Theory and Practice of Fusing Loops when Optimizing Parallel Dense Linear Algebra Operations

FLAME Working Note #64

Tze Meng Low
Bryan Marker
Robert van de Geijn

Department of Computer Science
The University of Texas at Austin
Austin, TX 7872
ltm@mezmedia.com,
bamarker@gmail.com,
rvdg@cs.utexas.edu

August 20, 2012

Abstract

Dense linear algebra (DLA) algorithms for distributed memory architectures are often implemented as sequences of highly optimized parallel implementations of individual sub-operations. This can hinder performance because better load balancing and/or reduction in communication overhead can be achieved when sub-operations are merged/fused together. In practice merging of parallel implementations is often avoided because each DLA operation can be instantiated through multiple algorithms and each such algorithm has its various implementations, creating a combinatorial search space. Our approach divides the optimization problem into two phases: one which identifies fusion-compatible algorithms and one that transforms the merged algorithms into high-performance parallel implementations. By exploiting high-level information related to loop invariants, multiple sets of fusion-compatible algorithms can be found without a need for an exhaustive examination of different combinations of implementations. Through the use of an automated tool, DxTer, that understands the cost of computation and collective communications, the merged algorithm that analytically attains the best performance can then be identified and code can be generated. We illustrate the benefits of this two-phase approach with performance numbers on an IBM Blue Gene/P system. This takes us one step closer to the complete mechanical generation of dense linear algebra libraries from expert knowledge.

1 Introduction

Our ultimate goal is to mechanically generate high-performance dense linear algebra (DLA) libraries. Input will be a high-level specification of the operations to be supported, expert knowledge about these algorithms, and knowledge about a target architecture. Output will be a highly optimized library for the target architecture. To reach this goal, what an expert does has to be carefully examined and made systematic to the point where it can be automated. One thing that an expert does when developing high-performance code for distributed memory architectures (clusters) is to merge loops so as to expose more opportunities for optimization [5]. This paper focuses on the theory and practice of automating this. While the results are general, we illustrate them within the setting of DLA and the FLAME project [11].

We attack the problem with a two-phase approach. Given two DLA operations, the first phase identifies pairs of loop-based algorithms that can be merged. For DLA, loop invariants (in the sense of Dijkstra and Hoare) can be constructed from the mathematical specifications of a given operation [4]. We prove that the mathematical specifications of the operations and the loop invariants provide all information needed to determine the legality of loop fusion. The second phase optimizes each resulting merged loop via Design-by-Transformation (DxT, pronounced “dext”) [16], an approach to software engineering that codifies design knowledge. A prototype tool that builds on DxT, DxTer, implements this second phase, exploring and choosing the best implementation based on cost estimates. Together they accomplish what an expert would do: identify multiple loops for each operation, determine which can be merged, analyze implementations, and generate code for the best solution.

2 Motivation

In this section, we motivate the two-phase approach. We do so by considering lower triangular matrix L and the operation that overwrites that matrix with its inverse, TriInv, and then multiplies the transpose of the result times itself, overwriting L with the lower triangular part (since the resulting matrix is symmetric), TrTrmm (Triangular Triangular Matrix Multiplication): $L := L^{-1}$; $L := L^T L$. This sequence of the two operations are the last two steps in computing the inversion of the symmetric positive definite matrix[5]. More importantly, they are representative of a large class of DLA operations.

2.1 Algorithms

The FLAME approach [4] to deriving algorithms yields three algorithmic variants[1] for each of these operations, presented in the left two columns of Figure 1 using the FLAME notation. For this paper, one has to understand only part of the FLAME approach for deriving algorithms. First, the triangular matrix is partitioned into quadrants: $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$. These quadrants track how the computation marches through the matrix (see Figure 1). Second, the final contents of L, in terms of these quadrants, are given by

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^{-1} & 0 \\ \hline L_{BL} \equiv -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} \equiv L_{BR}^{-1} \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^{T}\hat{L}_{TL} + L_{BL}^{T}\hat{L}_{BL} & \star \\ \hline L_{BL} \equiv \hat{L}_{BR}^{T}\hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR}^{T}\hat{L}_{BR} \end{array}\right)$$

for $L := L^{-1}$ and $L := L^T L$, respectively. Here $\hat{L}$ represents the original contents of L, 0 represents a block of zero elements, and $\star$ the symmetric part of the matrix that is neither stored nor computed. These are the recursive definitions of the operations, which we call the Partitioned Matrix Expressions (PMEs) of the respective operations. The PME is a succinct description of all the computations that needs to be performed in terms of regions of the matrix in order to compute the given DLA operation.

Finally, the state of the quadrants of L before and after each iteration for each of the algorithms in Figure 1 can be described by the conditions given above the algorithms in Figure 1. These conditions are called the loop invariants for the loop: Variant i maintains the state indicated by Invariant i above the

[1]Actually, the methodology yields more algorithmic variants than given in Figure 1. However, we will only need the given subset for our discussion.

$L := L^{-1}$

PME:

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

TriInv Invariant 1:

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

TriInv Invariant 2:

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^{-1} & \\ \hline L_{BL} \equiv \\ \quad -\hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} \equiv \hat{L}_{BR} \end{array}\right)$$

TriInv Invariant 3:

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

```
Algorithm: $L := L^{-1}$
Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)$
    where $L_{TL}$ is $0 \times 0$
while $m(L_{TL}) < m(L)$ do
  Determine block size $b$
  Repartition
  $\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
     where $A_{11}$ is $b \times b$
  ----------------------------
  TriInv Variant 1
  $L_{10} := L_{10}L_{00}$
  $L_{10} := -L_{11}^{-1}L_{10}$
  $L_{11} := L_{11}^{-1}$
  ----------------------------
  TriInv Variant 2
  $L_{21} := -L_{21}L_{11}^{-1}$
  $L_{21} := L_{22}^{-1}L_{21}$
  $L_{11} := L_{11}^{-1}$
  ----------------------------
  TriInv Variant 3
  $L_{10} := L_{11}^{-1}L_{10}$
  $L_{20} := L_{20} - L_{21}L_{10}$
  $L_{21} := -L_{21}L_{11}^{-1}$
  $L_{11} := L_{11}^{-1}$
  ----------------------------
  Continue with
  $\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
endwhile
```

$L := L^T L$

PME:

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^T\hat{L}_{TL} \\ \quad + \hat{L}_{BL}^T\hat{L}_{BL} & \\ \hline L_{BL} \equiv \hat{L}_{BR}^T\hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR}^T\hat{L}_{BR} \end{array}\right)$$

Trtrmm Invariant 1:

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^T\hat{L}_{TL} & \\ \hline L_{BL} \equiv \hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR} \end{array}\right)$$

Trtrmm Invariant 2:

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^T\hat{L}_{TL} \\ \quad + \hat{L}_{BL}^T\hat{L}_{BL} & \\ \hline L_{BL} \equiv \hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR} \end{array}\right)$$

Trtrmm Invariant 3:

$$\left(\begin{array}{c|c} L_{TL} \equiv \hat{L}_{TL}^T\hat{L}_{TL} \\ \quad + \hat{L}_{BL}^T\hat{L}_{BL} & \\ \hline L_{BL} \equiv \hat{L}_{BR}^T\hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR} \end{array}\right)$$

```
Algorithm: $L : L^T L$
Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right)$
    where $L_{TL}$ is $0 \times 0$
while $m(L_{TL}) < m(L)$ do
  Determine block size $b$
  Repartition
  $\left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & * & * \\ \hline L_{10} & L_{11} & * \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
     where $A_{11}$ is $b \times b$
  ----------------------------
  Trtrmm Variant 1
  $L_{00} := L_{10}^T L_{10} + L_{00}$
  $L_{10} := L_{11}^T L_{10}$
  $L_{11} := L_{11}^T L_{11}$
  ----------------------------
  Trtrmm Variant 2
  $L_{10} := L_{11}^T L_{10}$
  $L_{10} := L_{21}^T L_{20} + L_{10}$
  $L_{11} := L_{11}^T L_{11}$
  $L_{11} := L_{21}^T L_{21} + L_{11}$
  ----------------------------
  Trtrmm Variant 3
  $L_{11} := L_{11}^T L_{11}$
  $L_{11} := L_{21}^T L_{21} + L_{11}$
  $L_{21} := L_{22}^T L_{21}$
  ----------------------------
  Continue with
  $\left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & * & * \\ \hline L_{10} & L_{11} & * \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
endwhile
```

$L := L^{-1};\ L := L^T L$

```
Algorithm: $L := L^{-1};\ L := L^T L$
Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right)$
    where $L_{TL}$ is $0 \times 0$
while $m(L_{TL}) < m(L)$ do
  Determine block size $b$
  Repartition
  $\left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & * & * \\ \hline L_{10} & L_{11} & * \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
     where $A_{11}$ is $b \times b$
  ----------------------------
  Merged Variant 1
   $L_{21} := -L_{21}L_{11}^{-1}$            }
   $L_{21} := L_{22}^{-1}L_{21}$             } TriInv Variant 2
   $L_{11} := L_{11}^{-1}$                   }
   $L_{00} := L_{10}^T L_{10} + L_{00}$      }
   $L_{10} := L_{11}^T L_{10}$               } Trtrmm Variant 1
   $L_{11} := L_{11}^T L_{11}$               }
  ----------------------------
  Merged Variant 2
   $L_{21} := -L_{21}L_{11}^{-1}$            }
   $L_{21} := L_{22}^{-1}L_{21}$             } TriInv Variant 2
   $L_{11} := L_{11}^{-1}$                   }
   $L_{10} := L_{11}^T L_{10}$               }
   $L_{10} := L_{21}^T L_{20} + L_{10}$      } Trtrmm Variant 2
   $L_{11} := L_{11}^T L_{11}$               }
   $L_{11} := L_{21}^T L_{21} + L_{11}$      }
  ----------------------------
  Merged Variant 3
   $L_{10} := L_{11}^{-1}L_{10}$             }
   $L_{20} := L_{20} - L_{21}L_{10}$         } TriInv Variant 3
   $L_{21} := -L_{21}L_{11}^{-1}$            }
   $L_{11} := L_{11}^{-1}$                   }
   $L_{00} := L_{10}^T L_{10} + L_{00}$      }
   $L_{10} := L_{11}^T L_{10}$               } Trtrmm Variant 1
   $L_{11} := L_{11}^T L_{11}$               }
  ----------------------------
  Continue with
  $\left(\begin{array}{c|c} L_{TL} & * \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & * & * \\ \hline L_{10} & L_{11} & * \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$
endwhile
```

Figure 1: Algorithms for the two separate operations and the resulting merged algorithms. Updates like $L_{ij} := L_{ii}^{-1}L_{ij}$ are actually implemented as a *triangular solve with multiple right-hand sides* rather than inverting L_{ii} and multiplying. This is important for numerical stability reasons.

algorithm. A loop invariant describes which quadrants of the output matrix have been updated, how the quadrants have been updated and where the values used to update the quadrants originate.

In the FLAME methodology, the loop invariants are systematically derived from the PME and the algorithm is then derived from the loop invariant. For this paper, it suffice to know that that a loop invariant has been derived from the PME and each loop invariant represents an algorithm. The details of deriving the algorithm from the loop invariant is not pertinent.

2.2 Merging the algorithms

In [18, 5], it is extensively documented that, for operations very similar to those discussed in this paper, merging loops of DLA algorithms is beneficial, especially on cluster. There are at least two reasons: (1) Merging the loops exposes opportunities to combine communications[2] required for the operations in each of the separate loops and (2) the computation can often be better load balanced.

Now, in order to merge loops, there must be implementations of the algorithms available that can be merged. In ScaLAPACK [8], TriInv and TrTrmm are implemented by the functions `PDTRTRI` and `PDLAUUM` respectively. Examining the implementations of these functions in that library, one quickly recognizes that the loops for the chosen algorithms cannot be merged. In `PDTRTRI`, $L := L^{-1}$ is computed row-wise from the bottom row upwards (an algorithm not given in Figure 1 for space reasons) and `PDLAUUM` computes $L := L^T L$ row-wise from the top row downwards. As the two implementations march through L in opposite directions, they cannot be merged. Dependencies between the different regions of L also prevent the application of simple loop transformations such as loop reversal to transform the implementations into compatible versions for loop fusion to take place.

However, by carefully selecting the right algorithmic variants for the two operations, the expert can find multiple algorithmic variants that result from merging algorithms, as illustrated in Figure 1(Right). The benefit can be dramatic, as illustrated in Figure 2 where we show performance estimates generated by our automated system DxTer, to be discussed in Section 4. Of the three possible merged algorithms, only one merged variant (Variant 3) is predicted to perform better than the unmerged operations found in ScaLAPACK. This implies that (1) it is necessary to find multiple pairs of algorithms that can be fused and (2) it is important to be able to pick the correct merged algorithm to implement and optimize.

This still leaves a number of questions in our quest to automate the actions of an expert: (1) How does one easily determine that two algorithms can be merged? For a broad class of DLA algorithms, the answer can be found in how the loop invariants relate to the PMEs so that this determination can be easily automated; (2) Even if we know that there exist algorithms that can be merged, what if these algorithms are not found in ScaLAPACK? Fortunately, Elemental [17], a modern alternative to ScaLAPACK, *does* implement all the needed algorithmic variants. In Section 3, we also describe a constructive approach to identifying fusion-compatible loop invariants such that the derived algorithms can be merged; (3) Once merged, how do we optimize the merged loop and differentiate between all possible optimized variants? For this we employ DxT, as described later in the paper.

3 Phase 1: Principles for Identifying Fusion-Compatible Algorithms

The fundamental thesis of the first phase of our two-phase approach is that the PMEs and the loop invariants contain all information necessary to determine whether two loops can be merged.

[2]In the case of DLA, these communications are invariably collective in nature [20, 23].

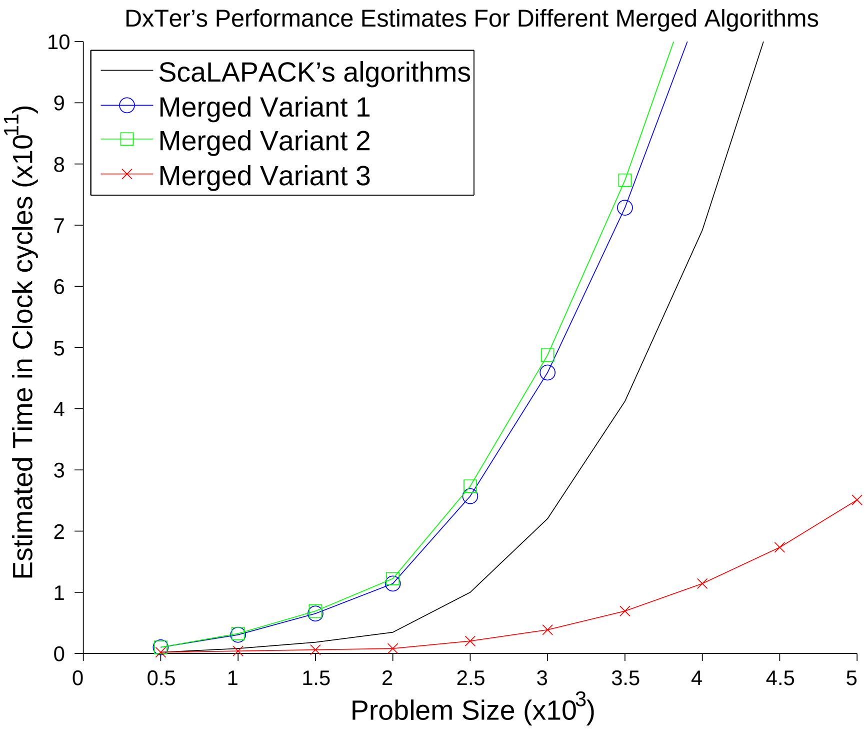

Figure 2: Performance Estimate of different merged algorithms compared to ScaLAPACK's unmerged algorithms

3.1 Example

Let us focus on TriInv Invariant 2 and Trtrmm Invariant 1:

$$\left(\begin{array}{c|c} X_{TL} \equiv \hat{L}_{TL}^{-1} & \\ \hline X_{BL} \equiv \cancel{-\hat{L}_{BR}^{-1}}\hat{L}_{BL}\hat{L}_{TL}^{-1} & X_{BR} \equiv \cancel{\hat{L}_{BR}^{-1}} \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|c} L_{TL} \equiv X_{TL}^{T}X_{TL}\cancel{+X_{BL}^{T}X_{BL}} & \\ \hline L_{BL} \equiv \cancel{X_{BR}^{T}}X_{BL} & L_{BR} \equiv \cancel{X_{BR}^{T}}X_{BR} \end{array}\right).$$

We show the loop invariant by striking out parts of the PME, to emphasize how the loop invariant and PME differ. Here we use X in order to indicate that the output of the first loop, X, is input to the second loop. It is important to remember that X overwrites L.

By comparing the loop invariant and the PME for the second operation, we notice that the second loop requires X_{TL} to partially update the top-left quadrant. Since X_{TL} was completely computed by the first operation (as no parts of the expression in the top-left quadrant was struck out and, thus, left uncomputed), the information is available for the second loop at the time the computation happens. Also, the first loop only partially updated the bottom-left quadrant, which means the second loop cannot yet alter values in the bottom-left quadrant, which it has not. We know this because all operations in the bottom-left quadrant have been struck out (and thus none of that computation has been performed). Finally, no information needed by the first loop has been overwritten by the second loop (the bottom-right quadrant has not been changed by the second loop). Thus, we conclude that the loops can be merged.

Now, compare TriInv Invariant 1,

$$\left(\begin{array}{c|c} L_{TL} \equiv X_{TL}^{-1} & \\ \hline L_{BL} \equiv \cancel{-X_{BR}^{-1}}X_{BL}\cancel{X_{TL}^{-1}} & L_{BR} \equiv \cancel{X_{BR}^{-1}} \end{array}\right)$$

and Trtrmm Invariant 2 (given above). Again, we use X to denote the output of the first loop. The problem this time is that, if merged, the second loop overwrites L_{TL} before the first loop has completed computing L_{BL}. Hence, the loops cannot be merged.

In our analysis above, we reason about the legality of loop fusion with the information in the PME and loop invariants, without an analysis of the code. **The advantage of being able to reason about loop fusion without code is that even when the update statements within the loops are implemented in terms of subroutine calls to highly optimized black-box libraries (which often is the case in the domain of DLA), loop fusion can still be performed as long as the status of the different regions of the output is captured by the loop invariant.**

In the rest of this section, we provide the theory for the analysis performed above and describe how it is applied to identify multiple pairs of algorithms that merge without an exhaustive search.

3.2 Notation and assumptions

Let us assume that the two linear algebra operations that we intend to merge have the following form:

$$B := F(\hat{B}, \hat{C}, \{A\});\ C := G(B, \hat{C}, \{A\}) \tag{1}$$

where $\{A\}$ represents the set of all (other) input matrices whose values are read but not written. We use $\hat{B}$ and $\hat{C}$ to emphasize that the inputs required by the operations are the original values of B and C. Notice that the set $\{A\}$ needs not be the same for both calls, but we can always (artificially) take the union of the separate sets. Similarly, 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.

Consider the algorithms in Figure 1. The initial partitioning, loop guard, repartitioning, and "continue with" all are related to indexing, and we will refer to this as the *loop skeleton*. It takes the place of the loop header, i.e. "`for ( i=0; i<n; i+=b){}`", in a typical C loop. The remainder (assignment to submatrices) is what we will call the *update statements*. We will require that the skeleton is identical for the two loops.

We summarize our assumptions as follows: (1) Loop-based algorithms for computing the operations march through B and C by partitioning them into regions (quadrants for L in Section 2); and (2) Both algorithms march through all operands in the same direction, corresponding iterations of the loops use the same block size b, and the size of the same matrix is used in the stopping criteria (loop guard) for both loops. These two assumptions imply that the loops always have the same number of iterations and at each step the same regions are exposed for all operands. We assume these conditions hold for the rest of the discussion in this section.

Let $\mathcal{P}^F$ denote the PME of the first operation in (1). Then $\mathcal{P}^F$ can be defined as

$$\mathcal{P}^F = \left(\begin{array}{c|c} B_{TL} \equiv \mathcal{P}^F_{TL} & B_{TR} \equiv \mathcal{P}^F_{TR} \\ \hline B_{BL} \equiv \mathcal{P}^F_{BL} & B_{BR} \equiv \mathcal{P}^F_{BR} \end{array}\right),$$

where $\mathcal{P}^F_X$ are legal mathematical expressions which express the computation required to be performed with the different regions of the input and/or output matrices in order to compute B_X for $X \in \{TL, TR, BL, BR\}$. This means that the operands of $\mathcal{P}^F_X$ must be regions of the input/output matrices of the first operation, $B := F(\hat{B}, C, \{A\})$. Hence the operands of $\mathcal{P}^F_X$ must come from the following set:

$$\left\{ \begin{array}{cccc} \hat{B}_{TL}, & \hat{B}_{TR}, & C_{TL}, & C_{TR}, \\ \hat{B}_{BL}, & \hat{B}_{BR}, & C_{BL}, & C_{BR} \end{array} \right\} \cup \left\{ \begin{array}{cc} \{A_{TL}\}, & \{A_{TR}\}, \\ \{A_{BL}\}, & \{A_{BR}\} \end{array} \right\}$$

In the FLAME methodology, a loop invariant represents a partial state of computation that contributes towards the result expressed in the PME. Therefore, a strategy for obtaining loop invariants is to remove subexpressions from the PME. This implies that a loop invariant $\mathcal{I}^F$ obtained through this method must be partitioned in the same manner as $\mathcal{P}^F$. In addition, the expressions in each quadrant of the loop invariant must be subexpressions of the expression in the corresponding quadrant of the PME. Hence,

$$\mathcal{I}^F = \left(\begin{array}{c|c} B_{TL} \equiv \mathcal{I}^F_{TL} & B_{TR} \equiv \mathcal{I}^F_{TR} \\ \hline B_{BL} \equiv \mathcal{I}^F_{BL} & B_{BR} \equiv \mathcal{I}^F_{BR} \end{array} \right)$$

where $\mathcal{I}^F_X$ is a subexpression of $\mathcal{P}^F_X$ for $X \in \{TL, TR, BL, BR\}$.

Notice that $\mathcal{P}^F_X$ and $\mathcal{I}^F_X$ are expressions that describe how B_X needs to be changed. We define functions F_X and f_X as follows:

$$F_X(\hat{B}_X) = \mathcal{P}^F_X \quad \text{and} \quad f_X(\hat{B}_X) = \mathcal{I}^F_X$$

where F_X and f_X treat operands other than B_X as constants and evaluates the expressions $\mathcal{P}^F_X$ and $\mathcal{I}^F_X$ respectively.

In addition, recall that $\mathcal{I}^F_X$ is obtained by removing subexpressions from $\mathcal{P}^F_X$. These removed subexpressions represent computation that will need to be performed on $f_X(\hat{B}_X)$ in the future in order to obtain the final results. Hence, we also define "Remainder" functions, $f^{\mathcal{R}}_X$ to represent the removed subexpressions for each region of B such that

$$F_X(\hat{B}_X) = f^{\mathcal{R}}_X(f_X(\hat{B}_X))$$

where $X \in \{TL, TR, BL, BR\}$, and operands other than B_X are treated as constants. The relationship between F_X, f_X and $f^{\mathcal{R}}_X$ can be summarized as follows: The "Remainder" functions are computations to be performed in the future, the functions f_X are computations performed in the past and together, they compute the same value as F_X which computes the desired final result. For simplicity, we say f_X(or similarly $f^{\mathcal{R}}_X$) updates B_X when we mean the computations represented by f_X (or similarly $f^{\mathcal{R}}_X$) updates B_X.

G_X, g_X, and $g^{\mathcal{R}}_X$ can similarly be defined for the second operation with the PME $\mathcal{P}^G$, loop invariant $\mathcal{I}^G$.

As we illustrated in Section 3.1, whether loops can be merged hinges on whether certain submatrices have been updated appropriately (so that they can be used for the computation associated with the second loop) and other submatrices have not yet been touched (because they are still needed for the first loop). This leads us to propose the following definition:

Definition 1 *The status of a region X of an output matrix B, denoted as $\sigma(B_X)$, takes on one of three values, depending on the relationship between the PME and the loop invariant:*

$$\sigma(B_X) = \begin{cases} \text{Fully Updated} & \textit{if } \mathcal{P}^F_X \equiv \mathcal{I}^F_X. \\ \text{Partially Updated} & \textit{if } \mathcal{P}^F_X \not\equiv \mathcal{I}^F_X \\ & \textit{and } \mathcal{I}^F_X \not\equiv \hat{B}_X. \\ \text{Not Updated} & \mathcal{I}^F_X \equiv \hat{B}_X. \end{cases}$$

We similarly define $\sigma(C_X)$ in terms of $\mathcal{P}^G_X$ and $\mathcal{I}^G_X$.

In essence, $\sigma(B_X)$ is Not Updated, means that the values in B_X are the original values from before the loop commenced. Similarly, when $\sigma(B_X)$ is Fully Updated, the values in B_X are the final result. This implies that B_X was updated in past iterations and will no longer be updated in future iterations of the first loop. Conversely, when $\sigma(B_X)$ is not Fully Updated, it means that computation must be performed on the values stored in B_X and during each iteration of the loop, some amount of computation will be performed on a subset of the values stored in B_X. We call this subset of B_X a *subregion* of B_X.

Definition 2 *In the FLAME notation (illustrated in Figure 1), a subregion of a region B_X is a subset of the values of B_X and is denoted with a lowercase subscript(e.g B_x) or a two-digit subscripts from the set* $\{00, 01, 02, 10, 11, 12, 20, 21, 22\}$.

3.3 Dependence analysis of loop invariants and PMEs

Unlike traditional dependence analysis which requires an examination of the implementation (code), our approach analyzes the essence of a loop-based algorithm, the loop invariant and the PME.

Recall that a loop invariant describes the state of computation at the start of every iteration of the loop. At the start of any given iteration, if $\sigma(C_Y)$ is not FULLY UPDATED, then it implies that values stored in C_Y do not contain the final result and more computation is required in order to completely compute C_Y. This means that there must exist at least one update statement $C_y := op_y(\ldots)$ in the loop where C_y is a subregion of C_Y. If such a statement does not exist then elements in C_Y will never be updated to their final values.

Now recall that the function $g_Y^{\mathcal{R}}$ represents the "remaining" operations that needs to be performed on C_Y in order to compute the final result. Since $g_Y^{\mathcal{R}}$ represents computation on all elements in C_Y, then it must represent computation on the values of all subregions of C_Y, including the subregion C_y. From this, we can deduce that the operands of the function $op_y(\ldots)$ must be subregions of operands of the function $g_Y^{\mathcal{R}}$.

Notice that the loop invariant is also true at the end of every iteration of the loop. An arbitrary update statement $B_x := op_x(\ldots)$ would have updated the values in B_x at the end of an iteration. Since subregion B_x must belong to some region B_X, and the function f_X represents computation that was performed in the past, then at the end of the iteration the function f_X must also represent the updates performed by $B_x := op_x(\ldots)$. This implies that the operands of $op_x(\ldots)$ must also be subregions of operands of the function f_X.

In short, the loop invariant and PME captures information regarding regions used in the past and regions that will be used in the future. For any given update statement $B_x := op_x(\ldots)$ or $C_y := op_y(\ldots)$, the operands to the update statement before it is computed is given by the operands to the remainder function $f_X^{\mathcal{R}}$ or $g_Y^{\mathcal{R}}$. After the update statement has been computed, the operands to the same update statement are identified from the operands to the function f_X and g_Y. Since we can identify, from the loop invariant/PME, the regions from which the operands of an update statement originate, we can also deduce the dependences between the update statements from the dependences between regions of the loop invariants/PME. This implies that traditional dependence analysis of code can be replaced with a dependence analysis of the loop invariant/PME.

3.4 Status and preservation of dependence after loop fusion

Given two loops with identical loop bounds/loop limits, traditional compilers determine if loop fusion can be legally applied to the given loops by ensuring that dependences are preserved under loop fusion. The condition to ensure dependences are preserved can be paraphrased with the following condition [26]:

Condition 3 *For dependence to be preserved after loop fusion, the dependence must flow from an update statement in iteration i of the first loop to an update statement in iteration j of the second loop where $j \geq i$.*

Traditionally, the loop is indexed by a loop counter that tracks the iteration number and accesses into the matrices/arrays are through indices. By tracking the loop counter and the indices, a compiler can reason about dependences between iterations i and j.

In contrast, as our starting point is the mathematical specification (PME and loop invariant) of the algorithms to be merged, there is no concept of such a loop counter, nor indices. However, an equivalent way of stating this condition is as follows:

Condition 3 (ALTERNATE) *Let two loops be merged by concatenating, in order, their update statements. Then for dependence to be preserved, the dependence between the two loops must flow from the update* *statements of the first loop to the update statements in the same iteration or future iterations of the second loop.*

This removes the need of a loop counter, or indices when arguing about the correctness of loop fusion: the condition for loop fusion is now cast in terms of dependences between the update statements of the two loops within the same iteration and dependences between the update statements of the first loop from past (inclusive of the current) to update statements of the second loop in future iterations.

3.5 The Theorem

When the loops are not fused, all computation of regions of B is completed before any computation of regions of C commenses. Suppose that $C_Y \equiv g_Y(\hat{C}_Y)$ holds at the end of the current iteration of the merged loop (as indicated in the loop invariant). Then, if B_X is a region of B on which (computations represented by) the function g_Y depends or B_X is updated by (computations represented by) g_Y, B_X must have been fully computed in a previous (or the current) iteration. What this means is that $\sigma(B_X) = $ Fully Updated must hold. Otherwise, either the computation of C_Y would use values of B_X that are not the final values and the computation of C_Y might not be correct, or computation that update B_X in the future will overwrite the values written into B_X by g_Y. Similarly, if B_X is not Fully Updated (meaning that the remainder function $f_X^{\mathcal{R}}$ is not the identity) and $f_X^{\mathcal{R}}$ depends on C_Y or C_Y will be updated by $f_X^{\mathcal{R}}$, then C_Y cannot yet have been updated. Otherwise, either the computation of B_X would (in general) not be using the original values in C_Y and might compute B_X incorrectly, or values in C_Y would be overwritten by $f_X^{\mathcal{R}}$ in the future. This would mean the results in C_Y might not be correct.

These observations motivate the following theorem:

Theorem 4 *Let B_X and C_Y be arbitrary regions of matrices B and C that were respectively updated by the first and second operation. In addition, let there be a dependence that requires B_X to be computed before C_Y. If the loop invariants for the two algorithms satisfy the following two conditions:*

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 dependences are preserved after loop fusion.

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 that are updated by the first and second operation, respectively. In addition, assume that the two conditions hold for the loop invariants for the respective algorithms, for these regions. Furthermore, assume that there exist a dependence such that B_X must be computed before C_Y and that this dependence is not preserved after loop fusing. 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 $f_X^{\mathcal{R}}$ and C_Y is not required by $f_X^{\mathcal{R}}$. Since C_Y is not updated by $f_X^{\mathcal{R}}$, it implies that C_Y is not the same region as B_X. In addition, since $f_X^{\mathcal{R}}$ represent 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. □

3.6 Merging more than two loops

Merging two DLA loop-based algorithms is a matter of ensuring that the different regions of B and C have the appropriate status. If we know the status of the regions of B and C after merger, then determining if additional DLA algorithms can be merged via loop fusion is just a matter of iteratively applying the analysis of the status for each new algorithm. We discuss how the status of a region of the merged algorithm can be determined from the status of the region of the separate loop invariants.

Recall that the update status of a region in the loop invariant is defined in terms of when the region is updated: either in the past, in the future or both. This means that if a region C_Y is only updated by the second algorithm, then the status of C_Y in the merged algorithm is determined by the status of the loop invariant for the second algorithm. Similarly, if B_X is only updated by the first algorithm, then $\sigma(B_X)$ in the merged algorithm is the same as $\sigma(B_X)$ in the loop invariant of the first algorithm.

For the case where the region B_X and C_Y refers to the same region and both algorithms update C_Y, it is necessary to examine if C_Y will be updated in future iterations. When both algorithms that were merged update C_Y only in past iterations ($\sigma(C_Y)$ is FULLY UPDATED), we know that no update to C_Y will occur in the future. Therefore, $\sigma(C_Y)$ will remain as FULLY UPDATED. Similarly, when both algorithms update C_Y only in the future iterations, $\sigma(C_Y)$ will remain as NOT UPDATED. For all other combinations of $\sigma(C_Y)$, either the first or second operation will update X_Y in future iterations. Hence, $\sigma(C_Y)$ is PARTIALLY UPDATED after merger.

3.7 Multiple pairs of fusion-compatible loop invariants

In order to find a single pair of fusion-compatible implementations, traditional compilers first analyze the given implementations to determine if they can be merged. If there exist a dependence that is not preserved after loop fusion, other loop transformations (e.g. loop reversal, loop interchange, etc) are applied to one or both of the implementations, and the new implementations are then analyzed again. Even when we know that fusion is desired, depending on the input implementations, different sequences of loop transformations may be required before implementations that merge can be found. Since compilers apply loop transformations in phases, the incorrect ordering of the transformation phases could potentially prevent compatible implementations from being identified. This is the phase-ordering problem [22, 1]. The problem is exacerbated when multiple pairs of fusion-compatible implementations are required of the traditional compiler.

In our setting, the information required to determine when two algorithms can be merged, namely the PME and loop invariants, are readily available for libraries developed as part of the FLAME project (`libflame` [24] and Elemental [17]). In these libraries, most algorithms were systematically derived from the PME of the operations and related loop invariants to be provably correct. Now, the same information can be used to determine if two algorithms can be merged.

Subroutines in DLA libraries (not resulting from the FLAME project) can also be annotated with information regarding the PME and loop invariants to enable extensible compilers (compilers that allow end-users to include domain-specific knowledge through annotations or transformational rules), such as Broadway[13], PetaBricks[3], Telescoping Languages[7], Active Libraries[25], to simplify the analysis of determining when subroutines can be merged.

3.8 Constructing fusion-compatible loop invariants

The astute reader may recognize that a library that encodes all algorithmic variants of commonly used operations will quickly become bloated and conducting a pair-wise search to identify fusion-compatible algorithms requires searching through a space that is combinatorially increasing with each additional operation to merge.

Instead, we describe how to constructively identify loop invariants for the loops that can be merged from the PMEs of the operations that we want to merge. The construction algorithm is similar to that described in [10] but we additionally impose the constraints described in Theorem 4 at the appropriate points in the algorithm when constructing the loop invariants. We illustrate the construction using the example from Section 2.

Recall that PMEs show all computation, in terms of regions, required to compute the final result. It follows that by removing appropriate subexpressions from the PME, loop invariants that represent partial computation can be obtained. Hence, in order to derive fusion-compatible loop invariants, we start with the separate PMEs as shown below:

$$\left(\begin{array}{c|c} L_{TL} := L_{TL}^{-1} & * \\ \hline L_{BL} := \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-}1 & L_{BR} := \hat{L}_{BR}^{-1} \end{array}\right); \quad \left(\begin{array}{c|c} L_{TL} := L_{TL}^{T}L_{TL} + \hat{L}_{BL}^{T}\hat{L}_{BL} & * \\ \hline L_{BL} := \hat{L}_{BR}^{T}\hat{L}_{BL} & L_{BR} := \hat{L}_{BR}^{T}\hat{L}_{BR} \end{array}\right).$$

At the end of the last iteration of the loop, the loop invariant must imply that the operation has been computed. This means that at the end of the loop, one of the subregions of L must contain the final result, which implies that expressions highlighted in red below must be part of the loop invariants:

$$\left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{-1} & * \\ \hline L_{BL} \equiv \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-}1 & L_{BR} \equiv \hat{L}_{BR}^{-1} \end{array}\right); \quad \left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{T}L_{TL} + \hat{L}_{BL}^{T}\hat{L}_{BL} & * \\ \hline L_{BL} \equiv \hat{L}_{BR}^{T}\hat{L}_{BL} & L_{BR} \equiv \hat{L}_{BR}^{T}\hat{L}_{BR} \end{array}\right).$$

One could have chosen $L_{BR} \equiv \hat{L}_{BR}^{-1}$ to be highlighted in red, which would mean that the algorithm computes from the bottom-right quadrant of L and proceeds towards the top-left. This would be similar to the algorithm implemented by ScaLAPACK for the TriInv. However, recall that our assumption was that merged algorithms must march through all operands in the same manner. Since dependences in the PME of Trtrmm requires us to compute Trtrmm from top-left to bottom-right, we restrict the algorithms that compute TriInv to those that compute L^{-1} from top-left to bottom-right.

At the start of the first iteration, the loop invariant must imply that no computation has been performed. This means that the bottom-right region of L for both TriInv and Trtrmm must contain the original value of L: $L_{BR} \equiv \hat{L}_{BR}$. Then by striking out the appropriate expressions in the respective PME to ensure that the bottom-right subregion of L contains its original value, we get the following expressions:

$$\left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{-1} & * \\ \hline L_{BL} \equiv \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-}1 & L_{BR} \equiv \hat{L}_{BR}^{\cancel{-1}} \end{array}\right); \quad \left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{T}L_{TL} + \hat{L}_{BL}^{T}\hat{L}_{BL} & * \\ \hline L_{BL} \equiv \hat{L}_{BR}^{T}\hat{L}_{BL} & L_{BR} \equiv \cancel{\hat{L}_{BR}^{T}}\hat{L}_{BR} \end{array}\right).$$

Next, we consider dependences between the two operations and apply the two conditions described in Theorem 4. Recall that a region is an operand of a remainder function if it is part of a subexpression that was removed from the PME. Therefore, if a region cannot be an operand of a remainder function, then it must not be struck off from the PME.

Notice that any loop invariant of the second operation which ensures that the final result is stored in L_{TL} at the end of the operation will not preserve the value of L_{TL} computed by the first operation i.e. $\sigma(L_{TL}) \neq$ Not Updated for the second loop. Condition 2 of Theorem 4 tells us that L_{TL} cannot be read/written by any remainder functions of the first loop invariant. This means that the expression $L_{BL}L_{TL}$ must be included in any fusion-compatible loop invariant for the first operation. A second dependence that exists between the two operations is the use of L_{BR}. This means that the first operation must compute the value of L_{BR} before the second operation can update L_{BL}. However, $L_{BR} \equiv \hat{L}_{BR}$, which means $\sigma(L_{BR}) \neq$ Fully Updated in the first loop. Condition 1 of Theorem 4 requires that the second operation not use L_{BR} in past iterations. This implies that the expression L_{BR}^{T} is must be stricken off from the PME for the second operation.

$$\left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{-1} & * \\ \hline L_{BL} \equiv \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} & L_{BR} \equiv \hat{L}_{BR}^{\cancel{-1}} \end{array}\right); \quad \left(\begin{array}{c|c} L_{TL} \equiv L_{TL}^{T}L_{TL} + \hat{L}_{BL}^{T}\hat{L}_{BL} & * \\ \hline L_{BL} \equiv \cancel{\hat{L}_{BR}^{T}}\hat{L}_{BL} & L_{BR} \equiv \cancel{\hat{L}_{BR}^{T}}\hat{L}_{BR} \end{array}\right)$$

The astute reader may recognize that the above set of expressions are the loop invariants for TriInv Variant 2 and Trtrmm Variant 2. These fusion-compatible loop invariants can be merged to create Merged Variant 2.

In order to find other fusion-compatible loop invariant pairs, we continue to examine the PME for subexpressions that can be optionally removed and when we remove a subexpression, we need to ensure that the conditions in Theorem 4 are maintained. Notice that for the second operation, the only expression that can be optionally removed from the PME is the expression $\hat{L}_{BL}^{T}\hat{L}_{BL}$ and in order for the expression not to be removed, Condition 1 of the Theorem 4 requires that $\sigma(L_{BL}) =$ Fully Updated for the first loop invariant. This implies $L_{BL} \equiv \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1}$.

Similarly, the only expression that can be optionally removed from the PME of the first operation is $\hat{L}_{BR}^{-1}$ from the expression in the bottom-left region of L. If $\hat{L}_{BR}^{-1}$ is removed, then $L_{TL} \equiv L_{TL}^{T}L_{TL}$.

The different ways of optionally removing subexpressions from the pair of PMEs are summarized in Figure 3. From Figure 3, it becomes obvious that there are three pairs of loop invariants that can be merged and these pairs of fusion-compatible loop invariants correspond to the three merged algorithms in Figure 1

Notice that with this constructive approach to identify fusion-compatible loop invariants, multiple pairs of loop invariants can be identified while avoiding the phase-ordering problem encountered by traditional compilers. In addition, by starting with the PMEs, there is no need to search a combinatorially increasing search space nor is there a need for a large library of DLA routines.

4 Phase 2: Practical Implications for Parallel Processing on Clusters

Design by Transformation (DxT) [16, 19] is an approach to software engineering that enables the codification of design knowledge for a domain. In our setting, it allows algorithms, expert knowledge about

Possible expressions for L_{BL} of first loop invariant

Possible expressions for L_{TL} of second loop invariant

$$\begin{tikzcd}
L_{BL} \equiv \hat{L}_{BR}^{-1}\hat{L}_{BL}\hat{L}_{TL}^{-1} \arrow[r] \arrow[dr] & L_{TL} \equiv \hat{L}_{TL}^{T}\hat{L}_{TL} + \hat{L}_{TR}^{T}\hat{L}_{TR} \\
L_{BL} \equiv \hat{L}_{BL}\hat{L}_{TL}^{-1} \arrow[r] & L_{TL} \equiv \hat{L}_{TL}^{T}\hat{L}_{TL}
\end{tikzcd}$$

Figure 3: Relationship between subexpressions in the two PME that can be optionally removed. Arrows denote combinations that preserve dependences after loop fusion.

those algorithms, and knowledge about target architectures to be encoded so that a tool can transform this knowledge into "high quality" implementations. The principle behind DxT is that knowledge used by an expert to develop implementations of algorithms can be encoded as a series of transformation rules that apply design knowledge of domain software. By iteratively/recursively applying the transformation rules used by the expert on an abstract description of an input algorithm, the abstract description is transformed into implementations composed from basic building blocks (routines) via a process similar to what an expert would go through when implementing the algorithm manually. To select between different implementations, DxT requires mathematical cost functions to be associated with each building block, so that the overall cost of an implementation can be estimated and a best implementation can be chosen. DxTer [15] is a prototype tool that implements DxT by automatically applying the transformation rules and evaluating the costs of resulting implementations.

4.1 Encoding dense linear algebra expert knowledge

Knowledge in DxTer can be classified into three broad categories, namely building blocks, refinements and optimizations. We discuss the knowledge that was encoded in DxTer for the domain of parallel DLA on clusters.

Building Blocks For the domain of parallel DLA on clusters, the building blocks of software are sequential implementations of common linear algebra operations found in the Basic Linear Algebra Subroutines (BLAS) [9] and LAPACK [2] and the collective communications found in the Message-Passing Interface (MPI) [21]. Associated with each of the building blocks is a cost estimate as a function of the matrix size and machine parameters (e.g., latency and bandwidth).

Refinements Refinements are transformation rules that represent expert knowledge on how to implement algorithms in terms of the building blocks. For our particular domain, refinements describe how an expert would transform a DLA update statement (e.g., $L_{10} := -L_{11}^{-1}L_{10}$) into a parallel implementation. This typically consists of some redistribution via a collective communication followed by local computation on each process (e.g., MPI process) followed by another collective communication to reduce local contributions to a global result and/or redistribute the result. An expert will choose how to parallelize a given update statement from a set of parallelization patterns that are known to be good. These desirable parallelization patterns identified by the expert are encoded as multiple refinement rules in DxTer. For example, two possible refinements for the Triangular Solve with Multiple Right-hand Sides (TRSM) operation that computes the update statement $L_{10} := -L_{11}^{-1}L_{10}$ are encoded in DxTer as shown in Figure 4.

Optimizations Optimizations are transformation rules that represent expert knowledge on how to optimize inefficient code patterns that show up repeatedly in domain software. Examples of optimizations include equivalences of sequences of DLA operations and/or collective communications [16], so inefficient patterns can be replaced with better implementations. For example, it is known that broadcast can be implemented as a scatter followed by an allgather and vise versa [6]. An expert understands this. DxT enables us to encode this knowledge. DxTer explores these options automatically and estimates the performance of resulting codes.

4.2 The output

The expert developer uses a Domain Specific Language (DSL) to implement DLA algorithms on clusters. Several exist, all implemented as libraries: ScaLAPACK [8], PLAPACK [23], and (more recently) Elemental [17]. We choose Elemental as the output language for DxTer because its implementations closely mirror the algorithms as presented in Figure 1 and it expresses computation and communication at a very high level of expression. It also typically achieves the best performance of all three libraries so that matching or surpassing the performance of the hand-coded implementation in Elemental is a significant achievement.

4.3 From an abstraction to multiple implementations

Our input to DxTer is an abstract DLA algorithm. The abstract algorithm is specified in two parts, a loop skeleton and a pipe-and-filter model of the update statements. The pipe-and-filter model is a directed acyclic graph (DAG) where vertices are abstractions representing computations and the edges represent data flow. This representation is similar in concept to the data dependence graph constructed by traditional compilers.

To transform this abstract algorithm into an implementation, refinement rules are applied on the vertices to refine (replace) each abstract vertex to either a building block or a subgraph of equivalent functionality but with more specific implementation choices. For example, the vertex representing the update statement $L_{10} := -L_{11}^{-1} L_{10}$ is refined by one of the refinements rules in Figure 4 into a subgraph whose vertices represent data redistributions and local computation. In cases where there are multiple possible refinements (as shown in Figure 4), one DAG is created from each application of a refinement rule. This creates multiple DAGs, representing a family of possible implementations. Optimization rules are applied in a similar fashion, where inefficient subgraphs are replaced with more efficient ones and additional DAGs are created when necessary. All created DAGs are stored to allow for cost comparison.

Eventually, the process of applying refinement and optimization rules yields many DAGs expressed in terms of building blocks. Each DAG represents one of many parallelization approaches and a distinct combination of optimizations. A cost for each implementation is then estimated by adding the costs of the building blocks[3]. A simple comparison of the cost allows the best[4]implementation to be identified, for which Elemental code is then generated.

4.4 Fusing automated

Now, let us assume that DxTer is to provide an optimized implementation for $L := L^{-1};\ L := L^T L$.

To allow DxTer to use the status of the loop invariants to perform loop fusion, we annotate the abstract algorithms of the separate operations with the status of the different regions of the output matrix. In this

[3]DLA algorithms tend to be block synchronous, which simplifies the cost estimation.

[4]There may be multiple candidates for best implementation, depending on (for example) the matrix size.

sense, DxTer is similar to the extensible compilers that understand domain-specific knowledge through annotations.

DxTer first picks a loop-based algorithm for each of the operations and uses the status of the regions to determine if the algorithms can be merged. If they can be merged, operands are added to the separate loop skeletons to create the loop skeleton for the merged algorithm. In addition, the separate pipe-and-filter models are then composed together to form the pipe-and-filter model of the update statements of the merged algorithm. The refinement/optimization of the merged algorithm is then performed.

DxTer repeats this process for all pairs of algorithms. In the end, the best merged algorithm is chosen from all the implementations that are thus generated.

4.5 Performance

In this section, we present performance results obtained on a BlueGene/P supercomputer. We tested on 8192 cores (2 racks), which have a combined theoretical peak of over 27 TFLOPS. Double precision arithmetic was used in all computation. For an input matrix $L \in \mathbb{R}^{n \times n}$, an operation count of $\frac{n^3}{3}$ floating-point operations (flops) was used for TRIINV and TRTRMM operations and $\frac{2n^3}{3}$ flops was used for the merged operation. We compare DxTer-generated Elemental code to ScaLAPACK performance across a range of algorithmic block sizes and present the best performance attained for each[5]. In all experiments, DxTer generated the same optimized parallel code as the expert who implemented Elemental so only comparison with ScaLAPACK is shown.

In Figure 5, we show results (from top to bottom) for $L := L^{-1}$ (TRIINV), $L := L^T L$ (TRTRMM), and the combined operation $L := L^{-1};\ L := L^T L$. In each graph, two-thirds of the theoretical machine peak is at the top of the graph.

"Inlined" codes are implementations of the algorithm where only refinement rules were applied. "Optimized" codes are implementations where both refinement and optimization rules were applied. "Fused" codes are those that have been fused and optimized, while "Unfused" codes are optimized version of the separate algorithms run separately.

All three graphs demonstrate that poorly-selected algorithmic variants result in bad performance. These variants have update statements that incur significantly more data communication, which is costly overhead on clusters. TRIINV variant 2, for example, has the update $L_{21} := L_{22}^{-1} L_{21}$, (a large triangular solve with few right-hand sides) which parallelizes poorly because the large submatrix L_{22} is communicated in each iteration of the loop. This communication is amortized over relatively little computation and thus the performance suffers. The fused algorithms with this variant similarly perform badly. For these bad variants, even DxTer optimizations cannot overcome suboperations that do not parallelize well, so the "inlined" code performs roughly the same as "optimized" code.

In Figure 5 (Bottom), we show DxTer-generated optimized implementations of the three fused algorithms in Figure 1 when compared to ScaLAPACK. Notice that Merged Variant 3 is the only merged algorithm that performs better than ScaLAPACK. This was exactly predicted by DxTer in Figure 2. Again, choosing the wrong variant of the fused algorithms produces very bad performance. These merged implementations were generated, parallelize and optimize automatically by DxTer through the use of the status of the regions.

To quantify the benefits of loop fusion, we compare the fused algorithm (Merged Variant 3) against the non-fused (but optimized) implementation (TRIINV Variant 3 + TRTRMM Variant 1). The fused algorithm performed approximately 10% better than the unfused one. The difference in performance can be attributed to DxTer applying optimization rules that reuse intermediate distributions and reduce communication.

[5]In other words, the graphs show performance for when the block sizes, which are tuning parameters, were optimized.

```
void DistTrsmToLocalTrsmVar1::Apply(Poss *poss, Node *node) const {
   Trsm *trsm = (Trsm*)node;

  //Distribute Triangular Matrix, L
  RedistNode *distL = new RedistNode(D_VC_STAR);
  distL->AddInput(node->Input(0),node->InputConnNum(0));

  //Distribute RHS Matrix, X
  RedistNode *distX = new RedistNode(D_VC_STAR);
  distX->AddInput(node->Input(1),node->InputConnNum(1));

  //Local Computation
  LocalTrsm *trsm = new LocalTrsm(trsm->m_side, trsm->m_tri,
                                  trsm->m_trans, trsm->m_coeff);
  trsm->AddInput(distL,0);  trsm->AddInput(distX,0);

  //Distribute X
  RedistNode *redistX = new RedistNode(D_MC_MR);
  redistX->AddInput(trsm,0);

  //Refine Abstract Vertex & Clean up
  poss->AddNodes(4, distL, distX, trsm, redistX);
  node->RedirectChildren(redistX,0);
  node->m_poss->DeleteChildAndCleanUp(node);
}

void DistTrsmToLocalTrsmVar2::Apply(Poss *poss, Node *node) const {
  Trsm *trsm = (Trsm*)node;

  //Distribute Triangular Matrix, L
  RedistNode *distL = new RedistNode(D_STAR_STAR);
  distL->AddInput(node->Input(0),node->InputConnNum(0));

  //Distribute RHS Matrix, X
  RedistNode *distX = new RedistNode(D_STAR_VC);
  distX->AddInput(node->Input(1),node->InputConnNum(1));

  //Local Computation
  LocalTrsm *trsm = new LocalTrsm(trsm->m_side, trsm->m_tri,
                                  trsm->m_trans, trsm->m_coeff);
  trsm->AddInput(distL,0);  trsm->AddInput(distX,0);

  //Distribute X
  RedistNode *redistX = new RedistNode(D_MC_MR);
  redistX->AddInput(trsm,0);

  //Refine Abstract Vertex & Clean up
  poss->AddNodes(4, distL, distX, trsm, redistX);
  node->RedirectChildren(redistX,0);
  node->m_poss->DeleteChildAndCleanUp(node);
}
```

Figure 4: 2 out of 6 different refinement rules encoded in DxTer for the TRSM operation that is used to implement $L_{10} := -L_{11}^{-1} L_{10}$. Notice that they differ only in the way the oeprands L and X are distributed.

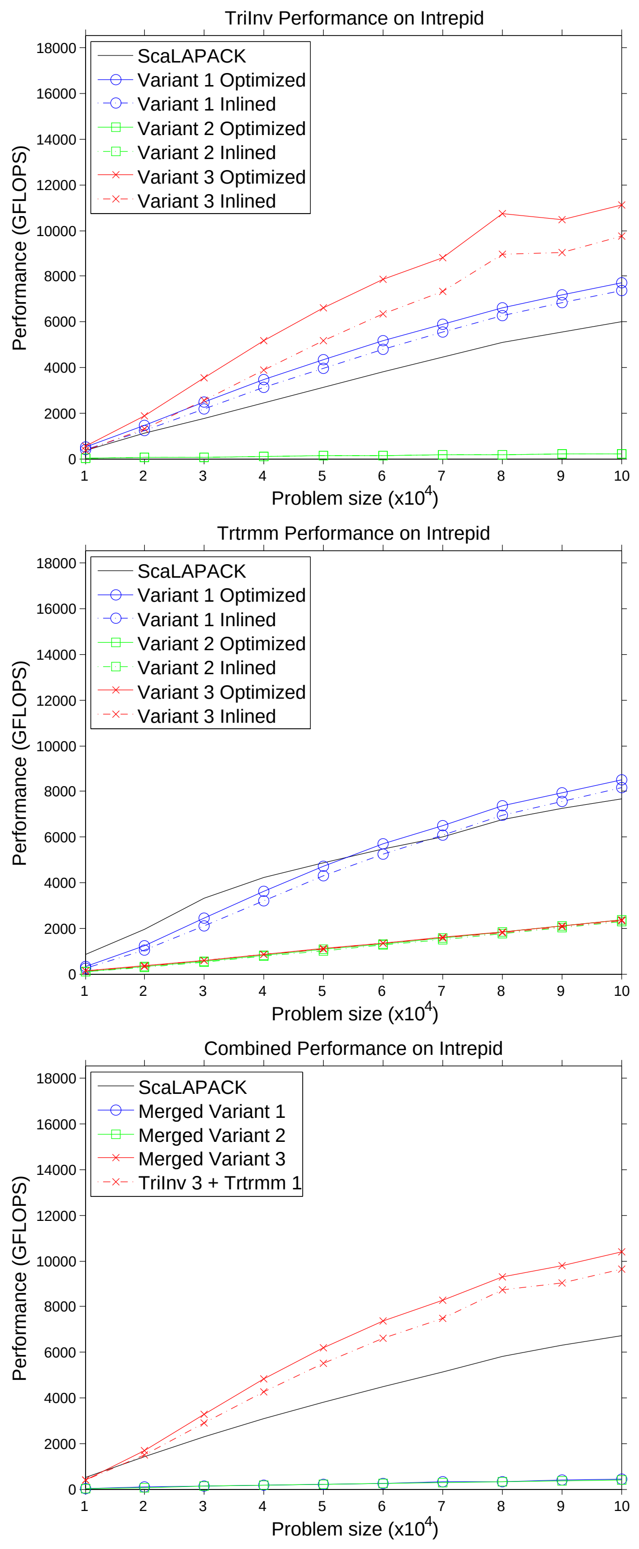

Figure 5: Performance results comparing algorithmic variants and ScaLAPACK on Intrepid. 2/3 of theoretical peak is at the top of the graphs. Variants 2 and 3 of TRTRMM are almost the same, so they appear on top of each other in the graph. Variant 3 is slightly faster.

5 Conclusion and Future Directions

In this paper, we described a two-phase approach that makes it practical for loop fusion to be applied when automatically optimizing DLA operations for distributed memory architecture.

The first phase analyzes the dependences in the mathematical specifications embedded in the PME and constructs loop invariants that possess the property that their derived algorithms can be merged through loop fusion. The theory presented in the paper, coupled with a constructive algorithm for generating loop invariants allows multiple pairs of algorithms to be identified while side-stepping the phase-ordering problem faced by traditional compilers. This work supports the hypothesis of the authors of [14] where it was suggested in the conclusion that the feasibility of loop fusion can be deduced from the loop invariants. We demonstrated how the theory can be used to annotate algorithms/subroutine code which extensible compilers can use to perform loop merging by annotating the status of regions in the refinement rules for our automated system, DxTer,to allow automated loop fusion.

The second phase uses an automated system, DxTer, to replicate the work of an expert when optimizing parallel DLA operations for clusters. DxTer implements an approach to software engineering called Design-by-Transformations (DxT). DxTer analytically estimates the relative performance of possible implementations and identifies if it is worthwhile to merge two given operations. Automatically generated optimized code from DxTer matches or surpasses both hand-coded implementations and implementations found in commonly used parallel libraries.

We believe that more can be done through a better understanding of the PME and loop invariant. In the area of annotations for extensible compilers, we note that the required annotations usually involve information as to whether an input to a subroutine has been (or may be) updated or remains pristine at the end of the subroutine[12, 13]. This information is then used for dependence analysis in place of actual code. In our analysis of loop fusion, no examination of code was performed. Instead, dependence analysis was performed on the loop invariants and PME. This seemed to suggest that it suffice to annotate a loop-based algorithm with its loop invariant (or properties of the loop invariant) instead of having to provide annotations for all subroutines that may be called by the algorithm.

Even if the presented loop merging analysis only finds practical application within DLA, the impact will be considerable. Opportunities for merging loops abound within the operations, algorithms, and libraries encountered in that domain, and the performance improvement is real. Because the analysis presented in this paper was reasoned in terms of disjoint regions of arbitrary shapes, though, we believe it can be extended to domains other than DLA.

Acknowledgments

This research was sponsored in part by NSF grants OCI-0850750 and CCF-0917167, a grant from Microsoft, and the UTAustin-Portugal Colab Program. Bryan Marker was sponsored by a fellowship from Sandia National Laboratories and an NSF Graduate Research Fellowship under grant DGE-1110007. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] 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.

[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. SIAM, 1999.

[3] 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.

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[5] 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.

[6] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783, September 2007.

[7] 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.

[8] 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.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] Diego Fabregat-Traver and Paolo Bientinesi. Automatic generation of loop-invariants for matrix operations. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications*, ICCSA '11, pages 82–92. IEEE Computer Society, 2011.

[11] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[12] Jichi Guo, Mike Stiles, Qing Yi, and Kleanthis Psarris. Enhancing the role of inlining in effective interprocedural parallelization. In *ICPP'11*, pages 265–274, 2011.

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

[14] 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.

[15] Bryan Marker, Don Batory, and Robert van de Geijn. DxTer: A program synthesizer for dense linear algebra. Computer Science report TR-12-17, Univ. of Texas at Austin, 2012.

[16] Bryan Marker, Jack Poulson, Don Batory, and Robert van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *International Workshop on Automatic Performance Tuning (iWAPT2012). Proceedings of VECPAR 2012 Conference*, July 2012.

[17] Jack Poulson, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Soft.* to appear.

[18] 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.

[19] Taylor Riche, Don Batory, Rui Goncalves, and Bryan Marker. Architecture design by transformation. Computer Science report TR-10-39, Univ. of Texas at Austin, 2010.

[20] Martin D. Schatz, Jack Poulson, and Robert A. van de Geijn. Scalable universal matrix multiplication algorithms: 2d and 3d variations on a theme. *ACM Transactions on Mathematical Software.* submitted.

[21] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[22] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM.

[23] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[24] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[25] Todd L. Veldhuizen. *Active Libraries and Universal Languages.* PhD thesis, Indiana University Computer Science, May 2004.

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