 FLAME algorithms compute primitive recursive functions . . . 118
9.3.1 $\mathcal{F}$ and the loop invariant 119
9.3.2 $\mathcal{H}$ and the update statements 119
9.3.3 $\mathcal{S}(k)$ and sweeping through the operands 120
9.4 Characterizing Domains Outside of Dense Linear Algebra . . . 120
9.5 Summary 126

Chapter 10. Conclusion and Future Directions **127**
10.1 Results 127
10.2 Future Work 129

Appendix **132**

Appendix 1. Table of Symbols **133**

Bibliography **134**

List of Figures

1.1 Expert implementations that call subroutines that implement other mathematical operations . . . 12

1.2 Resulting code after applying loop transformation in each of the different phases . . . 13

2.1 Blank FLAME derivation Worksheet . . . 20

2.2 PME and loop invariants for the Trmv-x operation . . . 24

2.3 Completed worksheet for deriving a loop for the Trmv-x operation using loop invariant 1 . . . 28

2.4 Matlab implementation of derived loop using the FLAME API 30

3.1 Completed FLAME Worksheet using the Remainder as input . 42

5.1 Loop invariants and remainders that represent algorithms with independent iterations for the Trinv-Alt operation. . . . 64

6.1 Top: Algorithms for the separate operations Trmv-x and Trmv-trans, and Bottom: Algorithm for the merged operation Trmv-merge 68

7.1 The different variants of Gemm, and the shapes of the operands, A, B and C. . . . 84

8.1 Enumerating different assignments of operations. Nodes represent f_X and g_X, edges represent paths taken by the algorithm when enumerating loop invariants . . . 112

8.2 Pairs of loop invariants whose loops can be merged . . . 113

Chapter 1

Introduction

The fundamental thesis of this dissertation is that the loop invariant (in the sense of Dijkstra and Hoare [29, 13]) can be used to formally derive loops, optimized for performance, in a goal-oriented fashion. We show that characteristics of the loop that impact performance can be identified *a priori* (i.e., before the loop is even implemented) from the loop invariant. This allows one to constructively identify loops that have desired characteristics (e.g., that yield good performance) when implemented.

1.1 Motivation

A loop is usually first derived, implemented, and then optimized either manually, or by an optimizing compiler. In order to optimize an implemented loop, it is typical for a traditional compiler to analyze the loop in order to recreate the information inherently lost or obscured in the input code. Often, how the loop is implemented can hinder the analysis process, thereby making it difficult to optimize the loop.

As an example, consider the following loop:

```
for (i = 0; i != n; ++i)
    foo(i+1, &x, 1, &L[i], n, &y[i], 1);
}
```

In order to analyze the loop, a traditional compiler will perform dependence analysis to create a partial ordering of the instances of the statement in the loop, which would subsequently be used to ensure that correctness is preserved.

Since the loop is implemented using a subroutine (`foo`), it is difficult for the traditional compiler to determine if the arrays `x`, `y` or `L` are used and/or updated in the subroutine. In such a situation, where the traditional compiler is unable to accurately determine if the elements in the arrays are being read or written, the traditional compiler is not able to determine if the instances of the statement can be reordered, and thus the traditional compiler is not able to optimize this loop. To overcome this loss of information, annotations-aware compilers [28, 10], which allow the programmer to specify which of the input arrays will be updated by the subroutine, have been developed by the compiler community.

One example of such an annotation is a description of the values in the arrays after the subroutine has been executed. Using such an annotation, elements in arrays that are read-only will retain their original values, and arrays that are updated will have values that differ from their original values. In formal methods, this description is simply the post-condition of the subroutine.

Because the subroutine is the only statement in the example loop, the post-condition must imply that the loop invariant (in the sense of Dijkstra and

Hoare) is true. In addition, the loop invariant must describe the inputs to the subroutine. Therefore, the loop invariant of the loop is a natural description for the updates in the body of the loop. We will see that we can reason about the reason about the body of the loop, even if it is written in terms of subroutine calls, by analyzing the loop invariant. This insight will be formalized in this dissertation.

We illustrate an analysis of a loop invariant with the loop given previously. In order to describe a loop invariant for the example loop, we first make the following observations:

– **The value of any element of an array must either be the final value, the original value or a partially updated value.** This implies that if there exists an element from an array whose value is either partially updated or final, then that array must have been updated by the subroutine `foo`. Similarly, if all elements in an array remain pristine after the subroutine `foo` has been executed, then that array is read-only, or not used.

– **After k iterations through the loop, the values in an updated array must be the result of k calls to the `foo` subroutine.** Recall that the loop body consist of exactly one call to the `foo` subroutine. This means that after k iterations, any changes to the values in the updated array must be from k calls to the `foo` subroutine.

– **The loop accesses the arrays in a systematic manner, allowing**

arrays to be described in terms of different regions. Consider the following sequence of subroutine calls to `foo` as the loop progresses:

```
foo(1, &x, 1, &L[0], n, &y[0], 1);
foo(2, &x, 1, &L[1], n, &y[1], 1);
                    .
                    .
                    .
foo(n, &x, 1, &L[n-1], n, &y[n-1], 1);
```

Notice that as the loop progresses, some of the inputs to the `foo` subroutine are changing in a systematic manner. At the end of k iterations, array `y` can be divided into two regions, y_T and y_B, where y_T describes addresses in array `y` that had already been passed into the `foo` subroutine, and y_B describes addresses in array `y` that have yet to be passed to the subroutine. Thus, array `y` can be envisioned to be partitioned as follows:

$$\texttt{y} : \left(\begin{array}{c} \left.\begin{array}{c} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{k-1} \end{array}\right\} y_T \\ \hline \left.\begin{array}{c} \psi_k \\ \psi_{k+1} \\ \vdots \\ \psi_{n-1} \end{array}\right\} y_B \end{array} \right)$$

Here, the thick line represents how far into the array `y`, the loop has progressed.

The above observations allow us to describe the loop invariant for the example loop following way:

$$\left(\frac{y_T \equiv f_T(L_T, x)}{y_B \equiv \widehat{y}_B} \right), \tag{1.1}$$

where the expression:

$$y_B \equiv \widehat{y}_B,$$

is used to denote that the region of the array `y` described by y_B contains original values.

As $y_T \not\equiv \widehat{y}_T$, it can be deduced from (1.1) that the array `y` has been updated by the subroutine `foo`. Recall that the thick line describe how far into the array `y` the loop has progressed. After $k-1$ and k iterations, y_T contains $k-1$ and k elements, respectively. This implies that exactly one element in array `y` is updated between the two iterations. In addition, we can deduce that the particular element updated in the kth iteration must be the kth element of the array `y`.

The fundamental observation is that, since the body of any loop (regardless of whether it is implemented in terms of subroutine calls or with primitive operations) can be replaced by a subroutine call, the loop invariant can be used as an annotation for the body of the loop. Therefore, an analysis of the loop invariant can be used in place of traditional dependence analysis.

In this dissertation, we show that the loop invariant can be used for more than an annotation for subroutines. Specifically, this dissertation will provide evidence, from the dense linear algebra (DLA) domain, that shows that information about the characteristics of the loops relating to performance are embedded within loop invariants. By analyzing loop invariants, these characteristics canbe used to select favorable loops for a particular architecture.

By raising the level at which the analysis of loops is performed, optimization techniques embedded in traditional compilers, and knowledge used by the DLA expert in optimizing DLA loops, is unified into a common framework. This facilitates the use of both sets of knowledge within the framework. More importantly, by working with loop invariants, loops that compute the correct result and execute in a desired manner can be formally derived in a goal-oriented fashion.

1.2 Loop Invariant

The term "loop invariant" is found in many fields of computer science. Across the different fields, the definitions of loop invariant often differ. For example, loop invariant variables in the compiler literature often refer to variables within a loop whose values across iterations do not change. This allows for the application of an optimization called *loop invariant code motion* (also known as code hoisting) which moves the computation of these loop invariant variables out of the loop [1]. This is not the notion of loop invariant we intend.

Even within the formal methods community, there are slight differences as to what constitute a loop invariant. In Hoare's work [29], a loop invariant is simply a condition that is true at the start of the loop and remains true after the execution of each iteration of the loop. Notice that Hoare's definition of a loop invariant allows for conditions that are not related to the execution of the loop (e.g., 0=0). By contrast, Dijkstra's notion of a loop invariant [13] is that of an assertion that is true at the end of every iteration of the loop.

In addition, the weakest precondition that ensures that the loop invariant is true at the end of each iteration must be implied by the loop invariant itself. Notice that in Dijkstra's definition, the loop invariant has to be related to the execution of the loop because it implies that the weakest precondition is satisfied.

In the context of this work, the definition of loop invariant that we adopt is closest to Dijkstra's in that our loop invariant is an assertion about the state of the values in *all* variables, *and it describes how those values have been computed* at the start and end of every iteration of a given loop. Since the assertion describes how all values at the start and end of every iteration have been computed, the updates that must happen within the loop is *prescribed* by the loop invariant. Therefore, our notion of a loop invariant can be used to derive the loop body in a goal-oriented fashion.

1.3 Background

Due to the importance of high performance dense linear algebra libraries to the scientific community, two communities in computer science have made optimizing dense linear algebra operations an area of focus in their respective fields. The linear algebra community has adopted a mathematical approach to optimization where a less efficient operation or sequence of operations and/or implementations are replaced by mathematically equivalent but more efficient operations and/or implementations. The compiler community has developed a transformational approach to optimization in which the com-

piler transforms the loops in the implementation of a linear algebra operation via a sequence of loop transformations to obtain an optimized implementation.

We illustrate the two different approaches with a concrete example, the triangular matrix-vector multiplication (Trmv-x), which can be described mathematically as

$$y = Lx$$

where y and x are vectors, and L is a lower triangular matrix.

1.3.1 Mathematical approach to optimizing DLA algorithms

The DLA expert, when required to optimize a DLA operation, will recognize that there are, typically, multiple algorithms that compute the same operation. The expert examines the operation to identify possible ways/algorithms in which the operation can be computed and then analyzes each possible algorithm to determine which of these algorithms should be implemented. Algorithms that are deemed inefficient and/or inefficient sequences of operations are replaced with more efficient ones. Let us now illustrate this manual process performed by the expert.

Examining the matrix-vector multiplication, the expert will note that the output y can be viewed as a linear combination of the columns of L, and y can be computed as a sequence of `axpy` (scaled vector addition) operations

as follows:

$$\left(\boxed{\begin{array}{c}\psi_0\\ \psi_1\\ \vdots\\ \psi_{m-2}\\ \psi_{m-1}\end{array}}\right) \stackrel{+}{=} \left(\begin{array}{c|c|cccc}\lambda_{0,0} & 0 & 0 & \cdots & 0\\ \lambda_{1,0} & \lambda_{1,1} & 0 & \cdots & 0\\ \vdots & \vdots & \ddots & \ddots & \vdots\\ \lambda_{m-2,0} & \lambda_{m-2,1} & \cdots & \lambda_{m-2,n-2} & 0\\ \lambda_{m-1,0} & \lambda_{m-1,1} & \cdots & \lambda_{m-1,n-2} & \lambda_{m-1,n-1}\end{array}\right)\left(\begin{array}{c}\chi_0\\ \boxed{\chi_1}\\ \vdots\\ \chi_{n-2}\\ \chi_{n-1}\end{array}\right),$$

where in each iteration of the loop, an `axpy` operation is performed. Each `axpy` operation scales a column of L by the corresponding element of x, χ_i, and the resulting scaled vector is accumulated into y.

Alternatively, y can be computed as a sequence of inner (dot) products as follows:

$$\left(\begin{array}{c}\psi_0\\ \boxed{\psi_1}\\ \vdots\\ \psi_{m-2}\\ \psi_{m-1}\end{array}\right) \stackrel{+}{=} \left(\begin{array}{ccccc}\lambda_{0,0} & 0 & 0 & \cdots & 0\\ \hline \lambda_{1,0} & \lambda_{1,1} & 0 & \cdots & 0\\ \hline \vdots & \vdots & \ddots & \ddots & \vdots\\ \lambda_{m-2,0} & \lambda_{m-2,1} & \cdots & \lambda_{m-2,n-2} & 0\\ \lambda_{m-1,0} & \lambda_{m-1,1} & \cdots & \lambda_{m-1,n-2} & \lambda_{m-1,n-1}\end{array}\right)\left(\boxed{\begin{array}{c}\chi_0\\ \chi_1\\ \vdots\\ \chi_{m-2}\\ \chi_{m-1}\end{array}}\right)$$

where in each iteration, an element (ψ_i) of y is computed by performing an inner product of the corresponding row of L with all of x.

For each of the two algorithms, there are two directions in which L can be access. Columns of L can be accessed left-to-right or right-to-left. Rows can be access from top-to-bottom or bottom-to-top. This yields two more algorithms that performs the same update but access L (and similarly, x or y) in a different manner.

Having identified multiple algorithms for computing the matrix-vector product, the expert will analyze the algorithms to determine their performance

characteristics. The expert would note that the `axpy`-based algorithms access elements of L column-wise, while the other variants access elements of L row-wise. If L is stored in column-major order (as is usually the case in the domain of DLA), then `axpy`-based algorithms will access elements of L with unit stride. This implies that there is spacial data locality, which reduces cache misses, and thus yields better performance than the other variants.

Another characteristic of the algorithms the expert would note is that there is more inherent parallelism in the inner-product-based algorithms. Notice that each inner-product computes a unique element of y. Therefore, all elements of y can be computed in parallel. While each `axpy` operation in the `axpy`-based algorithms can similarly be performed independently, their results must be accumulated together, which is inherently sequential. Therefore, the `axpy`-based algorithms have less parallelism.

Depending on the target machine architecture, the expert may choose to implement a different algorithm. On a sequential machine, the `axpy`-based algorithms would typically be better because of better cache performance. For an system with shared memory, the inner-product-based algorithms would be a better choice because of the inherent parallelism. Assuming a shared-memory implementation is desired, the expert will pick one of the inner-product variants and assign a subset of the inner-products to each of the processors so that the inner-products can be executed independently. A more sophisticated solution would combine these ideas.

The expert will notice that multiple inner products assigned to each

processor is mathematically equivalent to a smaller matrix-vector operation. A matrix-vector operation is more efficient than multiple inner product operations because the vector x is read once instead of multiple times. This implies that in order to compute the original matrix-vector operation, each processor can perform a smaller matrix-vector locally instead of multiple inner products. As discussed previously, this local matrix-vector operation can then be implemented with one of the `axpy` variants.

This optimization approach of replacing less efficient mathematical operations with more efficient ones is encapsulated in the way the expert implements his chosen algorithm. The implementations developed by the expert are often written in terms of subroutine calls to DLA libraries such as LAPACK [4] and the Basic Linear Algebra Subroutines (BLAS) [32, 16, 15] that implement other linear algebra operations. An example of how the four original variants of matrix-vector multiplication are implemented is summarized in Figure 1.1.

1.3.2 Compiler approach to loop optimizations

The focus of the compiler approach towards optimizing DLA algorithms is the loops that are inherent in DLA algorithms. By transforming the input loops via a sequence of loop restructuring transformations, known collectively as loop transformations, the optimizing compiler attempts to optimize the loops of the input implementations into a more efficient loop-based implementation that yields good performance.

Starting with an implementation of a DLA operation, a traditional

<pre>for (int i = 0; i < n; ++i){
  // BLAS call that performs
  // dot (inner) product
  y[i] = ddot(m, x[i],
                L[i*n], 1);
}</pre> Inner product from top to bottom
	<pre>for (int i = n; i >= n; --i){
	  // BLAS call that performs
	  // dot (inner) product
	  y[i] = ddot(m, x[i],
	                L[i*n], n);
	}</pre> Inner product from bottom to top

<pre>for (int j = 0; j < n; ++j){
  //BLAS call that performs
  //an axpy operation
  daxpy(n, x[j],
          L[j*m], 1, y, 1);
}</pre> `axpy` from left to right
	<pre>for (int j = n; j >= 0; --j){
	  //BLAS call that performs
	  //an axpy operation
	  daxpy(n, x[j],
	          L[j*m], 1, y, 1);
	}</pre> `axpy` from right to left

Figure 1.1: Expert implementations that call subroutines that implement other mathematical operations

compiler will first analyze the loop(s) and the statements within the body of the loop(s) to determine a partial ordering of all instances of the statements that will be executed over the course of the loop(s). Using the heuristics and analytical models built into the compiler, the compiler then determines which loop transformations to apply to the input loop(s), and the sequence/order in which the loop transformations are applied so that the output has the desired performance characteristic that should yield good performance on the desired machine architecture.

For instance, assume that the starting input was an implementation of one of the `axpy` variants, the compiler may first perform a loop optimization called loop interchange [2] so that the iterations of the outer loop are indepen-

```
for (j = 0; j < m; ++j){
  for (i = 0; i < n; ++i){
    y[i] += L[i,j]x[j]
  }
}
```
1) Input Code

```
for (i = 0; i < n; ++i){
  for (j = 0; j < m; ++j){
    y[i] += L[i,j]x[j]
  }
}
```
2) After loop interchange

```
for (j = 0; j < m; j+= 4){
  for (i = 0; i < n; ++i){
    y[i] += L[i,j]x[j]
  }
  for (i = 0; i < n; ++i){
    y[i] += L[i,j+1]x[j+1]
  }
  for (i = 0; i < n; ++i){
    y[i] += L[i,j+2]x[j+2]
  }
  for (i = 0; i < n; ++i){
    y[i] += L[i,j+3]x[j+3]
  }
}
```
3) After loop unroll

```
for (j = 0; j < m; j+= 4){
  for (i = 0; i < n; ++i){
    y[i] += L[i,j]x[j]
    y[i] += L[i,j+1]x[j+1]
    y[i] += L[i,j+2]x[j+2]
    y[i] += L[i,j+3]x[j+3]
  }
}
```
4) After loop fusion (Output)

Figure 1.2: Resulting code after applying loop transformation in each of the different phases

dent of each other. Next, another loop transformation, called loop unrolling [1, 53], may be applied to transformed loop, yielding a new intermediate loop-based code that will allow the next loop transformation to be applied. Finally, loop fusion [31, 47] may be applied to the intermediate code to combine the inner loops into a single loop that reduces memory accesses. The different intermediate loops, created after each loop transformation is applied, is shown in Figure 1.2.

The loop transformations applied and the sequence in which the loop transformations are performed is important. In this instance, if loop unrolling is not performed before loop fusion, then loop fusion cannot be applied. As loop transformations are applied in phases by the compiler, identifying which loop transformation must be applied and the order in which the transformations need to be applied is known as the phase-ordering problem [3], an NP-complete problem, in the compiler literature.

Recently, the compiler community has taken to using empirical search, or a hybrid of empirical searching and analytical modeling to overcome the phase-ordering problem [6, 5]. Essentially, a process that, repeatedly, applies transformations to the code, compiles and execute the code on the target machine, and determines if the transformation is beneficial is automatically performed by the compiler as a tuning step to avoid the phase-ordering problem.

1.4 A Calculus of Loop Invariants

Starting around 2000,the Formal Linear Algebra Methods Environment (FLAME) project [7] has systematized the process the DLA expert uses to discover multiple algorithms for the same operation through a systematic exploration of loop invariants for the same operation. The essence of the project is that each algorithm identified by the DLA expert can be represented by its loop invariant. This implies that the optimization performed by the DLA expert can be viewed as a replacement of the original loop invariant with the

loop invariant of the optimized loop.

Similarly, we argue that many of the loop transformations performed by the compiler can be viewed as a transformation of the loop invariant. Recall that a compiler optimizes a loop by applying a sequence of one or many loop transformations to the input loop in order to restructure it into an optimized output loop with a particular characteristic. Since each loop transformation changes the input loop to a different output loop, each loop transformation can be viewed as a function that maps the loop invariant of the input loop to the loop invariant of the output loop. This implies that a sequence of loop transformations can be viewed as a series of functions that maps the input loop invariant to the desired output loop invariant.

Our observation is that despite the different approaches (and different starting inputs) towards optimizing DLA loops, both approaches can be unified using a framework that views optimization of DLA loops as a change in the loop invariant. As compilers apply loop transformations in order to obtain output loops that have desirable characteristics for particular machine architectures, an understanding of how these desirable characteristics are described by the loop invariants of the output loops will allow the systematic discovery of the characteristics of the algorithms.

We have thus motivated characteristics of algorithms can be identified from their loop invariants.

1.5 Contributions

This dissertation contributes to the field of computer science in the following ways:

– It provides evidence that loop invariants can be used for more than program derivation and proofs of correctness. We show that loop invariants can also be used to determine characteristics that impact performance of the loop they represent. This allows one to identify loop-based algorithms that have the potential to perform well on a given machine architecture.

– The theories developed in this dissertation allows the generalization of the notion of goal-oriented programming from goals related to the correctness of the computation performed by the loop to goals related to the performance characteristics of the loop. The identification of loop invariant with a desired characteristic is shown to be so systematic that an algorithm for their identification is one contribution of this dissertation.

– This dissertation demonstrates that the phase-ordering problem encountered by compilers can be avoided if one takes a goal-oriented approach towards optimizing DLA loops. As compilers optimize loops by ensuring that the output loop has a particular characteristic, deriving the loop invariant of the output loop would allow one to avoid having to identify the sequence of one or more transformation that results in the output loop. The additional benefit of the goal-oriented approach is that

multiple loop invariants (hence, mutliple output loops) with the desired characteristic may be obtained. Obtaining multiple output loops where each output loop is the result of a different sequence of transformations, would exacerbate the phase-ordering problem using the tranformational approach employed by compilers.

– This dissertation unifies the optimization approaches of the traditional optimizing compiler, and the DLA expert into a common framework based on the analysis and optimization of loop invariants. This unification of the two approaches allows one to use both compiler and DLA expert knowledge to optimize DLA loops by raising the level at which analysis and optimization of loops are performed. In addition, it allows analysis and optimization to be performed on loops that include subroutine calls to black-box libraries.

1.6 Outline

This dissertation is structured as followed:

We start by providing a quick introduction to the FLAME methodology for deriving dense linear algebra algorithms from their mathematical specifications in Chapter 2. We end that chapter with a discussion on the relevance of the FLAME methodology to the rest of the dissertation.

In Chapters 3 and 4, we introduce and formalize the notion of the *remainder of a loop invariant*. We also discuss its importance for analysis

purposes and show how dependence analysis performed by a compiler can be raised to a higher level of abstraction with the use of the loop invariant and its remainder.

Chapters 5 through 7 provide examples of using the loop invariant and its remainder to determine if a particular optimization is legal (i.e. still computes the same output) and the resulting loop invariant after the optimization is performed.

In Chapter 8, we introduce a goal-oriented approach to deriving algorithms where the goal is not only to derive an algorithm that computes the correct result, but also to derive an algorithm with a particular desired characteristic. Using the theories developed in the previous chapters, we introduce an algorithm to construct loop invariants that represent loops with the desired performance characteristics.

Theoretical result that shows how the approach is applicable to domains other than DLA is given in Chapter 9. Finally, a summary of this dissertation is provided and future directions are identified in Chapter 10.

Chapter 2

Formal Linear Algebra Methods Environment

The Formal Linear Algebra Methods Environment (FLAME) is a project with the ultimate aim of mechanically generating and implementing high performance dense linear algebra libraries for any machine architecture from the mathematical specification of the desired functionality. In this chapter, we review the results from the FLAME project that are relevant to this dissertation.

2.1 FLAME Derivation Methodology

The FLAME derivation methodology is a systematic process that guides one through an 8-step derivation process by filling in the "worksheet" shown in Figure 2.1. We illustrate the FLAME methodology with the derivation of a loop for a triangular matrix-vector multiplication. Householder's notation will be used, where scalars, vectors and matrices are represented by lowercase Greek letters, lowercase and uppercase letters, respectively. Vectors are assumed to be column vectors. Row vectors are represented by transposed column vectors.

The triangular matrix-vector multiplication (TRMV-X) is typical of a

Step	Annotated Algorithm:
1a	$\{P_{pre}\}$
4	**Partition** **where**
2	$\{P_{inv}\}$
3	**while** G **do**
2,3	$\{(P_{inv}) \wedge (G)\}$
5a	**Repartition** **where**
6	$\{P_{before}\}$
8	S_{update}
5b	**Continue with**
7	$\{P_{after}\}$
2	$\{P_{inv}\}$
	endwhile
2,3	$\{(P_{inv}) \wedge \neg (G)\}$
1b	$\{P_{post}\}$

Figure 2.1: Blank FLAME derivation Worksheet

large class of dense linear algebra operations. Mathematically, the operation can be expressed as

$$y := Lx \tag{2.1}$$

where x and y are vectors, and L is a triangular matrix. In this discussion, L is assumed to be lower triangular, and dimensions of all operands are assumed to be conformal so that the operation is well-defined.

Step 1a, b: Defining P_{pre} and P_{post}. The first step of the FLAME derivation process is to identify the pre-condition (P_{pre}) and post-condition (P_{post}) of the TRMV-X operation. At the start of the algorithm, no computation has occurred. This means that the values of all operands (x, y and L) must be

their original values. Using $\widehat{x}$, $\widehat{y}$ and $\widehat{L}$ to represent the original values of x, y and L respectively, the pre-condition must be

$$P_{pre}: \quad x \equiv \widehat{x} \ \wedge \ y \equiv \widehat{y} \ \wedge \ L \equiv \widehat{L},$$

which states that, at the start of the algorithm, the current value of x is its original value $\widehat{x}$. The same can be said for y and L. Strictly speaking, P_{pre} also describes that the dimensions of x, y and L must be conformal and L must always be a lower triangular matrix. However, these conditions are assumed to be implicit.

Upon the completion of the loop, y must contain the final value, $\widetilde{y}$, while L and x must still contain their original values. This implies that

$$y = \widetilde{y} \equiv \widehat{L}\widehat{x} \quad \wedge \quad x = \widetilde{x} \equiv \widehat{x} \quad \wedge \quad L = \widetilde{L} \equiv \widehat{L}.$$

A more detailed description of the final value ($\widetilde{y}$) can be obtained by expressing $\widetilde{y}$ as the result of computations with partitioned matrices. Basic linear algebra states that matrix computation can be expressed in terms of operations on partitioned matrices so long as the different submatrices/regions of the matrices after partitioning remain conformal. Therefore, when x, y and L are partitioned into conformal regions in the following manner:

$$y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right), x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right), \text{ and } L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right),$$

Equation (2.1) can be re-written as:

$$\begin{array}{rcl} \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) & := & \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \\ & \equiv & \left(\begin{array}{c} L_{TL}x_T \\ \hline L_{BL}x_T + L_{BR}x_B \end{array}\right), \end{array}$$

which implies that, at the end of the to-be-derived loop,

$$P_{post}: \quad \left(\begin{array}{c} y_T = \widetilde{y}_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B = \widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B \end{array}\right) \quad \wedge$$
$$\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv \widehat{L}_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv \widehat{L}_{BL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR} \end{array}\right) \quad \wedge$$
$$\left(\begin{array}{c} \widetilde{x}_T \equiv \widehat{x}_T \\ \hline \widetilde{x}_B \equiv \widehat{x}_B \end{array}\right)$$

This expression, P_{post}, is also known as the *Partitioned Matrix Expression(PME)* in the FLAME terminology[1]. Since the last two terms assert that the final values of L and x are the same as their original values, these two terms are dropped to simplify the expression for P_{post}.

Notice that the PME is a recursive definition of the desired operation and it describes the operations performed with the different regions of L and x in order to compute the output y.

Step 2: Determine loop invariant. The next step of the derivation process is the identification of possible loop invariants to use for the subsequent steps in the derivation process.

Recall that the PME is an expression that describes what the different regions of the output y must be when the loop has completed. When the loop has not finished, it must be the case that some of the computations described in the PME have not been performed. *By deleting operations from the PME*

[1]The predicates asserting that x and L are not changed at the end of the loop are removed to simplify the expression for P_{post}.

and therefore asserting that only some of the operations needed to compute the output has been performed, potential loop invariants can be found.

A fundamental insight of DLA algorithms used during the process of deriving feasible loop invariants is that DLA loops sweeps through the operands in a systematic manner. Continuing with the TRMV-X example: At the start of the loop, y is not computed. As the loop progresses, parts of y are updated and the region of y that still contains the original value would shrink. At the end of the loop, the region containing the original value of y is empty and all of y must be fully computed.

A natural expression for the loop invariant that describes the current value of the different regions of y at the start and end of every iteration is:

$$P_{inv}: \quad \left(\begin{array}{c} y_T = \widetilde{y}_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B \equiv \widehat{y}_B \end{array} \right), \tag{2.2}$$

which states that the current values of the region of y that is above the thick lines (y_T) already contain the computed value $\widetilde{y}_T$ while the other region (y_B) still contains the original value ($\widehat{y}_B$). Here, the thick lines represent how far through the operands the loop has proceeded. Other loop invariants that yield loops for the TRMV-X operation are shown in Figure 2.2. For the rest of this example, the loop invariant in (2.2) will be used to derive a loop that computes TRMV-X.

Step 3: Determine loop guard G. After the loop completes, $\{P_{inv} \wedge \neg G\}$ is true. In addition, since the loop is now complete, P_{post} must also

$$\left(\begin{array}{c} y_T = \widetilde{y}_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B = \widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B \end{array}\right)$$ PME

$$\left(\begin{array}{c} y_T \equiv \widehat{y}_T \\ \hline y_B \equiv \widehat{L}_{BR}\widehat{x}_B \end{array}\right)$$ Loop Invariant 1	$$\left(\begin{array}{c} y_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B \equiv \widehat{L}_{BL}\widehat{x}_T \end{array}\right)$$ Loop Invariant 2
$$\left(\begin{array}{c} y_T \equiv \widehat{y}_T \\ \hline y_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B \end{array}\right)$$ Loop Invariant 3	$$\left(\begin{array}{c} y_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B \equiv \widehat{y}_B \end{array}\right)$$ Loop Invariant 4

Figure 2.2: PME and loop invariants for the TRMV-X operation

be true. Therefore, G must be chosen such that $\{P_{inv} \wedge \neg G\}$ implies P_{post}. When y_T contains all of y, it can be concluded that $L_{TL} \equiv L$ and $x_T \equiv x$, as the dimensions of the regions must remain conformal for P_{inv} to be a mathematically valid expression. This implies that:

$$y_T = \widetilde{y}_T \equiv \widehat{L}_{TL}\widehat{x}_T$$

is equivalent to:

$$y = \widetilde{y} \equiv \widehat{L}\widehat{x}$$

when $m(y_T) = m(y)$ where $m(.)$ returns the number of rows of the argument. Therefore, a valid choice for G is:

$$m(y_T) < m(y).$$

Step 4: Determine initialization. Step 4 is essentially an indexing step to ensure that the operands x, y and L are partitioned such that P_{pre} implies P_{inv}. Notice that if y_T is not empty (contains no elements), then some computation must be performed to update those elements of y in y_T in order for P_{inv} to be true. This cannot be the case since no computation has been performed before the loop has started. This implies that P_{pre} cannot imply P_{inv} unless y_T is empty. Therefore, the initial partitioning of y must be such that $m(y_T) = 0$, which implies that L_{TL} must be a 0×0 matrix and x_T is also empty.

Step 5a, 5b: Determine how to traverse the operands. Recall that the thick lines represent how far through the operands the loop has proceeded. To ensure progress is made in computing y, the region of y that still contains its original values, i.e. y_B, is repartitioned to expose the next subregion of y that is to be updated. This is indicated by thin lines as follows:

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right),$$

which exposes the top row of y_B[2]. At the end of the loop, the newly exposed subregion (ψ_1) of y_B is updated and is moved across the thick line in the following manner:

$$\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ \hline y_2 \end{array}\right),$$

[2] In this case, that top row of y_B happened to be a row of one element.

making the top row of the previous y_B, the bottom row of previous y_T. This achieves two objectives: (1) progress is ensured since the size of y_B that contains the original values has decreased and eventually y_B will be empty; and (2) the loop invariant is maintained in that y_T contains computed values while y_B still contains original values. In addition, when y is repartitioned, L and x also need to be repartitioned in a conformal manner to ensure that P_{inv} is still a valid expression. This implies that x and L are repartitioned at the start of an iteration as follows:

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10} & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right);$$

and at the end of an iteration, the different subregions of x and L are shifted in the following manner:

$$\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ \hline x_2 \end{array}\right) \text{ and } \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10} & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right).$$

Step 6: Determine P_{before}. Steps 5a and 5b are indexing operations that do not change the values of the operands. Therefore, the status before the update statement (P_{before}) can be obtained by textual substitution of the expressions obtained in Step 5a into the loop invariant, yielding:

$$P_{before}: \quad \left(\begin{array}{c} y_0 \equiv \widehat{L}_{00}\widehat{x}_0 \\ \hline \psi_1 \equiv \widehat{\psi}_1 \\ \hline y_2 \equiv \widehat{y}_2 \end{array}\right)$$

Step 7: Determine P_{after}. Similarly, the status of the operands after the update statements (P_{after}) has been performed can be obtained by textual substitution of the expression obtained in Step 5b into the loop invariant, yielding:

$$P_{after}: \quad \left(\begin{array}{c} y_0 \equiv \widehat{L}_{00}\widehat{x}_0 \\ \hline \psi_1 \equiv \widehat{l}_{10}\widehat{x}_0 + \widehat{\lambda}_{11}\widehat{\chi}_1 \\ \hline y_2 \equiv \widehat{y}_2 \end{array}\right)$$

Step 8: Determine the update statements, S_{update}. The update statements are now dictated by the current status of the output in Step 6 and the status they must be in at Step 7:

1. y_0 must contain $\widehat{L}_{00}\widehat{x}_0$ at the end of the iteration. Since it already contains that value at Step 6, no update to y_0 is required.

2. ψ_1 contains its original value $\widehat{\psi}_1$ at Step 6 and it needs to contain the value $\widehat{l}_{10}\widehat{x}_0 + \widehat{\lambda}_{11}\widehat{\chi}_1$ at Step 7. This means that ψ_1 needs to be updated in the following manner: $\psi_1 := l_{10}x_0 + \lambda_{11}\chi_1$.

3. y_2 needs to contain its original value $\widehat{y}_2$ at Step 7, which is already the current value of y_2. Therefore, no computation is required to update y_2.

The completed worksheet for the Trmv-x operation with the loop invariant:

$$\left(\begin{array}{c} y_T \equiv \widehat{L}_{TL}\widehat{x}_T \\ \hline y_B \equiv \widehat{L}_{BL}\widehat{x}_T + L_{BR}\widehat{x}_B \end{array}\right)$$

is given in Figure 2.3. Removing the assertions (grayed out rows) from the worksheet yields a loop that is derived to be correct.

Step	Annotated Algorithm: $y := Lx$
1a	$\{y \equiv \widehat{y}\}$
4	**Partition** $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, $L \rightarrow \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ **where** y_T has 0 rows, L_{TL} is 0×0, x_T has 0 rows
2	$\left\{\left(\begin{array}{c} y_T \equiv L_{TL}x_T \\ \hline y_B \equiv \widehat{y}_B \end{array}\right)\right\}$
3	**while** $m(y_T) < m(y)$ **do**
2,3	$\left\{\left(\left(\begin{array}{c} y_T \equiv L_{TL}x_T \\ \hline y_B \equiv \widehat{y}_B \end{array}\right)\right) \wedge (m(y_T) < m(y))\right\}$
5a	**Repartition** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$, $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ **where** ψ_1 has 1 row, λ_{11} is 1×1, χ_1 has 1 row
6	$\left\{\left(\begin{array}{c} y_0 \equiv L_{00}x_0 \\ \hline \psi_1 \equiv \widehat{\psi}_1 \\ y_2 \equiv \widehat{y}_2 \end{array}\right)\right\}$
8	$\psi_1 := L_{01}x_0 + \lambda_{11}\chi_1$
5b	**Continue with** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$, $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$
7	$\left\{\left(\begin{array}{c} y_0 \equiv L_{00}x_0 \\ \psi_1 \equiv L_{01}x_0 + \lambda_{11}\chi_1 \\ \hline y_2 \equiv \widehat{y}_2 \end{array}\right)\right\}$
2	$\left\{\left(\begin{array}{c} y_T \equiv L_{TL}x_T \\ \hline y_B \equiv \widehat{y}_B \end{array}\right)\right\}$
	endwhile
2,3	$\left\{\left(\left(\begin{array}{c} y_T \equiv L_{TL}x_T \\ \hline y_B \equiv \widehat{y}_B \end{array}\right)\right) \wedge \neg (m(y_T) < m(y))\right\}$
1b	$\{\widetilde{y} \equiv y \equiv Lx\}$

Figure 2.3: Completed worksheet for deriving a loop for the TRMV-X operation using loop invariant 1

2.2 FLAME APIs

Having derived a loop that is provably correct, it is then necessary to translate that loop into actual code. The FLAME APIs [9] is a set of interfaces for different programming languages that were designed to capture the correctness of the derived loop in code. This is achieved by making the code visually similar to the derived loop. In addition, by hiding details such as indices, storage methods and matrix dimensions through the use of objects and methods, commonly encountered programming errors are avoided.

The derived loop in Figure 2.3 when implemented with the FLAME Matlab API is given in Figure 2.4.

2.3 Relevance to this Work

We make the following observations:

- Using the FLAME API, the code is now visually similar to the derived loop. An analysis of the code is identical to the analysis of the derived loop.

- The derived loop is dependent on one key input, the loop invariant. With the loop invariant (identified in Step 2), a loop guard and the update statements in the loop body can be derived. In other words, the loop guard and update statements are prescribed by the loop invariant. Hence, the loop invariant is the essence of the derived loop and an analysis of the derived loop is equivalent to an analysis of the loop invariant.

```
function [ y_out ] = Trmv_x( L, x, y )

  [ LTL, LTR, ...
    LBL, LBR ] = FLA_Part_2x2( L, ...
                               0, 0, 'FLA_TL' );
  [ xT, ...
    xB ] = FLA_Part_2x1( x, ...
                         0, 'FLA_TOP' );
  [ yT, ...
    yB ] = FLA_Part_2x1( y, ...
                         0, 'FLA_TOP' );

  while ( size( LTL, 1 ) < size( L, 1 ) )

    [ L00,  l01,      L02,  ...
      l10t, lambda11, l12t, ...
      L20,  l21,      L22 ] = FLA_Repart_2x2_to_3x3( LTL, LTR, ...
                                                      LBL, LBR, ...
                                                      1, 1, 'FLA_BR' );
    [ x0, ...
      \chi1, ...
      x2 ] = FLA_Repart_2x1_to_3x1( xT, ...
                                    xB, ...
                                    1, 'FLA_BOTTOM' );
    [ y0, ...
      psi1, ...
      y2 ] = FLA_Repart_2x1_to_3x1( yT, ...
                                    yB, ...
                                    1, 'FLA_BOTTOM' );
    %------------------------------------------------------------%

    psi1 = l10t * x0 + lambda11 * chi1;

    %------------------------------------------------------------%
    [ LTL, LTR, ...
      LBL, LBR ] = FLA_Cont_with_3x3_to_2x2( L00,  l01,      L02,  ...
                                             l10t, lambda11, l12t, ...
                                             L20,  l21,      L22, ...
                                             'FLA_TL' );
    [ xT, ...
      xB ] = FLA_Cont_with_3x1_to_2x1( x0, ...
                                       \chi1, ...
                                       x2, ...
                                       'FLA_TOP' );
    [ yT, ...
      yB ] = FLA_Cont_with_3x1_to_2x1( y0, ...
                                       psi1, ...
                                       y2, ...
                                       'FLA_TOP' );
  end

  y_out = [ yT
            yB ];
return
```

Figure 2.4: Matlab implementation of derived loop using the FLAME API

– Since many loop invariants can be obtained by removing different sets of operations from the PME, the analysis of multiple loop invariants can be performed simultaneously with a single analysis of the PME.

In the rest of this dissertation, we show how the code analysis can be raised to a higher level of abstraction, by providing the theory and techniques for analyzing PMEs and loop invariants.

Chapter 3

The Loop Invariant and its Remainder

Recall that the loop invariant is an assertion that states how (possibly intermediate) values have been computed and stored in the different regions of the output operand at the start and end of every iteration. More computations must be performed in order to compute the final result. In this chapter, we introduce an expression called the *remainder of a loop invariant* that describes the additional computations required to compute the final result.

3.1 The Remainder

Since the loop invariant in the FLAME methodology is obtained by removing operations from the PME (which describes the final result), the operations that were removed from the PME must describe computation that still need to be performed in order to obtain the final result. This is apparent when a textual substitution of the loop invariant into the PME is performed.

We illustrate how to obtain an expression that describes computation that still has to be performed, by revisiting the TRMV-X ($y := Lx$) example.

For convenience, we replicate the PME and the loop invariant in (2.2):

$$\text{PME } (P_{post}) : \left(\frac{\widetilde{y}_T \equiv \widehat{L}_{TL}\widehat{x}_T}{\widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B} \right), \quad P_{inv} : \left(\frac{y_T \equiv \widehat{L}_{TL}\widehat{x}_T}{y_B \equiv \widehat{y}_B} \right).$$

Since P_{inv} asserts that some computations have been performed on y_T, and the current value of y_T is given by the expression $\widehat{L}_{TL}\widehat{x}_T$, we can substitute y_T for the sub-expression $\widehat{L}_{TL}\widehat{x}_T$ in the top expression of the PME:

$$\text{PME} : \quad \left(\frac{\widetilde{y}_T \equiv \overbrace{\widehat{L}_{TL}\widehat{x}_T}^{y_T}}{\widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B} \right),$$

which yields:

$$\left(\frac{\widetilde{y}_T \equiv y_T}{\widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B} \right). \tag{3.1}$$

Similarly, since the current value in y_B is $\widehat{y}_B$, y_B can be substituted for any occurrence of $\widehat{y}_B$ in the bottom region of the PME. Since $\widehat{y}_B$ does not appear in the P_{post}, the resulting expression of performing the substitution of y_B for $\widehat{y}_B$ is still given by (3.1).

The expression in (3.1) states that the final value of $\widetilde{y}_T$ is the current value of y_T. This implies that y_T is already fully computed and no future computation is required. Similarly, the expression also states that the value given by the operations $\widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B$ still needs to be performed in order to obtain $\widetilde{y}_B$. In other words, the expression in (3.1) tells us the remaining computation that must be performed on the current values in y in order to compute the final result. Since Expression (3.1) states what remaining computation must be performed, we term it the *remainder of the loop invariant*, or simply, the *Remainder*.

3.2 Generalization and Formalization

Having provided an example to illustrate the relationship between the loop invariant and its remainder, we now generalize and formalize the insights to arbitrary DLA operations with one output operand.

3.2.1 Prototypical operation

Assume that the operation that we intend to analyze is of the following form:

$$C := \mathcal{F}(C, \{A\})$$

where C is both an input and output operand, $\{A\}$ is a set of inputs whose values are read but never overwritten, and $\mathcal{F}$ is a function that operates over $C \cup \{A\}$. In addition, we assume that the output value ($\widetilde{C}$) and input value ($\widehat{C}$) of C are different, i.e. $\widetilde{C} \neq \widehat{C}$.

Let us assume that C has been partitioned into different regions as follows:

$$C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array} \right),$$

and the post-condition (PME), $\mathcal{P}^{\mathcal{F}}$, which asserts what the final values of the different regions of C must be, can be expressed in the following manner:

$$\mathcal{P}^{F} = \left(\begin{array}{c|c} \widetilde{C}_{TL} \equiv \mathcal{F}_{TL}(\widehat{C}_{TL}) & \widetilde{C}_{TR} \equiv \mathcal{F}_{TR}(\widehat{C}_{TR}) \\ \hline \widetilde{C}_{BL} \equiv \mathcal{F}_{BL}(\widehat{C}_{BL}) & \widetilde{C}_{BR} \equiv \mathcal{F}_{BR}(\widehat{C}_{BR}) \end{array} \right), \quad (3.2)$$

where each $\mathcal{F}_X$ for all $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$ represents all computation required to compute the final value $\widetilde{C}_X$ of region C_X from its original value $\widehat{C}_X$.

The expression in (3.2) is the PME of the operation in the FLAME terminology. Each $\mathcal{F}_X$ can be viewed as a mapping of the original contents of C_X, denoted $\widehat{C}_X$, to the final values, $\widetilde{C}_X$. All other operands to each $\mathcal{F}_X$ are treated as constants as the only region updated by $\mathcal{F}_X$ is C_X, including $\widehat{C}_Y$, $\widetilde{C}_Y$ and C_Y, where $Y \neq X$.

3.2.2 Other partitionings of operands

Operands can be partitioned into multiple regions in different manners. As we have seen in the Trmv-x example, one partitioning yields top and bottom portions of an operand, i.e.:

$$y := \left(\frac{y_T}{y_B} \right).$$

Apart from being partitioned into four regions, two commonly encountered ways of partitioning a matrix into regions are as follows:

$$C \rightarrow \left(\frac{C_T}{C_B} \right) \text{ or } C \rightarrow \left(\begin{array}{c|c} C_L & C_R \end{array} \right).$$

These different ways of partitioning a matrix are represented by our prototypical operation where some of the regions are degenerate regions. For example, the partitioning:

$$C \rightarrow \left(\frac{C_T}{C_B} \right)$$

is represented by our prototypical operation in the following manner:

$$C \rightarrow \left(\begin{array}{c|c} C_{T(L)} & \\ \hline C_{B(L)} & \end{array} \right),$$

where the regions to the right of the thick lines are empty (i.e. contain no values), and the regions to the left are relabeled as C_T and C_B. The partitioning that yields left and right regions of an operand can be similarly defined.

3.2.3 Expressing loop invariants in general

Any function, $\mathcal{F}_X$, for all $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, can always be expressed in terms of two simpler functions f_X and $f_X^{\mathcal{R}}$, such that:

$$\mathcal{F}_X(\widehat{C}_X) \equiv f_X^{\mathcal{R}}(f_X(\widehat{C}_X)).$$

A function $\mathcal{F}_X$ can be expressed in the desired form by setting either f_X or $f_X^{\mathcal{R}}$ to $\mathcal{F}_X$, and the other function (either $f_X^{\mathcal{R}}$ or f_X respectively) to the identity function, i.e. $\mathcal{G}(X) = X$.

Formulating $\mathcal{F}_X$ as two simpler functions allows one to interpret the necessary computation of $\widetilde{C}_X$ from $\widehat{C}_X$ as a two-step process. First, some of the operations of $\mathcal{F}_X$, represented by f_X, are performed to obtain an intermediate value C_X where $C_X \equiv f_X(\widehat{C}_X)$. The final value of $\widetilde{C}_X$ is then computed from C_X by performing the remaining operations of $\mathcal{F}_X$ (represented by $f_X^{\mathcal{R}}$) to compute $\widetilde{C}_X$.

A loop-based algorithm that computes $\widetilde{C}$ must inherently compute C incrementally. In each iteration of the loop, some additional computation must be performed to all or some regions of C. Therefore, a loop invariant that tracks the current value of C must assert that C only contains a partial/intermediate result. Hence, such a loop invariant, denoted $\mathcal{I}^{\mathcal{F}}$, can be

defined in terms of f_X's as follows:

$$\mathcal{I}^{\mathcal{F}} = \left(\begin{array}{c|c} C_{TL} \equiv f_{TL}(\widehat{C}_{TL}) & C_{TR} \equiv f_{TR}(\widehat{C}_{TR}) \\ \hline C_{BL} \equiv f_{BL}(\widehat{C}_{BL}) & C_{BR} \equiv f_{BR}(\widehat{C}_{BR}) \end{array} \right). \tag{3.3}$$

3.2.4 Expressing the remainder of a loop invariant

In Section 3.1, we illustrated for a specific example (namely, the TRMV-X operation) that the Remainder is obtained by a textual substitution of the loop invariant into the PME. In this section, we describe more precisely what we mean. Recall that the PME ($\mathcal{P}^{\mathcal{F}}$) is given by:

$$\left(\begin{array}{c|c} \widetilde{C}_{TL} \equiv \mathcal{F}_{TL}(\widehat{C}_{TL}) & \widetilde{C}_{TR} \equiv \mathcal{F}_{TR}(\widehat{C}_{TR}) \\ \hline \widetilde{C}_{BL} \equiv \mathcal{F}_{BL}(\widehat{C}_{BL}) & \widetilde{C}_{BR} \equiv \mathcal{F}_{BR}(\widehat{C}_{BR}) \end{array} \right),$$

which, as discussed in Section 3.2.3, is equivalent to:

$$\mathcal{P}^{\mathcal{F}} \equiv \left(\begin{array}{c|c} \widetilde{C}_{TL} \equiv f^{\mathcal{R}}_{TL}(f_{TL}(\widehat{C}_{TL})) & \widetilde{C}_{TR} \equiv f^{\mathcal{R}}_{TR}(f_{TR}(\widehat{C}_{TR})) \\ \hline \widetilde{C}_{BL} \equiv f^{\mathcal{R}}_{BL}(f_{BL}(\widehat{C}_{BL})) & \widetilde{C}_{BR} \equiv f^{\mathcal{R}}_{BR}(f_{BR}(\widehat{C}_{BR})) \end{array} \right).$$

Since a loop invariant has the general form:

$$\left(\begin{array}{c|c} C_{TL} \equiv f_{TL}(\widehat{C}_{TL}) & C_{TR} \equiv f_{TR}(\widehat{C}_{TR}) \\ \hline C_{BL} \equiv f_{BL}(\widehat{C}_{BL}) & C_{BR} \equiv f_{BR}(\widehat{C}_{BR}) \end{array} \right),$$

we know that the expression $f_X(\widehat{C}_X)$ is equivalent to C_X, for all $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. This means that we can replace the expression $f_X(\widehat{C}_X)$ in $\mathcal{P}^F$ with C_X, yielding:

$$\mathcal{R}_{\mathcal{I}^F} \equiv \left(\begin{array}{c|c} \widetilde{C}_{TL} \equiv f^{\mathcal{R}}_{TL}(C_{TL}) & \widetilde{C}_{TR} \equiv f^{\mathcal{R}}_{TR}(C_{TR}) \\ \hline \widetilde{C}_{BL} \equiv f^{\mathcal{R}}_{BL}(C_{BL}) & \widetilde{C}_{BR} \equiv f^{\mathcal{R}}_{BR}(C_{BR}) \end{array} \right).$$

Here, $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$, is the expression for what is known as the *remainder of the loop invariant (the Remainder)*. Again, in $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$, all other operands (including $\widehat{C}_Y$, $\widetilde{C}_Y$ and C_Y where $Y \neq X$) to each $\mathcal{F}_X$ are treated as constants.

3.2.5 Status of Computation

In describing $\mathcal{I}^{\mathcal{F}}$ and $\mathcal{R}_{\mathcal{I}F}$, we did not restrict what f_X and $f_X^{\mathcal{R}}$ must be. If $f_X(\widehat{C}_X) = \widehat{C}_X$, it means that no computation updated C_X in past iterations, and C_X contains its original value. Similarly, if $f_X(\widehat{C}_X) = \mathcal{F}_X(\widehat{C}_X) = \widetilde{C}_X$, it means that the final result already exists in the region C_X and no future computation needs to be performed on C_X. Therefore, for each region of the output, we can describe the *status of computation* for that region in the following manner:

Definition 3.2.1. *The status of computation of a region X of an output C, denoted as $\sigma(C_X)$, takes on one of three values, depending on the relationship between $\mathcal{F}_X$, f_X and $f_X^{\mathcal{R}}$:*

$$\sigma(C_X) = \begin{cases} \text{Fully Updated} & \text{if } \mathcal{F}_X = f_X. \\ \text{Partially Updated} & \text{if } \mathcal{F}_X \neq f_X \text{ and } \mathcal{F}_X \neq f_X^{\mathcal{R}}. \\ \text{Not Updated} & \text{if } \mathcal{F}_X = f_X^{\mathcal{R}} \text{ and } f_X \neq \mathcal{F}_X. \end{cases}$$

Now assume that for all $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, $f_X = \mathcal{F}_X$. This means that $\mathcal{I}^{\mathcal{F}} \equiv \mathcal{P}^{\mathcal{F}}$, which can be interpreted as $C \equiv \widetilde{C} \equiv \widehat{C}$ at the start of the first iteration of the loop. This implies that no computation of C is required. $\mathcal{I}^{\mathcal{F}} \equiv \mathcal{P}^{\mathcal{F}}$ is an example of a loop invariant that is no *feasible*, in that it does not yield a valid loop.

Definition 3.2.2. *A loop invariant is* feasible *if a provably correct algorithm can be derived from it and the computation performed in the loop body is not equivalent to a no-op.*

Generalizing the above example, we can identify the following necessary conditions for a feasible loop invariant:

Theorem 3.2.1. *Every feasible loop invariant for the operation* $C := \mathcal{F}(C, \{A\})$ *must satisfy the following:*

1. *There exists at least one* f_X *such that* $f_X \not\equiv \mathcal{F}_X$.

2. *There exists at least one* $f_X^{\mathcal{R}}$ *such that* $f_X^{\mathcal{R}} \not\equiv \mathcal{F}_X$.

Proof: We prove the conditions separately with proofs by contradiction.

Condition 1: Assume that $f_X = \mathcal{F}_X$ for $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. This means that $\mathcal{I}^{\mathcal{F}} \equiv \mathcal{P}^{\mathcal{F}}$. By definition, $\mathcal{I}^{\mathcal{F}}$ is true at the start of the first iteration of the loop. $\mathcal{I}^{\mathcal{F}} \equiv P^{\mathcal{F}}$ implies that the current value of C at the start of the first iteration is the final value $\widetilde{C}$. Therefore, no update needs to be performed in the loop and hence $\mathcal{I}^{\mathcal{F}} \equiv \mathcal{P}^{\mathcal{F}}$ is not a feasible loop invariant. Therefore, there exist at least one $f_X \neq \mathcal{F}_X$ for $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$.

Condition 2: Assume that $f_X^{\mathcal{R}} = \mathcal{F}_X$ for $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. This means that $\mathcal{I}^{\mathcal{F}} \equiv \widehat{C}$. At the end of the last iteration, $\mathcal{I}^{\mathcal{F}}$ must be true. $\mathcal{I}^{\mathcal{F}} \equiv \widehat{C}$ asserts that at the end of the last iteration, the value stored in C is its original value. If $\widehat{C} \equiv \widetilde{C}$, then nothing needs to be computed in the loop. If $\widehat{C} \not\equiv \widetilde{C}$, then $\mathcal{I}^{\mathcal{F}}$ does not imply the post-condition, which means the derived loop will not compute the correct result. In both cases, $\mathcal{I}^{\mathcal{F}} \equiv \widehat{C}$ is not a feasible loop invariant. Therefore, there exist at least one $f_X^{\mathcal{R}} \neq \mathcal{F}_X$ for $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$.

Since both conditions are true, the theorem is thus proven. □

3.3 Properties of the Remainder

Apart from simply being an expression that states what remaining computation must still be performed, the Remainder has several properties that are relevant to this work. We discuss these properties of the Remainder.

Every loop invariant has a corresponding Remainder. This follows from Theorem 3.2.1. Since for some $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, $f_X^{\mathcal{R}} \neq \mathcal{F}_X$, we know that a loop invariant is an assertion stating that some *proper* subset of the operations described in the PME has been performed. As the set of operations is a proper subset, there must be at least one operation in the PME that is not within that set of operations that forms the loop invariant. Therefore, some computation must still be performed, and thus there must be a Remainder that describes this remaining operation.

The Remainder is true at the start and end of every iteration. The Remainder was obtained by textual substitution of the loop invariant into the PME. This means that whenever the loop invariant is true, the substitution of the loop invariant into the PME will ensure that the resulting Remainder will remain true. Since the loop invariant is true at the start and end of every iteration of the loop, its Remainder must also be true at the start and end of every iteration.

An assertion about computation in future iterations. Recall that the Remainder is obtained by substituting the loop invariant into the PME. When performing this textual substitution, subexpressions in the PME are replaced by the current value of the operands as indicated by the loop invariant. This implies that the Remainder is an expression that computes the condition using the current values of the operands.

Now, our definition of a loop invariant is that it is an assertion about how the current values of the operands have been computed. At the end of an iteration, the current values in the operands must have been computed by either previous iterations or the current iteration. Since the Remainder computes the post-condition using those current values, the computation described by the remainder of the loop invariant could not have been performed in previous or current iterations of the loop. Therefore, the computation represented by the remainder of the loop invariant must be performed in future iterations of the loop.

Thus, the loop can either be derived using the loop invariant or its remainder. We illustrate this by using the Remainder in (3.1) as input to the FLAME derivation methodology. The completed worksheet using the Remainder is shown in Figure 3.1. Unsurprisingly, using either the loop invariant or the Remainder yields the same derived loop. This is because the loop invariant and its Remainder are duals of each other. While the former describes what has already been computed, and the latter describes what has yet to be computed.

Step	**Annotated Algorithm:** $y := Lx$
1a	$\{\widetilde{y} \equiv Lx\}$
4	**Partition** $y \rightarrow \left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right)$, $L \rightarrow \left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $x \rightarrow \left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right)$ **where** y_T has 0 rows, L_{TL} is 0×0, x_T has 0 rows
2	$\left\{\left(\begin{array}{c} \widetilde{y}_T \equiv y_T \\ \hline \widetilde{y}_B \equiv L_{BL}x_T + L_{BR}x_B \end{array}\right)\right\}$
3	**while** $m(y_T) < m(y)$ **do**
2,3	$\left\{\left(\left(\begin{array}{c} \widetilde{y}_T \equiv y_T \\ \hline \widetilde{y}_B \equiv L_{BL}x_T + L_{BR}x_B \end{array}\right)\right) \wedge (m(y_T) < m(y))\right\}$
5a	**Repartition** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \rightarrow \left(\begin{array}{c} y_0 \\ \hline \psi_1 \\ y_2 \end{array}\right)$, $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \rightarrow \left(\begin{array}{c} x_0 \\ \hline \chi_1 \\ x_2 \end{array}\right)$ **where** λ_{11} is 1×1
6	$\left\{\left(\begin{array}{c} \widetilde{y}_0 \equiv y_0 \\ \hline \widetilde{\psi}_1 \equiv L_{01}x_0 + \lambda_{11}\chi_1 \\ \widetilde{y}_2 \equiv L_{20}x_0 + l_{21}\chi_1 + L_{22}x_2 \end{array}\right)\right\}$
8	$\psi_1 := L_{01}x_0 + \lambda_{11}\chi_1$
5b	**Continue with** $\left(\begin{array}{c} y_T \\ \hline y_B \end{array}\right) \leftarrow \left(\begin{array}{c} y_0 \\ \psi_1 \\ \hline y_2 \end{array}\right)$, $\left(\begin{array}{c\|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & 0 & 0 \\ l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$, $\left(\begin{array}{c} x_T \\ \hline x_B \end{array}\right) \leftarrow \left(\begin{array}{c} x_0 \\ \chi_1 \\ \hline x_2 \end{array}\right)$
7	$\left\{\left(\begin{array}{c} \widetilde{y}_0 \equiv y_0 \\ \widetilde{\psi}_1 \equiv \psi_1 \\ \hline \widetilde{y}_2 \equiv L_{20}x_0 + l_{21}\chi_1 + L_{22}x_2 \end{array}\right)\right\}$
2	$\left\{\left(\begin{array}{c} \widetilde{y}_T \equiv y_T \\ \hline \widetilde{y}_B \equiv L_{BL}x_T + L_{BR}x_B \end{array}\right)\right\}$
	endwhile
2,3	$\left\{\left(\left(\begin{array}{c} \widetilde{y}_T \equiv y_T \\ \hline \widetilde{y}_B \equiv L_{BL}x_T + L_{BR}x_B \end{array}\right)\right) \wedge \neg (m(y_T) < m(y))\right\}$
1b	$\{\widetilde{y} \equiv y\}$

Figure 3.1: Completed FLAME Worksheet using the Remainder as input

Remark 1. *In our experience, it is sometimes better to use the Remainder as input to the FLAME derivation methodology. Using the Remainder can simplify the derivation process because the expression for the Remainder may be simpler than that for the loop invariant. An example of such a situation is described in* Poulson et al *[44].*

An assertion about future operands. Analyzing the loop invariant tells us what operations must have been performed in the past. In addition, regions of the (input and output) operands present in the loop invariant represent regions of data that are required for computation in past iterations. Similarly, analyzing the remainder of a loop invariant will yield information on the operations that must be performed in the future and the regions of data that will be required in future iterations.

3.4 Summary

In this chapter, we introduced the notion of a *remainder of a loop invariant*. The Remainder is the dual of the loop invariant in that the loop invariant describes computation that was performed in past iterations while its remainder describes computation that will be performed in future iterations of the same loop.

More importantly, the key insight in this chapter is the recognition that the loop invariant and its remainder describe computation in all iterations of the derived loop. This motivates the need to analyze both expressions

in order to determine the characteristic(s) of the derived loop. As the loop invariant and its remainder represent computation in past and future iterations respectively, analyzing both expressions will yield information about past and future computation.

In the rest of the dissertation, we will show that the analysis of both the loop invariant and its remainder yield information about all computations performed by the loop, and it is through these information about past and future computation that characteristics of the derived loop can be identified.

Chapter 4

Analysis at a Higher Level of Abstraction

Compilers require loop-based algorithms to be implemented as loop-based code with explicit indexing before dependence analysis can be performed. Inherently, when implementing an algorithm, knowledge about alternative algorithms is lost or obscured. In this chapter, we show that by using the status of computation of the different regions, the analysis performed by compilers can now be performed at a higher level of abstraction, namely at the level of the PME, loop invariant and its remainder, while retaining information that would otherwise be lost or obscured in the representation of the algorithm in code.

4.1 Types of Dependences

We begin with a discussion on the types of data dependence that are identified by compilers during dependence analysis.

4.1.1 True (Flow) dependence

True dependences, also known as flow dependences, are dependences that occur when a piece of data is computed by one statement and then read

subsequently by another statement in the future or a future instance of the same statement (read-after-write).

Consider the following code snippet:

```
C = A * B
D = E * C
```

Notice that `C` is computed by the first statement and then read by the next statement. This creates a dependence between the two statements such that the order of the statements cannot be swapped. If the statements are swapped, then in computing the value of `D`, a potentially wrong value of `C` will be used.

4.1.2 Anti-dependence

An anti-dependence occurs when a piece of data is read and then subsequently overwritten some time in the future.

Consider the following snippet of code:

```
D = E * C
C = A * B
```

The original value of `C` is used to compute `D`. That value is then subsequently overwritten by the second statement. This is a write-after-read situation where again the statements cannot be swapped. Notice that anti-dependence is a form of false dependence that can be eliminated with the use of additional

memory. By storing the original value of C in another variable, the two statements can now be swapped. This is because the original value of C is then not destroyed.

4.1.3 Output dependence

The last form of data dependence that is important when optimizing code is output dependence, which occurs in a write-after-write situation. The dependence between two statements is of the output dependence type when the two statements update the same piece of data.

We illustrate output dependence with the following example:

```
C = D * E
    ⋮
F = D * C
    ⋮
C = A * B
```

Notice that both statements update C. Any statement between the two statements (e.g. the statement F = D * C in the code segment above) that reads the value of C, will read that value computed by the first statement. Statements after the second statement will read the second computed value. Again, these statements cannot be swapped. This is because any reads of C between the two statements will yield a different value for C if the two statements are swapped. Similarly, reads of C after the second statement will also be reading

a the first computed value if the statements were swapped. This is another form of false dependence where the use of additional memory can be used to break the dependence.

4.2 Loop-carried Dependence

An orthogonal type of dependence that a compiler extracts during dependence analysis is loop-carried dependence. A loop-carried dependence is a dependence (regardless of whether it is a true dependence, anti-dependence, or output dependence) that exists between statements in two different iterations of the loop. We illustrate loop-carried dependence with the following example:

```
for (i = 1; i < n; ++i){
  a[i] += a[i-1];
  b[i] = a[i];
}
```

Notice that in order to compute `a[i]`, `a[i-1]` must already have been computed in past iterations. Since this is a read-after-write scenario, the dependence is a true dependence. In addition, because `a[i-1]` was computed in a previous iteration, this true dependence between two different instances of the first statement is a loop-carried dependence.

Contrast this with the true dependence between the first and second statement in the loop body. The dependence between these two statements occurs within the same iteration. Hence, this dependence is not a loop-carried dependence.

4.3 Analysis of $\mathcal{P}^{\mathcal{F}}$, $\mathcal{I}^{\mathcal{F}}$ and $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$

In this section, we show how the different types of dependences described previously can be identified at a higher level of abstraction, namely from the PME, the loop invariant and its remainder.

4.3.1 Dependencies in $\mathcal{P}^{\mathcal{F}}$

Data dependences occur when a piece of computed data is required in a subsequent computation (true dependence), or when data required is overwritten in subsequent operation(s) (anti-dependence, output dependence). This suggests that our analysis needs only to focus on data whose values are changed by the operation.

Recall that for a given operation,

$$C := \mathcal{F}(C, \{A\}),$$

only C is updated. In addition, the PME, $\mathcal{P}^{\mathcal{F}}$, describes the operations performed with the different regions of elements of the set $\{A\} \cup C$ in order to compute different regions of C. Therefore, we start our analysis of $\mathcal{P}^{\mathcal{F}}$ by differentiating between uses of the original value ($\widehat{C}$), and the final value ($\widetilde{C}$) of regions of C.

True Dependence. Let C_X, C_Y be arbitrary regions of C. If $\widetilde{C}_Y$ is required to compute C_X (i.e. $\widetilde{C}_Y$ appears in $\mathcal{F}_X$), then it must be the case that $\widetilde{C}_Y$ is computed before it can be used to compute C_X. Therefore, there must be

a true dependence between $\mathcal{F}_Y$, which computes $\widetilde{C}_Y$, and $\mathcal{F}_X$, which requires the value of $\widetilde{C}_Y$.

Anti-dependence. Let C_X, C_Y be arbitrary regions of C. If $\widehat{C}_Y$ is required to compute C_X, then it must be the case that the value of $\widehat{C}_Y$ must be preserved until the computation of C_X no longer requires the use of $\widehat{C}_Y$. Once $\widehat{C}_Y$ is no longer required to compute C_X, C_Y can be updated and overwritten. Since this is a write-after-read situation, the dependence between the functions $\mathcal{F}_Y$ and $\mathcal{F}_X$ must be an anti-dependence.

Output dependence. Recall that an output dependence occurs when the same piece of data is updated by two separate computations. From $\mathcal{P}^{\mathcal{F}}$, we know that an arbitrary region C_X is only updated by $\mathcal{F}_X$. There cannot be any output dependences between $\mathcal{F}_X$ and $\mathcal{F}_Y$ when $X \neq Y$, as both functions update different regions.

4.3.2 An illustration

We illustrate the presence of true and anti-dependences with a concrete example: the inversion of a triangular matrix (TRINV). The TRINV operation can be mathematically described as

$$L := L^{-1}, \text{ where } L \text{ is lower triangular.}$$

One possible PME for the TRINV operation is as follows:

$$\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline \widetilde{L}_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widetilde{L}_{TL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right).$$

Notice that there is a true dependence between L_{TL} and L_{BL}, as $\widetilde{L}_{TL}$ is required to compute L_{BL}. Hence, L_{TL} must be computed before L_{BL}, so that the value $\widetilde{L}_{TL}$ is available for use during the computation of L_{BL}.

There is also an anti-dependence between L_{BL} and L_{BR}. In order to compute $\widetilde{L}_{BL}$, the original content of L_{BR}, $\widehat{L}_{BR}$, is required. If $\widetilde{L}_{BR}$ was computed before $\widehat{L}_{BR}$ was used to compute $\widetilde{L}_{BL}$, the value of $\widehat{L}_{BR}$ would be overwritten and the computed value of $\widetilde{L}_{BL}$ would be incorrect. Hence, with the PME given above, the anti-dependence requires that L_{BL} be either fully computed or partially computed before the computation of $\widetilde{L}_{BR}$ commences.

4.3.3 Loop-carried dependences in $\mathcal{I}^{\mathcal{F}}$ and $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$

Having identified both true and anti-dependences in the PME, we show how to determine if the identified dependences are loop-carried.

Recall that each function $\mathcal{F}_X$, where $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, in $\mathcal{P}^{\mathcal{F}}$ can be expressed in terms of two functions f_X and $f_X^{\mathcal{R}}$ in the following manner

$$\mathcal{F}_X(\widehat{C}_X) \equiv f_X^{\mathcal{R}}(f_X(\widehat{C}_X)),$$

where f_X and $f_X^{\mathcal{R}}$ represents computations in past and future iterations.

Since loop-carried dependences are dependences between computations in two different iterations, it must mean that the dependence is between a computation performed in a previous iteration and another computation that will be carried out in a future iteration. This means that if $\widetilde{C}_Y$ is required in order to compute C_X, where $X, Y \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, and $X \neq Y$, and that

dependence is a loop-carried dependence, then $\widetilde{C}_Y$ must have been computed in the past and $\widetilde{C}_Y$ will be required in the future. This means that $\widetilde{C}_Y$ must be appear in $f_X^{\mathcal{R}}$.

Similarly, if the loop-carried dependence is an anti-dependence, then $\widehat{C}_Y$ must be read in the past and overwritten in the future. Therefore, $\widehat{C}_Y$ must appear in f_X where $X \neq Y$, $X, Y \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. In addition, if no computation was performed in the past or no computation will be performed in the future, then any dependences between C_X and C_Y cannot be loop-carried. This observation suggest the following theorem:

Theorem 4.3.1. *Let C_X and C_Y be two different regions of the PME and assume that there exists a dependence between C_X and C_Y such that C_X must be computed before C_Y. If either $\sigma(C_X) = \sigma(C_Y) =$ FULLY UPDATED, or $\sigma(C_X) = \sigma(C_Y) =$ NOT UPDATED, then the dependence is not loop-carried.*

Proof: We prove this theorem with a proof-by-cases, where each case is proven by contradiction.

Case 1: If $\sigma(C_X) = \sigma(C_Y) =$ FULLY UPDATED, then dependence is not loop-carried.

Assume $\sigma(C_X) = \sigma(C_Y) =$ FULLY UPDATED, and the dependence *is* loop-carried. Since $\sigma(C_X) = \sigma(C_Y) =$ FULLY UPDATED, no computation will be performed on either C_X or C_Y in the future. Since a loop-carried dependence is a dependence between a computation from a

past iteration and a computation in a future iteration, and no computation will be performed on either C_X or C_Y in the future, the dependence cannot be a loop-carried dependence. Hence a contradiction is obtained.

Case 2: If $\sigma(C_X) = \sigma(C_Y) =$ NOT UPDATED*, then dependence is not loop-carried.*

Assume $\sigma(C_X) = \sigma(C_Y) =$ NOT UPDATED, and the dependence is loop-carried. Since $\sigma(C_X) = \sigma(C_Y) =$ NOT UPDATED, then no computation was performed on either C_X or C_Y in the past. Since a loop-carried dependence is a dependence between a computation from a past iteration and a computation in a future iteration, and no computation was performed on either C_X or C_Y in past iterations, the dependence cannot be a loop-carried dependence. Hence a contradiction is obtained.

In both cases, a contradiction was obtained and hence, the loop dependence cannot be loop-carried if $\sigma(C_X) = \sigma(C_Y) =$ NOT UPDATED or $\sigma(C_X) = \sigma(C_Y) =$ FULLY UPDATED. □

So far, the discussion about loop-carried dependence assumes that C_X and C_Y are different regions of the output. We now examine the situation where $X = Y$.

Recall that f_X and $f_X^{\mathfrak{R}}$ are functions that update C_X in the past and future respectively. If f_X and $f_X^{\mathfrak{R}}$ both update C_X, this is a write-after-write situation which implies that there must be an output dependence between the two functions.

Theorem 4.3.2. *An output dependence exists if and only if* $\sigma(C_X) =$ Partially Updated.

Proof: An output dependence exists if and only if f_X and $f_X^{\mathcal{R}}$ both update C_X. Since f_X updates C_X if and only if $\sigma(C_X) \neq$ Not Updated, and C_X is updated by $f_X^{\mathcal{R}}$ if and only if $\sigma(C_X) \neq$ Fully Updated, then there exist an output dependence if and only if $\sigma(C_X) \neq$ Fully Updated $\wedge\, \sigma(C_X) \neq$ Not Updated. Since

$$\left(\begin{array}{c} \sigma(C_X) \neq \text{Fully Updated} \wedge \\ \sigma(C_X) \neq \text{Not Updated} \end{array}\right) \equiv (\sigma(C_X) = \text{Partially Updated}),$$

the theorem is proven. □

In addition, because the output dependence is between f_X and $f_X^{\mathcal{R}}$, this dependence is also loop-carried.

4.3.4 An additional illustration

We return to our Trinv operation to illustrate loop-carried dependence. Consider the following loop invariant and its remainder:

$$\begin{aligned} \mathcal{I}^{\text{Trinv}} &= \left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widetilde{L}_{TL} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right); \\ \mathcal{R}_{\mathcal{I}^{\text{Trinv}}} &= \left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv L_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv L_{BL} & \widetilde{L}_{BR} \equiv L_{BR}^{-1} \end{array}\right) \end{aligned}$$

We start with the true dependence and anti-dependence identified previously in Section 4.3.2. Since $\widetilde{L}_{TL}$ appears in f_{BL} and $\widetilde{L}_{TL}$ was computed by f_{TL} and $f_X = \mathcal{F}_{TL}$, this implies that the true dependence is *NOT* between a

computation in the past and a computation that will be performed in the future. Therefore, the true dependence is not loop-carried.

Similarly, recall that $\widehat{L}_{BR}$ is required to compute L_{BL}. Since $\widehat{L}_{BR}$ is required in past iterations ($\widehat{L}_{BR}$ appears in $\mathcal{I}^{\text{Trinv}}$), and $\widetilde{L}_{BR}$ is computed in future iterations, it follows that this anti-dependence is loop-carried.

Finally, because the status of computation for all regions of L are either Fully Updated or Not Updated, this implies that there is no output dependence in the loop represented by this pair of loop invariant and Remainder.

The following Matlab implementation of the loop with the above loop invariant illustrates the dependences identified through our analysis:

```
for i=1:n
    % LTL = LTL^{-1}
    L(i,i) = 1/L(i,i);

    % LBL = -LBR^{-1}*LBL*LTL
    L(i+1:n,i) = -L(i+1:n,i+1:n) \ L(i+1:n,i) * L(i,i);
end
```

The true dependence described by the loop invariant corresponds to the true dependence between the first and second statement in the loop body. During each iteration, `L(i,i)` is first computed by the first statement and then used in the second statement. Hence, the true dependence is not loop-carried. As `L(i+1,i+1)` is an element of the region `L(i+1:n,i+1:n)`, `L(i+1,i+1)` is used to compute `L(i+1:m,i)`. In addition, as `L(i+1,i+1)` will be overwritten by the first statement of the next iteration, this is a loop-carried anti-dependence. Finally, as each iteration only updates elements in the `i`-th column of the ma-

trix `L`, we know that no single element is computed in two different iterations. Hence, there cannot be an output dependence.

4.4 Summary

In this chapter, we demonstrated how within the domain of DLA, dependence analysis performed on code by traditional compilers can be performed at a higher level of abstraction on the PME $\mathcal{P}^{\mathcal{F}}$, loop invariant $\mathcal{I}^{\mathcal{F}}$ and its remainder $\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$.

By analyzing the expression for $\mathcal{P}^{\mathcal{F}}$, we show that true dependences and anti-dependences can be identified. Since $\mathcal{P}^{\mathcal{F}}$ describes all required computations, and these computations are divided up between the loop invariant and remainder, the dependences between the operations must be present in all pairs of loop invariants and remainders that are obtained. Hence, a single analysis performed on the PME allows us to analyze multiple pairs, consisting of a loop invariant and its remainder, and hence multiple algorithms at once. This preserves information about other algorithms that would have been lost or obscured if a chosen algorithm was implemented as code.

In addition, we presented a theory for identifying loop-carried dependences from the status of computations of the different regions. Since the functions f_X and $f_X^{\mathcal{R}}$ describe computations in past and future iterations respectively, a dependence between f_X and $f_Y^{\mathcal{R}}$ for any X and Y describes a dependence between computation in the past and in the future. This means that this dependence is a loop-carried dependence.

The insights and theory developed in this chapter allows compiler-like dependence analysis to be performed without having to analyze code. This allows dependence analysis to be applied even if the implementation uses subroutine calls to black-box libraries, as is commonly the case in the domain of DLA. **Indeed, the theory allows dependence analysis to be performed even before the algorithm has been derived!**

Chapter 5

Independent Iterations

A common way in which a DLA operation is parallelized is to divide the tasks performed by the loop amongst the processors such that each processor computes the different tasks independently [33, 2]. To implement such a parallelization scheme, these tasks have to be independent of each other. If a task is defined as all the updates that occur in an iteration of the loop, then tasks are independent when the iterations of the loop are independent.

5.1 Characteristics of Loop Invariants for Loops with Independent Iterations

Iterations are independent when a given iteration of the loop neither uses data computed by any other iteration in the loop, nor overwrites data that will be used by other iterations of the loop. In other words, there must not be any loop-carried dependence [2, 41].

Recall that, in Chapter 4, the theory behind identifying loop-carried dependence from the loop invariant and its remainder was developed for our prototypical operation. We thus begin our analysis for independent iterations with a simple lemma for our prototypical operation as described in Section 3.2.1:

Lemma 5.1.1. *If a loop has independent iterations, then for all* $X \in \{\mathsf{TL},\ \mathsf{TR},\mathsf{BL},\ \mathsf{BR}\}$*,* $\sigma(C_X) \neq$ *PARTIALLY UPDATED.*

Proof: We prove this lemma by contradiction. Assume a loop has independent iterations but there exists a region C_X such that $\sigma(C_X) =$ PARTIALLY UPDATED. Theorem 4.3.2 tell us that $\sigma(C_X) =$ PARTIALLY UPDATED implies that there exists an output dependence between a computation performed on C_X in the past, and a computation that will be performed on C_X in the future. Since the dependence is between a computation in the past and a computation in the future, this output dependence is loop-carried and hence, the iterations of the loop are not independent. Hence, a contradiction is obtained. Therefore, $\sigma(C_X) \neq$ PARTIALLY UPDATED if a loop has independent iterations. □

Consider the operation $x := Lx$ (TRMV), where L is lower triangular. This operation performs the same computation as TRMV-X, but the result is written back into x. Here, the assumption is that no temporary storage will be used. The PME of this operation and one of the loop invariants for computing it are given by:

$$\mathcal{P}^{\text{TRMV}} = \left(\frac{\widetilde{x}_T \equiv \widehat{L}_{TL}\widehat{x}_T}{\widetilde{x}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B} \right); \quad \mathcal{I}^{\text{TRMV}} = \left(\frac{x_T \equiv \widehat{x}_T}{x_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B} \right).$$

Recall that $\widehat{x}_T$ represents the original value of x_T. Notice that $\widehat{x}_T$ is required to compute both $\widetilde{x}_T$ and $\widetilde{x}_B$. In addition, when x_T is computed, $\widehat{x}_T$ will be overwritten. Therefore, $\widetilde{x}_B$ has to be computed before $\widetilde{x}_T$ in order to ensure that $\widehat{x}_T$ remains available. This implies that there is a dependence between the regions x_T and x_B and this dependence is an anti-dependence.

Given that the regions are now dependent, if x_T and x_B are computed in different iterations through the loop, the iterations will be dependent. This observation suggests the following lemma:

Lemma 5.1.2. *Let X and Y be two regions of the output and there exists a dependence between X and Y such that X needs to be computed before Y. If a loop has independent iterations, then regions X and Y are both either fully updated or not updated.*

Proof: We prove this via a proof by contradiction.

Assume that the loop has independent iterations but there exists regions X and Y that are neither both fully updated nor both not updated. Since the loop has independent iterations, Lemma 5.1.1 tells us that both $\sigma(X)$ and $\sigma(Y)$ cannot be partially updated. This implies that $\sigma(X) \neq \sigma(Y)$, since they are neither both fully updated nor both not updated.

Without loss of generality, assume that $\sigma(X) = $ FULLY UPDATED and $\sigma(Y) = $ NOT UPDATED. This means that $f_X = F_X$ and $f_Y^{\mathcal{R}} = F_Y$. This implies that X was computed in past iterations while Y will be computed in future iterations. Since there exists a dependence between X and Y, it follows that the iteration(s) that compute(s) Y must depend on the the iteration(s) that compute(s) X, and so the loop cannot have independent iterations which contradicts our initial assumptions. Hence, the lemma is proven. $\square$

Both Lemmas 5.1.1 and 5.1.2 are necessary conditions for a loop to have independent iterations. We prove that these conditions are also sufficient to

determine when a loop has independent iterations, with the following theorem:

Theorem 5.1.1. *A loop has independent iterations if and only if the following conditions are met:*

1. *No region of the loop invariant has a partially updated status, and*
2. *If there exists a dependence between regions X and Y, then the status of X must be the same as the status of Y.*

Proof: We prove the theorem in two parts:

Part 1: If a loop has independent iterations, then the loop invariant has the two characteristics.

This part follows the proofs for Lemmas 5.1.1 and 5.1.2.

Part 2: If a loop invariant has the two characteristics, then its loop has independent iterations.

We prove this direction with a proof by contradiction. Assume that the loop invariant has the required two characteristics but there exists at least one pair of iterations that are not independent. This implies that there is at least a loop-carried dependence.

Since no regions in the loop invariant are partially updated, then Theorem 4.3.2 tells us that there are no output dependences. This implies that any dependence is either a true dependence or an anti-dependence. Furthermore, as two regions that are dependent (either via true dependence

or anti-dependence) have the same status of computation, Theorem 4.3.1 tells us that that dependence is not loop-carried. This contradicts our assumptions and thus the loop must have independent iterations.

Since both directions are proven, the theorem holds. □

5.2 Application of Theory

We examine our complex example of the inversion of a triangular matrix (TRINV) to determine if there are algorithms with independent iterations. For convenience, we repeat the post-condition, $\mathcal{P}^{\text{TRINV}}$, below:

$$\mathcal{P}^{\text{TRINV}} = \left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline \widetilde{L}_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array} \right). \tag{5.1}$$

Notice that L_{BL} requires $\widehat{L}_{BR}$ and $\widehat{L}_{TL}$. This means that all three regions of L are dependent. If an algorithm with a loop invariant derived from $\mathcal{P}^F$ has independent iterations, then Theorem 5.1.1 requires that $\sigma(L_{TL}) = \sigma(L_{BL}) = \sigma(L_{BR}) \neq$ PARTIALLY UPDATED

However, $\sigma(L_{TL})$ = FULLY UPDATED or $\sigma(L_{TL})$ = NOT UPDATED implies that $\widehat{C} = \widetilde{C}$. This means that there are no feasible loop invariants from $\mathcal{P}^{\text{TRINV}}$ that have independent iterations. Therefore, *NONE* of the loop invariants obtained by removing operations from the PME in (5.1) yields an algorithm with independent iterations.

If, instead of overwriting L, the result was stored into another matrix

T, then the PME for this operation can be expressed as:

$$\mathcal{P}^{\text{TRINV-ALT}} = \left(\begin{array}{c|c} \widetilde{T}_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline \widetilde{T}_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & \widetilde{T}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array} \right).$$

In this situation, we know that there cannot be any true dependence or anti-dependence because every region T_X of T does not require either $\widehat{T}_Y$ or $\widetilde{T}_Y$ where $X \neq Y$. This also implies that Condition 2 of Theorem 5.1.1 is met.

Condition 1 of Theorem 5.1.1 tells us that a loop invariant for TRINV-ALT represents an algorithm with independent iterations when no regions are partially updated. Hence, the loop invariants represent algorithms with independent iterations can be easily identified. These loop invariants are shown in Figure 5.1.

5.3 Summary

In this chapter, we developed a theory that allows one to determine, from an analysis of the loop invariant, whether a loop has independent iterations. In addition, we also demonstrated the effectiveness of performing an analysis at a higher level of abstraction. A single analysis of the post-condition allowed us to determine that there are no algorithms, whose loop invariant was derived from the post-condition, with independent iterations.

	$\mathcal{I}^{\text{Trinv}}$	$\mathcal{R}_{\mathcal{I}\text{Trinv}}$
1	$\left(\begin{array}{c\|c} T_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv \widehat{T}_{BL} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \widetilde{T}_{TL} \equiv T_{TL} & 0 \\ \hline \widetilde{T}_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL} & \widetilde{T}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)$
3	$\left(\begin{array}{c\|c} T_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \widetilde{T}_{TL} \equiv T_{TL} & 0 \\ \hline \widetilde{T}_{BL} \equiv \widehat{T}_{BL} & \widetilde{T}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)$
5	$\left(\begin{array}{c\|c} T_{TL} \equiv \widehat{T}_{TL} & 0 \\ \hline T_{BL} \equiv \widehat{T}_{BL} & T_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)$	$\left(\begin{array}{c\|c} \widetilde{T}_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline \widetilde{T}_{BL} \equiv -\widehat{L}_{BR}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & \widetilde{T}_{BR} \equiv T_{BR} \end{array}\right)$
7	$\left(\begin{array}{c\|c} T_{TL} \equiv \widehat{T}_{TL} & 0 \\ \hline T_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & T_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)$	$\left(\begin{array}{c\|c} \widetilde{T}_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline \widetilde{T}_{BL} \equiv -\widehat{L}_{BR}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & \widetilde{T}_{BR} \equiv T_{BR} \end{array}\right)$

Figure 5.1: Loop invariants and remainders that represent algorithms with independent iterations for the TRINV-ALT operation.

Chapter 6

Merging DLA Loops

Merged operations are commonly encountered in the domain of DLA. A merged operation is an operation that provides the same functionality as multiple operations performed in sequence. In this chapter, we describe how loop-based algorithms for computing merged operations can be identified from the loop invariants and their remainders of loop-based algorithms for the separate operations.

6.1 Motivation

We illustrate the benefits of merged operations with an example that is representative of many situations in DLA:

$$\begin{array}{ll} y := Lx & (\textsc{Trmv-x}) \\ x := L^T y & (\textsc{Trmv-trans}). \end{array}$$

In the sequence of operations shown above, the vector y is first computed by the first operation (Trmv-x) and subsequently used in the second operation (Trmv-trans) to compute x. Notice that the separate operations will access L twice. Since matrix-vector operations consists of $O(n^2)$ floating point operations and (On^2) memory accesses, and memory accesses are orders of magnitude more costly than floating point operations, reducing the number

of times L is read from memory typically improves the performance of that sequence of operations [51, 47]. On distributed memory architectures, merged operations can also reduce communication cost and/or improve load-balance [45, 8].

A merged operation that replaces the separate operations TRMV-X and TRMV-TRANS is:

$$y := Lx;\ x := L^T y \quad (\text{TRMV-MERGE}),$$

which computes y and x at the same time. In other words, the two separate operations are replaced with a merged operation that reduces the number of times L is read, hence improving temporal locality and thus, improving performance.

6.2 Loop Fusion

Recall that DLA algorithms are typically implemented in terms of loops. Merged operations can, thus, be created by the application of a loop transformation, called loop fusion [31, 37], onto the separate loops for the separate operations that we want to merged. The result of loop fusion is that multiple (in this case, two) loops are merged into a single loop. Algorithms for computing the operations TRMV-X, TRMV-TRANS, and the merged operation TRMV-MERGE are given in Figure 6.1. Notice that the update statements of algorithms that compute TRMV-MERGE in Figure 6.1 are composed from the update statements of the separate algorithms concatenated in the exact

sequence in which the separate operations were performed.

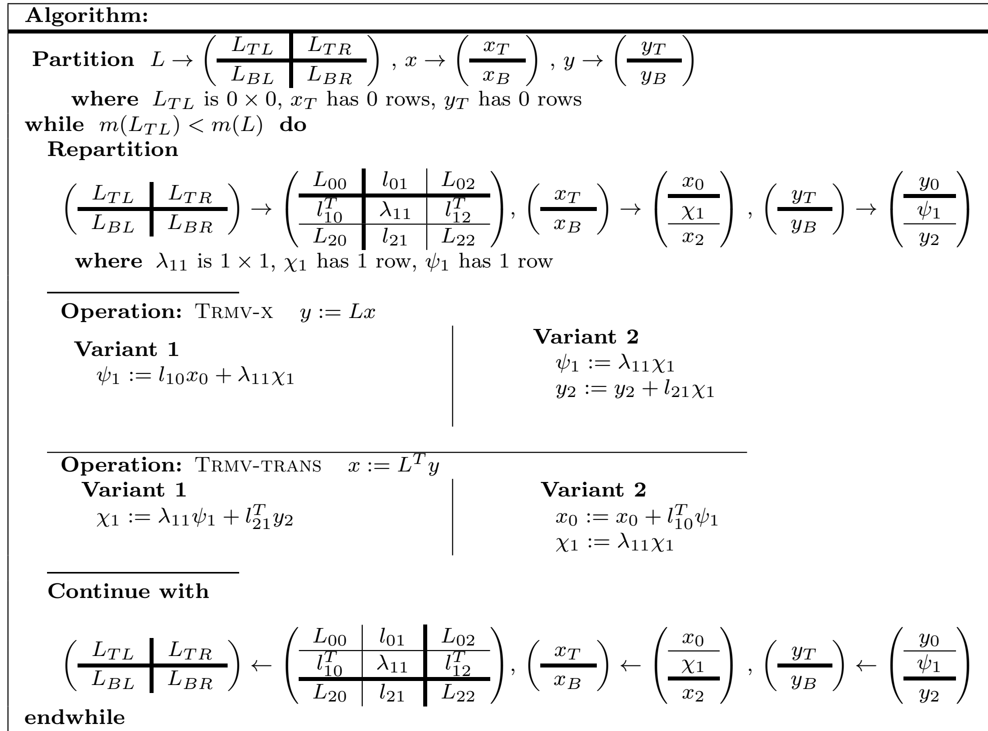
6.2.1 Loop fusion for DLA in practice

While loop fusion is a loop transformation that can be automatically performed by existing compilers, a number of hurdles hinders the wide-spread application of loop fusion in the domain of DLA.

As most high performance implementation in DLA are often implemented in terms of subroutine calls to highly optimized black-box libraries such as the Basic Linear Algebra Subroutines (BLAS), LAPACK and ScaLAPACK[11], this means that compilers are limited in their ability to analyze the implementation. This reduces the opportunity for loop transformations, including loop fusion, to be applied automatically by the compiler.

When a compiler is given implementations of the separate operations that cannot be merged, other loop transformations are required to be applied to one or both of the implementations. After a loop transformation is performed, the newly generated implementations are analyzed to determine if they can be merged. If not, the process is repeated. Since compilers apply loop transformations in phases, the incorrect ordering of the transformation phases could potentially prevent compatible implementations from being identified.

Furthermore, not all merged implementations are beneficial. This means that multiple merged implementations must be identified in order to determine the merged implementation that is most beneficial. This exacerbates the phase-ordering problem since the compiler now has to identify multiple



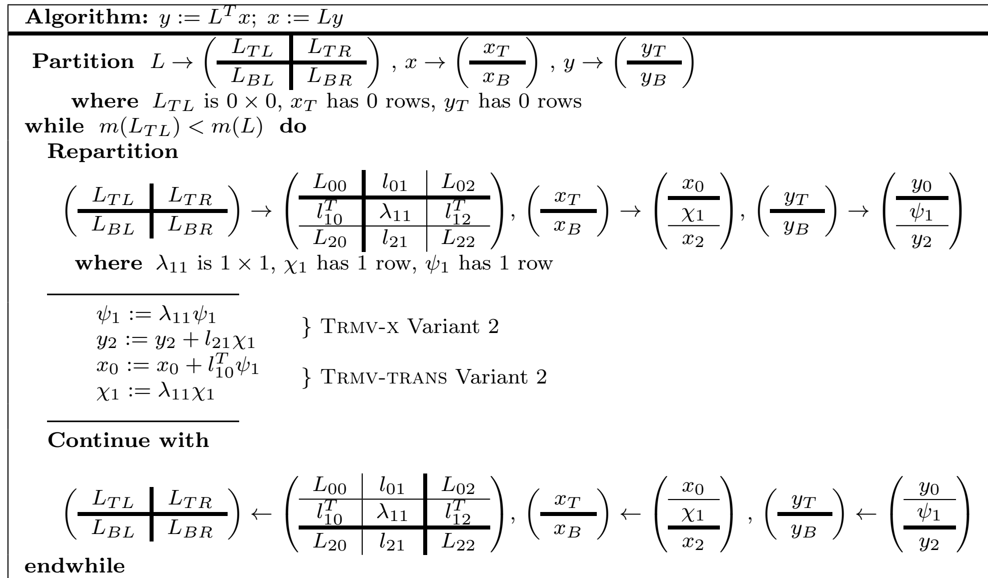


Figure 6.1: Top: Algorithms for the separate operations TRMV-X and TRMV-TRANS, and Bottom: Algorithm for the merged operation TRMV-MERGE

.

merged implementations.

Often, loop fusion is performed manually by the DLA expert. This overcomes some of the hurdles, such as implementations using subroutine calls, encountered by the compiler. However, this process is laborious and time-consuming. For each pair of operations that is to be merged, the DLA expert derives multiple algorithms that compute the separate operations. Next, each possible pair of algorithms is analyzed to determine if the algorithms can be merged. As there are often many algorithms that compute each of the separate operations, there is a combinatorial number of ways in which the separate algorithms can potentially be merged. As a result, a large number of analyses must be performed to determine if algorithms can be merged.

6.3 Characteristics of Desired Loop Invariants

Let us examine the separate operations TRMV-X and TRMV-TRANS to gain insights into loop fusion. For convenience, the separate operations are repeated below:

$$\begin{array}{lcll} y & := & Lx & \text{TRMV-X} \\ x & := & L^T y & \text{TRMV-TRANS}. \end{array}$$

Notice that there exist two types of dependence between the two operations. Firstly, y is computed by the first operation and then subsequently used. This is a true dependence. The other dependence between the two operations is an anti-dependence as x is used by the first operation and then subsequently overwritten. These dependences mean that (1) any algorithm that computes TRMV-X must use the original value $\widehat{x}$, and (2) any algorithm that computes

Trmv-trans must use the computed value of $\widetilde{y}$ in order to ensure that the two algorithms compute the correct result.

Now, let us examine the loop invariant and remainder of an algorithm for each of the two separate operations to determine if they can be merged. The loop invariants and remainders for Trmv-x and Trmv-trans are given below:

$$F = \text{Trmv-x}$$

$$\mathcal{I}^F = \left(\frac{y_T \equiv \widehat{L}_{TL}\widehat{x}_T}{y_B \equiv \widehat{L}_{BL}\widehat{x}_T} \right), \qquad \mathcal{R}_{\mathcal{I}^F} = \left(\frac{\widetilde{y}_T \equiv y_T}{\widetilde{y}_B \equiv y_B + \widehat{L}_{BR}\widehat{x}_B} \right)$$

$$G = \text{Trmv-trans}$$

$$\mathcal{I}^G = \left(\frac{x_T \equiv \widehat{L}_{TL}^T\widetilde{y}_T + \widehat{L}_{BL}^T\widetilde{y}_B}{x_B \equiv \widehat{x}_B} \right), \quad \mathcal{R}_{\mathcal{I}^G} = \left(\frac{\widetilde{x}_T \equiv x_T}{\widetilde{x}_B \equiv \widehat{L}_{BR}^T\widetilde{y}_B} \right).$$

Observe that $\mathcal{R}_{\mathcal{I}^F}$ indicates that $\widehat{x}_B$ will be used in future iterations to compute $\widetilde{y}_B$. This implies that at the start of an iteration, $\widehat{x}_B$ must not have been overwritten. Fortunately, $\mathcal{I}^G$ asserts that at the start of any iteration, $\sigma(x_B) =$ Not Updated, which implies that $x_B = \widehat{x}_B$. Hence, the anti-dependence between the two operations is preserved.

Similarly, we can use the status of computation to determine if the true dependence is preserved. The loop invariant $\mathcal{I}^G$ asserts that $\widetilde{y}_B$ was used in past iterations to compute x_T. In addition, an examination of either $\mathcal{I}^G$ or $\mathcal{R}_{\mathcal{I}^F}$ states that $\sigma(y_B) \neq$ Fully Updated. This means that when loop fusion is applied to the two algorithms, the value of y_B read by the algorithm for Trmv-trans has not been fully computed. Therefore, the true dependence between the two operations is not preserved. This implies that loop fusion,

when applied to the loops described by the loop invariants and remainders described above, is not legal.

6.4 Generalization

Let us generalize the above observations. Assume that there is a sequence of two operations

$$\begin{array}{l} \widetilde{B} := \mathcal{F}(\widehat{B}, \widehat{C}, \{A\}) \\ \widetilde{C} := \mathcal{G}(\widehat{C}, \widetilde{B}, \{D\}) \end{array}$$

that we would like to merge. Recall that $\widehat{B}$ and $\widetilde{C}$ represent the initial and final values of B and C respectively. As with our prototypical operation, described in Section 3.2.1, $\{A\}$ and $\{D\}$ represent operands that are read but not written.

It should be noted that it is possible that C and B are not used in the first and second operation, respectively, or that C and B are one and the same matrix. In the first case, this implies that no dependence exists between the two operations, while in the latter case, an output dependence exists between the two operations.

We assume that the two operations are prototypical operations and their PMEs can be expressed as follow:

$$\mathcal{P}^{\mathcal{F}} = \left(\begin{array}{c|c} \widetilde{B}_{TL} \equiv \mathcal{F}_{TL}(\widehat{B}_{TL}) & \widetilde{B}_{TR} \equiv \mathcal{F}_{TR}(\widehat{B}_{TR}) \\ \hline \widetilde{B}_{BL} \equiv \mathcal{F}_{BL}(\widehat{B}_{BL}) & \widetilde{B}_{BR} \equiv \mathcal{F}_{BR}(\widehat{B}_{BR}) \end{array} \right);$$

$$\mathcal{P}^{\mathcal{G}} = \left(\begin{array}{c|c} \widetilde{C}_{TL} \equiv \mathcal{G}_{TL}(\widehat{C}_{TL}) & \widetilde{C}_{TR} \equiv \mathcal{G}_{TR}(\widehat{C}_{TR}) \\ \hline \widetilde{C}_{BL} \equiv \mathcal{G}_{BL}(\widehat{C}_{BL}) & \widetilde{C}_{BR} \equiv \mathcal{G}_{BR}(\widehat{C}_{BR}) \end{array} \right).$$

We can generalize the observations discussed in Section 6.3 with the following theorem:

Theorem 6.4.1. *Let the two operations that are to be merged be $\mathcal{F}$ and $\mathcal{G}$, such that $\mathcal{F}$ is computed before $\mathcal{G}$. Let B_X and C_Y be arbitrary regions of matrices B and C that were respectively updated by $\mathcal{F}$ and $\mathcal{G}$. In addition, let there be a dependence that requires B_X to be computed before C_Y.*

If the following conditions are satisfied by the loop invariant and remainder of algorithms that compute two separate operations, $\mathcal{F}$ and $\mathcal{G}$:

1. *If B_X was updated by g_Y (i.e., B and C are the same matrix and B_X is C_Y) or B_X is required by g_Y, then $\sigma(B_X) =$ Fully Updated, and,*

2. *If C_Y will be updated by $f_X^{\mathcal{R}}$ (i.e., B and C are the same matrix and B_X is C_Y) or C_Y is required by $f_X^{\mathcal{R}}$, then $\sigma(C_Y) =$ Not Updated,*

then the algorithms derived from those particular loop invariants and remainders can be merged.

Proof: We prove the theorem with a proof by contradiction. Assume that B_X and C_Y are arbitrary regions in matrices B and C updated by $\mathcal{F}$ and $\mathcal{G}$, respectively. In addition, assume that the two conditions hold for the loop invariants for the respective algorithms, for these regions. Furthermore, assume that there exists a dependence such that B_X must be computed before C_Y and that this dependence is not preserved after loop fusion. We will show this leads to a contradiction.

Because the dependence is not preserved, either (1) C_Y has to be updated before B_X in the same iteration and B_X is used to compute C_Y (a dependence flows from C_Y to B_X in the same iteration); or (2) C_Y was updated in some iteration in the past and C_Y is used to compute B_X in future iterations (a dependence flows from B_X in future iterations to C_Y in past iterations). Each of these cases leads to a contradiction:

Case 1: A dependence flows from C_Y to B_X in the same iteration. Since the dependence requires B_X to be computed after C_Y has been computed, it follows that $\sigma(B_X) \neq$ FULLY UPDATED. The contrapositive of the first condition tells us that if $\sigma(B_X) \neq$ FULLY UPDATED, then B_X was not be updated by computations represented by g_Y, and B_X is not required by g_Y. Since B_X is not required by g_Y and not updated by g_Y, then B_X must not be required to update C_Y, and B_X cannot be the same region as C_Y. Hence, there cannot be a dependence between the B_X and C_Y. Therefore, a contradiction is obtained.

Case 2: A dependence flows from B_X in future iterations to C_Y in past iterations. Since C_Y is (either partially or fully) computed in past iterations, it follows that $\sigma(C_Y) \neq$ NOT UPDATED. $\sigma(C_Y) \neq$ NOT UPDATED and the contrapositive of the second condition tells us that C_Y is not updated by computations represented by $f_X^{\mathfrak{R}}$ and C_Y is not required by $f_X^{\mathfrak{R}}$. Since C_Y is not updated by $f_X^{\mathfrak{R}}$, this implies that C_Y is not the same region as B_X. In addition, since $f_X^{\mathfrak{R}}$ represents updates to B_X in future iterations, then C_Y must not be required to compute B_X in future iterations.

Therefore, there cannot exist a dependence between B_X and C_Y. Hence a contradiction is found.

In both cases, we have shown that if a fusion-preventing dependence exists, then a contradiction with one of our assumptions is found and thus no fusion-preventing dependence can exist if both conditions are satisfied. □

6.5 Merging More Than Two Operations

Recall that algorithms can be merged if the regions of the two loop invariants have the appropriate status of computation that preserves dependences. If we know that status of computation for all regions of the output of a merged operation, then identifying if a third (or more) loop(s) can be merged can simply be performed by repeated application of Theorem 6.4.1.

By definition, the status of computation of a region is determined by identifying when the region is updated (i.e. in the past, in the future, or both). Therefore, to identify the status of computation of a region for a merged operation, we similarly identify when the region has been or will be updated.

When the regions updated by the separate loops are disjoint (i.e. the two loops do not update the same region(s)), then the status of computation of a region in the merged operation is determined by the loop invariant of the separate loop which updates that particular region. As the regions being updated by the separate loops are disjoint, we are assured that a region updated

by one loop will not be updated by the other loop.

When the separate loops update the same regions, then we need to consider if a region X has been updated in the past, or will be updated in the future. If both operations fully update region X in the past, then we know that region X will not be updated by either of the operations in the future. Therefore, $\sigma(X) =$ FULLY UPDATED for the merged operation. Similarly, if both operations will only update region X in future iterations, then $\sigma(X) =$ NOT UPDATED. Otherwise, region X was updated in the past, and will be updated in future iterations. Hence $\sigma(X) =$ PARTIALLY UPDATED.

6.6 Application of the Theory

When L in the operation TRMV-MERGE is the inverse of the Cholesky factor of a Symmetric Positive Definite (SPD) matrix S, the operation TRMV-MERGE computes the operation

$$x := S^{-1}x, \text{ where } S = LL^T.$$

Here, S is a SPD matrix. To demonstrate how the theory developed in this chapter can be applied in practice, we apply loop fusion to the following three operations that are part of the operations required to compute $x := S^{-1}x$:

$$\begin{array}{ll} L := L^{-1} & \text{TRINV} \\ y := Lx & \text{TRMV-X} \\ x := L^T y & \text{TRMV-TRANS.} \end{array} \tag{6.1}$$

Consider the following set of loop invariants for the three operations

that we want to merge.

$$
\begin{aligned}
\mathcal{I}^{\textsc{Trinv}} &= \left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right); \\
\mathcal{I}^{\textsc{Trmv-x}} &= \left(\begin{array}{c} y_T \equiv L_{TL}x_T \\ \hline y_B \equiv L_{BL}x_T \end{array}\right);\ \textit{and} \qquad (6.2) \\
\mathcal{I}^{\textsc{Trmv-trans}} &= \left(\begin{array}{c} x_T \equiv L_{TL}y_T \\ \hline x_B \equiv \widehat{x}_B \end{array}\right).
\end{aligned}
$$

Examining the operations in (6.1), we note that the dependence between the TRINV and TRMV-X is a true dependence created by the need to update L. Since $\mathcal{I}^{\textsc{Trmv-x}}$ requires L_{TL} and L_{BL}, Theorem 6.4.1 tells us that $\sigma(L_{TL})$ and $\sigma(L_{BL})$ must both be FULLY UPDATED. Since $\mathcal{I}^{\textsc{Trinv}}$ fulfills those criteria, then the two loop invariants, $\mathcal{I}^{\textsc{Trinv}}$ and $\mathcal{I}^{\textsc{Trmv-x}}$, can be merged.

Since TRINV and TRMV-X compute L and y respectively, we know that the status of computation of y is determined only by the chosen loop invariant for TRMV-X. In addition, because a true dependence and an anti-dependence exist between the TRMV-X and TRMV-TRANS, Theorem 6.4.1 tells us that because $\widehat{x}_B$ is required by $\mathcal{R}_{\mathcal{I}^{\textsc{Trmv-x}}}$, $\sigma(x_B)$ = NOT UPDATED. Furthermore, because $\sigma(y_B) \neq$ FULLY UPDATED, then y_B cannot be an operand of $\mathcal{I}^{\textsc{Trmv-trans}}$. Since $\mathcal{I}^{\textsc{Trmv-x}}$ and $\mathcal{I}^{\textsc{Trmv-trans}}$ fulfills those criteria, we know that they can be merged. Hence, the set of three loop invariants shown in (6.2) yields algorithms that can be merged.

For the remaining $8 \times 4 \times 4 - 1 = 127$[1] possible sets of three loop

[1]There are at least 8, 4 and 4 different algorithms for the TRINV, TRMV-X, and

invariants, we leave these as an exercise for the reader to determine if they can be merged.

6.7 Summary

In this chapter, we developed the theory that allows one to determine the legality of loop fusion using the loop invariant and the remainder of the separate algorithms. By raising the level at which analysis is performed, this simplifies the analysis process performed by either the DLA expert or the compiler. However, there is still a need to examine a combinatorial number of possible combinations of loop invariants and remainders. In Chapter 8, we show how the developed theory can be used to **derive** loop invariants and remainders with the characteristics necessary to ensure that the derived algorithms can be merged, thus making the whole optimization process goal-oriented.

TRMV-TRANS operations respectively.

Chapter 7

Domain-Specific Heuristics

When optimizing DLA operations, the DLA expert has to manually examine multiple algorithms to determine if the algorithms possess desirable characteristics that will yield high performance on a given computer architecture. To limit the number of algorithms that must be explored, the DLA expert uses domain-specific heuristics. In this chapter, we describe some of the heuristics used by the DLA expert and show how heuristics can be applied to the loop invariant and its remainder to yield the same insights.

7.1 Unit Stride Data Accesses

For the domain of DLA, ensuring that data is accessed with unit stride is critical for attaining high performance. Often, data is packed into continuous storage so that data accesses are to contiguous memory with unit stride [51, 24]. For matrix multiplications and its many variants, the cost of packing can be amortized efficiently by the computations performed. However, for operations that have $O(n^2)$ memory accesses and $O(n^2)$ floating point computation, the cost of packing cannot be similarly amortized. For these operations (e.g. matrix-vector operations similar to those in the level-2 BLAS), it is im-

portant to identify algorithms that naturally access matrices with unit stride.

7.1.1 Matrix storage

In order to identify when matrices are accessed with unit stride, we need to discuss how matrices are stored. For the domain of DLA, matrices are usually stored in column-major order. This means that elements in the same column can be accessed with unit stride. This, in turn, implies that algorithms that access matrices one column at a time will access elements with unit stride. Similarly, matrices accessed by rows will not access elements with unit stride.

7.1.2 An example of access by columns

Now, let us examine the loop invariant and Remainder for two algorithms that compute the TRMV-X operation to gain insights into how to identify algorithms that access matrix elements with unit stride.

Consider the following loop invariants and their remainders for an `axpy`-based and an inner-product-based algorithm for the TRMV-X operation:

	`axpy`-based	inner-product-based
$\mathcal{I}^{\text{TRMV-X}}$	$\left(\frac{y_T \equiv \widehat{L}_{TL}\widehat{x}_T}{y_B \equiv \widehat{L}_{BL}\widehat{x}_T}\right)$	$\left(\frac{y_T \equiv \widehat{L}_{TL}\widehat{x}_T}{y_B \equiv \widehat{y}_B}\right)$
$\mathcal{R}^{\mathcal{I}^{\text{TRMV-X}}}$	$\left(\frac{\widetilde{y}_T \equiv y_T}{\widetilde{y}_B \equiv \widetilde{y}_B + \widehat{L}_{BR}\widehat{x}_B}\right)$	$\left(\frac{\widetilde{y}_T \equiv y_T}{\widetilde{y}_B \equiv \widehat{L}_{BL}\widehat{x}_T + \widehat{L}_{BR}\widehat{x}_B}\right)$

In both cases, L is partitioned in the following manner:

$$L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right),$$

where the thick lines denotes how far into matrix L the loop has progressed.

We start by examining the loop invariant and its remainder for the `axpy`-based algorithm. Recall that `axpy`-based algorithms for TRMV-X access the elements of L with unit stride. Notice that L_{TL} and L_{BL} are the only regions of L that appear in the loop invariant. Similarly, L_{BR} is the only region of L that appears in the Remainder. This tells us that elements of L that will be used in the body of the loop, must be part of L_{BR} at the start of an iteration, and will be part of L_{TL} or L_{BL} at the end of the iteration.

In addition, because of the way L is partitioned, the regions L_{TL} and L_{BL} represent the columns of L to the left of the vertical thick line (left columns), which implies that only the left columns of L were used in past iterations. Similarly, we know that:

$$\left(\begin{array}{c} 0 \\ \hline L_{BR} \end{array} \right),$$

represents the columns of L that are to the right of the vertical thick line, and L_{BR} represents the non-zero part of those columns of L. Furthermore, because L_{BR} appears only in the Remainder, we know that only columns to the right of the thick lines will be used in the future. This, coupled with the observation in the previous paragraph, implies that elements in the columns of L that are to the right of the thick lines will be used in an iteration. In addition, these

elements will form part of the columns of L that lie to the left of the thick lines. Hence, the elements must be accessed by columns, which validates the DLA expert's observation that `axpy`-based algorithms for TRMV-X access elements of L with unit stride.

Contrast this with the loop invariant and remainder for the inner-product-based algorithm, which access elements of L by rows. From the loop invariant and its remainder, we know that L_{TL} was used in the past, and L_{BL} and L_{BR} will be used in the future. Since L_{TL} represents the non-zero elements of the rows to the top of the horizontal thick line, and the regions L_{BL} and L_{BR} form the rows of L that lies below the horizontal thick line, elements of L must be accessed by rows. Hence, the DLA expert's observation that inner-product-based algorithm for TRMV-X do not access elements of L with unit stride.

7.1.3 The theory

Let us now generalize the insights in the previous section to our prototypical operation, where A is an operand (regardless of whether it is used as input, output, or both) that has been partitioned by in the following manner:

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right).$$

Without loss of generality, assume that A is swept through from top-left to bottom-right. In addition, assume that A_{TL} and A_{BL} appear only in the loop invariant and the remaining regions appear only in the remainder of the loop invariant.

At the start of any iteration, A will be repartitioned into subregions such that:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$$

Since the regions A_{TR} and A_{BR} appear only in the remainder, we know that these two regions will be used in any given iteration. This means that at least one of the following subregions:

$$\left(\begin{array}{c|c} a_{01} & A_{02} \end{array}\right) \text{ and } \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$$

will be used during the iteration. We also know that the remaining regions $(A_{00}, a_{10}^T, A_{20})$ will not be used in the iteration because A_{TL} and A_{BL} do not appear in the remainder of the loop.

At the end of the iteration, the thick lines are shifted such that:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right).$$

Since columns represented by A_{TR} and A_{BR} were not used in past iterations, this means that the subregions A_{02}, a_{12}^T and A_{22} were not used. Therefore, the subregions that were used during the iteration must be from the following set of regions: a_{01}, α_{11} and a_{21}. Since the three subregions are different segments of the same column of A, we know that A must be accessed by columns.

We can summarize this discussion with the following theorem:

Theorem 7.1.1. *Let S_A and $S_{A^{\mathfrak{R}}}$ be sets of regions of matrix A that appear in the loop invariant and the remainder, respectively. If $S_A \cap S_{A^{\mathfrak{R}}} = \emptyset$, and either*

1. $S_A \subseteq \{A_{TL}, A_{BL}\}$*, or*

2. $S_A^{\mathcal{R}} \subseteq \{A_{TL}, A_{BL}\}$

then A *is accessed by columns.*

Proof: The proof follows the previous discussion. □

7.2 Matrix Multiplication

For a DLA expert, implementing high-performance DLA code often boils down to finding algorithms that are rich in matrix multiplication. It has been shown that high-performance DLA loops can be attained when most of the computation within the loop is cast in terms of a highly optimized **GE**neral **M**atrix **M**ultiplication (GEMM) kernel that computes some variant of the operation $C := \alpha AB + \beta C$ [30, 24, 26, 51]. Without loss of generality, we will focus on $C := AB + C$, which we will also write as $C+ = AB$.

In general, there are three different algorithms to compute GEMM [26, 24]. The three variants of GEMM, namely the matrix-panel, panel-matrix and panel-panel variants, are shown in Figure 7.1.

Of particular interest to the DLA expert is the panel-panel (also known as the rank-k) variant of GEMM. The panel-panel variant is important for many architectures. For sequential architectures, this variant of GEMM is often the algorithm of choice [24, 51]. It has also been shown to asymptotically yield near-peak performance when correctly parallelized for distributed-memory architectures [49]. Hence, identifying whether a panel-panel matrix

Variant	Shape of Operands
Matrix-panel	$C \stackrel{+}{=} A \; B$
Panel-matrix	$C \stackrel{+}{=} A \; B$
Panel-panel	$C \stackrel{+}{=} A \; B$

Figure 7.1: The different variants of GEMM, and the shapes of the operands, A, B and C.

multiplication is absent or present allows the DLA expert to prioritize different algorithms that compute a given operation.

A key feature of the panel-panel variant of GEMM is that it is the only variant where both the dimensions of C are large, and both A and B have one small dimension. Therefore, by examining the shapes of the operands, A, B and C, of the GEMM operation, we can identify if the GEMM operation is of the panel-panel variant.

7.2.1 Examples of analysis with shapes of operands

Consider the following pair of loop invariant and remainder for the TRINV operation, which computes $L := L^{-1}$:

$$\overset{\text{Loop Invariant}}{\left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv -\widehat{L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right)}; \quad \overset{\text{Remainder}}{\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv L_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv L_{BL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)}.$$

Without loss of generality, assume that L is a $Nb \times Nb$ matrix, where N is the number of iterations, and b is the numbers of columns and/or rows computed in each iteration.

Notice that $\sigma(L_{TL}) = \sigma(L_{BL}) =$ FULLY UPDATED. This implies that the regions L_{TL} and L_{BL} contain the final result, and no further computation is required to update them. Hence, the panel-panel matrix multiplication is NOT required to update those regions of L. In addition, $\sigma(L_{BR}) =$ NOT UPDATED implies that computation must be performed on L_{BR}.

Since the final result must be stored in L_{TL}, we know that L is computed from top-left to bottom-right. In addition, at the start of an iteration, the regions of L are repartitioned into subregions in the following manner:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right),$$

where L_{11} is a small $b \times b$ matrix. Notice that L_{BR} is now represented by the subregions

$$\left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right). \qquad (7.1)$$

Since computation must be performed on L_{BR}, then computation must be performed on at least one of the subregions in (7.1).

At the end of the iteration, the subregions are then merged back into regions in the following manner:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right),$$

where, now, L_{BR} is L_{22}. Since $\sigma(L_{BR})$ must still be Not Updated, then we know that L_{22} cannot have been updated during the iteration. This also implies that the subregions that could have been updated during the iteration must be either L_{21}, L_{11} or both.

Since L_{21} and L_{11} are output operands and they also have a small column dimension (in this case, both subregions have a small number of colums, namely b), we know from examining their shape that the operation that updates either L_{21} or L_{11} is not a panel-panel matrix multiplication.

Let us consider another pair of loop invariant and remainder for the Trinv operation:

$$\overset{\text{Loop Invariant}}{\left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv \widehat{L}_{BL}\widehat{L}_{TL}^{-1} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right)}; \quad \overset{\text{Remainder}}{\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv L_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv \widehat{L}_{BR}^{-1} L_{BL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)}.$$

Notice that in this loop invariant, $\sigma(L_{TL}) =$ Fully Updated, $\sigma(L_{BL}) =$ Partially Updated, and $\sigma(L_{BR}) =$ Not Updated. This implies that the regions L_{BL} and L_{BR} will need to be updated in any given iteration.

Again, we consider what subregions of L will be updated during a given iteration. After repartitioning, we know that:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right),$$

where

$$L_{BL} = \left(\begin{array}{c} L_{10} \\ \hline L_{20} \end{array}\right), \tag{7.2}$$

and

$$L_{BR} = \left(\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & L_{22} \end{array}\right). \tag{7.3}$$

Since $\sigma(L_{BL}) =$ PARTIALLY UPDATED and $\sigma(L_{BR}) =$ NOT UPDATED, these imply that some of the subregions described in (7.2) and (7.3) must be updated during an iteration. Similar to the previous loop invariant for TRINV, because $\sigma(L_{BR}) =$ NOT UPDATED at the end of an iteration, we know that only the subregions L_{11} and L_{21} need to be updated, and due to their shapes (they only have a small number of columns, namely b columns), the operations that update L_{11} and L_{21} are not panel-panel matrix-multiplications.

As $\sigma(L_{BL}) =$ PARTIALLY UPDATED, we know that some updates must be performed on at least one of the two subregions, L_{10} and L_{20}. While we know that L_{10} has a small dimension (it has b rows), the same cannot be concluded for subregion L_{20}. Hence, we can conclude that a panel-panel matrix multiplication is not required in order to update L_{10}. However, it remains possible that L_{20} is updated by a panel-panel matrix multiplication.

7.2.2 The theory

It is trivially true that if our operation consists of only matrix additions, then no matrix multiplication is required. Similarly, for each region, C_X, where $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$ of our prototypical operation, if the function F_X, describing how C_X needs to be updated at the end of the loop, consists of only additions, then no panel-panel matrix multiplication is required to compute C_X. Hence, we can focus on the the case where there exists a matrix multiplication in F_X that computes an arbitrary region, C_X, of our prototypical operation.

Theorem 7.2.1. *Let X be a region of the output C, where $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. If $\sigma(C_X) =$ Fully Updated or $\sigma(C_X) =$ Not Updated, then the operation that updates C_X is not a panel-panel matrix multiplication.*

Proof: We prove this theorem with a proof by cases.

Case 1: $\sigma(C_X) =$ Fully Updated.

Let C_X be an arbitrary region of C where $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$. If $\sigma(C_X) =$ Fully Updated, then it implies that $C_X \equiv \widetilde{C}_X$, i.e. C_X already contains the final result. Therefore, no panel-panel matrix multiplication is required to compute C_X.

Case 2: $\sigma(C_X) =$ Not Updated.

Without loss of generality, let us assume that C is a $Nb \times Nb$ matrix that is computed from top-left to bottom-right. Here, N is the number

of iterations required to compute C and b is the number of columns, and/or rows exposed (also known as the block or tile size) during each iteration. This implies that at the start of an iteration, the different regions of C are partitioned into subregions in the following manner:

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right),$$

where C_{11} is a $b \times b$ matrix.

In addition, at the end of the loop, the regions are C are obtained by merging the subregions in the following manner:

$$\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right).$$

We want to prove that if $\sigma(C_X) = $ NOT UPDATED, for all values of $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$, then no panel-panel matrix multiplication is required to update C_X. We prove this by considering all possible values of X.

Case 2a: $C_X = C_{TL}$.

Since C is updated from top-left to bottom-right, this implies that C_{TL} contains all of C at the end of the last iteration. This means that C_{TL} must contain the final result ($\widetilde{C}$) at the end of the loop. Since $\sigma(C_{TL}) = $ NOT UPDATED at the end of the loop, it implies that C_{TL} contains the original value $\widehat{C}$ at the end of the loop. This contradicts our assumption that C_{TL} contains the final value at the

end of the loop, and thus a contradiction is obtained. Therefore, this situation cannot exist as $\sigma(C_{TL}) \neq$ Not Updated.

Case 2b: $C_X = C_{TR}$.

Notice that C_{TR} consists of the subregions C_{01} and C_{02} at the start of the iteration, and C_{02} and C_{12} at the end of the loop. Since $\sigma(C_{TR}) =$ Not Updated, it implies that C_{02} contains the original value at the start of the iteration, and C_{02} is not updated at the end of the iteration. Hence, no panel-panel matrix multiplication is required to update C_{02}.

Now, C_{01} and C_{12} may have been updated in the iteration. However, since C_{01} and C_{12} each have one small dimension, they cannot be the output of a panel-panel matrix multiplication. Therefore, when $\sigma(C_{TR}) =$ Not Updated, C_{TR} is not the output of a panel-panel matrix multiplication.

Case 2c: $C_X = C_{BL}$.

At the start of an iteration, C_{BL} consists of the subregions C_{10} and C_{20}. In addition, because $\sigma(C_{BL}) =$ Not Updated, both C_{10} and C_{20} contain their original value. At the end of the iteration, C_{BL} is made up of subregions C_{20} and C_{21}. Again, the values of C_{20} and C_{21} must be the original values because $\sigma(C_{BL}) =$ Not Updated. This implies that C_{20} cannot be updated during the iteration. In addition, because both C_{21} and C_{10} have one dimension that is small, C_{21} and C_{10} cannot be the output of a panel-panel matrix

multiplication.

Since C_{20}, C_{21} and C_{10} cannot be the output of a panel-panel matrix multiplication, therefore, C_{BL} cannot be updated by a panel-panel matrix multiplication when $\sigma(C_{BL}) =$ NOT UPDATED.

Case 2d: $C_X = C_{BR}$.

At the start of an iteration, C_{BR} is partitioned into C_{11}, C_{12}, C_{21} and C_{22}. At the end of the iteration, C_{BR} is C_{22}. Since $\sigma(C_{BR}) =$ NOT UPDATED, this means that C_{22} starts off being not updated at the start of the iteration, and must remain not updated at the end of the iteration. This means that C_{22} is not updated within the iteration. For the other three subregions (C_{11}, C_{12}, and C_{21}), each of them has at least one small dimension. Therefore, they cannot be the output of a panel-panel matrix multiplication, which means that C_{BR} cannot be updated by a panel-panel matrix multiplication. $\sigma(C_{BR}) =$ NOT UPDATED.

Since all $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$ leads to a situation where C_X cannot be updated by a panel-panel matrix multiplication, then C_X cannot be updated by a panel-panel matrix multiplication when $\sigma(C_X) =$ NOT UPDATED.

Since both $\sigma(C_X) =$ FULLY UPDATED and $\sigma(C_X) =$ NOT UPDATED imply that the operation that updates C_X is not a panel-panel variant of GEMM, the theorem is proven. □

Remark 2. *When a compiler parallelizes for a shared-memory system, one of the heuristics used is to identify algorithms where the outermost loop has independent iterations. This maximizes the size of the tasks that are issued to the different processors/threads. However, a loop with independent iterations means that no region in the loop invariant is partially updated, which means that there cannot be panel-panel matrix multiplication in the update statements. Since the DLA expert favors algorithms rich in panel-panel matrix multiplications, this means that the heuristic used by the compiler is unlikely to yield the algorithm favored by the DLA expert.*

7.3 Algorithms with Small Checkpoint Sizes

Checkpoint-and-restart is a commonly encountered technique for developing fault-tolerant DLA code [17, 27]. In essence, code is inserted into a DLA algorithm so that (1) the state of the loop is saved periodically (Checkpoint), and (2) the loop can be restarted from the last saved state when a hardware fault or software error occurs (Restart). It has been shown that the overhead of checkpointing can be reduced by either reducing the frequency at which checkpointing is performed, or reducing the amount of data saved at each checkpoint [36]. In this section, we focus on the latter, by describing how loop-based algorithms that require small checkpoint sizes can be identified.

7.3.1 Checkpoint-optimal algorithms

A common approach for ensuring that checkpoint sizes are small, is to only save data that have been modified since that last checkpoint [36, 39]. Since only modified data were saved, this implies that no unnecessary, or redundant data are saved at each checkpoint. Hence, the size of each checkpoint is small. In addition, by saving all modified data, no additional computation is necessary during restart.

Since the key to small checkpoint sizes is to ensure that no unnecessary, or redundant data are saved, we examine what sort of data need to be saved:

1. **Initial values of all input operands must be saved during an initial checkpoint before the loop-based algorithm commences.** This ensures that in the worst case, the entire algorithm can be re-executed using the initial values.

2. **Final values must be saved during a checkpoint.** This ensures that computed values are saved, and they need not be recomputed when restarting after a failure.

3. **All data required in order to restart from a checkpoint without additional (re)computation needs to be saved**. The purpose of saving all modified data is to ensure that no additional computation has to be performed before the loop can restart from the checkpoint. This minimizes the time between restart and the continuation of the original

algorithm, thus reducing the overhead of restart. In addition, the code inserted for restart is also simplified because only the restoration of data from a previous checkpoint is necessary.

Under these assumptions about the types of data that need to be saved during a checkpoint, each element of all operands is saved a minimum of once. If the operand is an input, then its elements are saved during the initial checkpoint. If an operand is an output, then the final value of that operand needs to be checkpointed. However, when an operand is both an input and an output, then its elements are checkpointed at least twice; once during the initial checkpoint, and when the final value has been computed.

When checkpoints can be inserted into a loop-based algorithm such that each element of all operands are saved the minimum number of times, and the algorithm can be restarted from a checkpoint without any recomputation, then we say that the algorithm is *checkpoint-optimal*.

Definition 7.3.1. *An algorithm is said to be* checkpoint-optimal *if all the following holds:*

- *Every element of an operand that is only an input, or only an output is saved exactly once.*
- *Every element of an operand that is both an input and an output operand, is saved exactly twice.*
- *Restarting from any checkpoint does not incur additional computation.*

7.3.2 Example of checkpoint-optimal algorithms

Consider the following pair of loop invariant and remainder for the TRINV operation:

$$\overset{\text{Loop Invariant}}{\left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv \widehat{-L}_{BR}^{-1}\widehat{L}_{BL}\widehat{L}_{TL}^{-1} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right)} ; \quad \overset{\text{Remainder}}{\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv L_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv L_{BL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)}.$$

Recall that L is computed from top-left to bottom-right. In addition, $\sigma(L_{TL}) = \sigma(L_{BL}) =$ FULLY UPDATED, and $\sigma(L_{BR}) =$ NOT UPDATED. Again, at the start of an iteration, L is repartitioned into subregions [1] in the following manner:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right),$$

and at the end of the iteration, the different subregions are merged in the following manner:

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right).$$

Notice that at the start of the iteration, λ_{11} and l_{21} were subregions of L_{BR}, and at the end of the iteration, λ_{11} and l_{21} become subregions of L_{TL} and L_{BL} respectively. From the status of computation, it can be concluded that at the start of the iteration, λ_{11} and l_{21} contain original values, and at the end of the iteration, the two subregions contain the final values.

[1] In practice, a blocked (tiled) algorithm is used to compute L. For simplicity, an unblocked algorithm is used in this example.

Notice that L_{22} remains uncomputed at the end of an iteration. Since we assumed that the original values have been saved during the initial checkpoint, we know that L_{22} does not need to be saved during any other checkpoints. However, because final values have to be checkpointed at least once, then at the end of an iteration, we have to decide which (or any) of the following subregions belonging to fully updated regions need to be checkpointed:

$$L_{00},\ l_{10}^T,\ L_{20}^T,\ \lambda_{11},\ \text{or}\ l_{21}.$$

Recall that the DLA expert is interested in small checkpoint sizes. A small checkpoint size implies that values that have been saved at a previous checkpoint should not be saved again. As λ_{11} and l_{21} have most recently been updated to their final values, we know that their values were not available at previous checkpoints and thus could not have been saved. Therefore, saving the data in these two subregions of L would ensure that no redundant data saved.

In addition, the subregions L_{00}, and L_{20} have two large dimensions, whereas l_{21} and l_{10}^T each has one small dimension. λ_{11} is the smallest of the five fully updated subregions where both of its dimensions are small. Hence, saving the values of subregions λ_{11} and l_{21} ensures that the checkpoint sizes are as small as, or smaller than most other combinations of subregions[2].

Now, recall that the DLA expert also wants checkpoints that require no

[2]The exception being only λ_{11} is saved during a checkpoint. However, saving only λ_{11} does not ensure that all final values are checkpointed

(re)computation during restart. Notice that L is computed column-wise. The Remainder tells us that pristine columns of L are required for computation in the future. This means that at the start of an iteration, a pristine column of L, namely the column:

$$\begin{pmatrix} 0 \\ \lambda_{11} \\ l_{21} \end{pmatrix},$$

is required. Since the computation of L only requires uncomputed data, and we know that those data were saved during the initial checkpoint, then no further computation is required during restart. Hence, the algorithm derived from the loop invariant is checkpoint-optimal.

Contrast this with the following loop invariant-Remainder pair for the TRINV operation:

$$\overset{\text{Loop Invariant}}{\left(\begin{array}{c|c} L_{TL} \equiv \widehat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv \widehat{L}_{BL}\widehat{L}_{TL}^{-1} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right)}; \quad \overset{\text{Remainder}}{\left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv L_{TL} & 0 \\ \hline \widetilde{L}_{BL} \equiv -\widehat{L}_{BR}^{-1}L_{BL} & \widetilde{L}_{BR} \equiv \widehat{L}_{BR}^{-1} \end{array}\right)},$$

where $\sigma(L_{TL})$ = FULLY UPDATED, $\sigma(L_{BL})$ = PARTIALLY UPDATED, and $\sigma(L_{BR})$ = NOT UPDATED. As with the previous example, L is repartitioned into subregions at the start of an iteration, and the subregions are merged at the end of the iteration. Comparing the subregions at the start and end of the iteration:

$$\text{At the start:} \quad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

$$\text{At the end:} \quad \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline l_{10}^T & \lambda_{11} & 0 \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right),$$

the status of computation tells us that λ_{11} and l_{01}^T are subregions where final values are computed. λ_{11} starts off containing its original value, and is fully computed at the end of the iteration. l_{01}^T is partially updated at the start of the iteration but is fully updated at the end of the iteration. Therefore, a checkpoint that saves data from these two subregions would ensure no redundant data are saved, and also ensure that the checkpoint sizes are small.

However, notice that the values in L_{BL} are in a partially updated state, and l_{21} is part of L_{BL} at the end of the iteration. This means that l_{21} started the iteration containing its original values but at the end of the iteration, l_{21} is partially updated. As the values in l_{21} have been modified, these intermediate values are not available after a failure has occurred. Hence, either these intermediate values are recomputed during restart, or the intermediate values need to be saved at each checkpoint. This means that this algorithm is not checkpoint-optimal.

7.3.3 The theory

A key insight that allows us to identify checkpoint-optimal algorithms is that if a loop invariant for an algorithm has partially updated regions, then the algorithm is not checkpoint-optimal. This insight can be summarized for our prototypical operation whose output C is partitioned into regions C_X, where $X \in \{\mathsf{TL}, \mathsf{TR}, \mathsf{BL}, \mathsf{BR}\}$ with the following lemma.

Lemma 7.3.1. *Let C_X be an arbitrary region of the loop invariant. If $\sigma(C_X) =$ PARTIALLY UPDATED, then the algorithm is not checkpoint-optimal.*

Proof: We prove the theorem with a proof by contradiction. Assume that the loop invariant for a checkpoint-optimal algorithm has an arbitrary region C_X where $\sigma(C_X) =$ PARTIALLY UPDATED. We want to show that such an algorithm is not checkpoint-optimal.

As $\sigma(C_X) =$ PARTIALLY UPDATED, it implies that at the end of a loop, the values in C_X are partially updated. We consider the following two cases:

Case 1: Values in C_X is saved at a checkpoint.

Since the values in C_X are intermediate and not final values, this means that at a future checkpoint, when the final values in elements in C_X are computed, those final values have to be checkpointed. This implies that the values in elements in C_X are saved more than the minimum number of times. Hence the algorithm is not checkpoint-optimal.

Case 2: Values in C_X are not saved at a checkpoint.

If the intermediate values in C_X are not saved at a checkpoint, then these intermediate values are not available after restarting from that particular checkpoint. Hence, additional computation must be performed to restore these partially-computed results during restart, before computation can proceed. Therefore, the algorithm is not checkpoint-optimal.

In both cases, when the loop invariant of an algorithm has regions whose status of computation is PARTIALLY UPDATED, the algorithm is not checkpoint-optimal. Hence, a contradiction is obtained and the lemma is proven. □

Now, for our prototypical operation:

$$C := \mathcal{F}(C, \{A\}),$$

an algorithm that computes C is checkpoint-optimal can be identified with the following theorem.

Theorem 7.3.1. *A loop invariant of an algorithm contains no partially updated region if and only if the algorithm is checkpoint-optimal.*

Proof: The forward direction follows the proof for Lemma 7.3.1.

We prove the backward direction by showing how to insert checkpoints into algorithms whose loop invariants contain no partially updated regions, such that the algorithms are checkpoint-optimal.

Checkpoint Strategy. We insert checkpoints in the following manner:

- **Initial checkpoint.** Since the inputs of our prototypical operation is $C \cup \{A\}$, then the initial checkpoint will save all elements of C and operands in $\{A\}$.
- **Other checkpoints at the end of each iteration.** Since C is an output, checkpoints will be inserted at the end of each iteration such that elements of C whose final values have been computed in that iteration will be saved at the checkpoint.

We prove that the algorithm is checkpoint-optimal with this checkpoint strategy by showing that the checkpointing strategy fulfills all requirements in the definition of a checkpoint-optimal algorithm:

Every element of an operand that is only an input, or only an output is saved exactly once.

The input-only operands are operands in $\{A\}$, and there are no output-only operands. Since the elements of operands in $\{A\}$ are saved only at the initial checkpoint, they are saved exactly once.

Every element of an operand, that is both an input and an output operand, is saved exactly twice.

Since C is both an input and an output, elements of C must be saved exactly twice. Elements of C are saved once during the initial checkpoint. In addition, the elements of C are saved at the checkpoint at the end of the iteration in which their final values are computed. Hence, elements of C are saved exactly twice.

Restarting from any checkpoint does not incur additional computation.

Since the loop invariant only consists of regions that are fully updated and not updated, it means that at the start of any iteration, elements of C are either pristine or contain their final values, and all other operands contain their original values. Since pristine values are saved during the initial checkpoint and the final computed values of C are saved in the iterations where they are computed, we know that the values at the

start of any iteration are saved at some previous checkpoint. Hence, no additional computation is required during restart.

Since our checkpointing strategy satisfies the definition of a checkpoint-optimal algorithm, an algorithm whose loop invariant contains no partially updated regions is checkpoint-optimal.

Since both directions have been proven, the theorem is proven. □

7.4 Summary

In this chapter, we showed that domain-specific heuristics employed by the DLA expert can be described in terms of constraints on the loop invariant and its remainder. By raising the level of abstraction at which loops are analyzed, we showed that characteristics of the loop, such as the absence of panel-panel matrix multiplication, that cannot be described using traditional compiler dependence analysis, can be identified from the loop invariant and its remainder. The examples presented in this chapter provide evidence for our claim that the DLA expert's approach to optimization can be unified with the compiler approach through the use of a framework that is based on a calculus of loop invariants.

Chapter 8

A Goal-Oriented Approach To Loop Invariants Derivation

In the previous chapters, we developed theories that allow one to determine if a loop has a particular characteristic from an analysis of the loop invariant and/or its remainder. This implies that one has to first derive multiple loop invariants, analyze them with the developed theory, and then discard loop invariants that do not have the desired characteristics.

In this chapter, we take the theories one step further by using them in a goal-oriented approach towards deriving loop invariants. We introduce a constructive algorithm that, given the PME and a desired characteristic of the loop as inputs, derives loop invariants with the desired characteristic. *It is through this goal-oriented approach that the phase-ordering problem can be side-stepped.*

8.1 Key Insights

Recall that loop invariants and their remainders are defined by the sets of functions $\{f_X\}$ and $\{f_X^{\mathcal{R}}\}$. These functions are obtained from the operations specified in the PME. Operations that are removed from the PME form the

functions in $\{f_X^{\mathfrak{R}}\}$, and those that were not removed from the PME form the set $\{f_X\}$.

In addition, the characteristics of loops (e.g. independent iterations and unit stride) can be described in terms of the status of computation. Since the status of computation is prescribed by the decompositions of operations in the PME into the functions in $\{f_X\}$ and $\{f_X^{\mathfrak{R}}\}$, the characteristics of loops can be determined by selectively choosing which operations from the PME belong to the functions from $\{f_X\}$ and $\{f_X^{\mathfrak{R}}\}$.

Furthermore, selectively determining if an operation in the PME belongs to f_X or $f_X^{\mathfrak{R}}$, allows us to preserve the status of computation that are required for the desired characteristic. This ensures that any loop invariant (and its remainder) that is derived will always have the necessary characteristic. If no feasible loop invariant can be derived in this manner, then no loop invariant derived from the input PME has the desired characteristic.

These insights suggest that a goal-oriented approach to deriving loop invariants with the necessary characteristic is achievable with a loop invariant derivation algorithm that preserves the status of computation required for the desired characteristic to appear in the loop.

8.2 Goal-Oriented Loop Invariant Derivation Algorithm

We present a constructive loop invariant derivation algorithm that takes as input, the PME and the theory that describes the necessary status of com-

putation for the desired characteristic, and returns a set (possibly empty) of loop invariants such that all returned loop invariants have the desired characteristic.

Our algorithm is similar to that described in *Fabregat-Traver and Bientinesi* [21]. The key difference between the two algorithms is that their goal is to systematically derive loop invariants, whereas the goal of our algorithm is to systematically derive loop invariants that yield loops with the desired characteristic. The theory described in this dissertation can be used to enhance the algorithm in *Fabregat-Traver and Bientinesi* to also obtain loop invariants that yield loops with the desired characteristic.

Our loop invariant derivation algorithm for constructing a loop invariant with the necessary characteristics comprises of three phases: Initialization, Construction, and Enumeration. We describe the three phases in detail.

8.2.1 Initialization

Theorem 3.2.1 tells us that for the loop invariants to be feasible, some operations described in the PME must have been performed in the past and certain operations must still be performed in the future. This implies that the status of computation of at least one region, C_X, of any feasible loop invariant is not FULLY UPDATED, and at least one region, C_Y (possibly the same region, i.e. $X = Y$) of any feasible loop invariant is not NOT UPDATED. This means that computation that requires $\widetilde{C}_X$ cannot be computed since $\sigma(C_X) \neq$ FULLY UPDATED. Similarly, regions that require $\widehat{C}_Y$ must be com-

puted to a state where $\widehat{C}_Y$ is no longer required.

Hence, because loop invariants have to be feasible, an initial set of constraints on the status of computations of at least one region is created. In addition, these initial constraints may impose further constraints on the status of computation of other regions. As such, these initial constraints seed the propagation of constraints in the subsequent phase, the Construction phase.

8.2.2 Construction

The Construction phase propagates the initial set of constraints on the status of computation, created during Initialization, to other regions of the loop invariant. Through propagating these constraints to other regions, the status of computation for other regions may be determined, which in turn may create constraints for even more regions. This process of propagating constraints is then repeated until one of the following conditions hold:

1. **The characteristic described by the theory cannot be preserved.** If the characteristic described by the theory cannot be preserved, then there exist no feasible loop invariant, obtained by removing operations from the given PME, that has the desired characteristic. In this case, the algorithm returns without finding a feasible loop invariant with the desired characteristic.
2. **All operations described in the PME have been divided up between the f_X's and $f_X^{\mathcal{R}}$'s.** When all operations have been divided

up between the f_X, and $f_X^{\mathcal{R}}$, then the loop invariant and remainder have been defined. Therefore, the constructed loop invariant is then returned as a feasible loop invariant, obtained by removing operations from the input PME, that possesses the desired characteristic.

3. **No new constraints are introduced, and some operations have not been assigned to either f_X or $f_X^{\mathcal{R}}$.** If no new constraints are introduced and the operations described in the PME have yet to be completely divided up between f_X and $f_X^{\mathcal{R}}$, then there may be multiple loop invariants that possess the desired characteristic. As no constraints, thus far encountered, requires the operations to be either in f_X or $f_X^{\mathcal{R}}$, then any of the operations that has yet to be assigned can be in either f_X or $f_X^{\mathcal{R}}$. Hence, the next phase of our algorithm enumerates through two sets of loop invariants; one that assigns the operation to f_X and another that assigns the operation to $f_X^{\mathcal{R}}$.

8.2.3 Enumeration

The Enumeration phase is executed when there are operations in the PME that have not been assigned to either f_X or $f_X^{\mathcal{R}}$.

An operation that has yet to be assigned exists because no constraint encountered so far requires it to be assigned either to f_X or $f_X^{\mathcal{R}}$. As such, the operation can be assigned either to f_X or $f_X^{\mathcal{R}}$. Therefore, this phase creates two sets of loop invariants where one set assigns the operation to f_X, and the other set assigns the same operation to $f_X^{\mathcal{R}}$.

Since assigning an operation to f_X or $f_X^{\mathcal{R}}$ may result in a new constraint being generated, the Construction phase is then repeated to propagate the new constraint to other regions such that the desired characteristic is preserved.

In addition, by assigning the operation to f_X (or $f_X^{\mathcal{R}}$), the number of unassigned operations is now reduced by one. Hence, we are ensured that progress is made towards assigning all operations in the PME to either f_X or $f_X^{\mathcal{R}}$, which is one of our two terminating conditions.

8.3 Illustration

We illustrate the constructive algorithm with an actual example, the merging of the Cholesky factorization, followed by the TRINV operation. These two operations are required in order to compute the inverse of a Symmetric Positive Definite matrix [8][1].

The inputs to our algorithm are as follows:

1. Theorem 6.4.1, which describes the status of computation required to ensure that loops can be merged, and

[1] The inversion of the Symmetric Positive Definite matrix is a sequence of three operations, but to illustrate the Enumeration phase, only the first two operations will be merged.

2. PMEs for the Cholesky factorization and TRINV, as given below:

   $\mathcal{F}$ = Cholesky

$$\mathcal{P}^{\mathcal{F}} = \left(\begin{array}{c|c} \widetilde{L}_{TL} \equiv Chol(\widehat{A}_{TL}) & 0 \\ \hline \widetilde{L}_{BL} \equiv \widehat{A}_{BL}\widetilde{L}_{TL}^{-T} & \widetilde{L}_{BR} \equiv Chol(\widehat{A}_{BR} - \widetilde{L}_{BL}\widetilde{L}_{BL}^{T}) \end{array} \right),$$

   where $L \equiv Chol(A) \ \wedge \ LL^T \equiv A$

   $\mathcal{G}$ = TRINV

$$\mathcal{P}^{\mathcal{G}} = \left(\begin{array}{c|c} \widetilde{T}_{TL} \equiv \widetilde{L}_{TL}^{-1} & 0 \\ \hline \widetilde{T}_{BL} \equiv \widetilde{L}_{BR}^{-1}\widetilde{L}_{BL}\widetilde{L}_{TL}^{-}1 & \widetilde{T}_{BR} \equiv \widetilde{L}_{BR}^{-1} \end{array} \right).$$

Initialization Phase In this phase, we need to identify operations that must be performed in the past and future. Recall that a loop invariant must assert that the final result has been computed at the end of the loop. This implies that

$$f_{TL}(L_{TL}) \equiv Chol(\widehat{A}_{TL}) \quad \text{and} \quad g_{TL}(T_{TL}) \equiv \widetilde{L}_{TL}^{-1}.$$

This implies that $\sigma(L_{TL})$ = FULLY UPDATED, and $\sigma(T_{TL})$ = FULLY UPDATED. Similarly, because the loop invariant also asserts that nothing has been computed at the start of the loop, then

$$g_{BR}^{\mathcal{R}}(T_{BR}) \equiv \widetilde{L}_{BR}^{-1},$$

and the Cholesky operation in the bottom-right region of $P^{\mathcal{F}}$ must be performed in $f_{BR}^{\mathcal{R}}$.

Construction Phase Since the Cholesky Factorization of L_{BR} cannot be performed, $\sigma(L_{BR}) \neq$ FULLY UPDATED. Theorem 6.4.1 tells us that L_{BR} must only be used in future computation. In addition, because L_{BR} is required to

compute T_{BL}, this implies that the operation $L_{BR}^{-1}L_{BL}$ must be computed in the future. Hence, $g_{BL}^{\mathcal{R}}$ must minimally compute $L_{BR}^{-1}L_{BL}$. Furthermore, this means that $\sigma(T_{BL}) \neq$ Fully Updated. This also implies that any operations involving L_{BR} in the post-condition for the Trinv cannot be fully updated.

For both $\mathcal{F}$ and $\mathcal{G}$, there are still operations in their post-conditions that we need to decide if they should be computed in the past or in the future. For example, for the Cholesky operation, we still need to determine if the operation $A_{BL}L_{TL}^{-T}$, required to compute L_{BL}, is performed in the past or in the future. This implies that the Enumeration Phase will be performed.

Enumeration Phase Since the operation $A_{BL}L_{TL}^{-T}$ for the Trinv operation can either be performed in the past or in the future, this phase of the algorithm will enumerate the different loop invariants that perform the operation $A_{BL}L_{TL}^{-T}$ in the past, followed by loop invariants that perform the operation in the future.

It is first assumed that $A_{BL}L_{TL}^{-T}$ is performed in the past. This implies that $\sigma(L_{BL}) =$ Fully Updated. Since there is a change in the status of $\sigma(L_{BL})$, the Construction Phase is repeated to propagate this change of status to other regions. It may be the case that at the end of the Construction Phase, there are still more operations that have not been divided between the loop invariant and its remainder. Then the Enumeration Phase is executed again, where now, in addition to performing the operation $A_{BL}L_{TL}^{-T}$ in the past, another operation (e.g. $A_{BR} - L_{BL}L_{BL}^{T}$) is also performed in the past. This

alternating execution of the Construction and Enumeration Phase creates a tree-like structure as shown in Figure 8.1.

When loop invariants that assume $A_{BL}L_{TL}^{-T}$ is performed in the past have been generated, the operation is then assigned to $f_X^{\mathcal{R}}$, indicating that it is performed in the future. Again, there is a change in the status of $\sigma(L_{BL})$ and the Construction phase is repeated. The resulting five pairs of loop invariants whose loops can be merged are given in Figure 8.2.

8.4 Summary

In this chapter, we introduced a constructive algorithm for deriving loop invariants for algorithms that possess a desired characteristic. With this constructive algorithm, the process of deriving a loop with a desired characteristic is now a goal-oriented process.

This generalizes the notion of goal-oriented programming advocated by eminent computer scientists such as Dijkstra [13, 14, 25], where the goal is not only to derive a loop that computes the correct output, but also to derive a loop that has a particular desired characteristic.

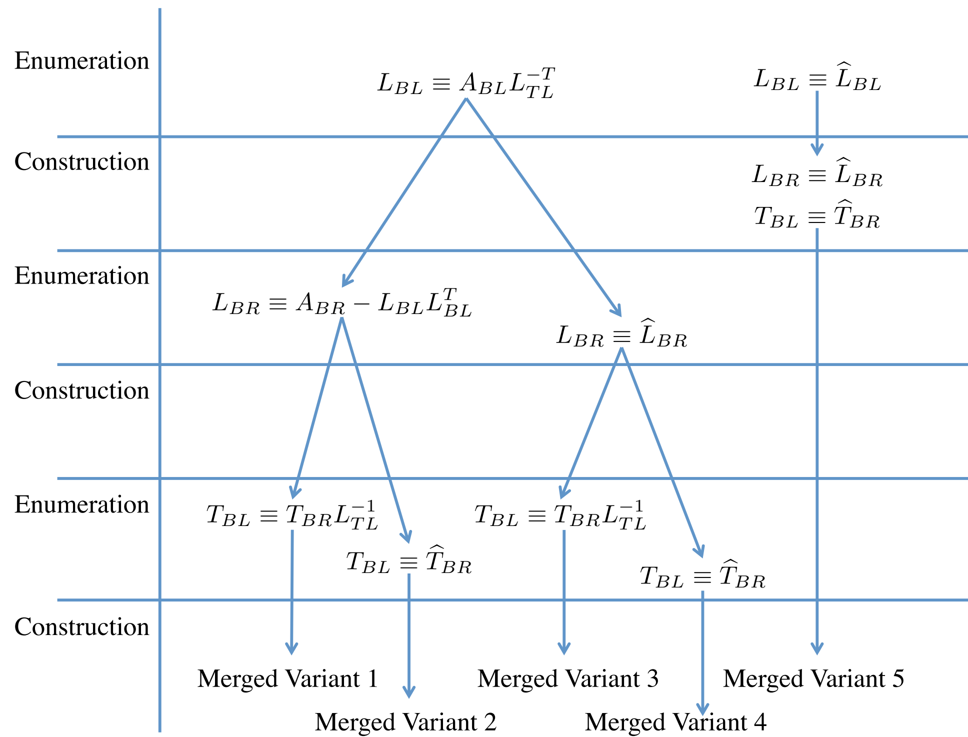


Figure 8.1: Enumerating different assignments of operations. Nodes represent f_X and g_X, edges represent paths taken by the algorithm when enumerating loop invariants

Merged Variant 1

$$\mathcal{I}^{\text{Cholesky}} = \left(\begin{array}{c|c} L_{TL} \equiv Chol(A_{TL}) & 0 \\ \hline L_{BL} \equiv A_{BL} L_{TL}^{-T} & L_{BR} \equiv A_{BR} - L_{BL} L_{BL}^{T} \end{array}\right),$$

$$\mathcal{I}^{\text{TRINV}} = \left(\begin{array}{c|c} T_{TL} \equiv L_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv L_{BR}^{-1} L_{BL} L_{TL}^{-1} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$$

Merged Variant 2

$$\mathcal{I}^{\text{Cholesky}} = \left(\begin{array}{c|c} L_{TL} \equiv Chol(A_{TL}) & 0 \\ \hline L_{BL} \equiv A_{BL} L_{TL}^{-T} & L_{BR} \equiv A_{BR} - L_{BL} L_{BL}^{T} \end{array}\right),$$

$$\mathcal{I}^{\text{TRINV}} = \left(\begin{array}{c|c} T_{TL} \equiv L_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv \widehat{T}_{BL} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$$

Merged Variant 3

$$\mathcal{I}^{\text{Cholesky}} = \left(\begin{array}{c|c} L_{TL} \equiv Chol(A_{TL}) & 0 \\ \hline L_{BL} \equiv A_{BL} L_{TL}^{-T} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right),$$

$$\mathcal{I}^{\text{TRINV}} = \left(\begin{array}{c|c} T_{TL} \equiv L_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv L_{BL} L_{TL}^{-1} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right).$$

Merged Variant 4

$$\mathcal{I}^{\text{Cholesky}} = \left(\begin{array}{c|c} L_{TL} \equiv Chol(A_{TL}) & 0 \\ \hline L_{BL} \equiv A_{BL} L_{TL}^{-T} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right),$$

$$\mathcal{I}^{\text{TRINV}} = \left(\begin{array}{c|c} T_{TL} \equiv L_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv \widehat{T}_{BL} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$$

Merged Variant 5

$$\mathcal{I}^{\text{Cholesky}} = \left(\begin{array}{c|c} L_{TL} \equiv Chol(A_{TL}) & 0 \\ \hline L_{BL} \equiv \widehat{L}_{BL} & L_{BR} \equiv \widehat{L}_{BR} \end{array}\right),$$

$$\mathcal{I}^{\text{TRINV}} = \left(\begin{array}{c|c} T_{TL} \equiv L_{TL}^{-1} & 0 \\ \hline T_{BL} \equiv \widehat{T}_{BL} & T_{BR} \equiv \widehat{T}_{BR} \end{array}\right)$$

Figure 8.2: Pairs of loop invariants whose loops can be merged

Chapter 9

Beyond Dense Linear Algebra

Recall that the fundamental observation is that an analysis of the code can be replaced by an analysis of the loop invariant because the loop invariant is the essence of the loop. If a loop invariant can be found, then that loop invariant is sufficient to derive an algorithm that computes the desired operation. The natural questions are "Can such a loop invariant be found?", and "Can the FLAME derivation process be relied upon to turn that loop invariant into an algorithm that can be implemented?". In this chapter, we examine the applicability of the approach beyond dense linear algebra.

9.1 Insights From Dense Linear Algebra Algorithms

In the DLA domain, algorithms for a desired DLA operation are implemented as a loop with finite iterations around sequences of other simpler DLA operations. This is inherently the nature of linear algebra operations, and is apparent when we examine the DLA software stack.

At the lowest level of the DLA software stack are the level 1 BLAS, which are operations on vectors. These operations are, themselves, loops around addition and/or multiplications of scalar elements. Using a single loop

with finite iterations around these level 1 BLAS subroutines, matrix-vector operations that make up the level 2 BLAS subroutines are then implemented. Level 3 BLAS routines and more complicated operations, incorporated in software libraries such as libFLAME and LAPACK, are then implemented as loops around the lower levels of the DLA software stack.

The FLAME derivation process captures these particular characteristics of DLA algorithms in that all FLAME derived algorithms are made up of a loop around other DLA operations and the loop has finite iterations. A finite number of iterations is ensured because the FLAME derivation process identifies a loop guard that is bounded by the size of one of the operands. When the loop has swept through the operand, the loop terminates.

These properties of algorithms derived using the FLAME derivation process implies that the operation is an example of a class of functions known as primitive recursive functions [12].

9.2 Primitive Recursive Functions (PRF)

The study of primitive recursive functions can be traced back to Dedekind in 1888, where he tried to understand the meaning of the natural numbers [12]. Skolem and other mathematicians showed that many interesting mathematical functions, such as addition, multiplication, exponentiation, the min (or max) function, are all primitive recursive [48]. However, it was only in 1935 that Rózsa Péter, considered the founder of Recursive Functions Theory [43], coined the term “primitive recursive”.

Definition 9.2.1. *The set of* Primitive Recursive Functions *is defined as the smallest set of functions that*

1. *Contains the following initial functions:*

   (a) *the Zero function,* $\mathcal{Z} : \mathbb{N}^n \to \mathbb{N}$ *where*

   $$\mathcal{O}(x_0, x_1, \ldots, x_{n-1}) = 0,$$

   (b) *the Successor function,* $\mathcal{S} : \mathbb{N} \to \mathbb{N}$ *where*

   $$\mathcal{S}(x) = x + 1,$$

   and

   (c) *the Identity (sometimes known as Projection) functions,* $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$ *where*

   $$\pi_i^n(x_0, x_1, \ldots, x_{n-1}) = x_i.$$

2. *Is closed under composition, i.e given primitive recursive functions* $\mathcal{G}, \mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_{k-1}$*:*

   $$\begin{array}{rcl} \mathcal{F}(x_0, x_1, \ldots, x_{n-1}) & = & \mathcal{G}(\; \mathcal{H}_0(x_0, x_1, \ldots, x_{n-1}), \\ & & \quad \mathcal{H}_1(x_0, x_1, \ldots, x_{n-1}), \ldots, \\ & & \quad \mathcal{H}_{k-1}(x_0, x_1, \ldots, x_{n-1})) \end{array}$$

3. *Is closed under primitive recursion, i.e:*

   $$\begin{array}{rcl} \mathcal{F}(x_0, x_1, \ldots, x_{n-1}, 0) & = & \mathcal{G}(x_0, x_1, \ldots, x_{n-1}) \\ \mathcal{F}(x_0, x_1, \ldots, x_{n-1}, \mathcal{S}(k)) & = & \mathcal{H}(x_0, x_1, \ldots, x_{n-1}, k, \mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k)), \end{array}$$

   where $\mathcal{H}$ *and* $\mathcal{G}$ *are primitive recursive functions, and the recursion is said to be with respect to variable* k*.*

Primitive recursive functions are defined on the natural numbers instead of floating point numbers or more complicated data structures such as vectors, matrices, tuples, trees and graphs. However, it has been shown that floating point numbers and other data structures can be mapped to the natural numbers using encoding and decoding functions (e.g. Cantor or Gödel encoding) that are themselves primitive recursive [46, 40]. Without loss of generality, we will continue the discussion assuming that there exists such encoding and decoding functions have been applied to map more complicated data structures to the natural numbers.

9.2.1 Important results regarding primitive recursive functions

A key result of primitive recursive functions that is pertinent to this discussion is that primitive recursive functions that are defined via primitive recursion can be defined iteratively [46, 22, 23].

Definition 9.2.2. *A function $\mathcal{F}$ is defined iteratively from the primitive recursive function $\mathcal{H}$ in the following manner:*

$$\begin{aligned} \mathcal{F}(x, 0) &= x \\ \mathcal{F}(x, \mathcal{S}(k)) &= \mathcal{H}^{\mathcal{S}(k)}(x) \\ &= \mathcal{H}^{k+1}(x) \\ &= \mathcal{H}(\mathcal{H}^{k}(x, k)) \\ &= \mathcal{H}(\mathcal{F}(x, k)), \end{aligned}$$

where $\mathcal{H}^k(x)$ denotes the result of applying $\mathcal{H}$ k times, successively.

Intuitively, the iterative definition of a primitive recursive function states that if no iteration is performed then nothing is computed and the

function simply returns the original value. Otherwise, the value computed in $k+1$ iterations is computed by applying the function $\mathcal{H}$ to the value computed after k iterations.

More generally, it is proven that a function is primitive recursive if and only if there exists a for-loop that computes the function [38].

9.3 FLAME algorithms compute primitive recursive functions

Let Ω be a DLA operation that can be derived using the FLAME methodology. This means that Ω can be computed with an algorithm in a finite number of iterations and the updates performed by the algorithm are DLA operations that can be derived using the FLAME process.

This implies that Ω is primitive recursive and there exists an iterative definition of Ω of the form:

$$\Omega(x) = \mathcal{F}(x, n) \quad \text{where} \left\{ \begin{array}{rcl} \mathcal{F}(x,0) & = & x, \\ \mathcal{F}(x,\mathcal{S}(k)) & = & \mathcal{H}(\mathcal{F}(x,k)). \end{array} \right.$$

Here, $\mathcal{H}$ is primitive recursive.

Notice that if the FLAME methodology is able to derive an algorithm for the function Ω, it means that the primitive recursive functions $\mathcal{F}$ and $\mathcal{H}$ are defined. In addition, there exists a parameter k which is monotonically decreasing due to the Successor function $\mathcal{S}$. We show, next, how these functions are related to different steps in the FLAME derivation process.

9.3.1 $\mathcal{F}$ and the loop invariant

Recall that the fundamental ingredient that underlies the FLAME derivation process is the identification of the loop invariant which asserts how the current values have been computed at the start and end of every iteration. Notice that the function $\mathcal{F}(x, k)$ computes the value obtained after k successive application of the function $\mathcal{H}$. In addition, the value computed by $\mathcal{F}(x, k)$ is an intermediate result that is used to compute the final value computed by $\Omega(x)$. Since the loop invariant describes how the current values after k iteration was computed, and the function $\mathcal{F}(x, k)$ computes the intermediate value after k applications of the function $\mathcal{H}$, it follows that the loop invariant must be equivalent to the function $\mathcal{F}(x, k)$, for $0 \leq k \leq n$.

9.3.2 $\mathcal{H}$ and the update statements

The last step of the FLAME derivation process is to derive the update statements by comparing the difference between the loop invariants described in Steps 6 and 7. Since the loop invariants in Steps 6 and 7 describe the value computed by the previous iteration and the value that will be computed at the end of the current iteration respectively, it implies that in the $\mathcal{S}(k)^{th}$ iteration of the loop, the loop invariant in Step 6 describes $\mathcal{F}(x, k)$ whereas the loop invariant in Step 7 describes $\mathcal{F}(x, \mathcal{S}(k))$. Since

$$\mathcal{F}(x, \mathcal{S}(k)) = \mathcal{H}(\mathcal{F}(x, k)),$$

it follows that $\mathcal{H}$ must be a function that represents all the update statements within the loop body. This implies that Step 8 of the FLAME derivation

process derives $\mathcal{H}$.

9.3.3 $\mathcal{S}(k)$ and sweeping through the operands

Recall that the FLAME derivation process ensures progress is made through the loop by systematically sweeping through the operands during Steps 5a and 5b. Steps 5a and 5b serve to repartition the regions of the operands to expose subregions and move these exposed subregions across the thick lines at the end of every iteration. When no more subregions can be exposed, the loop has ended.

The effect of Steps 5a and 5b is that the two steps ensure that the dimensions of the regions containing all of the operands at the start of the loop will decrease monotonically and will eventually become empty.

In the definition the function $\mathcal{F}$, the second parameter k and the Successor function $\mathcal{S}$ perform a similar function as Steps 5a and 5b in that they ensure progress is made and the recursion/iteration will eventually terminate.

9.4 Characterizing Domains Outside of Dense Linear Algebra

Now, we attempt to generalize the FLAME derivation process beyond the DLA domain. Specifically, we want to know if the FLAME derivation methodology is applicable when the desired function is primitive recursive and defined iteratively.

We know that if a primitive recursive function $\mathcal{F}$ is defined iteratively,

then there exist a primitive recursive function $\mathcal{H}$ and a particular value $\mathcal{S}(k)$ such that:

$$\mathcal{F}(x, \mathcal{S}(k)) \;=\; \mathcal{H}(F(x, k)).$$

For all $0 < i < \mathcal{S}(k)$, $\mathcal{F}(x, i)$ describes how intermediate values are computed after $\mathcal{H}$ is applied successively i times. In addition, $\mathcal{F}(x, i)$ describes the input values to $\mathcal{H}$ and $\mathcal{F}(x, \mathcal{S}(i))$ describes the output of applying $\mathcal{H}$ to $\mathcal{F}(x, i)$. Since $\mathcal{F}$ describes both the input and output of $\mathcal{H}$, $\mathcal{F}$ must be both the pre-condition and post-condition for $\mathcal{H}$. Therefore $\mathcal{F}$ must be, by definition, the loop invariant.

Now, given $\mathcal{F}$, we want to show that $\mathcal{H}$ can be derived from $\mathcal{F}$. We prove that $\mathcal{H}$ can be derived given $\mathcal{F}$, by proving the stronger statement that all primitive recursive functions can be derived from their pre-condition and post-condition.

Theorem 9.4.1. *If $\mathcal{F}$ is primitive recursive, then an algorithm that computes $\mathcal{F}$ can be derived from its pre-condition and post-condition.*

Proof: Since $\mathcal{F}$ is primitive recursive, then $\mathcal{F}$ must be defined as described in Definition 9.2.1. Hence, we prove the theorem via structural induction. Without loss of generality, let us assume that:

$$y = \mathcal{F}(x_0, x_1, \ldots, x_{n-1}),$$

and the pre-condition is given by:

$$P_{pre}: \quad y \equiv \widehat{y} \;\wedge\; \forall_{0 \le k < n} x_k \equiv \widehat{x}_k.$$

Base Case: $\mathcal{F}$ is an initial function.

We want to show that all three initial functions can be derived.

Case 1: $\mathcal{F}$ is the zero function, $\mathcal{O}$.

By definition, the zero function returns the value 0 for all possible input values. Therefore, the post-condition is given by $y \equiv 0$. Given the post-condition $y \equiv 0$, then an algorithm that computes $\mathcal{F}$ is

$$y := 0;$$

Case 2: $\mathcal{F}$ is the successor function, $\mathcal{S}$.

The successor function has exactly one input parameter, x_0, and returns the next value following the value of its input. Hence, its post-condition must be

$$\mathcal{P}_{post}: \quad n = 1 \ \wedge \ y \equiv x_0 + 1,$$

where n is the number of parameters of $\mathcal{F}$. Therefore, an algorithm that computes y must be

$$y := x_0 + 1;$$

Case 3: $\mathcal{F}$ is a projection function, π_i.

A projection function returns the value of one of the input parameters as the output value. Hence, for each projection function π_i, the post-condition of π_i is $y \equiv x_i$. Then, an algorithm that computes $\mathcal{F}$ is given by

$$y := x_i;$$

Inductive Cases

Inductive Hypothesis: Assume that there exists primitive recursive functions $\mathcal{G}, \mathcal{H},$ and $\mathcal{H}_i, i \in \mathbb{N}$ where algorithms that computes them can be derived from their pre-conditions and post-conditions. We want to show that an algorithm that computes a primitive recursive function $\mathcal{F}$, can be derived from its pre-condition and post-condition, if $\mathcal{F}$ is composed from the primitive recursive functions $\mathcal{G}, \mathcal{H},$ and $\mathcal{H}_i, i \in \mathbb{N}$.

Case 1: $\mathcal{F}$ is obtained via composition.

Since $\mathcal{F}$ is obtained via composition, there must exist primitive recursive functions $\mathcal{G}$ and $\mathcal{H}_i$, where $0 \leq i < k$ such that:

$$\begin{array}{rcll}\mathcal{F}(x_0, x_1, \ldots, x_{n-1}) & = & \mathcal{G}(& \mathcal{H}_0(x_0, x_1, \ldots, x_{n-1}), \\ & & & \mathcal{H}_1(x_0, x_1, \ldots, x_{n-1}), \ldots \\ & & & \mathcal{H}_{k-1}(x_0, x_1, \ldots, x_{n-1})).\end{array}$$

Notice that $\mathcal{F}$ can be computed in the following manner:

$$\begin{array}{rcl} y_0 & := & \mathcal{H}_0(x_0, x_1, \ldots, x_{n-1}) \\ y_1 & := & \mathcal{H}_1(x_0, x_1, \ldots, x_{n-1}) \\ & \vdots & \\ y_{k-1} & := & \mathcal{H}_{k-1}(x_0, x_1, \ldots, x_{n-1}) \\ y & := & \mathcal{G}(y_0, y_1, \ldots, y_{k-1}). \end{array}$$

This means that the post-condition for $\mathcal{F}$ can be described as follows:

$$P_{post} : \bigwedge_{0 \leq i < k} y_i \equiv H_i(x_0, x_1, \ldots, x_{n-1}) \ \wedge \ y \equiv \mathcal{G}(y_0, y_1, \ldots, y_{k-1}).$$

Hence, an algorithm for $\mathcal{F}$ can be derived if algorithms that compute $\mathcal{G}$ and $\mathcal{H}_i$ can be derived.

Since the input parameters of $\mathcal{F}$ are not changed by the sequence of computation, and the input parameters of $\mathcal{F}$ are the input parameters of $\mathcal{H}_i$ for all i, then the pre-conditions for all $\mathcal{H}_i$'s must be the pre-condition for $\mathcal{F}$. In addition, the post-conditions for $\mathcal{H}_i$'s must be $y_i \equiv H_i(x_0, x_1, \ldots, x_{n-1})$.

The parameters of $\mathcal{G}$ are the outputs of $\mathcal{H}_i$'s. This implies that the pre-condition for $\mathcal{G}$ must be:

$$\bigwedge_{0 \leq i < k} y_i \equiv \mathcal{H}_i(x_0, x_1, \ldots, x_{n-1}).$$

In addition, because $\mathcal{G}$ must compute the final result y, it follows that the post-condition for $\mathcal{G}$ must be the same as the post-condition for $\mathcal{F}$.

Since the pre-condition and post-conditions for $\mathcal{G}$ and $\mathcal{H}_i$'s can be identified from the pre-condition and post-condition of $\mathcal{F}$, then the inductive hypothesis allows us to conclude that algorithms for $\mathcal{G}$ and $\mathcal{H}_i$'s can be derived. Hence, an algorithm that computes $\mathcal{F}$ can be derived.

Case 2: $\mathcal{F}$ is obtained via primitive recursion.

Recall that functions obtained via primitive recursion can be defined iteratively. This implies that there exist a primitive recursive function $\mathcal{H}$ and a particular value $\mathcal{S}(k)$ such that:

$$\begin{aligned} \mathcal{F}(x_0, x_1, \ldots, x_{n-1}, \mathcal{S}(k)) &= \mathcal{H}^{\mathcal{S}(k)}(x_0, x_1, \ldots, x_{n-1}) \\ &= \mathcal{H}(\mathcal{H}^k(x_0, x_1, \ldots, x_{n-1})) \\ &= \mathcal{H}(F(x_0, x_1, \ldots, x_{n-1}, k)). \end{aligned}$$

This implies that $\mathcal{F}$ can be computed in the following manner:

$$\begin{array}{l}
y_0 \;:=\; (x_0, x_1, \ldots, x_{n-1}) \\
\;i \;:=\; 0 \\
while(i \leq k)\{ \\
\quad y_{i+1} \;:=\; \mathcal{H}(y_i) \\
\quad\quad i \;:=\; i+1 \\
\} \\
y \;:=\; y_{\mathcal{S}(k)}
\end{array}$$

and the post-condition for $\mathcal{F}$ is given by:

$$\begin{array}{rll}
P_{post}: & \forall_{0 \leq i \leq k}\; y_{i+1} \equiv \mathcal{H}(y_i) & \wedge \\
& y \equiv y_{\mathcal{S}(k)} & \wedge \\
& y_0 \equiv (\widehat{x}_0, \widehat{x}_1, \ldots, \widehat{x}_{n-1}) &
\end{array}$$

If an algorithm can be derived for computing $\mathcal{H}$, then an algorithm for $\mathcal{F}$ can be derived from its pre-condition and post-condition. This implies that we need to identify the pre-condition and post-condition for $\mathcal{H}$.

Since

$$\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k) \equiv \mathcal{H}(\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k-1)),$$

the $\mathcal{H}$ computes the desired output when the input is $\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k-1)$. Hence, the post-condition of $\mathcal{H}$ must be the post-condition of $\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k)$. In addition, the input to $\mathcal{H}$ is the output of $\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k-1)$. This means that the pre-condition for $\mathcal{H}$ must be the post-condition of $\mathcal{F}(x_0, x_1, \ldots, x_{n-1}, k-1)$. Hence the pre-condition for $\mathcal{H}$ is given by:

$$\begin{array}{ll}
\forall_{0 \leq i < k-1}\; y_{i+1} \equiv \mathcal{H}(y_i) & \wedge \\
y \equiv y_{k-1} & \wedge \\
y_0 \equiv (\widehat{x}_0, \widehat{x}_1, \ldots, \widehat{x}_{n-1}) &
\end{array}$$

Since the pre-condition and post-condition for $\mathcal{H}$ can be derived, then our induction hypothesis tells us that an algorithm for $\mathcal{H}$ can be derived. This implies that an algorithm that computes $\mathcal{F}$ can be derived from its pre-condition and post-condition.

Since algorithms for $\mathcal{F}$ can be derived for the base case where $\mathcal{F}$ is an initial function, and the inductive cases where $\mathcal{F}$ is either obtained via composition or primitive recursion, then by structural induction, all primitive recursive functions can be derived from their pre-conditions and post-conditions. □

9.5 Summary

In this section, we showed that the DLA algorithms derived using the FLAME derivation methodology compute functions that belong to the class of primitive recursive functions. In particular, we showed that the different steps in the FLAME derivation methodology map to the different primitive recursive functions, $\mathcal{F}, \mathcal{H}$, and $\mathcal{S}$ in the iterative definition of a primitive recursive function.

More importantly, we showed that primitive recursive functions that are defined iteratively have a loop invariant described by the function $\mathcal{F}$. In addition, we proved that given $\mathcal{F}$, algorithms that computes these primitive recursive functions can be derived. This implies that the FLAME derivation methodology extends to other domains beyond dense linear algebra as long as $\mathcal{F}$ can be described.

Chapter 10

Conclusion and Future Directions

The thesis of this dissertation is that the loop invariant can be used to formally derive loops that have the potential for attaining high performance, in a goal-oriented fashion. In this dissertation, we provided evidence supporting the thesis by demonstrating that characteristics of the loop that impact performance can be identified *a priori* from the loop invariant. In addition, we introduced a constructive algorithm for identifying loop invariants of loops that possess the desired characteristic when implemented.

10.1 Results

In this section, we reiterate the contributions of this work to computer science.

Loop invariants contain information regarding the performance characteristic of the loop. For the domain of DLA, we showed that performance characteristics of loops can be identified from their loop invariants (and remainders). This result supports the thesis of this dissertation in that the loop invariant is used to identify algorithms with specific performance char-

acteristics. The theories developed in this dissertation allow one to analyze and identify loop invariants for a given operation such that the identified loop invariants yield algorithms that have the potential to perform well on a given machine architecture.

Generalization of goal-oriented programming. The generalization of goal-oriented programming is demonstrated through the introduction of a constructive algorithm for identifying loop invariants with desired characteristics. In this dissertation, we generalized the notion of a goal from obtaining a provably correct loop to a goal that requires the derived loop to *also* possess a particular desirable characteristic.

Avoiding the phase-ordering problem when optimizing DLA loops. We showed that a goal-oriented approach towards optimizing DLA loops allows one to avoid the phase-ordering problem when optimizing DLA loops. By deriving the loop invariant(s) with the desired characteristic, one avoids having to identify the sequence of transformations that will restructure the input loop into an output loop with the desired characteristic. The goal-oriented approach has the additional benefit of potentially yielding multiple loop invariants with the desired characteristic. As there may be multiple loops with the desired characteristic, identifying different sequences of transformations would exacerbate the phase ordering-problem.

Unified optimization framework for loop transformations and DLA expert knowledge. For the domain of DLA, we showed that many loop transformations automatically applied by traditional compilers and heuristics used by the DLA expert can be unified under a framework based on a calculus of loop invariants. Through this unified framework, one can reap the benefits of compiler loop transformation and/or use expert knowledge as leverage to optimize DLA algorithms, without being hindered by the different representations (e.g., explicit indices, or black-box subroutine calls) required by the two approaches. We showed that dependence analysis and legality of commonly employed loop transformations in DLA can be determined through an analysis of the status of computation. This higher level of abstraction with the loop invariant allows analysis and optimization to be performed on loop-based code that includes calls to black-box libraries. DxTer [34], a prototype system that implements Design by Transformation (DxT) [35], implements these ideas by using status of computation to simplify the analysis and optimization of the DLA algorithms.

10.2 Future Work

Possible directions that build and extend the work in this dissertation are suggested in this section.

Extension of the theories. Recall that the prototypical operation consists of a single output operand that has been partitioned into four regions. As the

FLAME derivation methodology has been applied to operations where there are more than one output operand [50] or the output operand has been partitioned into more than four regions [18, 19], a natural extension is to prove that the analysis and theories in this dissertation extend to these operations that do not conform to our prototypical operation. For these non-conforming operations, the inputs to the FLAME derivation methodology are the PME and loop invariants that describe how regions of the output have been computed, which are also the exact inputs required for the analysis and theories developed in this dissertation. Hence, we believe that work in this dissertation can be extended to these non-conforming operations.

Non-performance related properties. We provided concrete examples for using the loop invariant to identify performance characteristics of the loop. In a future where power consumption of a chip is increasingly becoming a limit [20], the measure of what is a desired characteristic of an algorithm may be GFLOPS/Watt instead of maximum GFLOPS attained. Architecture research has shown that significant reduction in power consumption can be attained through delayed normalization, where intermediate values are kept in the accumulator for as long as possible [52, 42]. As each algorithm is unique in the way data is accessed and computed, we conjecture that identifying algorithms that allow normalization to be delayed as long as possible may, in essence, be similar to finding algorithms with the "right" data access and computation behavior. The "right" data access and computation would then

be the desired characteristic that we attempt to extract from the loop invariant of the loop.

Loop invariants of domains outside of DLA. The analysis and theories developed in this dissertation were based on the observation that a loop, in the DLA domain, sweeps through inputs and the output in a systematic manner. Hence, the loop invariant for such a loop can be described in terms of the status of computation of the different regions of the output operand. In addition, we showed that every primitive recursive function that is defined by primitive recursion has a loop invariant, and that given a loop invariant for a primitive recursive function, an algorithm that computes the function can be derived from the loop invariant.

Hence, a domain where (1) the operations are primitive recursive functions, (2) loops sweep through data in a systematic manner, and (3) loop invariants can be described in terms of the status of computation, seems to hold potential for extending the work in this dissertation. The natural question is "How would one describe such a loop invariant in domains outside of DLA?". A detailed analysis of loop-based algorithms in other domains is warranted.

Appendix

Appendix 1

Table of Symbols

$\alpha, \beta, \ldots$	Scalars variables
$a, b, \ldots$	Vectors variables
$A, B \ldots$	Matrices variables
$a_T, B_{TL}, \ldots$	Regions of variables
$\widehat{\alpha}, \widehat{a}, \widehat{A}, \ldots$	Original values in the variables α, a, and A
$\widetilde{\alpha}, \widetilde{a}, \widetilde{A}, \ldots$	Final values in the variables α, a, and A
$\mathcal{F}, \mathcal{G}, \mathcal{H}$	Operations for which algorithms need to be found
$\mathcal{P}^{\mathcal{F}}$	Partitioned Matrix Expression (PME) for the operation $\mathcal{F}$
$\mathcal{I}^{\mathcal{F}}$	A loop invariant for the operation $\mathcal{F}$
$\mathcal{R}_{\mathcal{I}^{\mathcal{F}}}$	The Remainder of loop invariant $\mathcal{I}^{\mathcal{F}}$
$\mathcal{F}_{TL}, \mathcal{G}_{TR}, \ldots$	Expression for different regions in the Partitioned Matrix Expression
$f_{TL}, g_{TR}, \ldots$	Functions that represent how the different regions in the loop invariant can be computed from the original values
$f^{\mathcal{R}}_{TL}, g^{\mathcal{R}}_{TR}, \ldots$	Functions that represent how the different regions in the Remainder need to be computed to obtain the final values
$\sigma(x)$	Status of computation for the region x

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] John R. Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 19(6):233–246, June 1984.

[3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *SIGPLAN Not.*, 39(7):231–239, June 2004.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. *SIGPLAN Not.*, 44(6):38–49, June 2009.

[6] Geoffrey Belter, E.R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *International*

Conference on High Performance Computing Networking, Storage and Analysis (SC), 2009.

[7] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, Tyler Rhodes, Robert A. Geijn, and Field G. Van Zee. Deriving dense linear algebra libraries. *Formal Aspects of Computing*, pages 1–13, 2012.

[8] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):1–22, 2008.

[9] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[10] Arun Chauhan, Cheryl McCosh, Ken Kennedy, and Richard Hanson. Automatic type-driven library generation for telescoping languages. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 51–. ACM, 2003.

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[12] Richard Dedekind. *Was sind und was sollen die Zahlen?* F. Vieweg und sohn., 1893.

[13] Edsger. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[14] Edsger.W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[15] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[16] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[17] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.*, 47(8):225–234, February 2012.

[18] Victor Eijkhout, Paolo Bientinesi, and Robert van de Geijn. Proof-driven derivation of Krylov solver libraries. Technical Report TR-10-02, The University of Texas at Austin, Department of Computer Sciences, 2010.

[19] Victor Eijkhout, Paolo Bientinesi, and Robert van de Geijn. Towards mechanical derivation of Krylov solver libraries. *Procedia Computer Science*, 1(1):1805 – 1813, 2010. ICCS 2010.

[20] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[21] Diego Fabregat-Traver and Paolo Bientinesi. Automatic generation of loop-invariants for matrix operations. In *Computational Science and its Applications, International Conference*, pages 82–92, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[22] M. D. Gladstone. A reduction of the recursion scheme. *Journal of Symbolic Logic*, 32(4):505–508, Dec 1967.

[23] M. D. Gladstone. Simplification of the recursion scheme. *Journal of Symbolic Logic*, 36(4):653–665, Dec 1971.

[24] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12, May 2008. Article 12, 25 pages.

[25] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[26] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*,

Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[27] John A. Gunnels, Robert A. van de Geijn, Daniel S. Katz, and Enrique S. Quintana-Ortí. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pages 47–56, Washington, DC, USA, 2001. IEEE Computer Society.

[28] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. *SIGPLAN Not.*, 35(1):39–52, December 1999.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, October 1969.

[30] Bo Kågström, Per Ling, and Charles Van Loan. Portable high performance GEMM-based level 3 BLAS. *Parallel Processing for Scientific Computing*, pages 330–346, 1993.

[31] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer Berlin Heidelberg, 1994.

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on*

Mathematical Software, 5(3):308–323, September 1979.

[33] Tze Meng Low, Robert A. van de Geijn, and Field G. Van Zee. Extracting SMP parallelism for dense linear algebra algorithms from high-level specifications. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP'05, pages 153–163, New York, NY, USA, 2005. ACM.

[34] Bryan Marker, Don Batory, and C.T. Shepherd. DxTer: A dense linear algebra program synthesizer. Technical Report TR-12-17, The University of Texas at Austin, Department of Computer Sciences, 2012.

[35] Bryan Marker, Jack Poulson, Don Batory, and Robert van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In Michel Dayd, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 362–378. Springer Berlin Heidelberg, 2013.

[36] Daniel Marques, Greg Bronevetsky, Rohit Fernandes, Keshav Pingali, and Paul Stodghil. Optimizing checkpoint sizes in the c3 system. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10 - Volume 11*, IPDPS '05, pages 226.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Tze Meng Low, Bryan Marker, and Robert van de Geijn. FLAME Working Note #64. Theory and practice of fusing loops when optimizing

parallel dense linear algebra operations. Technical Report TR-12-18, The University of Texas at Austin, Department of Computer Sciences, August 2012.

[38] Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd national conference*, ACM '67, pages 465–469, New York, NY, USA, 1967. ACM.

[39] Nichamon Naksinehaboon, Yudan Liu, Chokchai B. Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott. Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments. In *Proc. 8th Intl. Symp. on Cluster Computing and the Grid (CCGRID)*, 2008.

[40] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Number v. 1 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1992.

[41] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[42] Ardavan Pedram, Andreas Gerstlauer, and Robert A. van de Geijn. A high-performance, low-power linear algebra core. In *Proceedings of the ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, ASAP '11, pages 35–42, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Rózsa Péter. Über den zusammenhang der verschiedenen begriffe der rekursiven funktion. *Mathematische Annalen*, 110(1):612–632, 1935.

[44] Jack Poulson, Robert van de Geijn, and Jeffrey Bennighof. Parallel algorithms for reducing the generalized Hermitian-definite eigenvalue problem. FLAME Working Note #56. Technical Report TR-11-05, The University of Texas at Austin, Department of Computer Sciences, February 2011.

[45] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[46] Raphael M Robinson. Primitive recursive functions. *Bull. Amer. Math. Soc*, 53(10):925–942, 1947.

[47] Sharad Singhai and Kathryn McKinley. Loop fusion for data locality and parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*. Citeseer, 1996.

[48] Thoralf A. Skolem. *Begründung der Elementaren Arithmetik, etc.* Videnskapsselskapets. Skrifter. 1. Math.-naturv. Klasse. 1923.

[49] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[50] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. lulu.com, 2008.

[51] Field G. Van Zee and Robert A. van de Geijn. FLAME Working Note #66. BLIS: A framework for generating BLAS-like libraries. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Sciences, November 2012.

[52] Sriram R Vangal, Yatin V Hoskote, Nitin Y Borkar, and Atila Alvandpour. A 6.2-gflops floating-point multiply-accumulator with conditional normalization. *Solid-State Circuits, IEEE Journal of*, 41(10):2314–2323, 2006.

[53] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.