Copyright

by

Bryan Andrew Marker

2014

Design by Transformation:

From Domain Knowledge to Optimized Program Generation

by

Bryan Andrew Marker, B.S.C.S.; B.S.Math

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

May 2014

Design by Transformation:
From Domain Knowledge to Optimized Program Generation

Bryan Andrew Marker, Ph.D.

The University of Texas at Austin, 2014

Supervisors: Robert van de Geijn and Don Batory

Expert design knowledge is essential to develop a library of high-performance software. This includes how to implement and parallelize domain operations, how to optimize implementations, and estimates of which implementation choices are best. An expert repeatedly applies his knowledge, often in a rote and tedious way, to develop all of the related functionality expected from a domain-specific library. Expert knowledge is hard to gain and is easily lost over time when an expert forgets or when a new engineer starts developing code. The domain of *dense linear algebra (DLA)* is a prime example with software that is so well designed that much of experts' important work has become tediously rote in many ways. In this dissertation, we demonstrate how one can encode design knowledge for DLA so it can be automatically applied to generate code as an expert would or to generate better code. Further, the knowledge is encoded for perpetuity, so it can be reused to make implementing functionality on new hardware easier or it can be used to teach how software is designed to a non-expert. We call this approach to software engineering

(encoding expert knowledge and automatically applying it) *Design by Transformation (DxT)*. We present our vision, the methodology, a prototype code generation system, and possibilities when applying DxT to the domain of dense linear algebra.

Contents

Abstract **iv**

Chapter 1 Introduction **1**

1.1 Motivation . . . 2

1.2 Problem . . . 7

1.3 Our Solution . . . 9

1.4 The Grand Vision . . . 9

1.5 Related Work . . . 11

1.5.1 Software Engineering . . . 11

1.5.2 DLA and HPC . . . 13

1.6 Contributions . . . 17

1.7 Outline . . . 18

Chapter 2 Design by Transformation **19**

2.1 Representing Algorithms and Implementations . . . 19

2.2 Representing Design Knowledge . . . 20

2.2.1 Refinements . . . 20

2.2.2 Optimizations . . . 22

2.2.3 Graphs or Code? . . . 23

2.3 Grammar . . . 23
2.3.1 DSLs . . . 23
2.3.2 Exploring the Language . . . 25
2.3.3 A Family of DSLs . . . 26
2.3.4 Context Sensitivity . . . 28
2.4 Connection to Model Driven Engineering . . . 29
2.5 DLA Specifics . . . 30
2.5.1 Loops in an Acyclic Graph . . . 30
2.5.2 Type Information . . . 32
2.5.3 Correct by Construction . . . 32
2.6 Summary . . . 34

Chapter 3 Domain Structure **36**
3.1 Variants and Layering . . . 36
3.2 DLA Operations . . . 40
3.3 FLAME Algorithms in DxT . . . 43
3.3.1 Layer-Templatized Refinements . . . 44
3.3.2 An Abstract Layering Example . . . 45
3.4 Loop Transformations . . . 48
3.5 Going Lower . . . 50
3.5.1 Why Not Go Lower? . . . 50
3.5.2 Problems and Possible Solutions When Breaking Through . . . 51
3.6 Summary . . . 53

Chapter 4 DxTer **54**
4.1 Encoding Knowledge . . . 54

4.1.1 Nodes and Graphs 55
4.1.2 Node and Edge Properties 57
4.1.3 DAG Restrictions and Checking 58
4.1.4 Transformations 60
4.1.5 Output Code 63
4.1.6 Explaining Differences 65
4.2 Search 67
4.2.1 Basic Search 67
4.2.2 Phases and Culling 69
4.2.3 Saving the Search Space 72
4.2.4 Transformation Meta-Optimization 73
4.3 Summary 75

Chapter 5 Elemental **76**

5.1 Elemental 76
5.1.1 Elemental Basics 78
5.1.2 Parallelizing `Trmm` 79
5.1.3 Encoding the Algorithm with Elemental 82
5.2 BLAS3 85
5.2.1 Algorithms to Explore 86
5.2.2 BLAS3 Elemental Refinements 87
5.2.3 Redistribution Optimizations 88
5.2.4 Transpose Optimizations 92
5.2.5 The Knowledge Base 94
5.2.6 Cost Estimates 95
5.2.7 Search Space and Results 97

5.3 LAPACK-Level Operations 101
5.3.1 Cholesky 102
5.3.2 SPD Inversion 103
5.3.3 Two-Sided Problems 108
5.4 Locally-Best Search 111
5.4.1 Implementation Clusters 112
5.4.2 Locally-Best Refinements 119
5.4.3 The Axpy Heuristic 122
5.4.4 Are Heuristics Cheating? 124
5.5 Summary 125

Chapter 6 BLIS **126**

6.1 BLIS Layering 126
6.1.1 Sequential `Gemm` Implementation 128
6.1.2 Packing 133
6.1.3 DxTer Encoding 134
6.2 Parallelizing for Shared Memory 136
6.2.1 Parallelization Heuristic 136
6.2.2 Communicators 139
6.3 Encoding Multithreaded Parallelization 140
6.3.1 Quick Results 142
6.3.2 DxTer as a Productivity Enhancer 143
6.4 Performance Results 145
6.5 Heuristics vs. Testing 148
6.6 Summary 149

Chapter 7 Conclusion **151**

7.1 Contributions 151

7.1.1 A DLA Representation in DxT 152

7.1.2 A Prototype Generator 152

7.1.3 The Benefits of Encoding Design Knowledge 153

7.2 Future Work 154

7.3 Vision 155

Appendix A Two-Sided Trmm **157**

APPENDICES **157**

Bibliography **176**

Chapter 1

Introduction

The grand vision of our work is to change the way we view software libraries in an effort to alleviate the burden of expert developers by leveraging code generation, as described in Section 1.4. As this goal is ambitious, in this dissertation we focus on a domain that has been extensively studied and developed: *dense linear algebra (DLA)*. DLA is the example. The techniques are general.

For DLA, libraries are currently repositories of highly optimized code targeting a set of specific functionality on a particular class of hardware. We believe these libraries can be and should be repositories of fundamental domain-specific algorithms and expert software design knowledge about how to implement libraries for a particular class of hardware. Then, code for a user's application will be automatically generated from the encoded knowledge. It can even be optimized to the application's particular use of functionality. We see numerous benefits of this including better performing code, more maintainable code, and more easily extended code, all of which we touch on in this dissertation.

Our thesis is that DLA is an example of a domain that can be encoded

as dataflow graphs and that architecture-specific implementation knowledge can be encoded as graph transformations. From this, it is possible to produce code mechanically by carefully choosing and applying those transformations. This is significant because it means that a developer's rote task of exploring design options and choosing a high performance implementation is automatable for a set of domain functionality. The so produced implementation is explainable – in terms of transformations – and trusted for performance and correctness (given that the transformations are trusted). We call this approach to software engineering *Design by Transformation (DxT)*, pronounced "dext." This dissertation provides evidence in support of this thesis by focusing on the domain of DLA.

1.1 Motivation

In DLA, and in many other scientific computing domains, there is a standard set of functionality users expect. For example, the *Basic Linear Algebra Subprograms (BLAS)* [22, 23, 39] are commonly used matrix operations on which higher-level functionality (e.g., that of LAPACK [7], explained in Section 3.2, or `libflame` [61]) is implemented. DLA library users expect standard functionality to be implemented in high performance code so their applications perform well. To do so, DLA code must be specialized for the target architecture. For instance, distributed-memory, multithreaded, sequential, and GPU architectures each require customized code, often using different algorithms, *application programming interfaces (APIs)*, and programming paradigms. For each, one must become very knowledgeable of both DLA algorithms and the target hardware architecture to attain high performance.

As a result, there are few developers that can implement such functionality well and, therefore, their time is valuable. We will call them *experts*. When a new

architecture is targeted (e.g., when a new architecture is first released), a developer must become an expert and implement all expected functionality in high performance code. That code might have correctness bugs that must be discovered via testing or it might have performance bugs, where the developer missed opportunities to apply known optimizations. Often the resulting code is difficult to understand by a non-expert and cannot be easily explained to a new developer tasked with maintaining the code[1].

These issues are common to other domains. Experts are rare and valued for their ability to develop many related pieces of code well. They use their catalogue of algorithms, implementation options, and optimizations tricks to get the best performance possible. Often their job is rote, applying their knowledge repeatedly in different algorithm contexts. As computer scientists, we strongly believe that when a task becomes rote and the tools are available, automation should be employed.

DxT allows us to demonstrate the utility of automated code generation for a portion of the DLA software stack on distributed-memory, multithreaded, and sequential architectures. We encode fundamental architecture-agnostic DLA algorithms and architecture-specific design knowledge to alleviate the rote efforts of experts.

Figure 1.1 shows an algorithm for a BLAS operation in the FLAME notation [32, 33, 60] that computes *Triangular Matrix-Matrix multiplication (`Trmm`)*. There are eight versions of `Trmm`, but in this case it computes $B := BL$ with a triangular matrix L. We call this *`TrmmRLN`* since the triangluar matrix is on the right-hand side, lower-triangular, not transposed. This example is used throughout the dissertation to demonstrate how domain algorithms, a piece of domain knowledge, are

[1]It is common for developers to change jobs (e.g., graduate), so the overhead of bringing new engineers "up to speed" to continue development is a real and ongoing concern.

Algorithm: $[B] := \textsc{Trmm_rln_blk}(L, B)$

Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$
where L_{TL} is 0×0, B_L is $n \times 0$

while $m(L_{TL}) < m(L)$ **do**

Repartition

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where L_{11} is $b \times b$, B_1 has b columns

$B_0 := B_0 + B_1 L_{10}$ (`Gemm`)
$B_1 := B_1 L_{11}$ (`Trmm`)

Continue with

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

endwhile

Figure 1.1: Variant of `Trmm` to compute $B := BL$ (right, lower triangular, non-transposed, or `Trmm RLN`).

reused across architectures.

Throughout this dissertation, we hint at the fact that the DLA code generation process starts with algorithms like this that are an output of the "FLAME approach" to deriving algorithms [32, 33, 60]. This approach starts with a definition of a DLA operation and a family of loop invariants (in the sense of Dijkstra and Hoare [20, 35]) is derived from this definition. From each loop invariant, a loop-based algorithm is derived hand-in-hand with its proof of correctness. This then yields a family of algorithmic variants that one must explore and implement for each hardware architecture. Details of this process are not pertinent to our discussion other than briefly in Section 3.4.

In order to motivate the notation, though, we use this algorithm for `TrmmRLN`

and a simple derivation. We start by partitioning

$$B \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right) \quad \text{and} \quad L \rightarrow \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right),$$

where L_{00} is $k \times k$, L_{11} is $b \times b$, B_0 is $n \times k$, and B_1 is $n \times b$. Inserting these partitioned matrices into $B := BL$ yields

$$\left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right) := \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right) \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

or

$$\left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array} \right) := \left(\begin{array}{c|c|c} B_0 L_{00} + B_1 L_{10} + B_2 L_{20} & B_1 L_{11} + B_2 L_{21} & B_2 L_{22} \end{array} \right).$$

DLA algorithms, as in this case, are typically loop-based. Here, in the current iteration we assume that

- B_0 has already been updated[2] with the partial result $B_0 L_{00}$;
- B_0 is to be updated with the partial contribution $B_1 L_{10}$ in this iteration;
- the remaining $B_2 L_{20}$ contribution to B_0 will be performed in future iterations;
- B_1 has not yet been updated and is to be updated with $B_1 L_{11}$ in this iteration; and
- B_2 has not yet been updated and is to be updated in future iterations.

[2] This is an example of a *loop invariant*, an assertion made about the state of matrix quadrants at certain points of loop execution.

This means that in this iteration the updates

$$\begin{aligned} B_0 &:= B_0 + B_1 L_{10} \\ B_1 &:= B_1 L_{11} \end{aligned}$$

are to be performed.

The algorithm in Figure 1.1 expresses this. In each iteration, submatrices are partitioned from the input matrices. The loop body operations (called *update statements* or just *updates*) here are BLAS operations themselves. One is a recursive `Trmm` call on smaller matrices, submatrices of the inputs. The other operation is *General Matrix-Matrix multiply (`Gemm`)*, $C := \alpha AB + \beta C$), also on submatrices of the inputs.

In Figure 1.1, the *partition* and *repartition* operations at the beginning and end of each iteration, respectively, determine which part of the matrix forms each submatrix. The partitions move in each iteration.

The observed recursion on submatrices and call to `Gemm` are common in BLAS algorithms. For recursion, one layers different algorithms implementing the same operations, described in Section 3.1. Thus, the same algorithmic options are explored repeatedly to implement a software stack. `Gemm` is a widely-used operation in DLA, so one reimplements `Gemm` repeatedly by again exploring the same implementation options over and over. Higher-level DLA algorithms often have BLAS operation in their loop body, too, which requires implementation knowledge to be reused for higher-level operations. These are prime examples of an expert's rote efforts.

Consider the more complicated algorithm in Figure 1.2, which computes $A := L^H AL$. This is called *two-sided triangular matrix multiplication* and is detailed

in Section 5.3.3. The loop body includes a `Gemm` and two `Trmm` instances among other BLAS operations. In implementing DLA functionality for an architecture, one considerations how to parallelize `Trmm` and `Gemm` updates in the `Trmm` algorithm and then reuses that design knowledge for this more complicated operations. Throughout this dissertation, we demonstrate how expert implementation knowledge is repeatedly reused like this to implement the algorithms of Figure 1.1 and more complicated ones such as that of Figure 1.2. Further, we demonstrate how this decision process can be automated.

1.2 Problem

For domains like DLA, experts use software design knowledge to implement entire libraries of functionality for each new architecture. This is a largely rote and tedious process as knowledge is reapplied repeatedly in slightly different contexts. An expert is forced to go through this inefficient engineering work. Further, the rote nature of this software development leads to correctness mistakes as well as mistakes that decrease performance (e.g., not applying an optimization).

For other domains, one does not reuse the same knowledge repeatedly to develop related pieces of functionality. As we only store the result of applied knowledge (code)[3], the essential, important knowledge that leads to code is lost when somebody forgets it or retires. Large applications are too often trusted because of how long they have been used but not fully understood by the engineers maintaining them. For example, developers fear making changes (like adding parallelization) or adding functionality because they do not fully understand the software they are to maintain.

[3] We also store some knowledge in published papers and sometimes comments, but implementation knowledge is still incomplete.

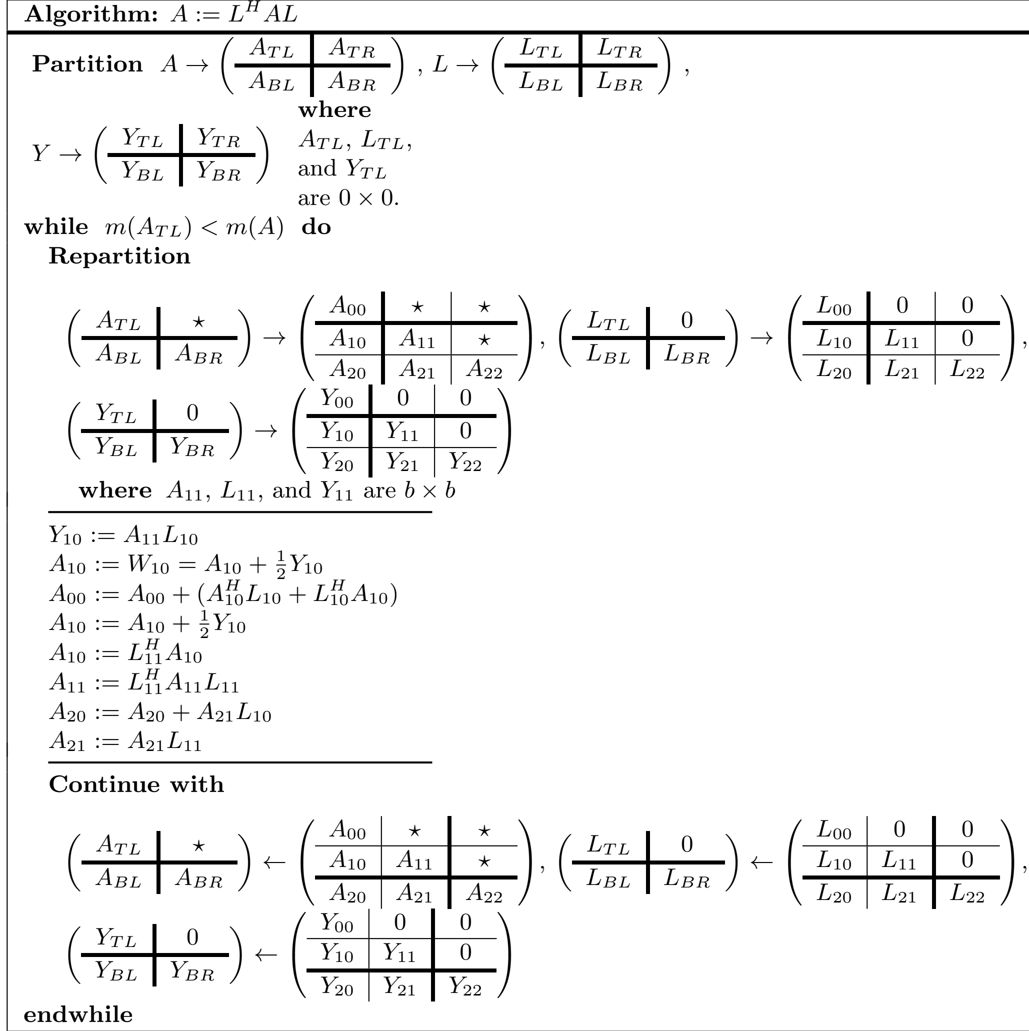

Figure 1.2: Blocked variant 4 for computing $A := L^H A L$.

1.3 Our Solution

We demonstrate how DLA design knowledge is represented with DxT for distributed-memory, multithreaded, and sequential architectures. We use APIs that we think of as *domain-specific languages (DSLs)*. Once encoded in our prototype system called *DxTer* ("dexter"), the knowledge, encoded in terms of graph transformations, is automatically applied and explored. Humans explore the same options and use cost (performance) estimates to choose the "best" performing code. Cost functions are another form of knowledge about the domain and the target architecture that we encode. DxTer uses cost functions to rank-order implementations in the search space similar to how a person does. It then chooses the "best" and outputs its code.

We demonstrate how the generated code for a distributed-memory target is the same or better than the hand-developed versions for a variety of operations. For a sequential architecture, we break down matrix-matrix BLAS operations in terms of explainable transformations. We then augment those transformations with knowledge to parallelize loops for multithreaded targets. In this case, we automatically generated code that did not exit. In fact, the desired final implementation was not known. Each time the developer had an implementation idea, a new parallelization scheme was added and DxTer evaluated it and all existing schemes for each of dozens of functions automatically without requiring human intervention. DxTer in this case was a productivity multiplier.

1.4 The Grand Vision

While this dissertation and the work behind it focuses on DLA, DxT is domain agnostic. We believe it to apply to any domain with a dataflow representation. We

expect that our DLA results can be replicated elsewhere: encode knowledge and automatically generate libraries of code for various architectures.

By encoding knowledge, we believe extending, maintaining, and learning software applications is easier. One has the design options explicitly exposed as knowledge and from those can make the decisions that yield software. Our "Grand Vision" is to see DxT or DxT-like approaches used by expert developers to engineer software much faster and more reliably (in terms of performance and correctness). Instead of only storing libraries as code, we need to store design knowledge.

Compared to engineering disciplines that have existed for centuries, software engineering is a relatively new. Structural engineers can determine how a bridge works or know if it is safe to add a room to a house. This is largely because commonly-accepted building principles are followed and the pieces that go into the structures are understood (a truss, for example, is an understood piece of a design). We want to get to a point in software engineering where a large system is similarly understandable.

We see transformations à la DxT as a possible way to accomplish this. We see the layering and abstractions DxT enables as a way to limit complexity by breaking it down into understandable pieces even if they are as numerous as the number of trusses or steal beams in a bridge. Software design knowledge is expressed as transformations. Particular applications will be explained via transformations and will be changed via transformations. Our Grand Vision is to apply DxT more widely, to more domains and to more complicated software systems to advance the state of software engineering.

Instead of having design knowledge in the head of a few experts who apply that knowledge to design code, we will have repositories of that knowledge stored

explicitly via transformations. The transformations will be proven correct, so resulting code is trusted for correctness. They will be understandable and teachable, so code is understandable and teachable. Software that was previously coded manually by experts through tedious and rote development will be automatically formed by a system that explores the same design options more thoroughly and with greater patience, so the resulting code is more trusted to achieve high performance.

This Grand Vision is lofty and years away, but we see DxT step in this direction.

1.5 Related Work

As the domain chosen in this work to illustrate DxT bridges software engineering and DLA / *high performance computing (HPC)*, we summarize related work from both communities here.

1.5.1 Software Engineering

The most closely-related work to DxT is decades old: rule-based relational query optimization [40]. Here, cost estimates and heuristics guide the choice of implementation details (based on problem size). A search space is explore to find a good implementation of a relational query at runtime. As code generation performance is important to getting small query runtime (which includes code generation time), heuristics are essential to searching the space quickly without resorting to an exhaustive search. DxT is a generalization of this idea.

With *program synthesis*, a single program is generated that implements specified functionality – it is difficult to get just one. With *program generation*, on the other hand, it is easy to get many implementations; the difficulty is in finding a

high-performing version, which leads to a search space of options. The work of [45] uses deductive program synthesis to prove the existence of an "object" meeting specified computation requirements. In finding that object (and proving its correctness), one is left with a prescription for computing the desired output. DxT uses correct-by-construction generation of implementations for a specification and searches many such implementations. One can imagine applying this approach to synthesize primitive implementations or the RHS graphs of transformations and use those as input to DxTer.

With the Amphion system [44], a user is guided in his development of a specification of functionality using a knowledge base containing domain-specific theory (using a context-sensitive, menu-driven GUI). Amphion then uses program synthesis and knowledge about the functions available in libraries to build an implementation of the specification. Thus, a developer does not need to learn about a library's functions, only the domain's operations. DxT is similar in goal, but uses program generation instead of program synthesis. Still, Amphion is a system with the same goal as DxT.

The Design Maintenance System [10] is a transformation-based, compiler-like tool that parses source code, performs a sequence of transformations, and outputs code in the same or different language. The sequence of applied transformations can be very long, so DMS only explores a single sequence, possibly "undoing" explored transformations to find a sequence that ends in an implementation for which code can be generated. With DxT, we start with high-level representations and explore many, much smaller sequences of transformations.

ReFlO [26] is a visual, graphical tool used to encode DxT architectures and transformations. Instead of automatically generating code as DxTer does, one uses

ReFlO to manually search an implementation space. One starts with a set of transformations and an architecture specification (a simple, high-level dataflow graph) and ReFlO presents, at any point, the refinements and optimizations that can be applied. Ideally, one day we will have a system that combines the graphical and interactive experience of ReFlO with the automation offered by DxTer.

1.5.2 DLA and HPC

Autotuning is an important way to improve performance of code automatically[4]. Autotuning customization is largely limited to selecting tuning constants (e.g., loop unrolling factors or algorithmic blocksizes) rather than selecting and optimizing algorithms. ATLAS [63], for example, does this for BLAS operations on some architectures. It explores tuning factors for a ire-determined algorithm. For each implementation option, code is generated, compiled, and run on the target machine. Code runtime is used to search the implementation space and choose a "good" version. DxT is different and complementary in that it generates a space of semantically equivalent implementations from a high-level understanding of how algorithms can be developed. We envision a comprehensive process that includes a DxTer-like tool to generate code followed by an autotuning step to then choose the best parameters like, for example, the algorithmic block size and process grid configuration for distributed-memory code. The work of [64] demonstrates how some of the empirical search in ATLAS can be removed via performance modeling. It talks about the possibility for a hybrid approach with empirical and analytic modeling. For now, we use cost estimates that guide DxTer to the best implementation(s) instead of empirical search, but this is not a requirement of DxT. We can envision DxTer incorporating such a hybrid technique.

[4] This section is a slight modification of a similar comparison given in [48].

The linear algebra compiler by Fabregat-Traver and Bientinesi [24] takes a specification of a mathematical operation and derives a family of algorithmic variants. It optimizes the algorithms by reusing variables and mapping to BLAS function calls. The output of this system produces the type of algorithms we use as input to DxTer, encoded as refinements. DxTer then optimizes the implementation by parallelizing for multithreading or distributed memory. One can envision a more complete code generation system that starts with a DLA operation specification, generates algorithmic variants, and inputs them to DxTer for implementation.

DxT is similar in goal to SPIRAL [52], which largely focuses on generating high-performance *Digital Signal Processing (DSP)* kernels. It starts with a mathematical description of the algorithm in a DSL and performs transformations similar to refinements and optimizations to recursively replace abstract operations with implementation code and to improve that code. It uses machine learning via online code compilation and performance testing to explore a huge space of implementations. DxT targets higher-level operations, built on lower-level functions like those of the BLAS, so we can utilize relatively accurate cost models instead of empirically-based search. Further, our search space remains manageable.

The *Built-to-Order (BTO) BLAS* [11] system automatically generates vector-vector and matrix-vector BLAS operations targeting sequential and multithreaded architecture. It uses a unique representation of algorithms and code to employ a genetic search of implementation options including loop fusion and parallelization. Both empirical testing and performance modeling are used to limit and explore the implementation space. In our work so far, we target higher-level functionality, which allows for analytic estimates to be sufficiently accurate for search.

We envision in the future relying on kernels generated from a SPIRAL or

BTO BLAS-like approach instead of the hand-developed and hand-tuned implementations at the lowest levels of code. Then, the DLA software stack, from lowest level code up, will be automatically generated.

The *Tensor Contraction Engine (TCE)* [8] aims to generate code for a tensor contraction expressed in a high-level representation (DSL). It applies (mostly loop) transformations to optimize over computational complexity, space complexity, communication cost, and then data access cost. These transformations and its cost models are similar in spirit to those of DxT. TCE specifically targets tensor contractions; DxT is general-purpose.

The Broadway compiler [34] had a similar goal as ours to encode expert knowledge to generate optimized code. Library functions were annotated, so Broadway could choose the best implementation of an interface at a call site. It was not able to optimize as DxT does, though, which prevented it from generating the "best" code. Further, it did not use a search space of implementations, which is necessary to avoid local minima when exploring optimizations.

Many domain-specific compilers exist to optimize code written for a particular problem type (e.g., DLA). They generally use DSLs to express algorithms in a convenient representation. Using a DSL, the compiler takes advantage of high-level domain knowledge. Similarly, we use DSLs extensively in DxT (see Section 2.1). Domain-specific compilers are generally written by compiler experts. They are largely non-extensible by users and their optimizations are generally difficult to understand to a non-compiler-expert. One can think of DxT as a way to build a lot of the functionality found in a domain-specific compiler, making it extensible to a domain expert who is not also a compiler expert.

The FLAME project is closely related to DxT. In [30], "The Big Picture"

expressed the idea of encoding algorithms and expert knowledge to mechanically generate code. There, optimized parallel code was also the goal, but the PLAPACK library [59] was the targeted DSL instead of Elemental, described in Chapter 5. Many implementations were generated and performance estimates were created from cost function annotations in the algorithms. Our work benefits from extra years of insights and experience, which enable a more sophisticated approach based on graph transformations (which is more general-purpose than DLA-specific applications). Further, DxT is a generalization of this idea.

DLA runtime schedulers like SuperMatrix [17] and PLASMA [21] use sequential code to form a dataflow task graph. BLAS and LAPACK function calls are replaced with scheduler-specific functions that add tasks to the dataflow graph representing computation. When the graph is executed, the runtime scheduler chooses where to run each task (e.g., on CPU cores or GPUs), possibly optimizing the schedule to reduce communication (e.g., between the GPU and CPU). This is useful to handle issues such as load imbalance as a runtime scheduler can compensate by scheduling tasks around a slow processing unit. The schedulers' dataflow graphs look similar to a DxT-style graph with loops unrolled. The runtime scheduler is optimized to perform well, so it uses heuristics to optimize the schedule without exploring a massive search space of options (a costly endeavor). The heuristics and scheduling optimizations can be represented in the DxT style (thus, the scheduler acts like a runtime version of DxTer). Further, DxTer can be augmented to output SuperMatrix code. Already encoded, FLAME-derived algorithms could be transformed to use SuperMatrix functions that add dataflow tasks to the runtime graph instead of calling BLAS or LAPACK functions directly. For distributed memory, static scheduling with DxT does well as load imbalance is less of a concern. Also,

there are many options to parallelize and optimize distributed-memory code (as seen by the size of search spaces described in Chapter 5), so a static schedule benefits from exploring many options.

1.6 Contributions

The contributions of our work, described in this dissertation, are:

- We see DxT as one way to elevate software engineering to a science. It promotes structure and more formal reasoning in software design and design decisions. This is a general contribution to computer science.
- We demonstrate how to encode DLA algorithms and architecture-specific implementation and optimization design knowledge as graph transformations to generate code for distributed-memory, multithreaded, and sequential architectures. Encoded knowledge is used to generate high performance code automatically that rivals hand-developed code. This is evidence that software for other architectures and domains with similarly representable knowledge can be automatically generated as well.
- We present our prototype DxTer to which one inputs graph transformations encoding design knowledge, knowledge about domain functions, and a graph representing functionality to be implemented. DxTer generates a search space of optimized implementations and outputs a single "best" using a performance estimate. We describe ways to prune that search space to reduce search time or to make the space tractable to explore.
- We describe benefits of encoding design knowledge other than just relieving an expert's work of implementing code. First, code is trusted for correctness

as transformations are reasoned (or proven) to be correct. Second, automatic generation often finds better implementations than a person develops since optimization mistakes can be made but not discovered in testing. Lastly, making design knowledge explicit allows it to be more easily taught to others and requires the designer to justify decisions (which can lead to design improvements). In this dissertation, we use transformations to explain how to parallelize DLA code for distributed memory and shared memory, which demonstrates the pedagogical utility of DxT.

1.7 Outline

In Chapter 2, we present DxT in a domain-agnostic way and then explain DLA-specific DxT characteristics. In Chapter 3, we detail the structure of DLA code, we describe the FLAME approach, and we discuss the common loop transformations used for DLA. We present DxTer in Chapter 4. In Chapters 5 and 6, we demonstrate how we automate code generation for distributed-memory and multithreaded systems, respectively. Lastly, in Chapter 7, we summarize our results and discuss future work.

Chapter 2

Design by Transformation

We introduce the basics of DxT: how to encode algorithms and domain knowledge. We do this without DLA examples since DxT is not DLA-specific. We then explain DxT constructs that are especially important for DLA and in the next chapter explain DxT via DLA and DLA via DxT.

2.1 Representing Algorithms and Implementations

An *algorithm* is a step-by-step procedure to perform computation. Algorithms are represented in DxT as dataflow graphs. A node represents computation; an edge represents dataflow. Nodes come in two flavors: *interfaces* and *primitives*. Interfaces have no implementation details. They represent functionality in terms of preconditions and postconditions on the input/output edges[1]. Interfaces are architecture agnostic since they do not map directly to code. Primitives have precondition and postcondition definitions of functionality as well as implementation details. They map directly to given architecture-specific code and have properties such as cost

[1]As informal descriptions of functionality are sufficient here, we omit preconditions and postconditions for readability throughout the dissertation.

when that code is executed. Computation time, memory usage, and power consumption are common examples of cost. Figure 2.1 is an example graph of the interfaces `FOO` and `BAR`. It represents functionality that needs to be implemented and converted to code.

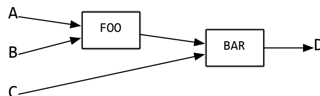

Figure 2.1: An example interface-only start graph.

Figure 2.2 is an implementation of the functionality of Figure 2.1. The two nodes (`FooFunc` and `BarFunc`) are primitives that map one-to-one to code (e.g., `D:=BarFunc(FooFunc(A,B),C)`) [2].

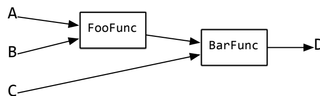

Figure 2.2: An example final implementation of Figure 2.1.

2.2 Representing Design Knowledge

In order to map Figure 2.1 to Figure 2.2, we employ hardware-agnostic and hardware-specific knowledge of the domain: knowledge about its operations and the interaction between them.

2.2.1 Refinements

A *refinement* encodes knowledge about how to implement an interface as an algorithm or primitive. It replaces an interface with a graph that satisfies the interface's

[2]The code generated from primitives need not be this simple, but it often is in practice.

preconditions and postconditions[3]. The replacement graph can contain (lower-level) interfaces and/or primitives to enable a hierarchy of functionality. Often, code is implemented in layers of abstraction or complexity, which is represented by a refinement hierarchy.

Figure 2.3 shows refinements for the `FOO` and `BAR` interfaces. For each, the *left-hand side (LHS)* of the bold arrow is an interface and the *right-hand side (RHS)* is a graph of primitives and/or interfaces. There are two refinements of `FOO`. The top-right represents a more efficient implementation, built from the primitives `FooFunc` and `Baz`. The bottom-right uses a node `FooAndOtherFunc`. The details of this node are not important. Let us consider it a primitive that is very expensive on the particular machine we are targeting, so while it is better in some cases, we avoid it here. (It might also be that the top refinement is only applicable when certain preconditions are met, otherwise the bottom must be used.) `BAR` only has one refinement, built from the primitives `InvBaz` and `BarFunc`. Figure 2.4 shows a refined version of Figure 2.1 obtained by applying two refinements.

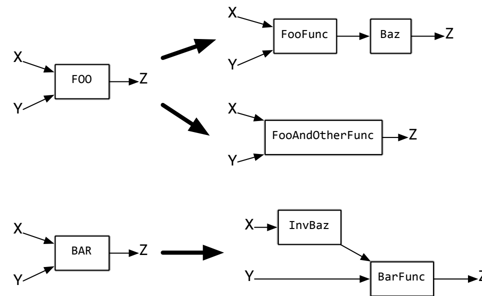

Figure 2.3: Refinement transformation examples.

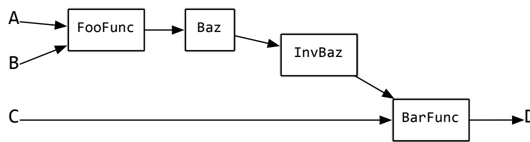

Figure 2.4: Intermediate graph that represents functional, but inefficient, code.

[3] In [27], requirements on how the graph can satisfy or strengthen the interface's preconditions and postconditions are described.

2.2.2 Optimizations

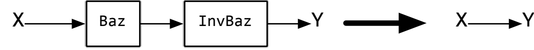

Figure 2.5: Optimization to remove unnecessary inversion operation.

While Figure 2.4 is a complete and correct implementation of Figure 2.1, it is not efficient when one knows that `Baz` followed by `InvBaz` is bad (e.g., `InvBaz` acts as an inverse of `Baz` so it is wasted computation). *Optimizations* encode knowledge about the interaction between domain components such as inverses. They express how a collection of nodes can be implemented in terms of another collection of nodes (i.e., one graph in terms of another). This is a basic metaoperation, replacing one algorithm (graph) with another. Figure 2.5 shows an optimization that encodes knowledge about these inverse operations. Applying the transformation of Figure 2.5 to the graph of Figure 2.4, we derive the optimized implementation in Figure 2.2.

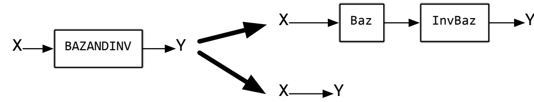

Figure 2.6: An alternate way to view an optimization.

Formally, the optimization of Figure 2.5 is represented by the relationship in Figure 2.6. It shows an interface `BAZANDINV` that can be implemented either as the LHS or RHS of the transformation in Figure 2.5. An optimization replaces the inefficient refinement to interface `BAZANDINV` and then refines to the efficient implementation. For convenience, we just show the direct optimizing transformation, like in Figure 2.5, when it is not useful to introduce `Baz` and `InvBaz`. This is a meta-optimization of the *knowledge base*, or the set of all transformations, where we recognize the structure of the domain and improve the encoded transformations

for performance and simplicity[4].

2.2.3 Graphs or Code?

Graphs of all primitives represent code as the primitives map directly to code. Graphs might encode additional information such as data type, as described in Section 2.5.2, but they still represent an implementation in code.

Viewing code as graphs allows us to keep important metainformation, but a graph and the code generated from it can be thought of as interchangeable in many ways (when the graph contains only primitives). For example, they are largely interchangeable when talking about what the implementation does, how it performs, and the changes we can make to it. Further, changes made to one can be equivalently made to the other.

2.3 Grammar

In this section, we explore the connection between the grammar of output code and the grammar of graphs in DxT. It is convenient to output code using a DSL (or to use an API as a DSL, as explained below) instead of code in a more general language like C++.

2.3.1 DSLs

There are two common ways to implement DSLs. The first uses a formally-defined grammar. A program written in the DSL can be parsed using that grammar, possibly with a domain-specific compiler. SQL is an example DSL implemented in such a

[4] There is another type of transformation called an *extension* [27], which is not needed for DLA so we do not discuss them.

way [18]. The second approach uses domain-specific APIs implemented via libraries, where a program in the DSL only calls those APIs. It is the second form of implementation that is common for DLA, but having an explicit grammar is equally viable.

Node types are derived from DSLs, which are an important part of DxT. At the start of manual code development, one might use a DSL based in mathematics (e.g., a DLA-specific notation used on paper) to describe functionality to be implemented. When encoding this functionality in DxT, interface types are limited to that DSL. The final code also uses a DSL API from which DxT *primitive types* (i.e., the operation types that are primitives) are chosen. We describe in subsequent chapters how the primitives we use belong to the Elemental and BLIS DSLs and the interfaces come from FLAME and DLA APIs.

It is convenient to target code to DSLs in this way. Then, we only need to support a limited number of node types, coming from the DSLs, instead of dealing with general purpose languages and a larger variety of expressible code. This limits the number of node types to represent and encode knowledge about and, therefore, the number of transformations encoded to optimize and implement those nodes. This is the same reason developers use DSLs when implementing code manually. For example, one does not have to consider lower-order or less-important details when using a well-designed DSL since those concerns are abstracted away. One only considers design decisions with a relatively small number of operations.

Of course, this creates the classic problems associated with language design. One must choose or design the right DSL, with the right abstractions and the right code patterns. As with Elemental and BLIS APIs, though, the key is expertise and effort. Well-designed DSLs aid developers and aid us in generating code automati-

cally.

We start with an algorithm given in an architecture-agnostic DSL. The goal is to make architecture-specific implementation choices and map the graph to code in an architecture-specific DSL. The various DSLs we work with form something of a family, where the domain's key programming constructs (e.g., loop control constructs for DLA) show up in each, possibly with architecture-specific implementation details.

2.3.2 Exploring the Language

The knowledge base of a domain and the target DSL form a hypergraph grammar [9, 55]. We do not detail the formalities of the grammar here, but do discuss the way the grammar is explored in DxTer. A sentence of the grammar is an implementation of functionality in the domain. The grammar provides a way to "reword" the sentence, implementing the same functionality in a different way.

Figure 2.7 shows the set of sentences for a DxT grammar $\mathcal{G}$. This is the language of the grammar, $\mathcal{L}(\mathcal{G})$. Within that set, there is a subset of sentences that have some architecture-specific details. One starts with a sentence (an interface-only program) P_0. One applies transformations to "reword" the sentence, searching, for example, for a high-performance implementation. In Figure 2.7, this is represented by moving from one point (sentence) to another in the language. By applying some refinements one reaches a sentence with some architecture-specific details, but there are still some interfaces to be refined before we have a complete implementation.

By applying enough refinements, one reaches a sentence with no interfaces that maps to code (P_4 here). This innermost region of the language $\mathcal{L}(\mathcal{G})$ contains all graphs that can map to code in the DSL for the target architecture. Notice that

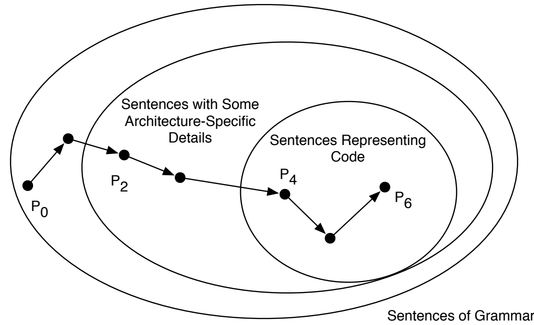

Figure 2.7: View of transformations exploring sentences of a grammar.

the grammar of this innermost region is *not* the same as the grammar for the DSL. A DxT graph of a program has a different grammar than the code to which that graph maps, though there is a strong relationship. For example, a DxT graph could have more limitations on the structure of code or a single primitive on a graph could map to multiple lines of DSL code.

Applying optimizations to P_4, one arrives at different code, represented by P_6. Each point explored from P_0 to P_6 implements the functionality of P_0 in different ways with different architecture-specific details.

2.3.3 A Family of DSLs

Commonly, multiple DSLs are utilized to implement domain functionality across architectures. One uses a DSL to encode domain algorithms. One also has a selection of related DSLs to implement those algorithms for particular architectures, using a different DSL for each target. Figure 2.8 visualizes such a relationship with two architectures.

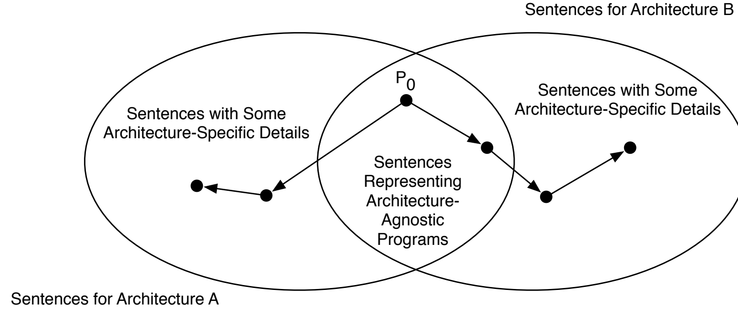

Figure 2.8: View of using multiple DSLs.

In this example, there are two architectures ARCHA and ARCHB. We have DxT grammars $\mathcal{G}_{\text{ARCHA}}$ and $\mathcal{G}_{\text{ARCHB}}$ to explore implementations for these two architectures, respectively. The two outer ovals of Figure 2.8 represent the DxT languages $\mathcal{L}(\mathcal{G}_{\text{ARCHA}})$ and $\mathcal{L}(\mathcal{G}_{\text{ARCHB}})$. Within each, one explores implementations to target the two architectures. The center area (the intersection of the two languages) contains sentences in the domain's language that includes no architecture-specific details. It contains algorithms without hardware-specific implementation details, so they can target either architecture.

The DSLs DSL_A and DSL_B are used to implement domain functionality for architectures ARCHA and ARCHB, respectively. $\mathcal{L}(\mathcal{G}_{\text{ARCHA}})$ and $\mathcal{L}(\mathcal{G}_{\text{ARCHB}})$ include graphs that represent a subset of the programs expressible in DSL_A and DSL_B, respectively[5]. The disjoint regions in Figure 2.8 represent algorithms with some architecture-specific decisions in those DSLs. Since $\mathcal{L}(\mathcal{G}_{\text{ARCHA}})$, for example, contains some graphs that do not represent DSL code (i.e., some graphs have architecture-agnostic interfaces) and not all DSL_A code is representable with $\mathcal{G}_{\text{ARCHA}}$, $\mathcal{G}_{\text{ARCHA}}$ is not the grammar for DSL_A.

In Figure 2.8, we show how one starts with P_0, a program to be implemented for a specific architecture. One uses transformations from the architecture-agnostic,

[5] One typically does not need to express in DxT all programs that can be written in a DSL.

interface-only grammar to explore implementations in the center no matter the architecture target. One also uses architecture-specific transformations to refine graphs to explore implementations specifically for ARCHA or ARCHB, where those transformations choose details for DSL_A or DSL_B, respectively.

Thus, with multiple DSLs, there are some grammar rules (transformations) that are explored regardless of the target architecture. They are reused each time a library of functionality is ported to a new machine and, we believe, should be encoded for posterity. One also uses DSL-specific rules to target a specific architecture. Those are reused for the various functionality being implemented.

2.3.4 Context Sensitivity

It is possible to have conditions on transformations beyond just matching the LHS in a graph. For example, the RHS of a refinement can have more constraining preconditions than imposed by the interface of the LHS. Consider the `SORT` interface with refinements shown in Figure 2.9. The top refinement is always valid. The bottom refinement would do nothing (i.e., just pass through the input), but it is only valid if the input `X` meets the condition that it is already sorted.

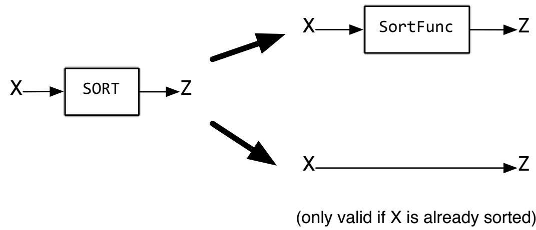

Figure 2.9: Two refinements of `SORT`, one of which has a condition for application.

Another example: the production rules of Figure 2.6 might only be equivalent if `X` meets certain conditions. If `X` is a list of data, `Baz` randomizes the input list's order, and `InvBaz` orders the data. Really, `Baz` and `InvBaz` (which is poorly named

in this case) are only inverses if `X` is sorted to begin with. Otherwise, the sorted data that comes out of `InvBaz` will not be the same as the unsorted data that goes into `Baz`. This condition on the applicability of the transformation is encoded with DxT.

Thus, the graph grammar of transformations can be context sensitive, and, in fact, DLA transformations are often context sensitive. Conditions such as problem size are common, as described in Chapter 3.

2.4 Connection to Model Driven Engineering

Model Driven Engineering (MDE) is an inspiration for our work [25, 36]. We follow the idea of starting with a *platform-independent model (PIM)* and refining to a *platform-specific model (PSM)* that targets a particular (code) artifact on a specific architecture. In DLA and other domains, refinements are insufficient to derive efficient, high-performance implementations. One must break through the boundaries around interfaces to optimize refinement-exposed components.

Optimizing transformations introduce some difficulty in the derivation process. Since they form a relationship between the refinements of different interfaces, one cannot simply choose the locally-best refinement (i.e., the refinement of an interface that is best) as in a dynamic programming approach. It is possible that suboptimal refinements of two interfaces allow for a optimization between the graphs exposed by refinements (called *cross-boundary* interfaces) that leads to the globally optimal design.

Therefore, with DxT, there is a search process to find the best implementation. In fact, there is a combinatorial search space of graphs derived from the starting interface-only graph. As described in Section 2.3, each point in the search

space is a valid implementation with varying amounts of implementation decisions made. The path from one point to another comes from the transformations that generate one implementation from the other. In Chapter 4, we discuss the way DxTer enumerates this space and searches for a "best" implementation.

2.5 DLA Specifics

For DLA, we use directed, acyclic multigraphs (DAGs)[6]. Each node can have multiple output values, so edge annotations specify which output value flows along the edge.

2.5.1 Loops in an Acyclic Graph

Loops are an essential algorithmic structure in DLA code[7]. In DxT, a loop is represented by "boxing off" in the graph. The subgraph within a loop structure represents loop-body operations. The graph of Figure 2.10 scales a vector v by π. It does so by iterating over the elements of v and performing scalar multiplication on each. The outer box represents a loop and the `ScalarMult` is a loop-body operation.

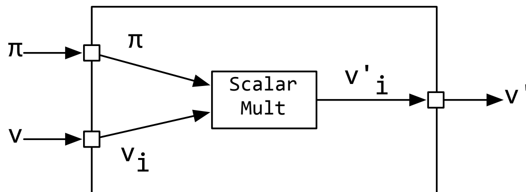

Figure 2.10: A graph with a loop, two input tunnels, and one output tunnel.

Tunnels on the edge of loops serve as a port between code outside of the loop and the loop-body code within. Tunnels can represent a pass-through such that the entire input is passed through to the loop body via its outgoing edges, as shown in

[6] In other domains [53], cycles can exist, but they do not exist in DLA.
[7] Our loop representation is based heavily on that of LabVIEW.

Figure 2.10. The same data is passed in on each iteration of the loop. In this case, the same value of π is used in each iteration, so it is passed through the tunnel.

Tunnels can also partition their input. These are called *split* tunnels. As described in Chapter 3, in DLA it is common to take submatrices of an input matrix in each loop iteration. Tunnels represent partitioning by passing submatrices to the loop body on outgoing edges. The boundaries and sizes of the submatrices change in each iteration of the loop, so different portions of the matrix get past in on each iteration. In this case, a different element of v is passed in on each iteration. We label that value in the graph as v_i for clarity.

Combine tunnels merge submatrices into a whole matrix on the output side of a loop. In this case, v'_i from each iteration is combined to form v'. Combine and split tunnels often come in pairs and the inputs to the combine tunnel always originally come from the matching split (though they may have passed through and been changed by loop-body operations). Thus, v has the same length as v'. We often omit the combine tunnel of a loop input that is read only since the output from the loop is the same as the input[8].

Split and combine tunnels have annotations to specify the direction of partitioning, blocksize, and so forth. For example, the split tunnel for v could iterate forwards or backwards through the vector. Instead of indexing a single scalar, it could have indexed a subvector with a fixed length or a length that depends on the iteration number. Thus, split tunnels look very similar to a data iterator construct.

One split tunnel on each loop is identified as the loop's *control* tunnel, which determines the number of loop iterations. The split tunnel for v is the control in this example, so it prescribes that the number of loop iterations is equal to the length

[8]In this case, any code that should take that data as input should be connected to the original producer of the data (i.e., the input to the loop) instead of the loop's output of the same data.

of the vector v. Throughout this dissertation, we generally focus on the loop-body subgraphs and do not visualize the loop itself, but the loop and its tunnels must be represented.

2.5.2 Type Information

As it is necessary to have runtime or compile-time type checking for code, it is also necessary to maintain and check type information on DxT graphs. For DLA, this could include edge's matrix sizes (across all iterations of loops), data type, structure (e.g., upper/lower triangular), and data distribution. For example, a Cholesky factorization node outputs a matrix that is either lower or upper triangular as a feature of the node type (lower or upper triangle Cholesky factor). Matrix sizes are important to keep track of, for example, to validate node preconditions are met. Chapter 4 discusses this in greater detail.

2.5.3 Correct by Construction

A starting interface-only graph for DLA is generally derived to be correct using the FLAME approach [60]. Even when this is not the case, domain experts often have great trust in the algorithms they develop to solve a task. We do not want to apply transformations that invalidate algorithm correctness, generating incorrect code from a correct algorithm. Therefore, we only use transformations that maintain correctness. How do we know transformations are correct?

First, it is important to understand how one trusts implementations of FLAME-derived algorithms. In Chapter 3, we discuss the structure of DLA algorithms and code. The key is that there is a finite set of commonly used operations on top of which algorithms are built and in terms of which code is implemented. As

we will see, the operations have a specific number of valid parameter combinations (ignoring different matrix/data values). It is standard to test the operations' various parameter combinations on a set of random matrices or matrices with carefully chosen structure to have great trust in the components. This can be thought of as unit testing for DLA. More complex algorithms are built on top of those components, using them in various ways. Those algorithms are similarly unit tested, which increases trust in the lower components (if they did not work, the higher-level algorithms would not work either). This is the accepted way to "trust" DLA code.

For now, we take a similar tactic for transformations. Some are obviously true (e.g., removing operations defined to be inverses of each other as in Figure 2.5). For others, we can reason about transformations to trust their correctness (as is often the case with parallelizing refinements).

In either case, most transformations get reused often in DxTer, and the code they yield gets tested for correctness. Errors in transformations are found quickly because errors in the code are found quickly and tracked back to the transformations that produced them. When they are fixed, all of the code derived from the corrected transformation is fixed.

Additionally, type checking is performed throughout DxTer. Many nodes have requirements on input data sizes, for example making sure that input matrix dimensions match up with each other. Loops ensure all partitioning tunnels have the same number of iterations as the control. Type checking like this often raises flags if a transformation is incorrect due to errors such as switched inputs.

Through code testing and type checking, we gain a great trust in the transformations in DxTer, which leads to a great trust in the correctness of output code. Eventually, we want to prove the correctness of transformations formally by proving

that the preconditions and postconditions of the RHS satisfy those of the LHS. In some cases, this requires a logic notation that we do not currently have (e.g., of data movement that results from communication components). This will be a future area of study.

2.6 Summary

A node on a DxT graph represents an operation. The operation has inputs and outputs, which are represented as incoming and outgoing edges on the graph. The operation's functionality is expressed in terms of preconditions and postconditions that may include type specifications (e.g., input data sizes). Nodes have two flavors: 1) interfaces, which represent functionality but do not have a specified implementation, or 2) primitives, which additionally include implementation details (e.g., code and execution time estimates).

There are two types of graph transformations needed for DLA. Refinements replace an interface with an implementation – a graph that uses primitives or lower-level interfaces and maintains the same precondition and postcondition specification of functionality. An optimization replaces a subgraph with another subgraph that implements the same functionality (has the same precondition and postcondition specification) but does so in a different way. Therefore, we can refine with architecture-specific implementations and optimize to implement functionality in better-performing ways.

Loops are an essential part of DLA code, so they must be represented in DxT DAGs. Figure 2.11 shows an example that arises when implementing `Gemm` ($C := \alpha AB + \beta C$). In Figure 2.11 (left), matrices A, B, and C are input to a loop, represented by the outer box. The loop body is represented inside this box. The

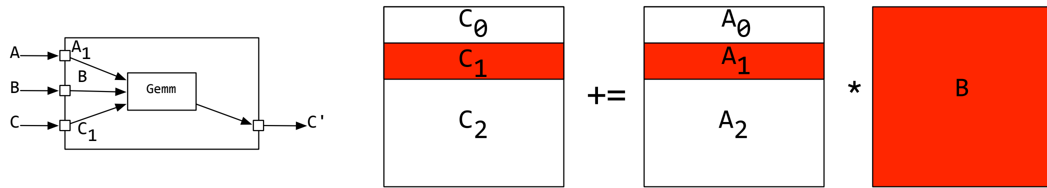

Figure 2.11: Example of loop (left) and computation of each iteration (right).

inner box, labeled `Gemm`, is executed on each iteration of the loop.

The smaller boxes on the left side of the loop box are loop tunnels. On each iteration of the loop, these tunnels pass submatrices into the loop body. The labels on edges coming out of tunnels specify the submatrix that flows on the edge, which Figure 2.11 (right) visualizes, in red. They change in each iteration. The tunnels also encode, for example, choices of partitioning direction and blocksizes, but these details are not shown in the visual representation.

Chapter 3

Domain Structure

While DxT is a general approach to software engineering, we target DLA here. Many of the lessons learned can be applied when using DxT in other domains. We now outline how the DLA software stack is developed by layering algorithms, reusing the same set of algorithmic knowledge repeatedly. After decades of polishing DLA software abstractions [7, 17, 50, 61], we have well-layered and understood code expressible in succinct and high-performance DSLs. We explain how this enables us to encode design knowledge with DxT and in later chapters demonstrate how this enables DxTer to explore implementation options.

3.1 Variants and Layering

The FLAME methodology [32, 33, 60] provides a way to derive a family of algorithmic variants for a DLA operation in a mechanical or automatic way [12, 24]. Figures 3.1- 3.3 show three variants for `Gemm` derived via this approach. DLA algorithms are typically loop based. In *blocked* algorithms, submatrices are exposed in each iteration of the loop, and loop-body operations use and update some of those

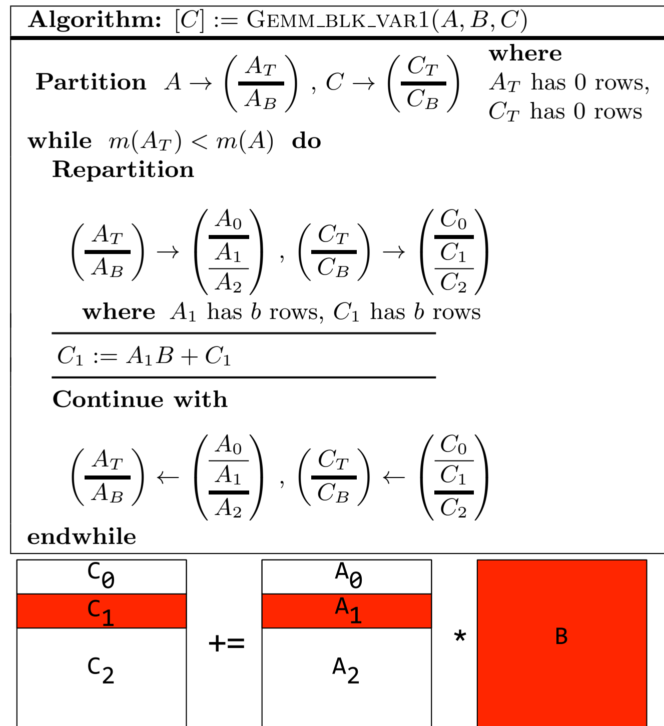

Figure 3.1: Variant 1 of Gemm to compute $C := AB+C$ (Normal, Normal) or Gemm NN with pictoral representation of how Gemm is broken down into small Gemm operations.

submatrices. By using blocks of submatrices, one can engineer code to take advantage of caches (keeping blocks in cache to reduce read / write time).x A blocksize, labeled b in these algorithms, is chosen to control the size of submatrices. *Unblocked* algorithms are similar but work on vectors and scalars. Unblocked algorithms can be thought of as blocked algorithms with the blocksize set to one ($b = 1$). Upon loop completion, the algorithm is finished and the final result is computed.

Algorithms generally have at least one update statement that is recursive: the operation performed by the algorithm is also performed on submatrices (thus, it has smaller input sizes). Each loop-body operation in the algorithms of Figures 3.1-3.3, for example, is a Gemm operation itself. Each algorithm reduces the size of

Algorithm: $[C] := \text{GEMM_BLK_VAR2}(A, B, C)$

Partition $A \rightarrow \left(A_L \,\middle|\, A_R \right)$, $B \rightarrow \left(\frac{B_T}{B_B} \right)$
where A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**

Repartition

$$\left(A_L \,\middle|\, A_R \right) \rightarrow \left(A_0 \,\middle|\, A_1 \,\middle|\, A_2 \right) , \left(\frac{B_T}{B_B} \right) \rightarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$$

where A_1 has b columns, B_1 has b rows

$C := A_1 B_1 + C$

Continue with

$$\left(A_L \,\middle|\, A_R \right) \leftarrow \left(A_0 \,\middle|\, A_1 \,\middle|\, A_2 \right) , \left(\frac{B_T}{B_B} \right) \leftarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$$

endwhile

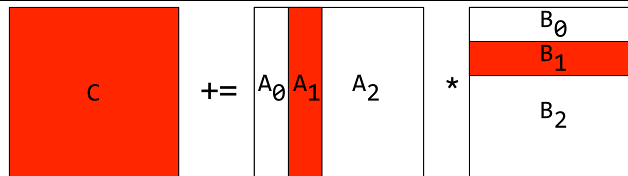

Figure 3.2: Variant 2 of `Gemm` to compute $C := AB + C$ (Normal, Normal) or `Gemm NN` with pictoral representation of how `Gemm` is broken down into small `Gemm` operations.

inputs in one dimension (m, k, or n) to the blocksize, b. The algorithmic variant's partitioning and blocksize determine the shape and size of the operands in recursive calls.

When implementing a DLA operation, one chooses algorithmic variants to reduce the problem size, where each variant does so along one or two dimensions. This is done to reduce operand sizes to the point that they can be kept in main memory, levels of cache, or registers, for example, to attain high performance.

The algorithm in Figure 3.1 partitions in the m dimension, so the `Gemm` update statement has an m-size of b. Another variant is used to implement that recursive `Gemm` by partitioning the problem in a different dimension, thus layering

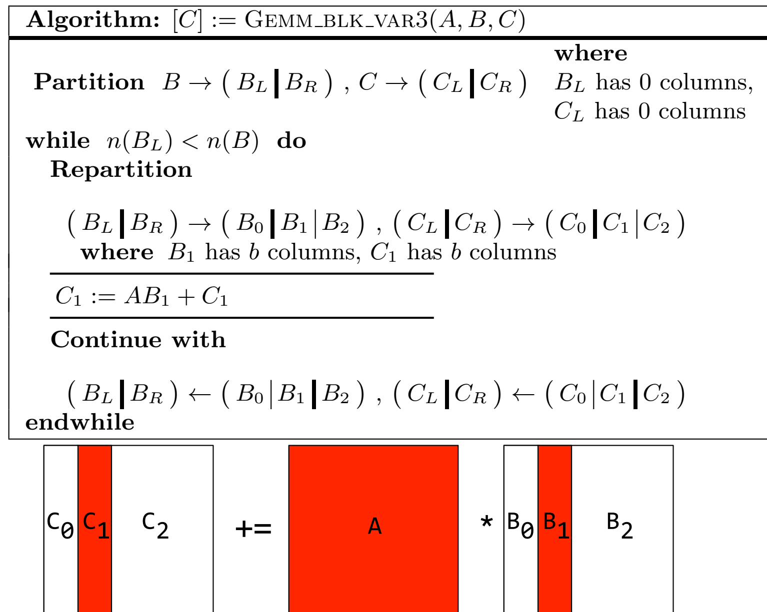

Figure 3.3: Variant 3 of `Gemm` to compute $C := AB + C$ (Normal, Normal) or `Gemm NN` with pictoral representation of how `Gemm` is broken down into small `Gemm` operations.

algorithms. A developer uses deep knowledge of the target architecture to choose which algorithmic variants to layer. He does so to reduce communication between caches, processors, and so forth, which is discussed in the following chapters. With enough layers, the innermost subproblem can be implemented by a primitive, perhaps calling a library function or scalar multiply and add.

To give perspective: in Chapter 6, we explain how algorithms are layered such that data is sized to remain in each layer of cache (one algorithm layer for each layer of cache). For `Gemm`, there are often six layers of the three algorithms [62]! The smallest subproblem is implemented by bringing data into registers and calling scalar multiply and add. Loop transformations are applied for the inner layers, but these basic algorithms are the starting point. For distributed memory, `Gemm` is implemented with only one of those algorithms, and then a sequential library's `Gemm` function is

called (which itself may have six layers) [31]. With all of this knowledge reuse when implementing libraries manually, we want to encode the basic algorithms and target-specific implementation details once and reuse them automatically instead.

3.2 DLA Operations

There is a small set of commonly used update statements that show up repeatedly in more complex DLA algorithms. This set has been made into the de-facto standard called the BLAS. The matrix-matrix subset of the BLAS, found in blocked algorithms, are the *level-3 BLAS (BLAS3)*, listed in Figure 3.4[1]. Each of these operations has multiple versions with small differences (e.g., transposition of operands); a count of the versions is shown in Figure 3.4.

`Gemm` is the most commonly used BLAS3 operation because one can attain high performance from it and then build other operations to attain high performance with it [38]. Recall the algorithm of Figure 1.1 for `trmm`. The loop-body operations are annotated with their BLAS3 names in parenthesis. As mentioned, recursive calls are common. Further, a call to `Gemm` is typical for most BLAS3 algorithms [38]. Since architecture-specific `Gemm` implementation knowledge must be reused repeatedly to develop all BLAS3 operations, DxT program generation can help with automated design knowledge reuse.

DLA or higher-level scientific algorithms are implemented in terms of the standard BLAS interfaces so code is portable. Code is linked to a BLAS library implemented for a particular architecture (e.g., for a specific model of an Intel single-core processor). Then, it can be easily retargeted to a different architecture by relinking with a different BLAS library (e.g., for a multicore AMD processor).

[1]Vector-vector BLAS operations are called level-1 and matrix-vector are called level-2.

BLAS3	# of Versions	Sample Operation
Gemm	4	$C := \alpha AB + \beta C$
Hemm	4	$C := \alpha AB + \beta C$
Her2k	4	$C := \alpha(AB^H + BA^H)\beta C$
Herk	4	$C := \alpha AA^H + \beta C$
Symm	4	$C := \alpha AB + \beta C$
Syr2k	4	$C := \alpha(AB^T + BA^T)\beta C$
Syrk	4	$C := \alpha AA^T + \beta C$
Trmm	8	$B := \alpha BL$
Trsm	8	$B := \alpha A^{-1}B$

Figure 3.4: BLAS3 operations and the number of versions of each.

Thus, the higher-level code does not need to change (much or at all) to retarget architectures and achieve good performance.

The benefits of this approach include 1) portability, 2) a limited amount of functionality that needs to be implemented for portability (see Figure 3.4), 3) simplicity of algorithm code using a limited number of operations, and 4) the BLAS3 can be implemented with high performance, so algorithms coded in terms of them can also attain high performance.

The BLAS standard is implemented in many libraries. For example, Intel's MKL [3] is a closed-source library purchased for high-performance BLAS implementations on Intel and other x86 processors. nVIDIA's closed-source CUBLAS [4] provides the BLAS for nVIDIA GPUs. An open-source and free BLAS library is provided at [5], often referred to as the "Netlib BLAS" or "reference implementation." It does not have specialized code per processor, so its performance is generally lacking and is, therefore, used as a reference as a correct implementation. There are many more BLAS libraries.

The newest member is BLIS [62, 65], an open-source framework for developing BLAS libraries that refactored the techniques for implementing the BLAS

pioneered by Goto [28, 29] While other open-source BLAS libraries are implemented with little concern for code readability and maintenance, these were fundamental design goals for BLIS while still achieving high performance. BLIS enables one to implement all BLAS functionality quickly by requiring only a small number of functions to be written for a target architecture. Previously, when new architectures came online, one had to either 1) purchase a vendor-implemented BLAS library (which is not guaranteed to exist or perform well), 2) live with poor performance provided by the Netlib BLAS, 3) wait for somebody else to implement the BLAS in open source, 4) implement all BLAS functionality from scratch, or 5) become an expert with an open-source BLAS library and shoehorn it to fit the new architecture. The first three options are the most commonly used. As new architectures are frequently released, users are often left with a waiting period for a new BLAS library or left to accept inferior performance. Instead, one learns a small amount about BLIS, leaves almost all of its code untouched, and plugs in a few, relatively short pieces of architecture-specific code.

LAPACK [7] is a library that standardized higher-level DLA functionality like matrix factorization schemes (e.g., Cholesky), eigenvalue decomposition, and solvers for special forms of equations. LAPACK is built on BLAS operations to attain performance and portability on general-purpose processors. When moving LAPACK to a new processor, a tuned, architecture-specific BLAS library is linked and high performance is generally attained. LAPACK itself is built for sequential processors, but just like the BLAS there are libraries that implement LAPACK-level functionality on all architecture classes (e.g., ScaLAPACK [14] for distributed memory).

For LAPACK-level and BLAS-level operations, FLAME-derived algorithms

are implemented in various hardware-specific libraries. BLIS provides BLAS functionality, targeting sequential architectures (and we generate multithreaded BLAS functionality using BLIS as a DSL in Chapter 6). libflame [61] provides LAPACK-level functionality for sequential CPU, multithreaded CPU, and multiGPU architectures (and links to a BLAS library for CPUs or GPUs). Lastly, Elemental [50] provides BLAS and LAPACK-level functionality for distributed-memory (and links to sequential BLAS and LAPACK libraries for on-node functionality).

3.3 FLAME Algorithms in DxT

FLAME-derived algorithms are mathematical specifications without architecture-specific implementation details, so they can be used on any hardware. Loop-body operations match the mathematical functions implemented in the BLAS and LAPACK libraries in terms of the computation to be performed, but they do not specify how to perform it. We can think of the BLAS and LAPACK standards as DLA DSLs for describing algorithms (along with FLAME-like loop structures like matrix partitioning).

The main steps to implement an operation in high-performance code are first to choose which algorithmic variant to implement from the family of options, then to implement the loop-body operations in architecture-specific code, and finally to optimize the combination of loop-body code. One might just call the relevant architecture-specific BLAS or LAPACK library for each loop-body operation, but this generally hides considerable inefficiencies caused by data movement (demonstrated in Chapters 5 and 6) that can be optimized away by exposing lower-level implementation decisions.

3.3.1 Layer-Templatized Refinements

Thinking of the operations of the BLAS and LAPACK standards and loop structures as a DSL, we want to represent the DSL code in DxT. We define nodes by the computation, communication or movement of data, or partitioning/looping they perform. Nodes have preconditions on input data like conformality of input matrix sizes or matrix structure required for the computation. Nodes are labeled in our graphs by their BLAS/LAPACK names, but some of the additional type information/conditions are also encoded in DxTer, described in Chapter 4.

Each node type is annotated with the software layer it targets. For example, `Gemm` is found in a distributed-memory library, a sequential BLAS library, and at multiple layers within each, so it is tagged with one of these layers.

We use the following convention: each layer of software has a layer number. Layer 0 is always the most abstract layer, where the algorithm has no implementation details. From there, layer numbering is architecture-specific. When dealing with distributed-memory software, primitives belong to layer 2 (described in Chapter 5) and map to sequential BLAS library function calls. With multithreading, primitives belong to layer 3 or 4 (Chapter 6). In later chapters, we omit layer tags as they are understood within a context or they are expressed via the names of nodes in the DAG.

We encode each FLAME-derived algorithmic variant as a refinement. It is templatized on the *left-hand side (LHS)* node's layer and the *right-hand side (RHS)* nodes are labeled with a larger layer. As node layers increase with each refinement, refinement recursion is guaranteed to terminate. `Gemm` is refined in terms of a lower software layer / higher layer number `Gemm` and there is a finite and small number of legal layers. Refinement templates are instantiated to suit the particular architecture

layer being targeted.

There is another way of accomplishing the same goal of limiting recursion. Each layer of recursion decreases the operands' sizes, so the LHS of each refinement could have constraints based on the operand sizes. Refinements partition along particular dimensions, so they are only applicable if the length of those dimensions is greater than the blocksize used by the refinement.

Doing so effectively limits recursion, however it is useful to talk explicitly about layers of algorithms. It is often the case (as shown in Chapter 6) that an expert knows which algorithmic variant to use at a particular layer. Having refinements that are layer-templatized allows one to encode this knowledge by only instantiating with desired layers.

3.3.2 An Abstract Layering Example

Specific layering and concrete examples are presented in future chapters. Here, we derive a hypothetical software stack with three layers. From top to bottom – outermost to innermost – they are called `LAYER0`, `LAYER1`, and `LAYER2`. Any `LAYER2` node is a primitive that maps to a given implementation (i.e., a function call).

Some templatized refinements are shown in Figure 3.5 for `Trmm` and `Gemm`. The refinements (a), (b), and (c) encode algorithmic variants 1, 2, and 3 (Figures 3.1-3.3) of `Gemm`, respectively, and (d) shows the variant of `Trmm` of Figure 1.1. Σ, μ, and Ω are layer template parameters. While partitioning directions and blocksizes are encoded in DxTer, we omit those details here.

Consider a `LAYER0 Gemm` operation by itself, shown in Figure 3.6 (a). This graph represents that we want to implement `Gemm`. We need to apply refinements to target specific hardware (in this case, we are deriving the high-performance imple-

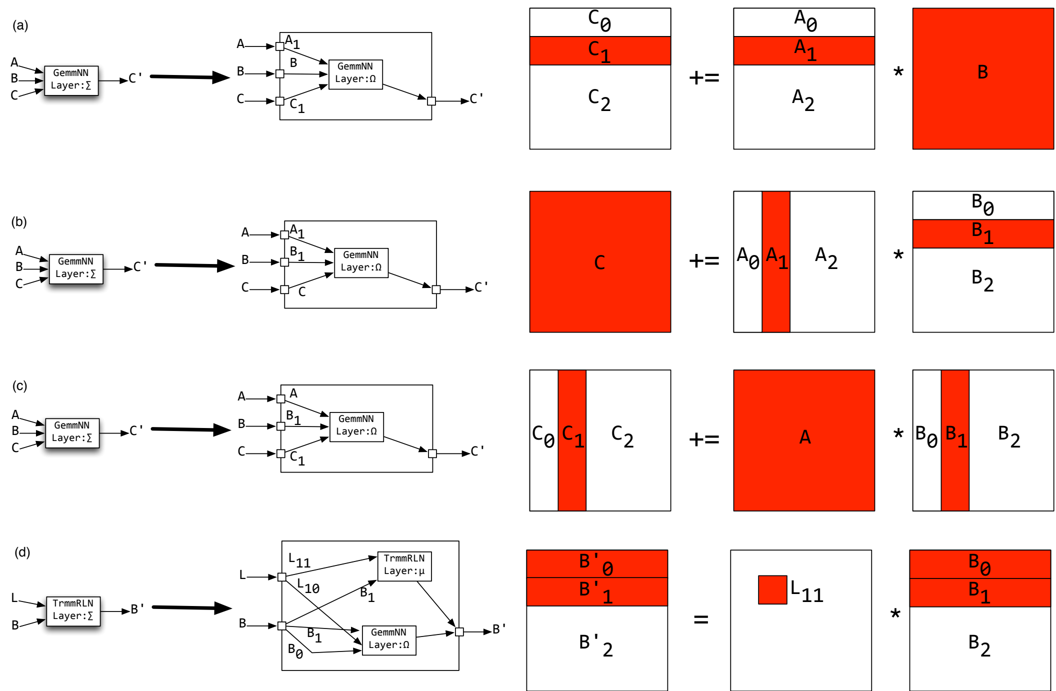

Figure 3.5: Four refinements templatized on LHS and RHS node layers where Σ, μ, and Ω are template parameters chosen for each refinement.

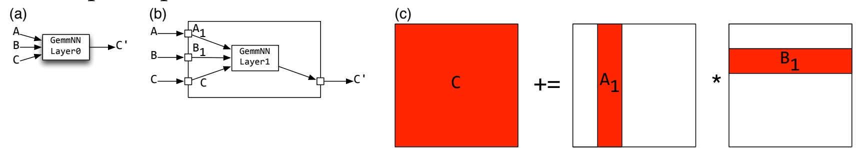

Figure 3.6: First refinement of (a) `LAYER0` `Gemm` operation to (b). (c) shows the partitioning of this loop, with the current iteration shown in red.

mentation used in BLIS and described in Chapter 6). In this example, an expert knows `Gemm` is best refined using the transformation of Figure 3.5 (b) with $\Sigma :=$ `LAYER0` and $\Omega :=$ `LAYER1`. The result is shown in Figure 3.6 (b).

Then, the remaining `LAYER1` `Gemm` operation is refined using the transformation of Figure 3.5 (c) with $\Sigma :=$ `LAYER1` and $\Omega :=$ `LAYER2`. The final design is the graph of Figure 3.7 (a). This graph only contains primitives, which map directly to DSL code. This is the high-performance implementation found in BLIS, where

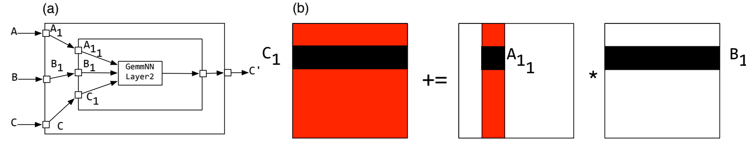

Figure 3.7: (a) Refinement of graph in Figure 3.6 (b) to a graph of only the primitive `LAYER2 Gemm`. (e) shows the iterations of the inner loop over the outer loop (red portion) in Figure 3.6 (c), with the current iteration shown in black.

the submatrices of A and B are targeted to stay in particular levels of cache, as we describe in Chapter 6.

The nested loops form the layers of the software. Figure 3.6 (c) shows the iterations of the outer loop of Figure 3.7 (a) or of the sole loop in Figure 3.6 (b) with the current iteration in red. Figure 3.7 (b) overlays the iterations of the inner loop over the iterations of the outer loop with the current iteration in black. It is this small operation (in black) that the primitive `LAYER2 GemmNN` operations implements (called for each iteration).

If we start with a `LAYER0 Trmm` operation (shown in Figure 3.8 (a)), we can apply the refinement of Figure 3.5 (d) with template parameters $\Sigma :=$ `LAYER0`, $\Omega :=$ `LAYER1`, and $\mu :=$ `LAYER2`. Figure 3.8 (b) shows the resulting graph. Then, we can reuse the `Gemm` refinement of Figure 3.5 (c) with $\Sigma :=$ `LAYER1` and $\Omega :=$ `LAYER2`. The resulting graph (Figure 3.8 (c)) contains two primitive/`LAYER2` nodes (`Gemm` and `Trmm`), so it maps directly to code.

For these two implementations, we needed three instantiations of the refinements of Figure 3.5, one of which was used to implement both `Gemm` and `Trmm`. When implementing all of the BLAS3 operations, `Gemm`-related transformations are reused repeatedly (a rote reapplication of knowledge). In the following chapters, we demonstrate how this structure enables automatic code generation, including adding layers and retargeting layers of design knowledge to a new architecture.

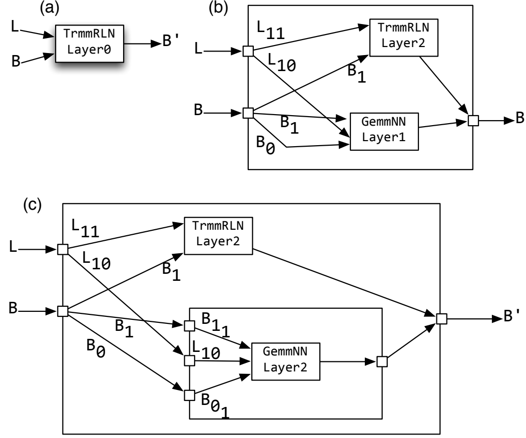

Figure 3.8: Derivation of `LAYER0` `Trmm` operation to a graph of only `LAYER2` primitives.

3.4 Loop Transformations

Loop transformations are essential to achieving high performance in DLA code. Loop fission, strip mining, and unrolling are commonly applied at low layers of code [62]. Those layers, though, are not discussed in this dissertation and are taken as primitives in the code output from DxTer (for now). Loop fusion, on the other hand, is a transformation commonly applied at high levels of the stack. With loop fusion, experts or DxTer can better optimize communication for distributed memory (Chapter 5) and data copying for sequential code (Chapter 6).

Compilers generally cannot perform loop fusion and optimizations on Elemental- or BLIS-level code. Code becomes too obfuscated by domain abstractions for compilers to determine loop dependencies and read/write patterns, so domain-agnostic compiler analysis is ineffective on DSL abstractions. With DxTer, high-level knowledge of loops and node computation can be encoded. For FLAME-derived loops (e.g., those shown in Figure 3.5), we know the way submatrices are read and written. Therefore, along with encoding the loops themselves, we annotate loops with higher-level knowledge so DxTer can perform loop fusion.

In [41, 42], we describe the loop/algorithm knowledge needed and give proofs for the fusion criteria DxTer uses. Here, we only present the basics used in DxTer without proof. FLAME algorithms are derived starting with a statement about the result of the operation's computation, the *Partitioned Matrix Expression (PME)* [32, 33, 42, 60]. A loop invariant is derived from the PME for each algorithmic variant; it expresses what portion of the final computation (described by the PME) is completed at the beginning and end of each loop iteration. Both of these are properties about each quadrant of the input and output matrices.

For each output matrix, the loop invariant tells us if each quadrant is not updated (unchanged), partially updated (changed, but not holding the final result), or fully updated (holding the final result); each quadrant is exactly one of these. Further, by inspecting the loop-body operations, we can say which quadrants are read. Lastly, for each matrix we know how the matrices are accessed because loop tunnels store the direction in which matrices are partitioned. This information is sufficient to determine if two loops can be fused.

For example with two loops, if a particular quadrant of the first loop's output is not fully updated but the second loop has an operation that reads it, fusion is not legal. Why? Because the second loop's update operations would read non-final results. Also, if the second loop updates a quadrant that the first loop reads, fusion is not legal. Otherwise, the first loop's operation would be computing with results changed by the second loop's instead of the results only obtained by the first loop (as intended).

Knowledge about update status is encoded on architecture-agnostic loops found in refinements, so fusion can be applied at any level of the software stack automatically when those refinements are employed. This is a great example of

knowledge reuse. Previously, loop fusion was performed by different people (or the same person repeatedly) for libraries targeting different architectures, so they manually do it for each level of the stack. Often, this sort of optimization is forgotten or missed because of its complexity. DxTer provides loop fusion for free.

In addition to the above, we also tag loops if their iterations are independent, which is determined by the PME and loop invariant. When they are independent, the iterations can be executed in parallel across threads in a multithreaded system. This is described in detail in Chapter 6.

3.5 Going Lower

In this work, we generate code for the high-level Elemental and BLIS DSLs. In both cases, when an expert develops code, he accepts a certain layer as the lowest to consider. Primitives are internally implemented in some way that he largely ignores. He generally only considers their cost and preconditions and postconditions, the same information used with DxT. With this information, he makes decisions on how to implement higher level functionality. One might ask how a developer decides where to "draw the line" of consideration.

3.5.1 Why Not Go Lower?

Why are the primitives accepted as the lowest layer of interest? Why not break through and go lower? In some cases, there would be minimal benefit to expose lower level details. Low-level operations might have multiple implementations (refinements) internally. They dispatch to the best choice at runtime based, for example, on problem size. The overhead of such runtime decisions is minimal. We expose refinements at higher levels in the hope of optimizing operations between interface

boundaries. For some of the lower level operations, there is no such opportunity, so we do not need to explore and expose their refinements.

In the cases where there is the possibility of cross-interface-boundary optimizations, the performance gain is a much lower-order term. If it were not, after all, an expert would not have been satisfied with the abstraction layers of the DSL. Therefore, the benefit of breaking through is not (currently) worth the effort to encode lower level knowledge.

3.5.2 Problems and Possible Solutions When Breaking Through

Perhaps optimizing some lower-order primitives will be worthwhile in the future. Further, it might simply be useful to generate lower level code to automate more of an experts' work. Below are some reasons why we have not done this (yet) and some ideas for the future.

First, cost functions are less reliable for lower-order terms. For example, the behavior of the cache gets more difficult to predict and more important to consider when dealing with choices of read and write strides in data movement. The CPU is difficult to predict when performing out-of-order execution, prefetching, and so forth. This means that it would likely be necessary to compile and run code to determine the performance (cost) of candidate implementations. This is not possible with the high-level operations with which we currently deal because of long runtimes. With lower-level operations, though, runtimes are shorter so this is viable albeit a slight departure from our current approach. When experts cannot predict cost with analytic estimates, they may also have to run code and time it. It would still be useful for the developer to have a tool like DxTer to direct him to an implementation. It would generate code instead of requiring an expert to do it. Either a tool like

DxTer could be all-in-one (generating the code and testing it) or it could be an aid to the expert in generating code, which an expert would then time and from which he would then learn. The latter case is what happened with multithreaded BLIS (Chapter 6).

Second, the combinatorial search space of implementations gets much larger when breaking through boundaries. A way around this is to employ phased generation (like DRACO [49]) to remove bad designs between phases. DxTer could perform the search just as it does now with existing interface boundaries, then the top n-best implementations would be kept and the rest would be thrown away. Next, DxTer would break through (refine) to lower levels with only those n implementations. Thus, the search space would not be the full combinatorial size. Also, DxTer could partition the entire design graph into disjoint subgraphs (cliques) using some heuristic and only explore optimizations within those subgraphs instead of the fully-connected possibilities.

Third, there are existing solutions to generate code for some of the common DLA primitives, described in Section 1.5. DxTer could identify regions of code that might be ripe for optimization. It could then use a dynamic programming approach, outsourcing the optimization of a graph of operations currently considered primitives to other generative approaches (e.g., Spiral [52]). The returned programs would be optimized code and, hopefully, cost estimates would be included that DxTer would use to search. This is a hybrid approach with cost models for high-level decisions and empirical tests for lower-layer decisions.

It is important to note that there would be many opportunities for reused knowledge if DxTer is ever applied to lower level DLA kernels. The tricks experts use are often transformations on the same algorithms as derived with FLAME. For

example experts perform fission, unrolling, strip mining, and so forth to the `Gemm` algorithms shown above to get the low-level computation kernels used as primitives in Chapter 6.

3.6 Summary

We demonstrated how FLAME algorithms are represented in DxT. Since these algorithms are architecture agnostic, we templatize them on a layer of the hardware stack. A layer is a mapping from a higher-level of abstraction to a lower-level that exposes some implementation detail (be it software or hardware). Layer instantiation parameters could represent, for example, a distributed-memory layer or a shared-memory layer. They could also represent a particular layer of code; for example, a chosen layer instantiation could target an encoded algorithm to a specific nested loop. Thus, we can target encoded domain knowledge to a particular hardware architecture.

Chapter 4

DxTer

We present DxTer [1], a prototype developed to explore some of the ideas behind DxT: encoding design knowledge as transformations, searching a space of implementation options, rank ordering implementations by cost estimates, and generating programs automatically. DxTer was designed to be a prototype for use by a DxT expert, so it is limited in usability features. Still, it exposes many ideas that could be used in a production-quality version.

4.1 Encoding Knowledge

DxTer is an object-oriented, C++ program with OpenMP directives for basic multithreaded parallelism within `for` loops. One starts DxTer with a knowledge base (the set of transformations) and a single graph, representing functionality to be implemented. DxTer generates a search space of implementations and outputs a single "best" piece of code based on a particular problem size[1].

[1] One can imagine a loop around this process to generate many implementations for a range of problem sizes. At program runtime, when the problem size is known, the "best" code for that size would be executed.

Figure 4.1 compares a traditional compiler to DxTer. A traditional compiler has hardware knowledge and optimizations encoded internally (they are not extensible by software engineers). In some cases, domain-specific transformations are also encoded internally. A compiler takes source code is input and outputs an executable. With DxTer, hardware knowledge (architecture-specific transformations and cost functions) and domain-specific transformations are part of the inputs. An algorithm to be implemented, represented as a graph, is input, and DSL code is output. This code is then input to a traditional compiler.

With a system like DxTer, one encodes domain algorithms in a knowledge base, which is reused for each hardware architecture. A developer would also learn about new hardware targets and encode implementation options for each, adding them to the hardware-specific knowledge base. DxTer would be run on each desired algorithm's graph for each target hardware architecture. As described in Chapter 6, DxTer can even be run each time new implementation ideas are developed and encoded to enable a developer to explore software design options more easily and quickly.

In this section, we discuss DxTer's representation of graphs and transformations. In the next section, we discuss the search process.

4.1.1 Nodes and Graphs

DAGs are represented by nodes that reference each other. Nodes have *producer references* to the nodes that provide their inputs; these are stored in an ordered list. Nodes also have *consumer references* to the nodes that use output data. Because nodes can produce more than one output (e.g., split tunnels output multiple submatrices), references include the output number used. The object representing a

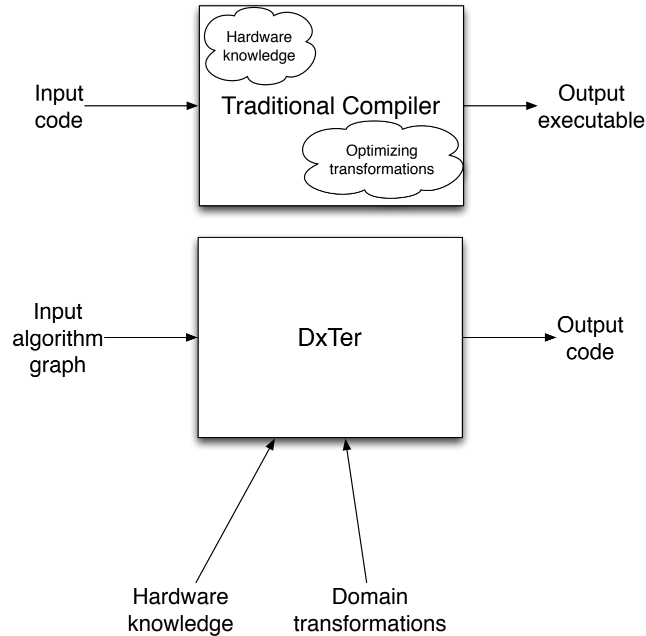

Figure 4.1: High level comparison of a compiler and DxTer.

DAG keeps track of all nodes it contains.

As is standard in MDE, all DxTer graphs must obey a *metamodel*. Loosely, the metamodel specifies (restricts) the structure of legal graphs. The metamodel is domain-specific, but there are some domain-agnostic restrictions in DxTer. The main restrictions are:

- Graphs must be directed acyclic multigraphs. This omits the DxT representation for some domains (e.g., the crash fault-tolerant services of [54]), but DAGs also allow many simplifying assumptions in DxTer's analysis algorithms. This is standard in the intermediate representation of compilers: DAGs simplify analysis algorithms. For example, it is easier to write a graph traversal algorithm without having to deal with cycles. Loops are represented using the structure described in Section 2.5.1

- All DAGs are *weakly connected* such that if all edges are changed to be undirected all nodes can be reached from all others.

- *Output nodes* represent the output of a graph. For each DAG, they are specified in an *output node list*. If all edges in the DAG are reversed, DxTer can reach all nodes from at least one output node. This is useful for some of the analyses of DxTer. This requirement also leads to easier identification of useless computation. When the output from a node `B` is not used, then either `B` is doing useless work (since no node is using its results) or `B`'s output is an output of the function being generated. In the latter case, the node should be added to the output node list. In the former case, `B` should not be on the graph. During execution, DxTer throws errors about `B` since it should be removed from the graph or added to the output node list.

4.1.2 Node and Edge Properties

Nodes are instances of a subclass of ***`Node`***. `Node` includes the data structures for consumer and producer references. A subclass of `Node` is ***`DLANode`***, of which all DLA-specific nodes are a subclass. Nodes are queried for properties of the nodes themselves or of the outgoing edges. For DLA, output matrix sizes, data distribution (explained in Chapter 5), and variables names are common properties of output edges while cost is a property of the node. These properties are queried via virtual methods on `DLANode`, which each subclass implements. Nodes query input properties and either "pass the value through" to the output properties or compute some function of the inputs' properties.

DxTer uses the *cost* property of primitives to determine if one DAG encodes a better implementation than another. The cost function used for DLA (for now) is

an estimate of runtime based on how much computation is performed, data is communicated, and so forth. The following chapters describe the details. Each primitive calculates its own runtime cost based on the size of inputs and other properties like data distribution. The cost of an entire DAG is calculated by summing the costs of each primitive. Summing is sufficient for DLA. For other domains with a more complicated cost calculation, DxTer would be extended.

Liveness analysis and similar compiler-type analyses can be performed via other properties. `Node` has a property to query whether or not an input (which in code is input via a variable) is overwritten and output or just read by the node. If the node does overwrite the input variable, then the node keeps it live. This blurs the line between dataflow graphs (which have no notion of a node overwriting the input) and practicality where a node can represent a function call that overwrites input data. Section 4.1.5 discusses this in greater detail.

4.1.3 DAG Restrictions and Checking

In Section 2.5.3, we described how type checking is used to provide additional trust in the graphs DxTer produces. `Node` has a virtual function called *`Prop`* (propagate). `Prop` is called on all nodes after a transformation is applied to propagate properties and check nodes for correctness. Any `Node` subclass can override `Prop` to add class-specific checks as long as it also calls its parent class's `Prop`. `Node`'s `Prop` method checks, for example, that each node's producers and consumers uphold their bidirectional connection, a requirement on all domains' DAGs. Further, each node checks that the DAG instance holding it knows about the node. These are basic checks to ensure the DAG is constructed well.

`Prop` is an opportunity for each node to check operation-specific features. For example, `Gemm` nodes check that there are exactly three input matrices and that consumer nodes only read the 0th output[2] since `Gemm` only has one output. Further, it checks that input matrix sizes conform (e.g., the inner dimensions of input A and B match). For Elemental, there also checks on data distribution and for BLIS there are checks on sizes meeting architecture-specific criteria. Checks like these are useful to discover if a transformation has errors like miswiring operation inputs/outputs.

A lot of these checks are similar to those performed at runtime in Elemental or BLIS. Granted, they are very low-order terms, but if Elemental or BLIS code were only generated by a DxTer-like system that guarantees such properties, the runtime checks could be removed or disabled.

`Prop` is also used by some nodes to propagate and cache properties like cost or output sizes. These can be non-trivial computations that are queried often during search, so caching the information amortizes the computation cost across many queries.

DxTer does not currently keep track of type information like symmetry or matrix structure. Such information could be useful in the future to ensure, for example, that the triangular matrix input to a `Trmm` operation is indeed triangular or to allow for more math-level transformations/optimizations. For example, a novice user might employ `Gemm` for multiplying a triangular matrix and DxTer could transform it to the (better performing) `Trmm` operation, but one must add type information to the inputs. Such transformations are similar to those performed by Fabregat-Traver and Bientinesi's compiler [24]. These properties are not necessary for our current work.

[2]We use 0-based indexing.

4.1.4 Transformations

When developing transformations, it is helpful to think about them in a pictorial way as in Figure 4.2. This transformation applies when a triangular matrix is inverted (via `TriInv`) and the result is input to `Trmm`. It replaces that with a `Trsm` and triangular matrix inversion. This can improve numerical stability because `Trsm` is more stable in some cases than using `Trmm` on an explicitly inverted matrix. When implementing this transformation in DxTer, the code is not a picture, it is C++. That code searches for a graph pattern (that of the LHS) and replaces that pattern by forming the RHS graph and patching it into the graph.

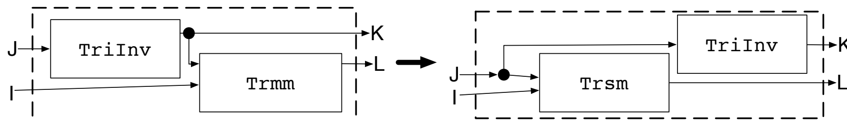

Figure 4.2: A sample transformation possibly to improve numerical stability.

Transformations are implemented in DxTer as a subclass of `Transformation`. One instance of each transformation is added to DxTer's knowledge base at the beginning of execution. Transformations can be templatized, so instantiations are created at that time with the desired parameters.

`Transformation` has three virtual functions. One returns the name of the transformation (e.g., "Trmm on Inverted Triangle to Trsm with Inversion"). When code is output from DxTer, the list of transformations is also output to explain how the implementation was derived, so meaningful names are useful. Comparing two implementations' transformation lists allows one to know how design decisions differed [10].

`Transformation` also has virtual functions `CanApply` and `Apply`, both of which are passed a `Node` pointer to a node on a DAG, which we call `box`. When comparing

```
DLANode *producer = (DLANode*)(box->Producer(0));
if (producer->GetNodeClass() == TriInv::GetClass() &&
    producer->GetLayer() == m_layer &&
    box->GetLayer() == m_layer)
{
  return true;
}
else
  return false;
```

Figure 4.3: Sample DxTer code of `CanApply` for the transformation of Figure 4.2.

the LHS of a transformation to a graph (to find if it applies), it is useful to consider one node in the LHS a root for comparison. One finds a `box` node of the same type as the root in the LHS and then compares the subgraph around `box` on the graph to the subgraph around the node in the LHS. Generally, the choice of which node to consider the `box` does not matter much and can be made arbitrarily, so programming transformations is made easier.

Each transformation has a particular `box` type to which it can apply (i.e., the type of the matching node on the LHS). DxTer maintains a lookup table mapping node types to transformations that can apply to improve scalability (performance). One generally adds some transformations for one node type, some for another, and so forth and does not add many transformations that only apply to one node type (i.e., transformations are usually distributed across node types). As transformations are added to DxTer, the number that can apply to any one node type grows significantly slower than the total number added because of this distribution. When determining which transformations apply to a particular node, the list of transformations that can apply to that node type is retrieved. Each transformation's `CanApply` is called with `box` pointing to the node on the graph. Without such a list, all transformations in the knowledge base would be tested and almost none would apply.

The `CanApply` function compares the surrounding subgraph of `box` to the

transformation's LHS. Figure 4.3 demonstrates this for Figure 4.2. This transformation applies when `box` is a `Trmm` node. First, the first producer of `box` (i.e., the node that provides the triangular matrix input) is retrieved. Then, the producer's type is checked since the transformation only applies if the type is `TriInv`. Lastly, the producer's and `box`'s layers are checked. The transformation is templatized on the layer to which it is applied, so it only applies when both the boxes are part of that layer (`m_layer`). If all of these conditions are met, `CanApply` returns `true` and returns `false` otherwise.

As this is C++ code, transformations can be more complicated than this, possibly with more variability in the LHS. Generally, though, transformations are roughly as complicated as this example (possibly dealing with three times as many nodes in the worst case). It is beneficial to keep transformations simple so that one can reason better about their correctness.

When a transformation is applicable, the graph is copied and the transformation's `Apply` function is called on the target `box` of the copy. This is discussed in greater detail in Section 4.2. Figure 4.4 demonstrates the step-by-step application of the transformation. The starting subgraph is the LHS and the ending subgraph is the RHS.

Figure 4.5 shows code for the `Apply` function with comments to match the panes of Figure 4.4[3]. First, the `Trmm` and `TriInv` nodes are identified. Then, a new `Trsm` node is created. The `Trsm` and `Trmm` nodes should have the same properties (e.g., side, data type, and so forth), so the new node is created with those properties. `trsm` is added to the graph's node list and is given the appropriate inputs (both the producer node and its output number are specified). The consumers of `trmm` are redirected to `trsm`. Finally, `trmm` is removed from the graph. If `trmm` had producer

[3]This code is slightly simplified, but is similar to what is found in DxTer for this transformation.

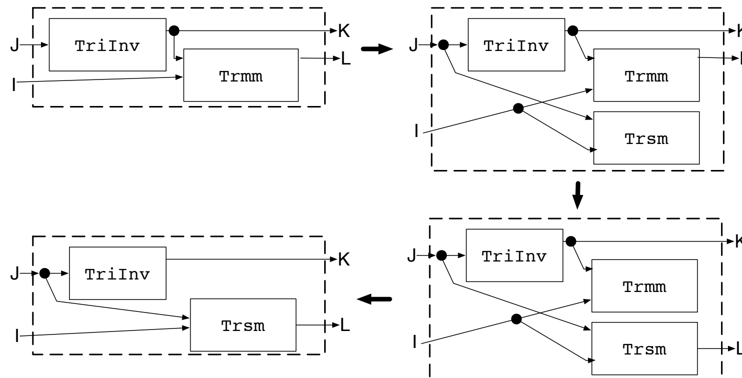

Figure 4.4: The step-by-step application of the transformation of Figure 4.2 in DxTer.

```
//Start at first pane
Trmm *trmm = (Trmm*)node;
TriInv *triInv = (TriInv*)(trmm->Producer(0))

//Create new Trsm and wire inputs as on second pane
Trsm *trsm = new Trsm(m_layer, trmm->m_side, trmm->m_tri, trmm->m_diag,
                      trmm->m_trans, trmm->m_coeff, trmm->m_type);
trmm->m_graph->AddNode(trsm);
trsm->AddProducer(triInv->Producer(0), triInv->ProducerConnNum(0));
trsm->AddProducer(trmm->Producer(1), trmm->ProducerConnNum(1));

//Rewire consumers as on third pane
trmm->RedirectConsumers(trsm);

//Clean up to end with RHS
trmm->m_graph->DeleteConsumerAndCleanUp(trmm);
```

Figure 4.5: Sample DxTer code of `Apply` for the transformation of Figure 4.2.

nodes that were only producing outputs used by `trmm`, those producer nodes should be removed too (otherwise they would be wasteful computation and would violate the final assumption listed in Section 4.1.1). This removal process is continued recursively upward.

4.1.5 Output Code

When a node is able to print, its `PrintCode` method is called. `PrintCode` is a purely virtual function defined on `Node`. `PrintCode` is usually specialized to the node's

layer tag since a `Gemm` primitive has different implementation code in BLIS than in Elemental. With `PrintCode`, producing output code for functional / dataflow graphs should be easy. In practice, producing output code is not so simple.

While we treat the graphs throughout this dissertation as functional representations, for DLA (and other domains) nodes represent functions with side effects. For example in a true dataflow graph, a `Gemm` node accepts three inputs, A, B, and C, all of which are unmodified by the node, and the output $\alpha AB + \beta C$ is assumed to resided in a new variable. In the actual output code of a `Gemm` node, though, one does not want to incur the memory overhead of allocating new space for C as the C variable can simply be updated with the result of `Gemm` (i.e., $C := \alpha AB + \beta C$). This is standard in DLA. As a result, DxTer must be careful in how it outputs code.

Figure 4.6 is a contrived example. A is input to both nodes and overwritten by $Op1$. The correct code needs to call $Op2$ first so its A input is unchanged, and then $Op1$ can execute. To ensure this, node types "know" which input(s) they overwrite. While printing a graph to code, DxTer keeps track of which nodes have printed. With a functional graph, when all producers for a node x have printed, x can print. As the nodes here are not strictly functional, DxTer must first check that all nodes using x's overwritten input(s) have already printed. After they have, x can print since it will not overwrite a variable another node must still read. If there is more than one consumer of x that overwrites it, DxTer throws an error since the graph (and the code it represents) is illegal[4].

Graph input values are represented by the `InputNode` class[5]. `InputNode` instances have one output, which is given a variable name (output property). Other

[4] This has happened when a transformation has a bug, so this is another form of DxTer aiding in correctness checking.

[5] `InputNode` instances are assigned input properties like problem size that are propagated through the rest of the graph.

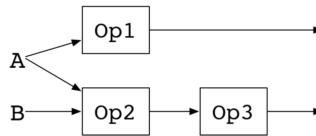

Figure 4.6: A graph that cannot be treated in a standard dataflow way when printing.

nodes provide a variable name for their outputs, too, which is usually the name of their overwritten variables. In some cases, the output variable names are a function of the input variable names (e.g., `inputTempVal` or `A00` for a temporary variable node or split, respectively).

Thus, variable names are propagated from `InputNode` nodes through the graph. For the graph of Figure 4.6, `Op1` and `Op2` propagate the output variable names `A` and `B`, respectively. The (correct) output code is `B:=Op3(Op2(A,B)); A:=Op1(A)`.

This sort of analysis and variable propagation or reuse is similar in goal to what traditional compilers do when allocating registers to produce code from *single static assignment (SSA)* form. Here, we benefit from only having a single way to allocate / reuse variables due to restrictions such as only having one consumer of an output override the variable.

4.1.6 Explaining Differences

A goal of this work is to generate the same or better code than a developer. When DxTer produces different code, there are three explanations. First, there could be a bug in DxTer or the knowledge base, which we discuss in Section 2.5.3.

Second, there may be a hole in the knowledge base. The developer might have used some optimization, refinement, or improved cost estimate or performance intuition that is not encoded in DxTer. In this case, the difference(s) is usually identified easily by comparing output code and hand-developed code, and the new

transformation or cost estimate is added to DxTer. This can be useful to make explicit some formerly hidden trick the developer used to write better code[6].

Lastly, the developer could have made a mistake. DxTer's implementation space should include all of the implementations a developer would produce (assuming all of his knowledge is encoded). One can compare the hand-developed code to all implementations in the search space. If the developer wrote incorrect code, leading to incorrect results, his implementation would not be in the search space as it only contains correct implementations. Experts' mistakes often come down to indexing bugs or similarly small errors. Those are usually easy to spot by comparing correct and incorrect code.

Otherwise, he might have developed a suboptimal implementation by choosing a bad refinement or by missing optimizations. In this case, the suboptimal implementation would be found in DxTer's search space[7]. DxTer keeps track of the transformations applied to derive all implementations, so one can compare the transformation list of the suboptimal and "best" implementation. The deviation in those lists explains which of the developer's choices were suboptimal.

Whenever DxTer produces different code than the developer's, there is something to learn. Either the developer's mistakes are brought to light or new implementation knowledge is identified and encoded (which has pedagogical value). Both cases demonstrate some utility in using DxT.

[6] We imagine differences could be automatically or semi-automatically (with expert intervention and guidance) discovered when reverse engineering code.

[7] In Section 5.4, we describe how the search space is limited by omitting clearly-bad implementations. In this case, the developer's bad implementation might not be in DxTer's search space, but an expert can quickly convince himself that DxTer's code is much better than his.

4.2 Search

With an understanding of how nodes and transformations are represented, we now explain DxTer's search process. First, we present the basics and then discuss optimizations to reduce the size of the search space to improve search time. These optimizations were necessary to make the search tractable for complicated algorithms with many implementations.

4.2.1 Basic Search

DxTer starts with a single DAG, called the *seed* of the search space. DxTer iteratively generates new graphs implementing the seed's functionality in different ways. These graphs are stored in a set called *ImplSet* (implementation set). ImplSet contains only unique graphs (detailed below).

The simplistic view of DxTer's search is that in each iteration, every node of every graph in ImplSet is tested to see if any transformation in the knowledge base can apply. When a transformation can apply, the graph to which it applies is duplicated, and the transformation is applied to the duplicate. The new graph is compared to each existing graph in ImplSet. If it is different than all existing graphs, `Prop` is called on the new graph so nodes update their properties such as cached output sizes and costs as well as to check for legal graph construction. The graph is then added to ImplSet. DxTer iterates until no new graphs are added, meaning all transformations that can apply have been applied. At this point, DxTer attempts to fuse any pairs of loops it can on all graphs. If new graphs are added, DxTer iterates again, checking only new graphs for transformations that apply. This continues (transform, fuse, transform, fuse, and so forth) until there are no new graphs generated. At that point, all graphs' costs are summed and the lowest-

cost graph is converted to code. This explanation of DxTer's search is sufficient to understand its program generation, but the actual implementation of search is more optimized.

First, any time a transformation is applied, the `box` node on the original graph (i.e., the node to which the transformation applies) is tagged with the transformation. Then, that node is not checked again for that transformation's applicability. The transformation was already applied and the resulting graph is in ImplSet, so there is no reason to check again. When graphs are duplicated, the nodes' transformation list is cleared so previously applied transformations can apply to the node's duplicated version. In some cases, a transformation applies multiple times to the same node (or, really, the node's duplicated version). Each time, the graph is duplicated, the transformation is applied to the surrounding subgraph, and then the transformation can be applied again, so the process is repeated.

For example, consider an optimization to remove redundant sort operations. If a graph has three sort operations (e.g., $sort(sort(sort(x)))$), the optimization is applied once. The new graph has one less sort (i.e., $sort(sort(x))$). The transformation can be applied again to the new graph, so the node's transformation list should not prohibit that. The original node (on the graph with three sorts), though, does not need the transformation applied again because the result (i.e., $sort(sort(x))$) is already in ImplSet. Similarly, DxTer keeps track of which loops are fused so it does not regenerate existing graphs.

Graph comparison is optimized by keeping a hash for each graph. The hash is computed from a string that lists all of the nodes' types in a format similar to $node1(node2(x), node3(y, node4(z)))$. This hash is computed once and reused for each comparison of a graph against a potential addition to ImplSet. When hashes

collide, a more detailed comparison is performed. DxTer starts with the `InputNode` nodes and compares the two graph's node types along the edges until it reaches the output nodes, which have no outgoing edges. If it does not find a deviation between the two graphs up to that point, the graphs are the same.

When new graphs are created in an iteration of the search, they are not immediately added to ImplSet. Instead, they are added to a temporary set. At the end of a search iteration, the graphs in the temporary set are merged into ImplSet (with checks for duplication). Using this approach, the graphs in ImplSet are analyzed in parallel for applicable transformations without locking since ImplSet is read-only until the current iteration concludes.

After a graph has been checked for applicable transformations, the graph is marked as "done." After that, it is not re-evaluated for applicable transformations.

Fusion is only applied to two loops that have an input in common or when one loop uses the output produced by another loop. If these relationships do not hold, there is no opportunity for optimization (with our current optimizations), and the resulting fused loop might actually perform worse since additional computation in the loop could evict data from cache.

4.2.2 Phases and Culling

When manually coding, one does not necessarily choose how to implement (refine) interfaces first and then optimize code, but it is beneficial to separate these steps in DxTer via *phases*, which stage when certain transformations are applied. Consider a graph with three interfaces (`A`, `B`, and `C`). If each has 4 refinements that expose only primitives, there are $4^3 = 64$ implementations of the graph without interfaces. Then, many optimizations could apply.

With DxTer's search, there would be $4^2 = 16$ graphs with the `A` interfaces and all combinations of `B` and `C` refinements. The optimizations that apply to these refined implementations would create even more graphs. Similarly, there would be graphs with `B` and `C` interfaces. We know, though, that all graphs with interface `A` do not represent valid code since an interface does not generate code. Each would be transformed by `A` refinements at some point, but that would just create graphs that already exist – there are 64 graphs that are already fully refined and more graphs with the optimizations already applied to those 64 graphs. In category theory, alternate paths to the same points in a space (i.e., the same graph) are called a *commuting diagrams*. While generating the search space, this happens often when applying refinements and optimizations.

With phases, DxTer decreases the size of the search space and limits the work done to create duplicate graphs by ordering refinements and optimizations. Between phases, graphs with interfaces that should have been refined are *culled*, or removed from ImplSet. After a graph with an interface is refined, it is no longer useful because it has already yielded any implementations that can be created. Therefore, DxTer removes such graphs from its search space between phases because they only lead to additional work; any optimizations applied to them lead to graphs that already exist, creating commuting diagrams with useless graphs as midpoints.

Culling is based on the layer annotation of nodes. A phase is linked to a layer in the software, so refinements from that layer should be performed within that phase (and a requirement of refinement layers is that they are monotonically increasing). Any interface for that layer is redundant at the end of the phase.

In practice, introducing multiple phases of optimization is not generally useful for culling because we cannot say that some of the graphs are no longer useful

because of how locally-bad optimizations can be chained together to generate a globally-good implementation. In practice, most optimizations apply to primitives, so they can all be applied together (in the same phase). Some optimizations apply to interfaces, in which case they must be mixed with the initial refinement phases so graphs on which such optimizations apply do not get culled before they are applied.

Optimization phases do, though, provide structure to code generation. For example in Chapter 6, we explain how one adds an extra optimization phase to retarget sequential BLIS code to a multithreaded system. That entire phase can be cleanly omitted or included in the search depending on the target. We imagine in the future that optimization phases could be added to change low-level implementation details with small performance changes. This would increase the size of the search space, so one could omit or include these phases depending on 1) how long he is willing to wait for an implementation and 2) how important the best performance is for an implementation. Phases could offer such customizability to DxTer's search.

When adding transformations to DxTer, one specifies to which phase the transformation belongs. In each phase, DxTer only considers the relevant transformations. Between phases, the list that keeps track of which transformations have been applied to a node is cleared. The tag that marks graphs as fully transformed is also cleared. There are new transformations in the next phase and they can be applied differently, so all graphs and nodes should be re-evaluated. With the exception of simplifiers (described below), we do not have an example of a transformation that is used in more than one phase.

Unlike compiler phases [6, 37], the phases of DxTer are easily determined and ordered. For example, with Elemental code generation described in Chapter 5, the first phase refines (choosing an algorithm and parallelization scheme) and the

second phase optimizes. With the BLIS work of Chapter 6, each layer of the code (i.e., choice of nested algorithmic variant) is linked to a search phase. The refinement phases are followed by an optimization phase, which is followed by a parallelization phase (if a multithreaded architecture is targeted). There is little to no ambiguity in which transformations belong to which phase because the DLA DSLs or code structure leads to a natural ordering. This may not be the case with other domains, but it has been true so far.

4.2.3 Saving the Search Space

Generating a large search space takes time. We see DxT used in the future to aid a developer in generating a library of code, including many complicated functions. If that developer is still exploring implementation options, it would be inefficient to re-generate all implementations with largely the same knowledge base each time a new transformation is added. Instead, it would be useful to save the search space generated with a particular knowledge base. Then, when a new transformation is added, that search space would be loaded and the new transformations would be applied to all implementations to generate only the new graphs.

When saving a DxTer search space, graphs are *flattened*, or stored to disk, with all objects' pointers. Nodes are flattened with the `Node` pointers for consumers and producers along with all type-specific information (e.g., coefficients, upper/lower triangular, and so forth). When loading, new objects are created to match the saved objects and a map is kept from old pointers to new pointers. The old pointers are replaced by new pointers on the loaded graph by querying the map.

For various sample algorithms, loading a search space took 5-10% of the time it took to generate the search space. The size of each search space is not

astronomical as in some similar work (such as SPIRAL [52]). We give example sizes in Chapter 5. We believe the key is that we are automating what a person would do manually. When doing this by hand, it must be feasible to search the space effectively (i.e., doing it sufficiently to get good performance). Experts have worked hard to develop interfaces that enable a separation of concerns, so one focuses only on the important decisions at each level of the stack and does not have to deal with all details concurrently. Considering all details concurrently, even those relating to low-order performance costs, makes the search space very large and difficult to search manually.

4.2.4 Transformation Meta-Optimization

Just as DxTer must optimize code for particular hardware, one must optimize transformations for DxTer. For now, this is a manual effort, but automating it will be an interesting area of future work. Below are some ways we optimize the knowledge base to improve performance significantly.

Some transformations are always worth applying. For example, there is no reason to explore implementations with redundant communication. For an optimization that removes redundant communication, DxTer should not copy the graph and apply the optimization to the copy (thus leaving the unoptimized graph in ImplSet). Instead, the optimization should be applied directly – it is always worth applying. Such optimizations are called *simplifiers* [46] since they often simplify graphs (removing extra nodes) and simplify the search space by removing subspaces of inefficient graphs. Simplifiers are tagged as such when adding them to DxTer. After DxTer duplicates a graph and applies a standard transformation, it applies simplifiers anywhere it can. You could think of simplifiers as a micro-phase that

"cleans up" and simplifies graphs.

It is often useful to produce code in terms of small transformations. That way each minor change can be taught, understood, and/or proven correct. DxTer keeps graphs with each transformation applied and not applied in ImplSet. The combinatorial search space that results from many small transformations (instead of fewer large transformations) becomes massive.

Further, some transformations increase implementation cost (lead to an always-worse graph) but expose details that allow for a subsequent optimization (a better graph in the end). We show examples of this in Section 5.2.3

To optimize the knowledge base, we form *merged transformations* that apply multiple transformations as one. By merging transformations, DxTer does not explore suboptimal implementations and instead only explores graphs that come from the combination of transformations. We merge small transformations if they are not useful to explore independently. We merge transformations that increase cost with the optimizations that are subsequently expected to decrease cost. Merged transformations can significantly improve the size of the search space (and time for its generation). We give a detailed example in Section 5.2.3.

Future Metaoptimization

Simplifiers and merged transformations are examples of metaoptimizations to the knowledge base. For now, these are created by a person who knows both the knowledge base and DxTer well – an expert. To some extent, they require a "feel" for the search process in addition to some tuning effort to get the metaoptimization right.

In the future, we want such metaoptimization to be automated. The knowledge base would be optimized by a system, which recognizes when rules are never

useful in abstentia, for example. Then, the system would merge them. The system could also label rules as simplifiers and would omit unnecessary rules.

While the viability of this goal is an open research question, it would be useful to the end-user of a DxTer-like system. He would think only about his domain and software and would not consider the search process. He would encode his knowledge as transformations and a system would optimize those to form the working knowledge base used in a high-performance search.

In Section 5.4, we discuss how we further limit the search space by reducing the number of refinements that are explored.

4.3 Summary

DxTer is a prototype into which domain and hardware knowledge is input to enable it to also take an input graph representing desired functionality and output an architecture-specific, optimized implementation.

Knowledge is encoded as refinements and optimizations (which form a knowledge base), which DxTer uses to transform the input graph into a search space of implementations. DxTer uses cost estimates to rank order these and output the best performing.

Further, knowledge about domain operations (both hardware specific and agnostic) is encoded as part of node specifications. A specification is encoded in a C++ class that includes requirements on the number of inputs and outputs, datatypes, data sizes, and so forth as well as cost estimates for code performance (for primitives).

Chapter 5

Elemental

We present how we encode knowledge in DxTer to generate code for Elemental [2, 50]. We do so for all BLAS3 operations and a subset of Elemental's LAPACK-level operations. We start with the BLAS3 and then reuse the encoded knowledge for other functions since the BLAS3 are a basis for higher level functionality[1].

5.1 Elemental

We now discuss the basics of Elemental and explain how an Elemental expert (Jack Poulson) can manually develop an algorithm for a BLAS3 operation optimized for *distributed memory architectures (clusters)*. We explain code in terms of transformations because that was the mechanism by which we reverse-engineered Jack's code to identify expert design knowledge. While the expert did not necessarily view his task with transformations in mind, the resulting code can be forward-engineered by transformations. Further, these transformations are reusable, understandable, and independent pieces of DLA knowledge.

[1] This chapter is based on material in [46, 47, 48].

Algorithm: $[B] := \text{TRMM_RLN_BLK}(L, B)$

Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c|c} B_L & B_R \end{array}\right)$ **where** L_{TL} is 0×0, B_L is $n \times 0$

while $m(L_{TL}) < m(L)$ **do**

Repartition

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} B_L & B_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

where L_{11} is $b \times b$, B_1 has b columns

$B_0 := B_0 + B_1 L_{10}$ (`Gemm`)
$B_1 := B_1 L_{11}$ (`Trmm`)

Continue with

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c|c} B_L & B_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_0 & B_1 & B_2 \end{array}\right)$$

endwhile

Figure 5.1: Variant of `Trmm` to compute $B := BL$ (right, lower triangular, non-transposed, or `Trmm RLN`).

We use our running example, the algorithm in Figure 5.1 (duplicated from Figure 1.1 for convenience) for `Trmm` as it is prototypical for how we encode knowledge to generate parallel code. It is also prototypical of how all BLAS3 can be implemented by casting most computation in terms of `Gemm` [38]. The primary concern is to get maximal parallelism from $B_0 := B_0 + B_1 L_{10}$ while a secondary concern is to parallelize $B_1 := B_1 L_{11}$ (see Figure 5.1) and to minimize necessary communication.

It is well known that hiding all parallelism within the separate update statements can introduce redundant communication and/or synchronization. This means that one cannot simply implement each of the update statements (or call implementations) in locally-best ways. Instead, the collection of update statements must be implemented and optimized in concert. We want to chose and expose implementa-

tion details of each statement to allow for optimization of the whole loop body.

5.1.1 Elemental Basics

Elemental is a library of DLA operations as well as a framework for parallelizing DLA algorithms, which we use as a DSL. Elemental is built on the *Message-Passing Interface (MPI)* [58], where p cluster processes are viewed as a two-dimensional grid, $p = r \times c$. For the default distribution of data (matrices), Elemental uses a 2D element-wise cyclic distribution, labeled $[M_C, M_R]$ where M_C and M_R represent partitions of the index space that provide a filter to determine which row and column indices are assigned to a given process[2]. There are a handful of other one and two-dimensional distributions of matrices, examples listed in Figure 5.2, that are used to redistribute data so that efficient local computation can be utilized.

Elemental is written in C++ and encodes matrices and attributes (including distribution) in objects. In order to parallelize a computation, matrices are redistributed from the default distribution to another to enable local computation to be performed independently by all processes, after which the result is placed back into the original distribution (possibly with a reduction operation such as sum). In Elemental, redistribution is accomplished using the overloaded "=" operation in C++, which hides the (MPI) collective communication required to perform data redistribution efficiently. This makes the Elemental software engineer more productive because he need not concern himself with low-level details of MPI function calls and data rearrangement for every redistribution. For our purposes, Elemental is a DSL for the output of DxTer. Local computation is implemented by linking Elemental code to a sequential BLAS or LAPACK library.

[2]We do not further explain the reason for different distribution names because they are out of the scope of this dissertation. Details are in [50, 56].

Distribution	Location of data in matrix
$[*,*]$	All processes store all elements
$[M_C, M_R]$	Process $(i\%r, j\%c)$ stores element (i,j)
$[M_C,*]$	Row i of data stored redundantly on process row i%r
$[M_R,*]$	Row i of data stored redundantly on process column i%c
$[*,M_C]$	Column i of data stored redundantly on process row i%r
$[*,M_R]$	Column i of data stored redundantly on process column i%c
$[V_C,*]$	Rows wrapped around proc. grid in column-major order
$[V_R,*]$	Rows wrapped around proc. grid in row-major order
$[*,V_C]$	Columns wrapped around proc. grid in column-major order
$[*,V_R]$	Columns wrapped around proc. grid in row-major order

Figure 5.2: Distributions on a $p = r \times c$ process grid for parallelizing DLA algorithms.

5.1.2 Parallelizing `Trmm`

We now examine the actions of an Elemental expert to develop an optimized parallel algorithm for `Trmm`. We do so in terms of transformations, first explaining the refinements that parallelize suboperations and then optimizations that are subsequently applied.

`Trmm` could be any of the following operations: $B = LB, B = L^T B, B = UB, = U^T B, B = BL, B = BL^T, B = BU$, and $B = BU^T$, where L and U are lower and upper triangular matrices, respectively. Each of these eight possibilities is implemented separately with different algorithms. Here, we focus on $B = BL$ for which Figure 5.1 gives one of several algorithmic variants an expert considers. The inputs L and B have the default $[M_C, M_R]$ distribution. The updates `Trmm` and `Gemm` in Figure 5.1 are parallelized by redistributing submatrices, performing local computation (via calls to sequential BLAS3 routines) on each process, and (if necessary) reducing and/or communicating the result.

An expert considers the various ways to parallelize each suboperation. The three parallelization schemes for the `Gemm` update statement keep the `A`, `B`, or `C` matrix *stationary*, avoiding costly redistribution from $[M_C, M_R]$. These are shown in

Figure 5.3. The best choice generally keeps the largest matrix stationary as communication is expensive, so movement of the largest matrix is avoided (we discuss this further in Section 5.4). `D*` boxes (i.e., those that start with `D`) are to be parallelized for clusters. `L*` boxes are primitives that represent local, sequential computation on each process (with no collective communication hidden internally). These names, explained below, specify the boxes' layers. The primitives with $\rightarrow$ redistribute data from the LHS distribution to the RHS distribution. The `SumScatter` box is a form of Elemental redistribution that performs a `ReduceScatter` collective operation on the first operand and stores the result in the second operand [2]. `TEMP` boxes create a temporary storage matrix with the specified distribution. The input matrix provides `TEMP` with problem size information, but its data is not changed.

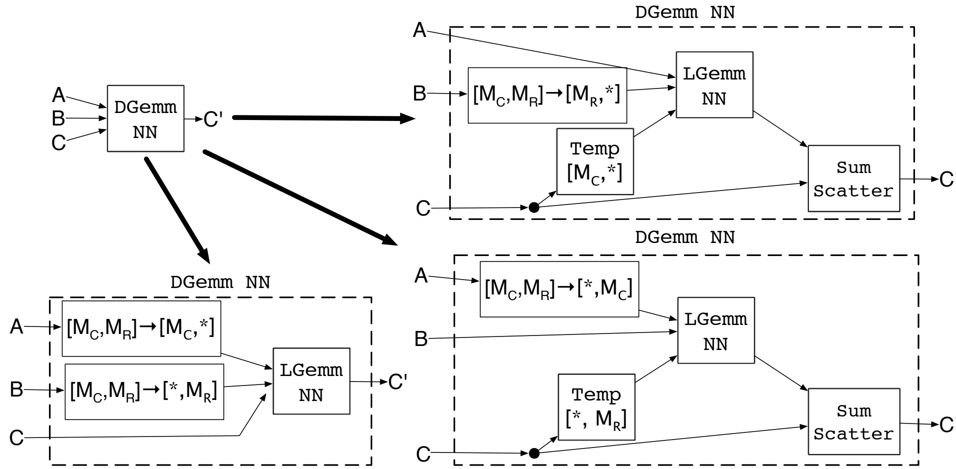

Figure 5.3: `Gemm` refinements.

In this case, B_0 (defined in Figure 5.1) is largest, so the stationary `C` refinement (bottom-left in Figure 5.3) is best. To parallelize `DGemm` with stationary B_0, we redistribute L_{10} (to $[*, M_R]$) and B_1 (to $[M_C, *]$), after which a local `LGemm` is performed in parallel on all processes, calculating disjoint portions of B_0.

To parallelize $B_1 := B_1 L_{11}$, an expert understands that if L_{11} is duplicated

to all processes (distribution $[*, *]$) and B_1 is redistributed so that any one process owns complete rows of this matrix (e.g., distribution $[V_C, *]$), then $B_1 L_{11}$ can be computed in parallel by calling a sequential `LTrmm` on each process with local data. But the expert would also consider many other distributions for B_1, given in Figure 5.2, before arriving at this particular choice. Each possible refinement distributes computation differently, requiring different communication and different local computation, offering a balance between communication (overhead) and parallelism in computation. One generally chooses less parallelism for the small amount of computation, which incurs less communication. Figure 5.4 shows a templatized refinement for `DTrmm`, both the left and right-hand side flavors. For large problems, one refinement may be best because the cost of communication (which enables parallelism) is amortized over more computation.

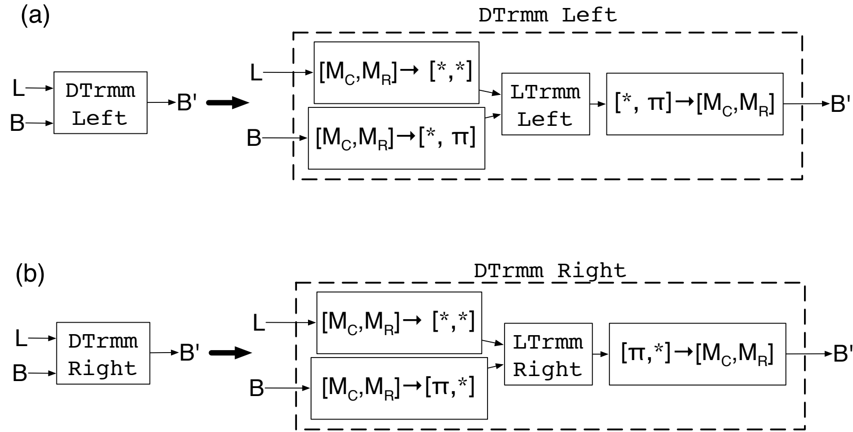

Figure 5.4: `DTrmm` refinements. Π is a templatization parameter limited to $*, M_C, M_R, V_C, V_R$.

We focus on a large problem size for B and L here, but an expert would serve the user best by providing a set of optimized `Trmm` implementation variants for a range of problem sizes. DxTer, therefore, optimizes for various problem sizes, as explained in Section 5.2.7, choosing different instantiations of the `DTrmm` refine-

ment. Here, we use the refinement with a $[V_C, *]$ distribution of B_1 in subsequent discussions.

5.1.3 Encoding the Algorithm with Elemental

Elemental variable declarations[3] and loop code are straight-forward and uninteresting, so we do not show it here. The Elemental code for parallelized update statements (using the refinement choices above) is given in Figure 5.5, with the graphical form shown in Figure 5.6. This is close to the code found in the Elemental library, but requires additional optimizations, explained below, that explore alternate ways to redistribute data. Figure 5.6 shows the subgraph to be optimized highlighted in red.

```
B1_MC_STAR = B1;
L10_STAR_MR = L10;
LocalGemm( NORMAL, NORMAL, 1.0, B1_MC_STAR,
           L10_STAR_MR, 1.0, B0 );
L11_STAR_STAR = L11;
B1_VC_STAR = B1;
LocalTrmm( RIGHT, LOWER, NORMAL, NON_UNIT, 1.0,
           L11_STAR_STAR, B1_VC_STAR );
B1 = B1_VC_STAR;
```

Figure 5.5: Parallelized code for Figure 5.1.

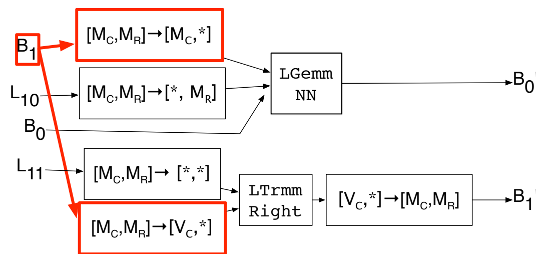

Figure 5.6: Refined loop body for Figure 5.1 that matches code of Figure 5.5. An inefficiency is highlighted by thick red boxes

[3]By Elemental convention, variables are named by the submatrix stored, appended with the distribution name for readability except for the default distribution $[M_C, M_R]$.

Notice how matrix B_1 is redistributed from $[M_C, M_R]$ to $[M_C, *]$, denoted $[M_C, M_R] \rightarrow [M_C, *]$, and then as $[M_C, M_R] \rightarrow [V_C, *]$. The $[M_C, M_R] \rightarrow [V_C, *]$ redistribution can be implemented with an `AllToAll` collective or it can be implemented in terms of the two redistributions, $[M_C, M_R] \rightarrow [M_C, *] \rightarrow [V_C, *]$, which is an `AllGather` followed by a memory copy. This alternative implementation option is encoded in the optimization of Figure 5.7 (a).

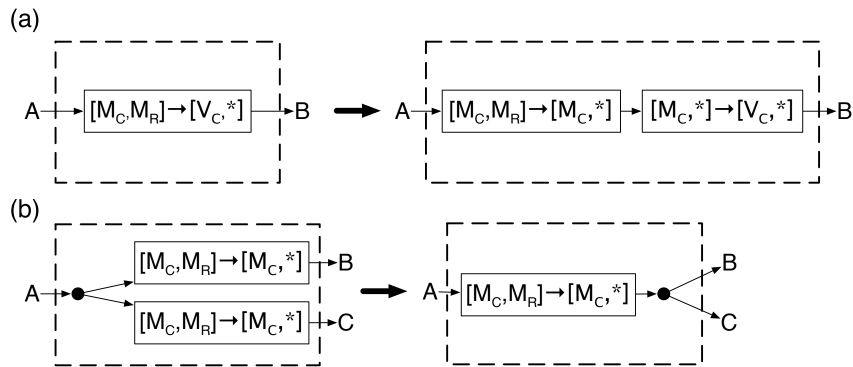

Figure 5.7: Optimization to change collective communication used to perform redistribution $[M_C, M_R] \rightarrow [V_C, *]$ and remove redundant communication.

Figure 5.8 (a) shows the highlighted, inefficient subgraph of Figure 5.6. Figure 5.8 (b) shows the subgraph after applying the optimization to implement the redistribution in a different way. Here, the highlighted subgraph has the data of B_1 redistributed in the same way twice.

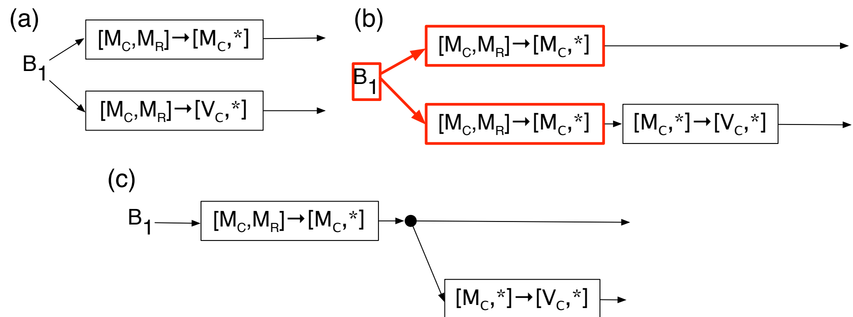

Figure 5.8: Step-by-step optimization of highlighted subgraph in Figure 5.6.

Now, an expert applies the optimization of Figure 5.7 (b) to this subgraph to form Figure 5.8 (c). The whole graph, shown in Figure 5.10, maps to the optimized code, which *is* in the Elemental library, shown in Figure 5.9.

```
B1_MC_STAR = B1;
L10_STAR_MR = L10;
LocalGemm( NORMAL, NORMAL, 1.0, B1_MC_STAR,
           L10_STAR_MR, 1.0, B0 );
L11_STAR_STAR = L11;
B1_VC_STAR = B1_MC_STAR;
LocalTrmm( RIGHT, LOWER, NORMAL, NON_UNIT, 1.0,
           L11_STAR_STAR, B1_VC_STAR );
B1 = B1_VC_STAR;
```

Figure 5.9: Optimized code for Figure 5.1.

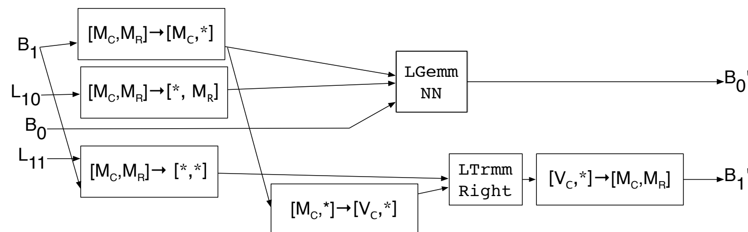

Figure 5.10: Graph representing the optimized `Trmm RLN` loop body code of Figure 5.9.

This final code is the result of two parallelizing refinements, one optimization to explore an alternate implementation of $[M_C, M_R] \to [V_C, *]$, and one optimization to remove a redundant redistribution. Each transformation is easy to understand individually, but learning and manually exploring the options and choosing the best combination is not easy and/or is tedious. It takes considerable knowledge and experience to do this well.

5.2 BLAS3

With the basics of Elemental explained above, we can show how we encode BLAS3 and Elemental knowledge in the DxT style to enable DxTer to generate all BLAS3 implementations automatically[4].

General rules for attaining high performance are that communication and redundant computation should be reduced and the portion of time spent in high-performing computation kernels should be maximized. On a single (multicore) CPU, communication is data movement between cache layers. With GPUs communication is data movement between devices and the host computer. With clusters, communication is movement between processes.

The important design decisions for Elemental deal with a small number of computation operations. For the parallel BLAS3, high-performance implementations call sequential BLAS3 kernels for suboperations (e.g., `LGemm`). Further, Elemental code requires redistribution operations (collective communication) between a finite number of supported distributions. Only knowledge regarding these redistributions needs to be encoded, and much of that, as shown below, is repetitive[5]. These are the primitives in terms of which DxT graphs will ultimately be defined.

The best implementations come down to the right combination of a small number of operations. The transformations to generate those implementations can be very simple. The rest of this section demonstrates these points.

[4] This text is adapted from [47].

[5] Many other ways to distribute data exist. Elemental only uses a small number of options to limit software complexity while still achieving high performance.

5.2.1 Algorithms to Explore

As there is generally no single algorithmic variant that works for all architectures (cluster, sequential, and so forth) or for all problem sizes, we want to encode many variants so DxTer can choose the best implementation of all. As described in Section 3.3, FLAME-derived algorithms are mathematical in nature and architecture independent, so we encode them and also encode architecture-specific transformations needed to yield efficient implementations tailored to (rough) problem sizes.

We represent BLAS3 operations in a graph with nodes named after the operations they represent (e.g., to optimize the `Trmm RLN` operation, the starting graph to be implemented consists of a single node labeled `Trmm RLN`). These are purely mathematical abstractions with no implementation details, so we label them with the abstract-most layer number. These operations can be combined in a graph with other nodes to compose higher-level functionality. In this section we focus just on implementations of the BLAS3 functions in isolation, and hence start with a graph with one node (i.e., the input to DxTer). In the next section, BLAS3 operations are combined to form LAPACK-level (higher level) functionality.

For each BLAS3 operation (e.g., `Trmm`), a refinement for each known algorithmic variant is encoded in DxTer. These refinements replace an interface with a graph representing a variant's loop and loop body operations. For blocked algorithms like in Figure 5.1, the update statements are BLAS3 operations themselves, operating on smaller submatrices. The part of the loop that does not include the update statements we call the *loop skeleton*, which is represented with the loop structure described in Section 2.5.1.

The refinement of node `Trmm RLN` for the algorithm of Figure 5.1 is a loop with update statements `Trmm RLN` and `Gemm NN`. This refinement is shown in Fig-

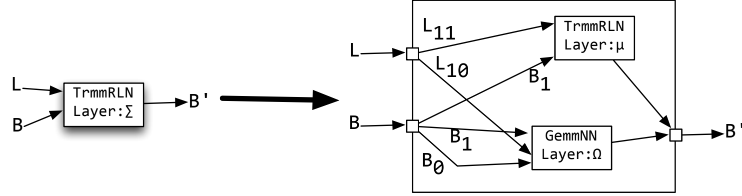

Figure 5.11: Refinement to encode algorithm of Figure 5.1.

ure 5.11 (duplicated from Figure 3.5 (d)). We change the refinement's node layers to specialize the algorithm to our architecture. In this case, we want to replace an abstract-layer node with a *distributed* layer node. For brevity, we leave off prefixes for the abstract-layer nodes and prefix with `D` for distributed-layer nodes.

Recall that local computation operations (e.g., `LocalGemm`) are labeled starting with `L`. This means their layer is *local*, which are all primitives. `L*` boxes map to local computation primitives that have no collective within. The distributed layer is one level of abstraction up. Distributed-layer interfaces need to be implemented (refined) in terms of local-layer primitive boxes to map to Elemental code.

5.2.2 BLAS3 Elemental Refinements

When implementing `Trmm RLN` above, an expert first uses a FLAME algorithm refinement, which results in distributed-layer operations. An expert then implements those operations by choosing from the ways to redistribute the operands to enable computations to be performed in parallel across a machine by calling locally-sequential computation on each core (e.g., via a call to a sequential (local) BLAS3 function). The result then needs be re-redistributed to the default $[M_C, M_R]$ distribution if it is not already distributed as such. To encode parallelization options for each of the `D*` boxes, we add refinements that have the building blocks of the local BLAS3 calls and redistribution operations.

Figures 5.3 and 5.4 are examples of parallelizing refinements. The options of Figure 5.4 parallelize computation over the process grid's rows or columns or over the entire grid depending on which templatization parameter $\pi \in \{*, M_C, M_R, V_C, V_R\}$ is chosen. An expert considers these options based on other operations in the loop body, the problem size, and so forth. Each possible refinement is included in the DxTer knowledge base. The refinement of Figure 5.4 (a) with $\pi = V_C$ was used for the code of Section 5.1.3.

Figure 5.3 shows the refinements for `DGemm NN`, which is the version of `DGemm` without transposition (i.e., `A` and `B` are both Normal instead of Transposed). There are small variations on these refinements for the three transposed versions of `DGemm` (`NT`, `TN`, and `TT`). An interested reader can discover them by looking at the Elemental library's `Gemm` implementations [2, 56], which DxTer reproduces.

All other `D*` BLAS3 functions have refinements that are comparably simple, but the particular parallelization schemes are not important here. The fixed set of Elemental distributions enable the most useful (and some less useful) ways to parallelize BLAS3 operations. These schemes are encoded in the DxTer knowledge base found at [1].

5.2.3 Redistribution Optimizations

Refinements are sufficient to attain parallel, executable code, but combinations of costly redistribution operations need to be optimized to remove inefficient communication. For that, we use optimizations on redistribution boxes.

Experts explore various ways to implement communication. Figure 5.12 (taken from [56]) demonstrates this. Each vertex is a distribution. Each edge is labelled with a collective communication operation that can accomplish the redis-

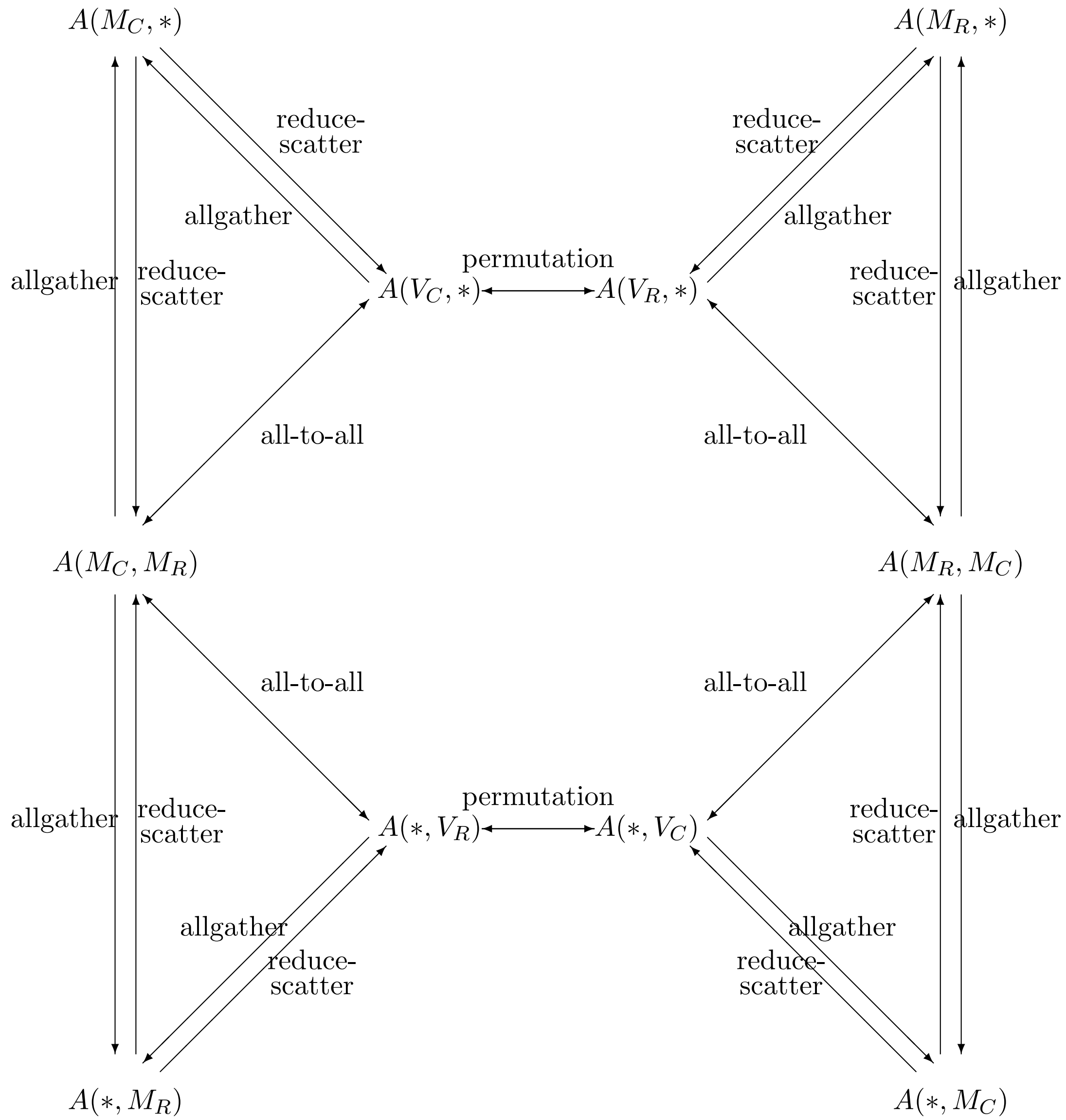

Figure 5.12: Summary of the communication patterns for redistributing matrix A from [56].

tribution from the start to the end distribution. An expert explores ways to get from one distribution on this diagram to another, considering if intermediate distributions of data are already available in the surrounding code.

Redistribution operations are implemented with default MPI collective communication. Figure 5.12 demonstrates other implementation options. Exposing hidden redistribution implementations and exploring alternative implementation enables an expert or DxTer to optimize the overall communication pattern of an implementation, possibly combining communication exposed by refinements of different update statements.

In some cases, Elemental implements "=" as a series of redistributions. One example is $[M_C, M_R] \rightarrow [V_R, *]$, which utilizes an intermediate distribution $[V_C, *]$ (i.e., with $[M_C, M_R] \rightarrow [V_C, *] \rightarrow [V_R, *]$) as shown in Figure 5.12. Optimizations like that of Figure 5.13 (c) expose such details. The template optimizations of Figure 5.13 (a) and(b) can then be employed to remove inverse or redundant redistributions, respectively, that were hiding behind redistributions. These optimizations are applied often by experts. Further, these optimizations are always worth applying when the inefficient LHS graphs are found. One should never keep redundant or inverse redistribution operations, so these optimizations are labeled as simplifiers in DxTer (Section 4.2.4).

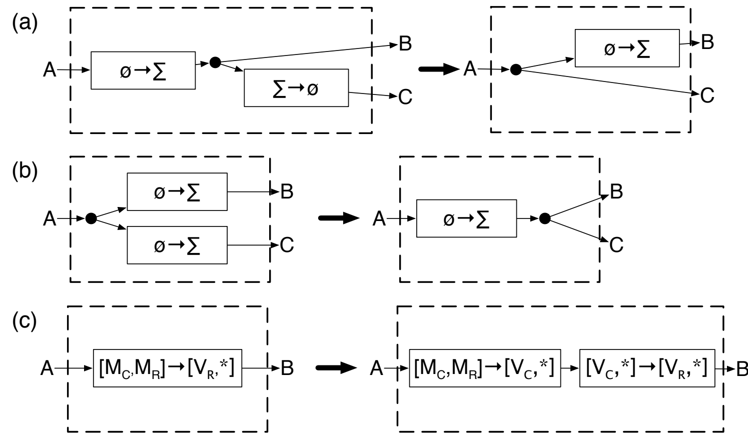

Figure 5.13: Templatized optimizations to remove inverse (a) and redundant (b) redistribution operations. Σ and ϕ can be any Elemental distribution. (c) An optimization to expose a hidden intermediate redistribution..

Optimizations like that of Figure 5.14 (a) (which is the same as Figure 5.7) are employed to explore alternate implementations of redistributions. If redistributions around the $[M_C, M_R] \rightarrow [C_C, *]$ operation already redistribute the data to $[M_C, *]$, then exposing the alternate redistributions enables a better overall implementation because an unnecessary redistribution to $[M_C, *]$ can be removed using the optimization of Figure 5.13 (b). These transformations replace a node with a subgraph that uses a *different* implementation, which will allow DxTer to explore subsequent optimizations.

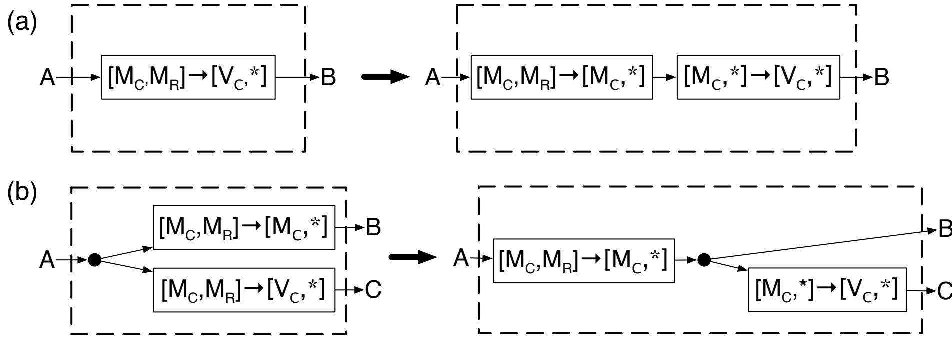

Figure 5.14: (a) An optimization to explore an alternate redistribution implementation and (b) an example of a merged transformation.

By itself, the transformation of Figure 5.14 (a) is never useful[6]. It always reduces performance unless a subsequent optimization removes a redistribution. Therefore, we encode such the combination of this transformation and the subsequent optimization as one transformation, shown in Figure 5.14 (b). By applying this to the highlighted portion of Figure 5.6, we arrive at the optimized loop body implementation shown in Figure 5.10 without exploring intermediate (and always bad) graphs.

This is a merged transformation (Section 4.2.4). By merging transformations,

[6]This demonstrates how "optimization" does not necessarily mean the transformation improves performance, but changes the implementation to possibly allow a subsequent transformation to improve performance.

we reduce the search space (sometimes by roughly half). In [46], we introduced this idea with Figure 5.15 (a) merged with Figure 5.13 (b) to form Figure 5.15 (b). There are eight versions of this transformation that are implemented using a templatized version of Figure 5.13 (d). Template parameters are limited to distributions that make sense for this optimization.

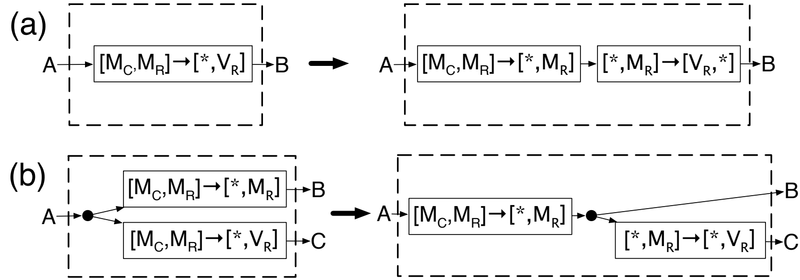

Figure 5.15: Another example of a merged transformation, where (a) is merged with Figure 5.13 (b) to form (b) here.

5.2.4 Transpose Optimizations

One of the greatest benefits of DxT is how easily new optimizations and refinements can be added and automatically applied to all algorithms. The above rules and others like them are sufficient to generate code for all BLAS3 operations. Further, that code performs well on many cluster architectures. On some, though, these rules are insufficient for good performance.

When the expert developer of Elemental tested his code on an IBM BlueGene/P machine using PowerPC 450 processors, he discovered that further optimizations were needed to improve memory access. He had to review all existing code to apply these optimizations repeatedly. With DxTer, new optimizations can be added to the knowledge base and automatically applied by regenerating all code, relieving the expert's burden. Comparing DxTer's generated code to the expert's

manually-created code, we found many instances where the expert missed these optimizations because he did not remember or have time to update existing code.

The optimizations affect the way data is read and written. With a non-unit stride, the penalty for accessing memory on an IBM BlueGene/P architecture is much greater than on many architectures. When MPI collectives are used behind a redistribution, data is packed and unpacked into send and receive buffers. The stride is often the number of rows or columns in the process grid because of the way data is redistributed.

To mitigate this substantial penalty, Elemental redistributions were added to (conjugate-)transpose data during communication, which results in more data copied with unit stride. With many BLAS3 functions, operand matrices can be transposed during computation, so data transposed during redistribution is untransposed during computation.

This optimization requires deep knowledge of Elemental because not all redistribution patterns can or should be transposed. Further, knowledge of BLAS3 functions is needed to identify when inputs can be transposed (to undo redistribution transposition). Figure 5.16 shows a transpose optimization on one input to `LGemm`. This case shows up in the stationary-`C` refinement of Figure 5.3, among others. Here, the "`B`" input to the `LGemm` box comes from a redistribution that can be transposed. This transformation of Figure 5.16 transposes the redistribution operation and changes the `LGemm` box to undo that transposition (i.e., `NN` to `NT`). This transformation applies to the code in Figure 5.10. The result of applying it generates the high-performance implementation of `Trmm RLN` generated by DxTer and now found in the Elemental library (it previously was not found in the library as explained before).

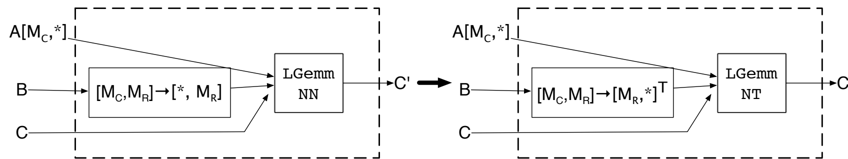

Figure 5.16: Optimization to transpose communication.

Many BLAS3 operations have transposition transformations similar to this. The actual optimization is built in a more versatile way than this figure suggests. Knowledge of redistributions is built into the nodes: they are queried to determine if they can or should be transposed. Transpose optimizations look for BLAS3 nodes and the inputs to them that can be transposed, and query the input redistribution box (if it is a redistribution) to see if it can be transposed during communication.

5.2.5 The Knowledge Base

The graph transformations we have illustrated are no more complicated than those we have not. Abstractly, they are all simple graph rewrites that capture deep domain knowledge of DLA and its encoding in Elemental. Had we chosen another cluster DLA library that did not have a cleanly-layered design, we suspect we would have been less successful or not successful at all. We can not stress enough that the key to the simplicity of our rewrite rules is that they capture relationships between fundamental levels of abstraction in DLA library design. If these abstractions are encoded inelegantly, transformations are likely to be substantially more complex.

Redistribution optimizations are templatized for use by many communication patterns (Figure 5.13 (a) and (b)). Similarly, the transformations (algorithm and parallelization refinements) for symmetric and Hermitian (the complex datatype equivalent of symmetric) BLAS3 operations are largely identical so the same knowledge can apply to both sets of operations. Further, `Trmm` and `Trsm` operations share

Type	Unique	Total
Algorithm refinement	19	30
Parallelization refinement	14	31
Redistribution optimization	32	758
Redistribution transposition	6	22
Total	71	841

Figure 5.17: Rule count in DxTer's BLAS3 knowledge base.

many of the same algorithms (with minor differences and `Trmm` switched for `Trsm`) and refinements. As a result, there is a lot of templatization and reuse of rules.

Figure 5.17 shows the unique (i.e., counting each template once) transformations encoded in DxTer to generate high-performance implementations for all BLAS3 operations. It also shows the total number of transformations that are generated from those unique pieces of knowledge using templates (different distributions, symmetric and Hermitian, etc.).

5.2.6 Cost Estimates

In Section 4.1.2, we describe how nodes have a cost property based on input problem sizes. The cost of all nodes is summed for each each implementation graph in DxTer's search space and the lowest cost is output. The input sizes consider all iterations of loops (even when loops are nested). DLA is generally optimized for minimal runtime, so that is our cost. For our work, we use γ to mean the time it takes to perform a single FLOP (i.e., the number of CPU cycle). Then, the cost of computation and communication can be estimated in terms of CPU cycles. As a system is evaluating cost estimates, more sophisticated cost estimates can be used when they are available.

When implementing code in Elemental, one does not specialize for a particular cluster architecture. Instead, one implements assuming a large process grid

Operation	Cost
`LocalGemm` $(m \times k \times n)$	$\gamma 2mkn$
`LocalTrsm RLN` $(n \times n,\ m \times n)$	γmnn
`A11_Star_Star = A11` $(m \times n)$	$\alpha \lceil \log_2 p \rceil + \beta \frac{p-1}{p} mn$
`A21_MC_Star = A21_VC_Star` $(m \times n)$	$\alpha \lceil \log_2 c \rceil + \beta \frac{c-1}{c} \frac{m}{r} n$

Figure 5.18: Representative first-order approximations for the cost of operations.

and uses rough estimates of communication on a generic cluster. For DLA we have reasonable cost estimates. First-order approximations for sequential operations can be given in terms of the number of floating point operations that are performed as a function of the size of operands. The coefficient γ roughly captures the quality of the operation's implementation and the speed of the machine. For example, matrix multiplication, $C = AB$, where C, A, and B are $m \times n$, $m \times k$ and $k \times n$, respectively, takes time (costs) $\gamma 2mkn$. The cost of every computation kernel can be approximated by the operation count multiplied by γ[7].

The data redistributions found in Elemental are implemented using MPI collective communication routines. Lower-bound costs of the common algorithms under idealized models of communication are known [16] in terms of coefficients α and β, which capture the latency and cost per item transferred, respectively. For example, redistributing an $n \times n$ block of A_{11} as in line `A11_Star_Star = A11` on p processes requires an `allgather` operation, which has a lower-bound cost of approximately $\alpha \log_2(p) + \beta \frac{p-1}{p} n^2$. α and β are set to be reasonable multiples of γ (100 and 100,000, respectively, are good choices).

Sample cost functions from the `Trmm RLN` example are in Figure 5.18. These only include higher-order terms and are first-order approximations meant to dis-

[7] A second-order approximation would take algorithm performance variation into account, but for now we stick to first-order approximations since this is generally good enough for an expert implementing algorithms by hand.

tinguish good (lower-cost) implementations of an algorithm from others. These estimates are good enough for the examples we have studied so far, but we expect to improve them to find the best code for more complicated algorithms.

For example, we have encountered situations where a collective communication operation is suboptimally implemented on a specific architecture while some other architectures provide hardware support for the same operation. As a mechanical system is evaluating the cost functions, they could be made much more sophisticated (complicated). One could use empirical timing information, for example. Further, more accurate timing will be necessary when finding the "best" implementation for specific problem sizes (instead of just "big," "medium," or "small" problems as we talk about below). Then, the crossover points are more difficult to determine with these rough estimates. The point is that, since we have an automated system, design space exploration and customization is easily accomplished.

5.2.7 Search Space and Results

We now present the performance of DxTer-generated code for the level-3 BLAS. All tests in this section were taken on an IBM BlueGene/P machine built from PowerPC 450 processors. We tested on 8192 cores (2 racks), which have a combined theoretical peak of over 27 TFLOPS. Two-thirds of peak performance is shown at the top of the graphs. Double-precision arithmetic was used for all computation. For all runs, we tune by hand the blocksize and process grid configuration and choose the best-performing run. The algorithm and implementation selections of DxTer account for the vast majority of performance; tuning the blocksize provides a small performance boost (5-10%).

BLAS3 implementations for clusters must be tailored to the problem size

BLAS3	# of Versions	# Implementations generated per variant	Compared to hand written
`Gemm`	12	378	Added transpose
`Hemm`	8	16,884	Same
`Her2k`	4	552,415	Same
`Herk`	4	1,252	Same
`Symm`	8	16,880	Same
`Syr2k`	4	295,894	Same
`Syrk`	4	1,290	Same
`Trmm`	16	3,352	Better algorithms
`Trsm`	16	1,012	Added transpose; new implementations

Figure 5.19: DxTer code generation statistics for the BLAS3.

and parameter combination. Consider, for example, `Gemm`: $C := AB + C$. `Gemm` is best provided in a library with different implementations for when each of the three input matrices is the largest (to minimize communication of it) and for each of the four combinations of "A" and "B" being transposed. As a result, Elemental offers $12 = 3 \times 4$ `Gemm` implementations. Implementations of `Trmm` could minimize communication of each of its two input matrices (whichever is biggest) and there are three parameters that lead to eight different algorithms and parallelization schemes, yielding a total of $16 = 2 \times 8$ implementations. The second column of Figure 5.19 lists the number of implementation versions for each BLAS3 operation.

For each version of each operation, we tested DxTer's ability to generate code. We ran DxTer's search on different problem sizes, making them relatively bigger or smaller depending on what sort of implementations we wanted. As the cost estimates are rough, it was only necessary to set problem sizes to be roughly large or small (i.e., 80,000 vs. 5,000). The third column of Figure 5.19 shows the total number of implementations generated by DxTer. Different parameters lead to different implementations (because different starting algorithms are used). For

versions with the same parameter combination (but different matrix sizes), the same implementations are generated, but the cost estimates rank-order them differently. This count includes the repeated implementations that are re-generated for each of the versions. Each "best" implementation is determined within 30 minutes for all operations; the majority take less than a minute.

Many of the differences between implementations are due to the variety of ways in which data can be redistributed and transposed. Consider the number of transformations dealing with redistributions, shown in Figure 5.19. There are four algorithmic versions for `Her2k`, but only one parallelizing refinement for `DHer2k` in their loop bodies. This does not lead to many implementations options. The large space is the result of the many ways to redistribute and transpose operands to the local computation.

When the Elemental developer first implemented the BLAS3, he explored a portion of these search spaces. At that point, he did not apply transposition optimizations because the Elemental API did not allow for transposed redistribution. The benefit of an automated approach to software engineering is that when an optimization is encoded, one can automatically regenerate all code with the optimization included (as demonstrated further in Chapter 6). The last column of Figure 5.19 provides a qualitative comparison of DxTer's implementations to the code in Elemental.

Figure 5.20 (top left) compares representative versions of each of the double-precision, real BLAS3 functions with problem sizes along each dimension of 50,000. We show performance from ScaLAPACK[8], Elemental, DxTer without optimization

[8]ScaLAPACK performance is often below that of Elemental because of the differences in how data is distributed and redistributed, which algorithms are used, etc. The particular details are unimportant here as the goal is to generate Elemental code, and ScaLAPACK is presented just for reference. An interested reader can find some details in [50].

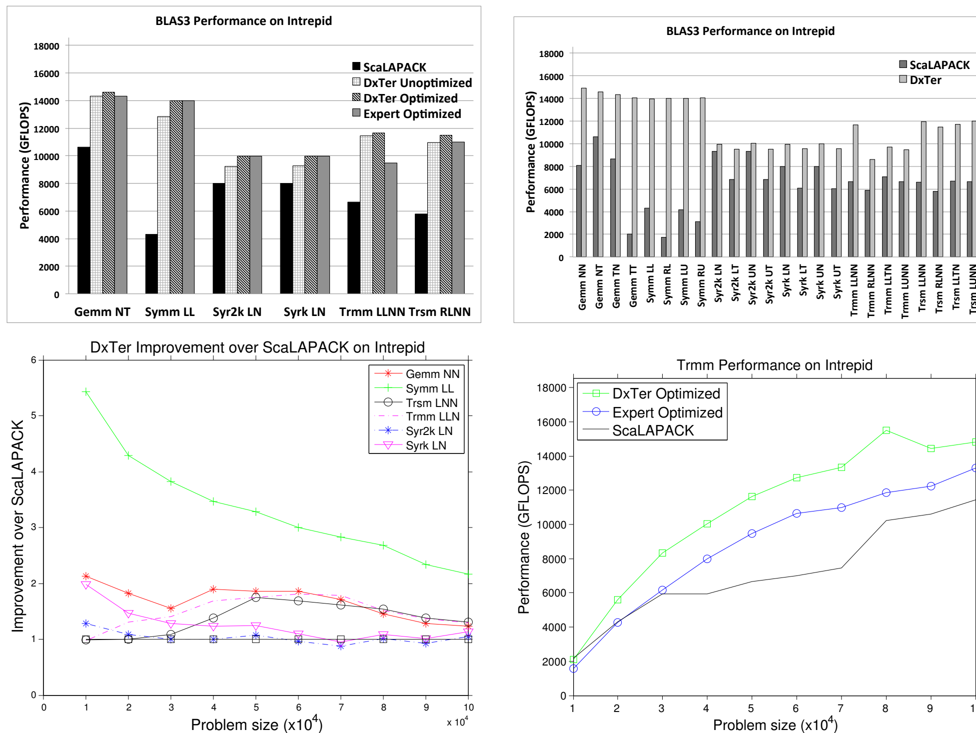

Figure 5.20: Performance of real BLAS3 functions. Problem size is 50,000 along all dimensions for top graphs.

(only parallelizing refinements), and DxTer with optimization. In many cases, the expert and DxTer produced the same implementations, but there were some notable improvements. *In all cases, DxTer generated implementations that were the same or better than the expert.*

For `Gemm`, the expert missed a number of transposition opportunities that improved performance. DxTer determined when those transpositions were worthwhile (the cost functions predicted runtime decreased) and generated code that incorporated the optimization.

For `Trsm`, DxTer again found a missed transposition opportunity in one variant. Figure 5.20 (top left) shows this is a modest improvement, but it is worthwhile

and it came without human effort. The improvement is greater for smaller problem sizes. Additionally, the expert had not implemented some of the `Trsm` versions. DxTer had sufficient knowledge to generate code for all versions.

The greatest DxTer successes came when studying `Trmm`. DxTer has three algorithms encoded for each of the "left-side" and "right-side" versions of `Trmm`. DxTer explored all implementations of these algorithms and chose as best a different algorithm than chosen by the expert. He did not explore the algorithm in Figure 5.1. Figure 5.20 (bottom right) shows the performance of DxTer's implementation over the expert-optimized version.

Figure 5.20 (top right) shows many parameter combinations for the real BLAS3 functions. We compare DxTer's predicted-best implementations against ScaLAPACK's implementations. The majority of these are the same as Elemental, so we omit its performance. Figure 5.20 (bottom left) shows a sample of these functions across a range of problem sizes, demonstrating DxTer-generated Elemental code performs better than or roughly equal to that of ScaLAPACK. Figure 5.20 (bottom right) shows the performance improvement DxTer gained when exploring many algorithms for `Trmm`, choosing one that is better than what the expert developer of Elemental used, highlighting the utility of automatic code generation.

5.3 LAPACK-Level Operations

We now examine results of using DxTer to generate more complicated code, i.e., LAPACK-level operations[9]. In some cases, DxTer generated the code before the bulk of the BLAS3 work above was completed and in other cases it was generated after. Either way, there is a lot of knowledge reuse when generating the code described in

[9] This text was taken from [47]

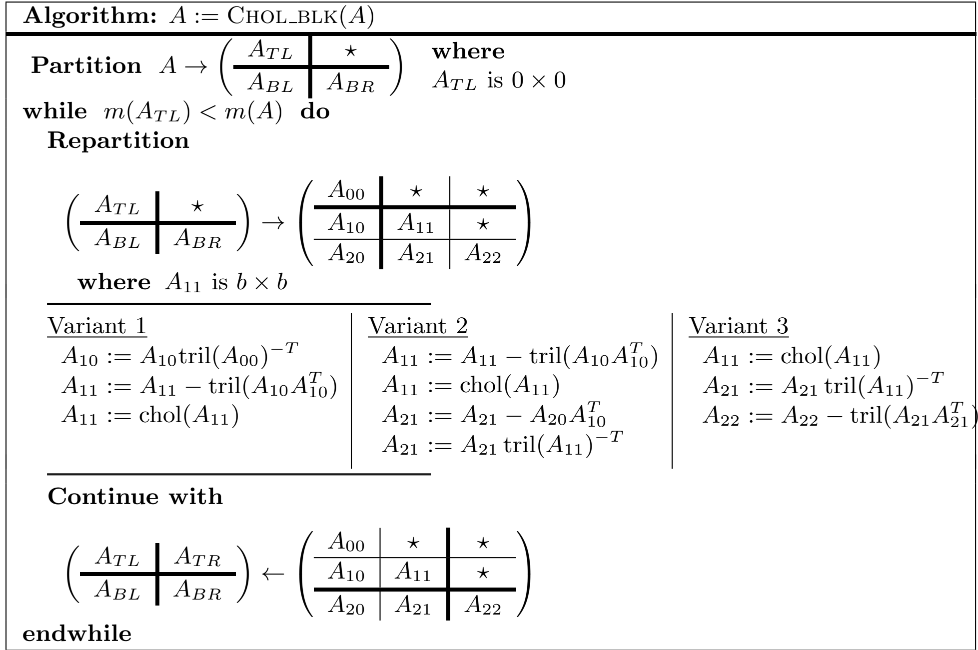

Figure 5.21: Blocked algorithms for computing the Cholesky factorization. *tril* is the lower-triangular portion of the matrix.

this section since the algorithms use BLAS3 operations as building blocks. In this section, we use a mix of results from the BlueGene/P architecture, described above, and a Xeon-based cluster based at the Texas Advanced Computing Center. We used 20 nodes, each with 2 Intel Xeon hexa-core processors running at 3.33 GHz. The combined theoretical peak performance of all 240 cores is 3.2 TFLOPS.

5.3.1 Cholesky

The first operation targeted for code generation was Cholesky factorization. Cholesky factorization takes as input a *symmetric/Hermitian, positive-definite (SPD/HPD)* matrix A, which is stored in lower or upper triangular form, and outputs the Cholesky factor such that $A = LL^T$ or $A = U^TU$, depending on if the input is

lower or upper triangular. A is overwritten with its Cholesky factor. Figure 5.21 shows three variants of Cholesky factorization for a lower-triangular matrix.

The updates are largely BLAS3 operations (`Trsm`, `Syrk` or `Herk`, and `Gemm`). These are discussed in Section 5.2. The remaining operation is Cholesky factorization of a small block. In Elemental, this is implemented by gathering all input data to distribution $[*, *]$ so all processes hold all of the data. Then, one calls a sequential implementation of Cholesky factorization. The result is then redistributed to $[M_C, M_R]$ (which is just a local memory copy). Figure 5.22 shows the refinement encoding this implementation knowledge. `LChol` has a cost of $\gamma\frac{n^3}{3}$.

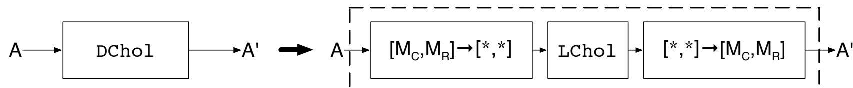

Figure 5.22: Refinement for Cholesky factorization.

All three Cholesky variants are encoded in DxTer, which yields 294 graphs and chooses a variant 3 implementation as best. That output code is the same as the expert implemented. Figure 5.23 shows performance results on a Xeon cluster compared to ScaLAPACK. The "Inlined" results are from code generated by DxTer if only refinements are applied (i.e., no optimizations). This demonstrates how only calling parallelized implementations of the loop body operations hurts performance because there are hidden inefficiencies. "Optimized 1" is the optimized implementation without transposing optimizations while "Optimized 2" includes those optimizations.

5.3.2 SPD Inversion

Cholesky factorization is a component of a more complicated operation, SPD Inversion [13] (and the complex analogue HPD Inversion). This operation takes an

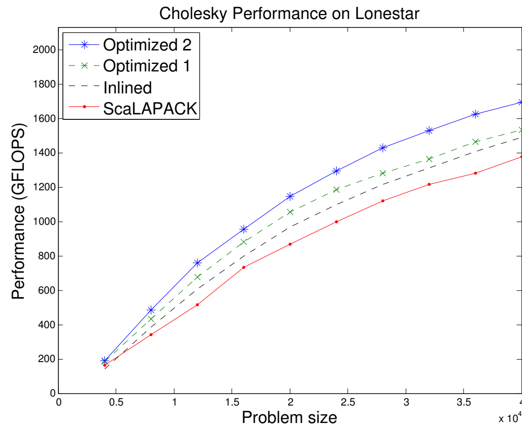

Figure 5.23: Cholesky implementation performance on a Xeon cluster. Two-thirds of peak is shown at the top of the graph.

SPD matrix A and inverts it using Cholesky factorization, triangular matrix inversion (`TriInv`, and triangular-triangular matrix multiple (`Trtrmm` not `Trmm`): $A := chol(A); A := A^{-1}; A := AA^T$

Figure 5.24 shows the various algorithms for `TriInv` and `Trtrmm`. All of these were encoded in DxTer, building on the same transformations that were used for the BLAS3. Additional refinements for `DTriInv` and `DTrtrmm` updates were created by copying the `DChol` refinement and replacing the node types because they can be implemented with the same $[*, *]$ distribution.

The work in [13] demonstrated how some of the variants' loops can be fused. For example, variant 2 of `TriInv` and variant 1 `Trtrmm` can be fused. Choosing the right variants of each of these operations allows all three to be fused. The result is a loop body that enables many opportunities for optimization of redistributions that are found in the various algorithms. Choosing the wrong variants, on the other hand, limits fusion potential. With DxTer, multiple variants are always explored and loop fusion is performed automatically when the loops are tagged as necessary (Section 3.4). DxTer, therefore, finds the implementation with all operations' loops

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 0×0

while $m(L_{TL}) < m(L)$ **do**

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

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 0×0

while $m(L_{TL}) < m(L)$ **do**

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

Figure 5.24: Algorithms for `TriInv` and `Trtrmm`.

fused. Figure 5.25 shows the loop body of the fully-fused algorithm.

DxTer's "best" implementation at this point was different than the expert's version. Since the DxTer-generated code was different, either the expert had missed something or DxTer did not have all of the expert's knowledge encoded (Section 4.1.6). In this case, there were two missing pieces of expert knowledge (transformations) that needed to be encoded in DxTer.

First, the optimization in Figure 5.26 was missing. Explicitly inverting a

$$
\begin{aligned}
A_{11} &:= Chol(A_{11}) \\
A_{01} &:= A_{01}A_{11}^{-1} \\
A_{00} &:= A_{00} + A_{01}A_{01}^{T} \\
A_{12} &:= A_{11}^{-T}A_{12} \\
A_{02} &:= A_{02} - A_{01}A_{12} \\
A_{22} &:= A_{22} - A_{12}^{T}A_{12} \\
A_{01} &:= A_{01}A_{11}^{-T} \\
A_{12} &:= -A_{11}^{-1}A_{12} \\
A_{11} &:= A_{11}^{-1} \\
A_{11} &:= A_{11}A_{11}^{T}
\end{aligned}
$$

Figure 5.25: Loop body of SPD matrix inversion.

triangular matrix can cause numerical instability. Instead of using the matrix inverse with `Trmm`, one can use the original triangular matrix with `Trsm`. `Trsm` does not explicitly invert the matrix, so it slightly improves numerical stability. Upon identification, this optimization was added to DxTer.

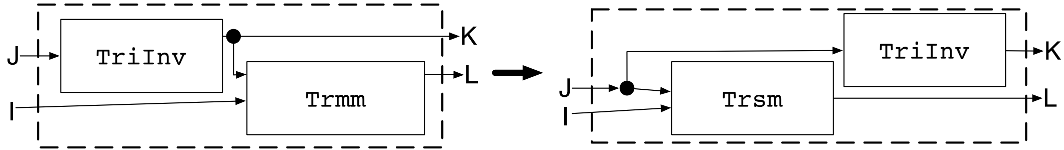

Figure 5.26: Optimization to improve numerical stability slightly.

The other missing optimization, shown in Figure 5.27, reorders operations in the implementation. Effectively, this exploits the associativity property of matrix multiplication: $(AL^{-1})B + C = A(L^{-1}B) + C$. Instead of using the result `Trsm LLN` for the first input to `Gemm`, this transformation uses the result of `Trsm RLN` in the second input. Reordering computation like this allows for subsequent communication optimization.

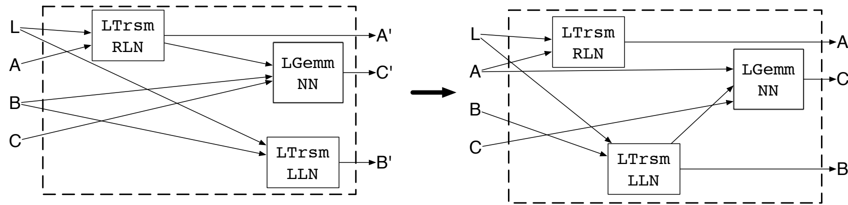

Figure 5.27: Optimization to reorder operations: $(AL^{-1})B + C$ to $A(L^{-1}B) + C$.

With these additional optimizations, DxTer produces the same implementation as the expert developed. It is the culmination of 36 transformations and fusion of the three operations' loops. Figure 5.28 shows performance on a Xeon cluster. This graph shows performance for both the non-fused DxTer code (optimized without fusing the three loops) and the fully fused and optimized code. ScaLAPACK's implementation is shown, too. That implementation does not fuse the three operations, so they are called sequentially.

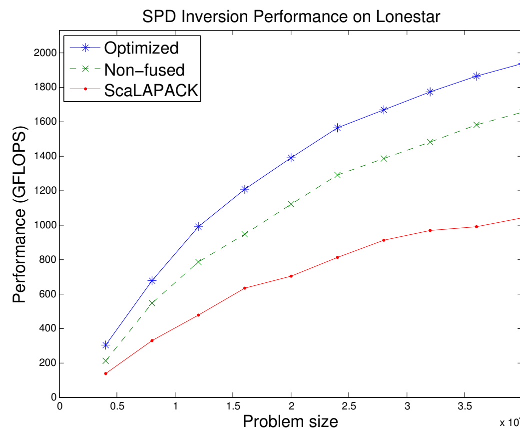

Figure 5.28: Performance of SPD Inversion on a Xeon cluster.

The `TriInv` and `Trtrmm` operations can be used individually, so we tested DxTer's implementations of them, too. For `TriInv`, DxTer's output code was slightly different than the expert's code. He had not applied the optimization of Figure 5.26, so DxTer's code was slightly more numerically stable. For `TrTrmm`, DxTer's code was

different, but the cause was a coding error made by the Elemental developer. He called a function with the wrong submatrix. DxTer's code was correct (by construction). These are examples of how automated code generation can be useful. Knowledge is used to develop code of particular interest and it is automatically applied to all operations (as with the transposing optimizations applied to the BLAS3).

5.3.3 Two-Sided Problems

The generalized eigenvalue problem is formulated as $Ax = \lambda Bx$, where A is Hermitian and B is HPD. Two-sided triangular solve (`TwoSidedTrsm`, $A := L^{-1}A^{-H}$) is employed to reduce the generalized eigenvalue problem to a standard Hermitian eigenvalue problem [7, 51]. Two-sided triangular matrix multiplication (`TwoSidedTrmm`, $A := L^H AL$) is used to reduce the generalized Hermitian-definite eigenvalue problem $ABx = \lambda x$ to a standard Hermitian eigenvalue problem [7, 51]. These two related operations are built on BLAS functionality (as well as recursive calls on a small block, implemented in a sequential LAPACK-level library).

With FLAME, one can derive five variants for each of these operations [51]. Figure 5.29 shows the best variant for each. We can encode the interesting variants[10] in DxTer. We can again copy the `DChol` refinement, replace the operations, and support refinements for `DTwoSidedTrmm` and `DTwoSidedTrsm`.

Using just this additional knowledge and existing BLAS3 knowledge (e.g., `Trmm` refinements and redistribution optimizations), DxTer can generate code for these operations. The implementations require about 30 transformations (as opposed to four for the running `Trmm` example). Figure 5.30 shows the final implementation graph DxTer generates for `TwoSidedTrmm`. For both operations, DxTer

[10] One variant of `TwoSidedTrmm` has an extra $O(n^3)$ computation, so it is not considered for implementation or encoded in DxTer as it is suboptimal on all reasonable architectures.

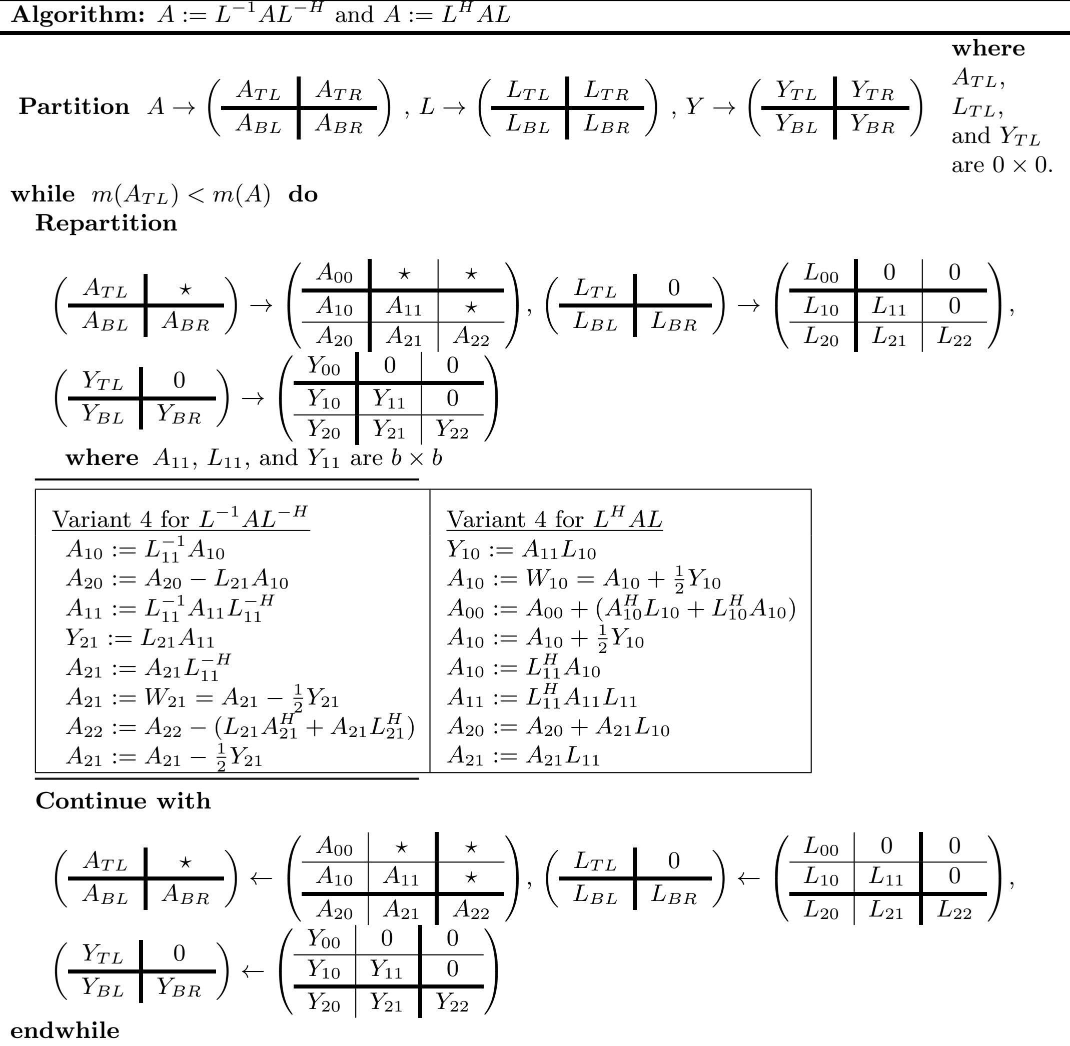

Figure 5.29: Blocked variant 4 for computing $A := L^{-1}AL^{-H}$ and $A := L^{H}AL$.

generates an implementation that is slightly better than the Elemental developer's. In Figure 5.30 (which is digitally enlargeable), we highlight the code differences in gray. They are the result of applying transposition optimizations. The appendix details the derivation of this implementation.

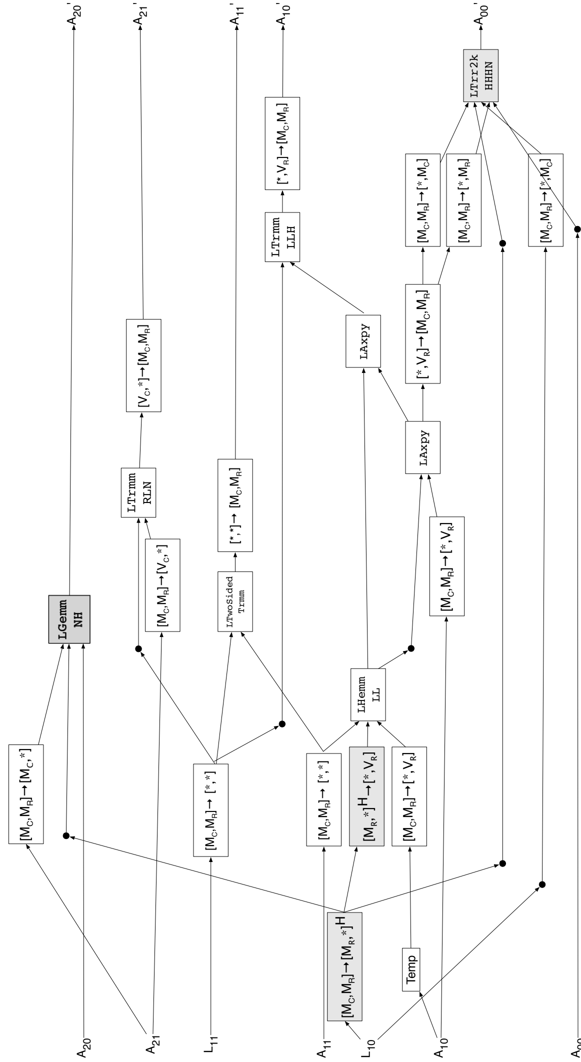

Figure 5.30: Final graph for `TwoSidedTrmm` with code improvements over hand-developed code highlighted in gray

Figure 5.31 shows the performance of DxTer-generated code for `TwoSidedTrmm` on the BlueGene/P machine. It shows the improvement of the optimized code over ScaLAPACK. The refined but unoptimized DxTer code is also shown. The performance of `TwoSidedTrsm` is almost identical to `TwoSidedTrmm` performance, so we do not show it. Further, the hand-developed code is only slightly lower performing, so we do not show it. Nonetheless, the optimizations DxTer found were worthwhile, so they have been incorporated into Elemental.

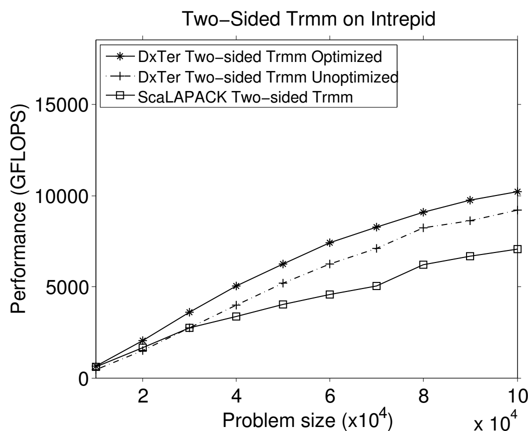

Figure 5.31: Two-sided trmm performance on a BlueGene/P architecture. Two-thirds of peak is at the top of the graph.

5.4 Locally-Best Search

While DxTer found many performance improvements over hand-developed code, it is nonetheless impressive that the Elemental developer did so well. There are so many implementation choices available (algorithms, parallelization schemes, and optimizations) that he navigated. In this section, we give some insight into how the Elemental DSL leads an expert to develop a valuable intuition for design decisions. We are able to do so because with DxT we encode those design decisions explicitly.

There are “stair-stepped performance curves” in the implementation search space, as shown in Figure 5.32 (Left) and explained in detail below. Steps arise from design decisions and how good they are with respect to performance. Some decisions make a big impact on performance (leading to a jump) and some make smaller impacts (leading to different implementations that are close together).

Experts make their decisions so effectively when exploring a massive implementation space by learning, through experience, which design decisions are the most important to get “right” (i.e., those that impact performance most). Further, they know which options for those decisions are best because the options are limited by the DSL. We explain how we leverage this intuition to limit the number of refinements DxTer explores. This significantly reduces the search space and makes it tractable for some operations (along with the use of merged transformations and simplifiers).

5.4.1 Implementation Clusters

DxTer estimates the cost of implementations to rank-order them and choose the “best.” We described how cost estimates do not need to be very accurate with respect to actual runtime as long as they rank order them correctly. They are good enough to compare implementation choices to determine which are best.

With FLAME, one can derive three Cholesky factorization algorithmic variants, shown in Figure 5.21. We can use DxTer to explore the implementation options for each of these. First, we explore the implementation of variant 3 since it has the fewest options. The first update, $A_{11} := chol(A_{11})$, has one refinement (parallelization scheme in Elemental). The third update, $A_{22} := A_{22} - \text{tril}(A_{21}A_{21}^T)$, also only has one option.

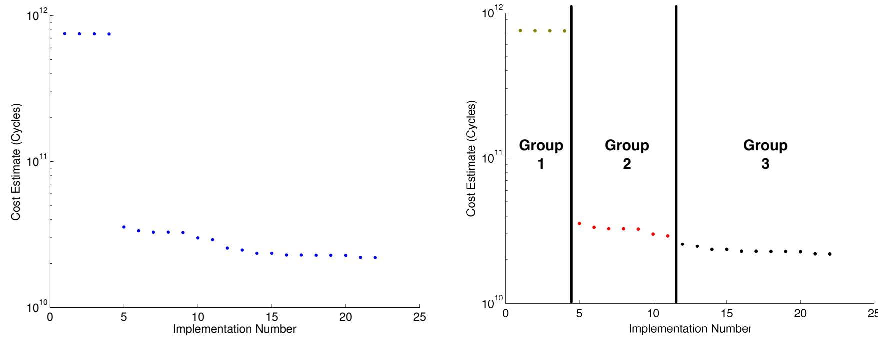

Figure 5.32: (Left) Costs and (Right) clusters of Cholesky variant 3 implementations with $k = 3$.

The middle update, $A_{21} := A_{21}\,\text{tril}(A_{11})^{-T}$, has five options. Figure 5.4 (b) shows the templatized refinement of `Trmm` on the right-hand side (with $\pi \in \{*, M_C, M_R, V_C, V_R\}$)[11]. With this refinement, the option of $\pi = *$ offers no parallelization – all processes perform the same computation. The options of $\pi = M_C$ or $\pi = M_R$ offer some parallelization with some redundancy. With $\pi = V_C$ or $\pi = V_R$, there is no redundancy; each process computes a different portion of the result. These options trade off parallelism and communication cost. With enough computation, additional communication cost is worthwhile to gain parallelism. The best decision is mostly determined by the problem size while surrounding code is a secondary consideration.

In Figure 5.32 (Left), we show the cost estimates of all implementations DxTer generates for variant 3 with a problem size of 80,000. They are ordered from greatest cost on the left to least cost (most efficient) on the right. In Figure 5.33 (Right), we show the result of clustering implementation costs with k-means clustering ($k = 3$ here).

[11] This refinement is actually further templatized in DxTer to represent `Trsm`, too, since the same refinement works for both. The nodes are labeled if the triangular matrix is inverted (for `Trsm`) or not (for `Trmm`).

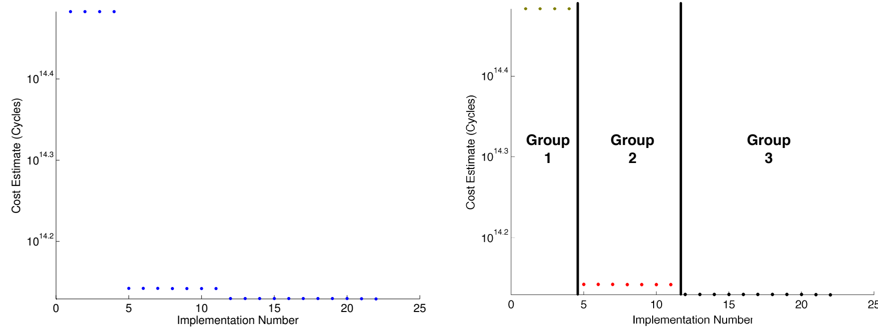

Figure 5.33: (Left) Costs and (Right) clusters of Cholesky variant 1 implementations with $k = 3$.

K-means clustering partitions data (implementation costs in this case) into k clusters to minimize $\sum_{i=1}^{k} \sum_{x_j \in C_i} |x_j - \mu_i|$ where C_i is the ith cluster, x_j is the jth piece of data in C_i, and μ_i is the mean of the data in C_i.

The group on the left is significantly worse than the two groups on the right. The group on the left uses the $\pi = *$ refinement, which an expert knows is a bad choice for a large problem size. The variation within the group is due to optimizations, which can only provide small performance changes once the bad choice of $\pi = *$ is made. The lower-performing of the other two groups (the middle group) consists of implementations using $\pi = M_C$ or $\pi = M_R$ and the best group's implementations use $\pi = V_C$ or $\pi = V_R$. Thus, all implementations in the search space can be separated based, largely, on which `Trsm` refinement is used.

Cholesky variant 1, like variant 3, has a `Trsm` operation, $A_{10} := A_{10}\text{tril}(A_{00})^{-T}$. The third operation is Cholesky, again, which has only one refinement. The second operation is `Herk`, which only has one refinement. In Figure 5.33 (Left), we show the cost estimates for variant 1 implementations. Again, there are three main groups that are distinguished by the `Trsm` refinement one chooses, as demonstrated by the clustering in Figure 5.33 (Right).

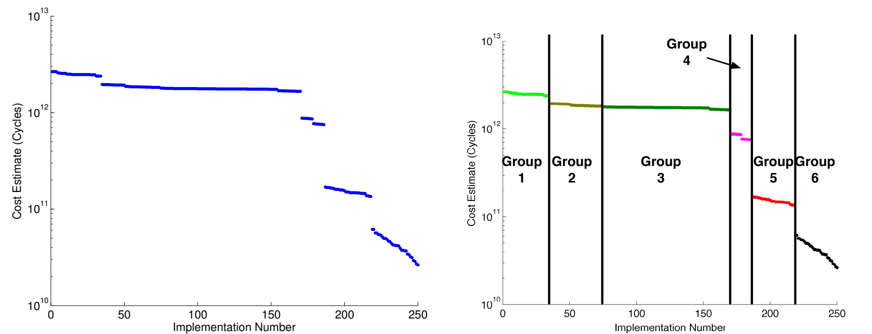

Figure 5.34: (Left) Costs and (Right) clusters of Cholesky variant 2 implementations with $k = 6$.

The stark difference between the three groups is important for human developers. DxTer uses a cost estimate of `Trsm` to determine which implementation options are best. A human developer knows that the suboptimal `Trsm` refinements, for a large problem size, are not worth even considering. Cost estimates lead an expert to rule out the bad options so he does not have to consider them in the context of other implementation options. Thus, he does not need to explore the entire combinatorial implementation space. Instead, the cost estimates allow him to prune consideration substantially. For this variant, one must only consider the refinement of one interface. For more complicated algorithms with many operations to implement/refine, being able to limit consideration is a great boon. Further, the Elemental API / DSL makes it easy to recognize that decisions for `Trsm`, in this case, are the most important to get "right" while redistribution optimizations are less important. This is generally true across algorithms.

Cholesky variant 2 is more complex than the other variants. In addition to `Trsm`, variant 2 has a `Gemm` update. `Gemm` has three refinements (Figure 5.3), so there are $5 \times 3 \times 3 = 45$ refinements of the algorithm (`Trsm` has 5 refinements). With optimizations, DxTer generates 250 implementations of variant 2, the costs of which

are shown in Figure 5.34 (Left).

Recall that `Trsm` has three levels of parallelization within the five refinements. This variant has 2 `Gemm` updates, but one accounts for a small amount of the overall computation, so we only consider the large one (the third update, overwriting A_{21}). `Gemm` has three parallelization schemes, but we can think of one keeping the largest matrix stationary while the other two do not. Therefore, `Gemm` has two basic schemes like `Trsm` has three. Figure 5.34 (Right) demonstrates the result of clustering with $k = 6$. Indeed, the $3 \times 2 = 6$ clusters from the refinements are clear. The right three clusters (best performing) all use the `Gemm` refinement that keeps the largest input (A_{20}) stationary. The left three clusters use a combination of the other two `Gemm` refinements. These are the main two clusters, which are clear to an expert just looking at the algorithm because `Gemm` accounts for the majority of FLOPS performed by the algorithm, so it is most important to get "right." Within the main clusters, the differentiation between the three subclusters is the class of `Trsm` refinements used.

For this more complicated algorithm, the developer is again able to use cost functions to make the important decisions: how to implement `Gemm` and `Trsm`. If those decisions are not made correctly, he is stuck exploring a suboptimal cluster of implementations. Once in the right cluster, he only needs to explore optimization options. As finding the right cluster (i.e., pruning the search space) is made relatively easy with cost functions, the developer spends most time optimizing. A developer becomes an expert as he gains "intuition" of which refinements are best (i.e., which lead to the best cluster) by gaining experience with the cost functions.

So what about choosing the right algorithmic variant? Cholesky has three, so we would hope that one can use some intuition to guide to the best variant. Fig-

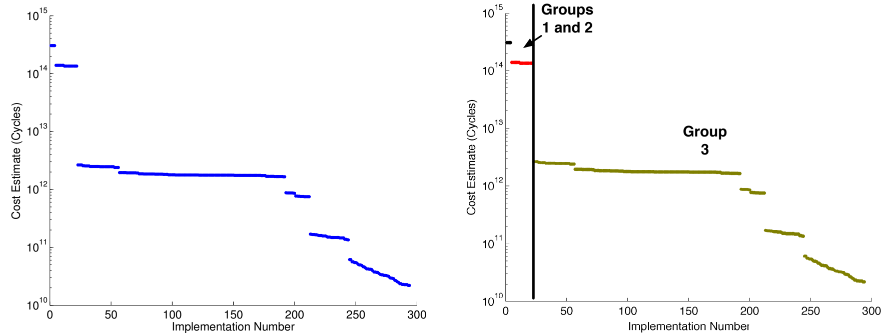

Figure 5.35: (Left) Costs and (Right) clusters of all Cholesky implementations with $k = 3$.

ure 5.35 (Left) shows the costs of implementations of all variants. Figure 5.35 (Right) shows the implementations clustered into three groups. The left two clusters both use variant 1. Cluster 1 of Figure 5.35 (Right) uses the $\pi = *$ `Trsm` refinement for variant 1 and Cluster 2 use the other `Trsm` refinements. Cluster 3 includes implementations of both variants 2 and 3. Even when clustering with a larger k, variant 2 clusters are intermingled with variant 3 clusters.

This clustering makes sense to an expert. Expert intuition is to use algorithms rich in rank-k updates. *Rank-k* updates are characterized by computation with a relatively small inner dimension (the k dimension) and (much) larger outer dimensions. These generally perform well because rank-k updates parallelize well. $A_{10} := A_{10}\text{tril}(A_{00})^{-T}$ in variant 1 is not a rank-k update and accounts for a majority of the algorithms' FLOPS. It is a `Trsm` operation that is particularly bad with all variant 1 implementations because A_{00} is redistributed to $[*, *]$, which is a very costly communication (`AllToAll`). Thus, an expert knows to avoid variant 1, and the clustering in Figure 5.35 (Right) visualizes this using cost functions to quantify this intuition.

Thus, an expert can prune the search space substantially by ruling out the

Variant 1	Variant 2
$Y_{21} := A_{22}L_{21}$ $A_{21} := A_{21}L_{11}$ $A_{21} := W_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} := L_{11}^H A_{11} L_{11}$ $A_{11} := A_{11} + (A_{21}^H L_{21} + L_{21}^H A_{21})$ $A_{21} := A_{21} + \frac{1}{2}Y_{21}$ $A_{21} := L_{22}^H A_{21}$	$A_{10} = L_{11}^H A_{10}$ $A_{10} = A_{10} + L_{21}^H A_{20}$ $Y_{21} = A_{22}L_{21}$ $A_{21} = A_{21}L_{11}$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$ $A_{11} = L_{11}^H A_{11} L_{11}$ $A_{11} = A_{11} + (A_{21}^H L_{21} + L_{21}^H * A_{21})$ $A_{21} = A_{21} + \frac{1}{2}Y_{21}$
Variant 4	Variant 5
$Y_{10} := A_{11}L_{10}$ $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^H A_{10}$ $A_{11} := L_{11}^H A_{11} L_{11}$ $A_{20} := A_{20} + A_{21}L_{10}$ $A_{21} := A_{21}L_{11}$	$Y_{10} := A_{11}L_{10}$ $A_{10} := A_{10}L_{00}$ $A_{10} := W_{10} = A_{10} + \frac{1}{2}Y_{10}$ $A_{00} := A_{00} + (A_{10}^H L_{10} + L_{10}^H A_{10})$ $A_{10} := A_{10} + \frac{1}{2}Y_{10}$ $A_{10} := L_{11}^H A_{10}$ $A_{11} := L_{11}^H A_{11} L_{11}$

Figure 5.36: Loop bodies for $A := L^H AL$. Variant 3 is omitted because it performs an additional $O(n^3)$ operations and is therefore never better than the other variants.

majority of refinement options that lead to the worst groups. In Section 5.4.2, we explain how we replicate an expert's intuition to limit the DxTer search space. For choosing an algorithmic variant, one can occasionally rule out a variant with larger-order computation as with the variant of `TwoSidedTrmm` in Figure 5.29.

One can also rule out a fraction of the variants with expensive non-rank-k-update operations. That is, the expert thinks a few steps ahead: not just about the computation cost of the algorithm but also available refinements about the loop-body operations. He knows that non-rank-k-update operations do not have good-performing refinements.

5.4.2 Locally-Best Refinements

Cholesky is fairly simple to implement in Elemental in terms of the number of transformations required and the lines of code. `TwoSidedTrmm`, on the other hand, is not. Figure 5.36 shows four of the five algorithmic variants presented in [51]. Each of these is more complicated than Cholesky (in terms of the size of the search space and number of lines of code). In exploring all implementations, DxTer's search space is too massive to enumerate fully.

Even when exploring only two variants, DxTer halted after a day of computation when the system ran out of memory [46]. In Section 5.2.3 we presented metaoptimizations (simplifiers and merged transformations) developed to limit the search space sufficiently to explore two of the variants. Utilizing the lessons from above, we can do better to enable DxTer to search all variants of the two-sided problems.

DxTer explores implementations by refining interfaces (first phase), culling graphs with interfaces, and then optimizing the remaining graphs (second phase). As cost functions can lead one to distinguish the "good" group of refinement options from the "bad" group, DxTer can use cost functions to explore only the good parallelizing refinements. By reducing the number of refinements applied in the first phase, the combinatorial explosion of the second phase is limited.

The second phase for distributed memory can end with 100-times or more implementations as it starts with, depending on the complexity of starting graphs. Therefore, reducing the graphs generated in the first phase has a significant impact on the total number of implementations searched.

In DxTer, there are two immediate subclasses of `Transformation`. `SingleTrans` represents a single transformation. `MultiTrans` contains multiple refinements that

all apply to the same node type. When a `MultiTrans` applies to a particular node type, the RHS graphs of the refinements are evaluated for their cost given the node's input sizes. The refinements are ordered by those RHS costs. DxTer can omit the worst refinements, so only the n best refinements are explored. Effectively, this leads to a greedy search in the first phase where only the best n refinements are explored. We call this the *n-locally-best* refinements. Optimizations might make some of the locally-worst refinements better in the global search, but clustering shows optimizations generally have lower-order performance effects.

Locally-Best Results

In Figure 5.37 (Top Left), we show the costs and of all `TwoSidedTrmm` variants' implementations when only exploring the n-locally-best refinements using $n = 2$. DxTer generated a total of 8,136 implementations. The best, as described in Section 5.3.3, is slightly better than the hand-developed code because the Elemental developer did not apply transposition optimizations. Even with such a limited search (i.e., $n = 2$), DxTer generated a better implementation than an expert produced.

In Figure 5.37 (Top Right), the best group contains 680 implementations, which have a mixture of variants. In Figure 5.37 (Bottom), we cluster with $k = 50$, which is large enough to show many groups without having very small groups. The best five groups (containing 569 implementations) all use variant 4 of the algorithm. The sixth-best group (containing 52 implementations) use a combination of variants 4 and 5. This means that while 4 is the best variant to use, 5 is not "so bad" since it is in the top 10% of implementations. A developer could have made the mistake of using variant 5, but the expert developer was able to pick variant 4 as best out of all of the options. In [51], performance of these four variants is tested on

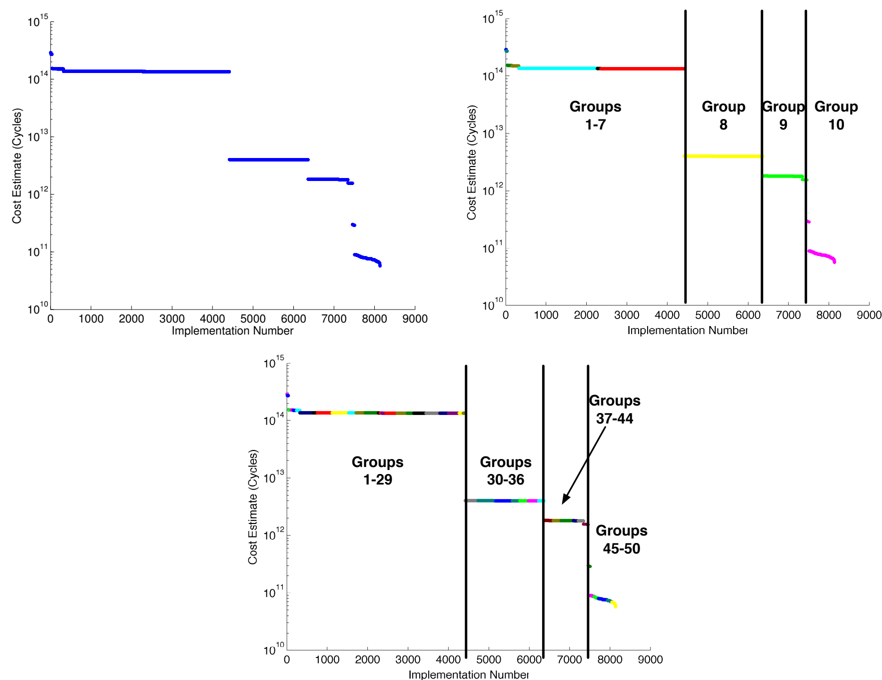

Figure 5.37: (Top Left) Costs and (Top Right and Bottom) clusters of `TwoSidedTrmm` implementations generated with on the two-locally-best refinements explored and clustered with (Top Right) $k = 10$ and (Bottom) $k = 50$.

2,048 cores. Those results show that a variant 4 implementation outperforms other variants and that a variant 5 implementation is better than variants 1 and 2, which DxTer determined analytically.

Using the two-locally-best heuristic, DxTer is able to explore all of these variants in *four minutes* on a dual-core system with 16 GB of memory instead of running out of 96 GB of memory after a day of search without the heuristic and generating millions of implementations (most of which were bad). With a three-locally-best heuristic, DxTer generates 85,448 implementations and takes *52 minutes* and still outputs the same best implementation.

We only apply this search heuristic for BLAS3 interfaces and their paral-

lelizing refinements. It does not apply to the choice of algorithmic variant. As demonstrated above, the choice between variants is difficult without exploring refinements. One can use a cost like FLOPS to remove, for instance, variant 3 of `TwoSidedTrmm`. Since such excessively-expensive variants are not encoded in DxTer in the first place, this does not reduce the search space. One could also construct a cost function to penalize non-rank-k-update operations [41]. This is future work.

We use this heuristic for the operations described in this chapter, including `TwoSidedTrmm`, and it leads to a significantly reduced search space. We have found $n = 3$ to be a good balance between search space reduction and implementation quality. In cases where the search space can be fully enumerated without the heuristic, DxTer outputs the same "best" implementation with $n = 3$ as with no locally-best search. In the cases where full enumeration is not viable (due to computation or space constraints), the heuristic still leads to the same or better implementations than chosen by the developer of Elemental.

5.4.3 The Axpy Heuristic

This search heuristic does not "consider" the context of a refinement within the graph[12]. It is possible for a refinement selected with only local consideration to lead to a globally suboptimal implementation. Optimizations generally cross interface boundaries, so they can make a two locally suboptimal refinements globally best by removing an inefficient piece of code.

`DAxpy` is an interface for which this is common. `Axpy` implements $y := \alpha x + y$ where x and y are vectors or matrices and α is a scalar (it is a level-1 BLAS operation). `DAxpy` performs $O(n^2)$ computation on $O(n^2)$ data, so the redistribution cost is a significant component of the implementation cost – it is $O(n^2)$ and in

[12] This section is adapted from [46].

practice the coefficient is larger than that for computation.

Redistributions are often optimized away across interface boundaries. For higher-level BLAS3 operations, the communication cost is a lower order term compared to the computation on large problem sizes, so locally-best choices that do not consider redistribution optimizations are sufficient. Even for small problem sizes, n-locally-best heuristics seem to perform well, but these are less of a consideration anyway for distributed-memory DLA libraries. `Axpy` is different.

For `DAxpy`, we developed a special heuristic to limit the search space. `DAxpy` refinements are templatized over a single distribution (Ω in Figure 5.38), which can be instantiated with any Elemental distribution (Figure 5.2). That is ten refinements of `DAxpy`. The algorithms in Figure 5.36 have two `DAxpy` operations, which lead to a 100-times increase in the search space. In the context of the entire graph, x or y can already be distributed as Ω or the output y' could be redistributed as Ω as part of refinements upstream or downstream of the `DAxpy` interface. In these cases, the optimization to get rid of inverse redistributions (i.e., of Figure 5.13 (a)) applies to get rid of an extraneous $O(n^2)$ cost. This optimization makes the `DAxpy` refinement using that distribution worth exploring.

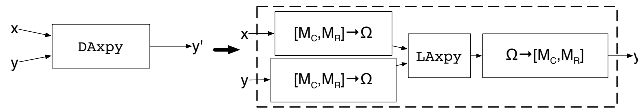

Figure 5.38: `DAxpy` refinements templatized on distribution Ω.

We use a simple heuristic to limit the ten refinements explored. For each instantiation on Ω, DxTer searches the graph immediately around the interface in `CanApply`. If an input or output is already distributed as Ω, then the refinement is allowed. A subsequent optimization then removes one or more of the redundant redistributions. Not only does this limit the search space considerably and result

in good implementations, but it also makes sense to an expert developer. If data is already distributed in a particular way, an expert would reuse that distribution to implement `Axpy`.

5.4.4 Are Heuristics Cheating?

One might consider adding a heuristic like the `DAxpy` refinement limitation a cheat since we are not allowing DxTer to explore options and instead guiding DxTer to a choice we already know is good. Similarly, the idea of merged transformations might seem like a cheat. These are ways to limit the search space in a problem-specific way (or interface-specific way). Depending on how one thinks about DxT, it is both a way to encode expert knowledge and a way to empower an expert to be more productive using an automated tool like DxTer. From that viewpoint, one *should* encode the "tricks" or "rules of thumb" an expert uses.

When an expert does not trust his tricks or wants to test their efficacy, he should not encode them and DxTer should search all options. If this leads to a prohibitively large search space, then DxTer should be improved (maybe using a MapReduce paradigm on a distributed-memory system [19] or similar engineering optimizations). When an expert does trust his rule of thumb, though, it should be encoded for DxTer to exploit just as he would manually. This improves DxTer's search time.

Further, this makes the trick explicit. Future developers can learn tricks by examining the knowledge base or questioning how DxTer generates a particular implementation. No longer will a developer's hard-earned expertise vanish when he leaves a project, and no longer will experts forget tricks after years of not using them.

For now, the downside is that such tricks are embedded in the search space exploration code instead of as an addition to the knowledge base. In the future, we wish for such tricks to be encoded in a separate knowledge base used to guide search. Domain-specific, target-specific, or general heuristics would be added to this knowledge base as needed or desired.

5.5 Summary

We have encoded knowledge about DLA and Elemental to generate code automatically that is the same or better than what an expert hand developed.

By making the implementation space explicit, we are able to recognize features of distributed-memory DLA code. "Stairs" of performance show us which design decisions impact performance the most (often parallelizing refinements) and which have less of an impact (optimizations). This allows us to use a heuristic to limit implementations to only the best stairs, which substantially reduces the search space. We expect these lessons to apply to some other domains, providing similar benefits to search / code-generation time.

Chapter 6

BLIS

We now talk about algorithm generation for multithreaded BLAS3 operations using BLIS as a DSL. We start with sequential implementations and add parallelism to generate multithreaded implementations. With sequential versions, as with Elemental, we endeavored to encode knowledge to produce algorithms just as an expert would. With shared memory, on the other hand, an expert was developing new ways to parallelize code and used DxTer to explore his ideas. We explain how DxTer allowed the developer to explore his ideas quickly, relieving him of the rote and tedious task of implementing all BLAS3 operations manually each time he had a new idea (and fixing compilation errors and testing them for correctness)

6.1 BLIS Layering

Because DLA operations are usually recursive (with suboperations operating on smaller problem sizes), algorithms are *layered*. As described in Chapter 3, this means that an algorithm is chosen for the outer blocking, then another is chosen for the next level, and so forth. until the problem is small enough to implement directly

with scalars and basic operations like addition and multiplication.

One of the highest-performance open-source BLAS libraries is the GotoBLAS, which is specialized for many architectures [28, 29]. This library implements the BLAS3 using layered algorithms to keep particular pieces of data in specific caches. In [29], cost models explained how Goto's choices achieved high performance. Code is specialized to particular architectures by adjusting algorithmic blocksizes and by hand-coding kernels in assembly to optimize low-level CPU behavior (e.g., down to considerations of prefetching and out-of-order execution). Not all library code had to be ported to a new architecture, but a lot of work is required to tune assembly code.

BLIS is a new framework that enables one to instantiate a BLAS library (i.e., provide all of the functionality in the BLAS standard and more) relatively easily by providing a small set of architecture-tuned kernels. BLIS provides functionality layered on top of these kernels to implement BLAS operations.

BLIS uses the GotoBLAS algorithm structure with additional abstractions to make porting easier. This simplifies encoding BLIS implementation knowledge in DxTer; it would have been much more difficult with the GotoBLAS (if possible at all) due to the irregular structure of its code and lack of abstraction. Without good abstraction, code generation is more difficult as the target DSL is more complicated and includes more primitives. More needs to be encoded to achieve the same functionality (and performance in this case).

Here are some basics about a CPU's cache structure, which motivates the GotoBLAS and BLIS layering. CPU registers have very fast reads and writes, but there are few of them. The level-1 (L1) cache holds much more data, but is slower. The L2 cache is larger still and is slower than L1. Some processors have an L3

cache (larger and slower than L2). The registers and caches are on the processor, which is then connected to main memory via a relatively slow communication bus. Figure 6.1 shows this structure.

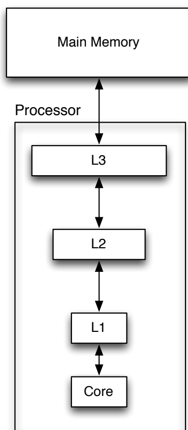

Figure 6.1: Cache and memory structure for single-core CPU.

For an operation like `Gemm`, we decompose a computation into a series of smaller `Gemm` operations. We do that by employing layers of FLAME-derived algorithms and target particular pieces of data at each layer for specific caches[1].

6.1.1 Sequential `Gemm` Implementation

Figures 6.2, 6.3, and 6.4 show the three FLAME-derived algorithmic variants for `Gemm` (copied from Chapter 3). They can be thought of as reducing the problem size in a single dimension (m, k, and n, respectively) since they partition the input matrices along those dimensions and the loop body is a `Gemm` operation itself with a smaller problem size. This recursion is implemented with different algorithmic variants, thus layering. We explain BLIS algorithm layering from the top (where

[1]One cannot instruct data to be bound to particular levels of cache, so one organizes computation such that the processor's cache eviction heuristics keep the data in the desired cache as much as possible. Generally, the most recently used data is kept in the cache closest to the registers and data is pushed to further-away caches as new data is accessed.

Algorithm: $[C] := \text{GEMM_BLK_VAR1}(A, B, C)$

Partition $A \rightarrow \left(\frac{A_T}{A_B}\right)$, $C \rightarrow \left(\frac{C_T}{C_B}\right)$ **where** A_T has 0 rows, C_T has 0 rows

while $m(A_T) < m(A)$ **do**

Repartition

$$\left(\frac{A_T}{A_B}\right) \rightarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \rightarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$

where A_1 has b rows, C_1 has b rows

$C_1 := A_1 B + C_1$

Continue with

$$\left(\frac{A_T}{A_B}\right) \leftarrow \begin{pmatrix} A_0 \\ \hline A_1 \\ \hline A_2 \end{pmatrix}, \left(\frac{C_T}{C_B}\right) \leftarrow \begin{pmatrix} C_0 \\ \hline C_1 \\ \hline C_2 \end{pmatrix}$$

endwhile

Figure 6.2: Variant 1 of `Gemm` computes $C := AB + C$ (Normal, Normal) or `Gemm NN`.

the size of the problem is arbitrarily large) down to a small size. Computation on this small size is implemented with a BLIS-specific primitive called the *macrokernel*. This is described later.

The outer-most algorithm partitions in the n dimension (Figure 6.4). We call the block size used b_n, which we specify later. Figure 6.5 demonstrates how this loop partitions the full matrices (first row) in the n dimension into smaller problems (shown on the second row). For the loop-body `Gemm` suboperation, the algorithm that partitions in the k dimension (Figure 6.3) is used. We call the blocksize used for that algorithm b_k. The `Gemm` sub-operation for the k-dimension loop is a rank-k update, shown in the third row of Figure 6.5 At this point, the `Gemm` suboperation/rank-k update has a portion of B that is $b_k \times b_n$, and there are two nested loops around this operation.

Note that a different portion of the B matrix is read in each iteration while A and C data are reread and/or rewritten across iterations of the outer or inner loop.

Algorithm: $[C] := \text{GEMM_BLK_VAR2}(A, B, C)$

Partition $A \rightarrow \left(A_L \,\middle|\, A_R \right)$, $B \rightarrow \left(\frac{B_T}{B_B} \right)$ **where** A_L has 0 columns, B_T has 0 rows

while $n(A_L) < n(A)$ **do**

Repartition

$$\left(A_L \,\middle|\, A_R \right) \rightarrow \left(A_0 \,\middle|\, A_1 \,\middle|\, A_2 \right), \quad \left(\frac{B_T}{B_B} \right) \rightarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$$

where A_1 has b columns, B_1 has b rows

$C := A_1 B_1 + C$

Continue with

$$\left(A_L \,\middle|\, A_R \right) \leftarrow \left(A_0 \,\middle|\, A_1 \,\middle|\, A_2 \right), \quad \left(\frac{B_T}{B_B} \right) \leftarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$$

endwhile

Figure 6.3: Variant 2 of `Gemm` computes $C := AB + C$ (Normal, Normal) or `Gemm NN`.

This means that if we can bring the portion of B (referred to by B_1 in Figure 6.3) into cache and keep it there throughout the `Gemm` suboperation, the substantial cost of reading $b_k \times b_n$ data from main memory is incurred only once and amortized over $O(2mb_kb_n)$ FLOPS. To this end, we try to keep the piece of B in the L3 cache. Therefore, b_k and b_n are chosen such that the portion of B takes up as much of L3 as possible without being evicted during computation. Generally, $b_k \approx 256$ and $b_n \approx 4,096$, but actual values are architecture-dependent.

Next, we use the m-dimension algorithm of Figure 6.2 to decompose the suboperation further. The blocksize $b_m \approx 256$ is used. The suboperation uses all of the $b_k \times b_n$ panel of B and multiplies it by a $b_m \times b_k$ block of A. We bring the block of A into L2 once and structure computation such that it remains there for the suboperation (as with B in L3 at the higher layer). This amortizes $b_m \times b_k$ data reads over $O(2b_mb_kb_n)$ computations. b_m and b_k are typically chosen such that A takes up about half of the L2. This suboperation is called a *macrokernel* in BLIS.

Algorithm: $[C] := \text{GEMM_BLK_VAR3}(A, B, C)$

Partition $B \rightarrow \left(B_L \| B_R \right)$, $C \rightarrow \left(C_L \| C_R \right)$ **where** B_L has 0 columns, C_L has 0 columns

while $n(B_L) < n(B)$ **do**

Repartition

$\left(B_L \| B_R \right) \rightarrow \left(B_0 \| B_1 | B_2 \right)$, $\left(C_L \| C_R \right) \rightarrow \left(C_0 \| C_1 | C_2 \right)$

where B_1 has b columns, C_1 has b columns

$C_1 := AB_1 + C_1$

Continue with

$\left(B_L \| B_R \right) \leftarrow \left(B_0 | B_1 \| B_2 \right)$, $\left(C_L \| C_R \right) \leftarrow \left(C_0 | C_1 \| C_2 \right)$

endwhile

Figure 6.4: Variant 3 of `Gemm` computes $C := AB + C$ (Normal, Normal) or `Gemm NN`.

We consider macrokernel as primitives for encoding knowledge and generating code. For consistency, though, we mention that a macrokernel is implemented by layering the n-dimensional and then the m-dimension algorithms again, with much smaller blocksizes (generally 4-8). The inner-most suboperation is sized to bring A, B, and C elements into registers, compute the `Gemm` operation, and then write the result back to C in main memory. This inner-most suboperation is called the *microkernel* in BLIS. Within a macrokernel, a small portion of B is pulled from the L3 cache into the L1 cache and remains there for microkernel calls.

In the GotoBLAS, macrokernels were assembly-coded for particular architectures. In BLIS, a macrokernel is architecture-agnostic and it is the microkernel that is specialized. Thus, somebody porting BLIS to a new architecture only needs to implement the relatively small suboperation in architecture-specific code to get good performance[2] This makes BLIS much easier to port [65]. Further, with architecture-specific details only in the lowest layers of code, BLIS is both easier to understand [65] and to encode as DxTer knowledge.

[2]The microkernel is implemented by layering more of the FLAME-derived algorithms and applying standard loop transformations like unrolling.

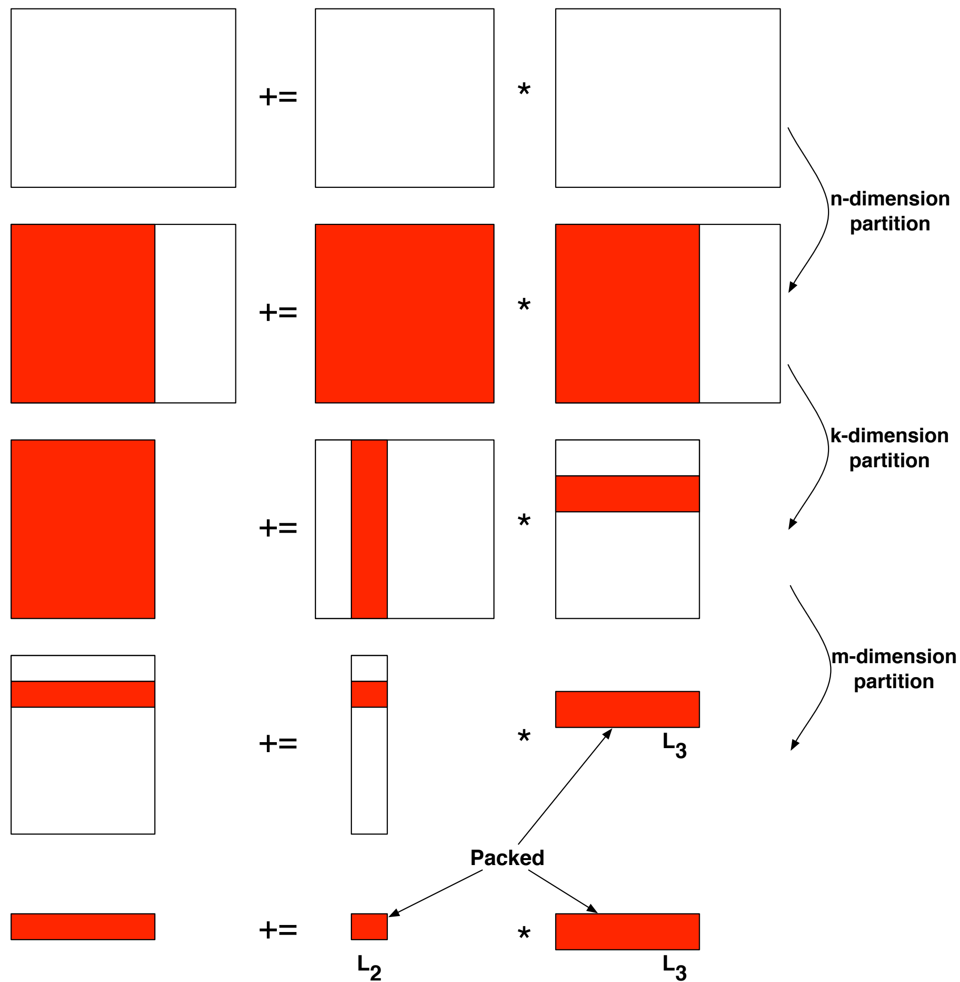

Figure 6.5: Partitioning of `Gemm` in BLIS.

BLIS was developed to reduce the number of specialized functions used to implement each operation (and reduce the number of functions to port to a new architecture). The result is fewer primitives for which to encode knowledge in DxTer. From the experience of developers, the time required to port BLIS is hours to days instead of weeks, as with the GotoBLAS.

6.1.2 Packing

One additional piece of design knowledge is essential to performance. Cache eviction policies are very sensitive to the structure of data being read/written. Data is often stored in memory such that there is a large stride to access each successive element. This would lead to the data being evicted from cache and reread from main memory even if the same submatrix is reread in each iteration of a loop. To avoid this, data is *packed*. A physically-contiguous piece of memory [29, 62] is used as a *buffer*. The portion of B reused within a rank-k update or the portion of A reused within a macrokernel is packed into the buffer. This means the data is read from main memory and written to the buffer in a different format [29, 62]. Roughly, packing transposes data as needed and puts every 4-8 elements of the data together in the buffer. This allows the microkernel to pull contiguous elements from the buffer into registers. A macrokernel accesses data carefully so the B buffer stays in L3 and the A buffer stays in L2.

Data movement like this seems inefficient, but there is an $O(n^3)$ computation performed on $O(n^2)$ data. The cost of packing is amortized across computation (the data has to be read from main memory at least once anyway) and the benefit of keeping data in cache improves performance considerably – so data is pulled from the cache and not pulled from main memory on each read [29, 62][3]. C is not packed because it is read less often by the microkernel and a well-implemented microkernel hides the cost of reads and writes behind computation. The packing operations are architecture-agnostic in BLIS and are not specialized. We consider them primitives in our work.

Other BLAS3 operations are implemented in a similar way because all of

[3]If the L3 cache does not exist on a target architecture, the cost of rereading data from main memory is still decreased considerably after packing B.

their algorithms have loop-body suboperations that are either the operation itself or the operation and `Gemm` (as with Figure 5.1). Thus, implementation knowledge about `Gemm`, like what we explained above, is used throughout BLAS3 operations. Further, knowledge about layering loops for the n, k, and m dimensions (in that order from top down) is repeatedly applied for all BLAS3 operation versions.

Some operations (e.g., `Trmm`) require a special macrokernel that is built on the `Gemm` microkernel. `Trsm` requires a special microkernel [62]. Others use the `Gemm` macrokernel by specially packing data. For example, `Symm` computes $C := \alpha AB + \beta C$ like `Gemm`, but A is symmetric. That means that either the data above or below the diagonal are not stored since it is the same as the data across the diagonal. When packing A for `Symm`, data are explicitly copied from across the diagonal, so it is in the form expected by the `Gemm` macrokernel. Most other operations can similarly copy data in a special way or zero-out data for specially-structured matrices and then use the standard `Gemm` kernels. `Trmm` is implemented by zeroing-out data above or below the diagonal for the triangular matrix for small blocks along the diagonal such that the `Gemm` microkernel can be used. This reuse of computation kernels leads to a significant improvement in porting productivity. One only needs to implement the `Gemm` and `Trsm` microkernels to attain high performance for all BLAS3 functionality.

6.1.3 DxTer Encoding

For each version of the BLAS3 operations, FLAME provides a family of derived algorithmic variants. Each variant partitions computation in one or more dimensions, so algorithms are layered to decompose in the n, k, and m dimensions, in that order, just like `Gemm`. The inner most computation is then properly sized for macrokernels ($b_m \times b_k \times b_n$).

To generate all versions of the BLAS3 operations, DxTer was augmented with new node/box types like the packing operations and macrokernels. Further, loops were extended to output not just Elemental-style loops (with Elemental-style submatrix partitioning) but also BLIS-style loops.

Lastly, the Elemental BLAS3 nodes/boxes were extended to represent BLIS operations. We did not need to develop an entire node class hierarchy of BLAS3 operations since it already existed for Elemental. Existing Elemental nodes were augmented to hold a label to specify if the node is an Elemental, a BLIS, or another flavor of BLAS3 node (to enable future work). This is when the layer labels discussed in Section 3.3.1 were added. Transformations could then be applied to nodes with specific labels (e.g., BLIS-specific `Gemm` transformations).

Additionally, the nodes' graph-to-code functionality was augmented to output BLIS code. As nodes were reused, we could utilize all of the algorithm refinements enumerated in Figure 5.17. The transformations were easily templatized using layer labels to specify if they work on Elemental or BLIS node types. We expect the work updating the nodes and transformations is largely a one-time effort because they are now templatized to support other libraries. In the future, just the output code behavior needs to be added for a new target DSL (e.g., for GPUs).

With these changes, DxTer generates code for all BLAS3 operations. Figure 3.4 list the versions of BLAS3 operations. Only 36 of those needed to be implemented since BLIS uses a form of templatization so the real-datatype code works for complex datatype operations, too. All 36 pieces of generated code are effectively the same as hand-developed. Hand-developed code has a different style than DxTer produced code, with a negligible difference in performance between the two approaches. Note that there is not much of a search space as the preferred implementation for

each operation is known in terms of loop layering, use of primitives, and so forth.

6.2 Parallelizing for Shared Memory

Multithreaded CPUs have the same levels of cache as sequential, but more threads. Figure 6.6 visualizes this with two-way sharing at each point. Instead of talking about a core, we now talk about threads (there may be multiple threads per core). There can be multiple threads sharing a single L1, multiple L1 caches per L2, multiple L2 caches per L3 (if it exists), and multiple processors sharing main memory.

In each case, there is communication among lower-level resources – threads sharing an L1 cache can communicate directly or through the L1 cache. Details of how this communication works are generally lower level than a BLAS developer considers, so we do not discuss them further. If two threads access the same memory, the hardware handles communication between resources. The developer only explicitly considers locking to prevent race conditions and which resources are shared (and therefore cost less to access together).

6.2.1 Parallelization Heuristic

Previously, we described how we want a block of A to stay in L2, a panel of B in L3, and a small portion of B to stay in L1. This is known to reduce data communication between main memory and cache layers [29]. In [57], BLIS's `Gemm` is parallelized in a way that tries to keep data in the same levels of cache. All of the threads that share that cache also share the stored data to complete the computation together. The idea is that one wants to reduce data movement between caches (analogous to reducing data movement between processes with distributed-memory computing) using a heuristic to guide how computation is parallelized.

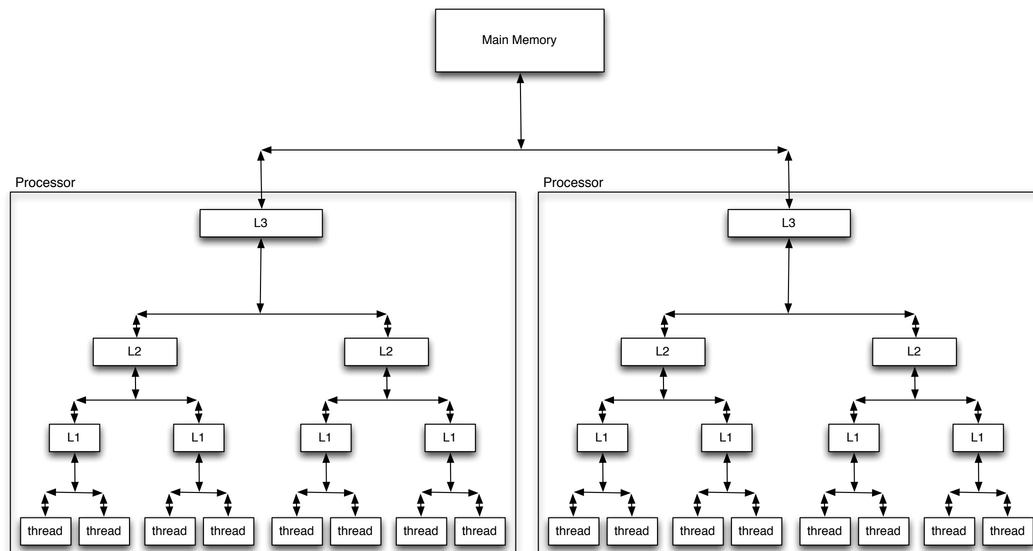

Figure 6.6: Cache and memory structure for a multicore CPU.

A *heuristic* here is an experienced-based way to design software. It is not guaranteed to lead to the best design, but in this case it is the way a knowledgeable developer chooses to implement algorithms without exploring and testing all possibilities. The heuristic we present below is a way to achieve good performance, but it will have to be adapted for other architectures, especially future many-core systems.

This type of parallelism is similar to that of Elemental: *data parallelism*, where each thread performs the same operation (i.e., running the same code) with different data. To accomplish this, we take the sequential code above and parallelize the loops such that different threads (or groups of threads) get disjoint portions of the loops' iterations. We now walk through the layers of algorithms and describe which are parallelized and to what degree with this heuristic.

Note that the algorithm of Figure 6.3 has a dependency between iterations.

The C matrix is both read and written in each iteration. This means if the loop is parallelized (e.g., two threads compute disjoint iterations), there must be a mutual-exclusion lock on C to prevent a race. This serialization limits scalability of parallelism. With the other algorithms, each iteration of the loop accesses a different portion of C, so they have no such dependency.

We now consider the two remaining `Gemm` algorithms for parallelization. In Figure 6.4 (the outer `Gemm` loop), the B and C matrices are partitioned in the n dimension. In each iteration, B is further partitioned in the k dimension by the inner loop and the $b_k \times b_n$ submatrix is packed and meant to stay in L3. Following the heuristic, we want all of the threads sharing the same L3 (i.e., all the threads on a processor) to use the same portion of B.

This means that we can parallelize the n-dimension loop such that for an iteration, all threads on a single processor (i.e., one L3) are assigned that iteration. Then, the k-dimension loop is run on all threads concurrently. Within that loop, the threads on the processor work together to pack the same portion of B and then work together to perform the rank-k update. We divide iterations of the n-dimension loop evenly between all of the processors.

We now have a portion of the `Gemm` that each processor must complete. We are left with the m-dimension loop (Figure 6.2) around a macrokernel. In each iteration of that loop, a portion of A is packed and meant to stay in the L2 for macrokernel execution. This means that all threads that share an L2 should work together to perform each iteration of the m-dimension loop. They work together to pack A and then work together within a macrokernel. Thus, we parallelize the m-dimension loop such that the L2s on each processor are given a roughly equal portion of the m-dimension loop's iterations.

Within a macrokernel, there are n and m-dimension loops. The n-dimension loop pulls a small portion of B into L1 and calls the microkernel. This loop can be parallelized across L1 caches. The m-dimension loop within can be parallelized across threads sharing the L1 cache.

6.2.2 Communicators

To parallelize sequential `Gemm` as described above, threads that share a resource need to be able to communicate with each other (e.g., share data, locks, and barriers). Further, we need to be able to divide iterations into portions for the various resources. Think of this as having the entire n dimension of the `Gemm` problem broken up based on how many processors there are. To enable this, we use hierarchical thread communicators based on the structure of Figure 6.6. A *hierarchical thread communicator* is a structure that keeps track of threads involved in each communicator so communication can take place and each communicator is aware of how many sub-communicators it has. This idea comes from communicators in MPI, which partitions processes into groups to enable group-wide communication [58].

Figure 6.7 shows the hierarchical communicator structure that mirrors the architecture of Figure 6.6. Each communicator enables communication across all of the threads it includes (e.g., *ProcComm* communicator includes all threads on the processor as shown in Figure 6.7). Further, there is a function that takes the size of a matrix dimension (e.g., n) and the communicator over which to parallelize (e.g., *GlobalComm*, which includes all threads on the system), and splits up the length of the dimension into equal chunks over which each sub-communicator (e.g., `ProcComm`) is responsible for computing. If there are no sub-communicators for a communicator used to parallelize (e.g., if there are not multiple processors), there

is no parallelization for the loop.

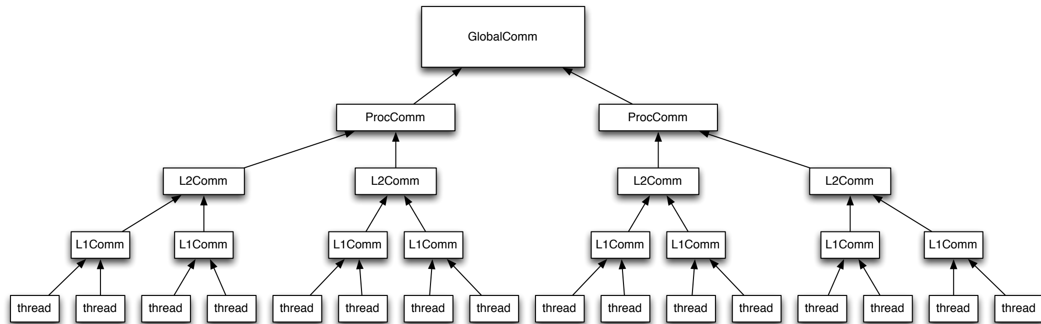

Figure 6.7: Communicator layout to mirror the architecture of Figure 6.6.

6.3 Encoding Multithreaded Parallelization

Once the heuristic for `Gemm` parallelization was developed, we went about encoding optimizations to parallelize sequential `Gemm` code with DxTer. DxTer produces the same code as before and then optimizations tag loops, packing operations, and macrokernels with communicators to parallelize them. Three new transformations were added to parallelize the n and m-dimension loops and the macrokernels. The output code for the loops and the primitives needed to be changed, which was easy. Loops and the primitives were extended to include a communicator tag, which denotes if and how they are parallelized.

Cost estimates were updated for parallelized loops and primitives by only counting $\frac{1}{p}$ of the iterations or computation, where p is the number of threads in the communicator. A small penalization was added per thread for a barrier (so DxTer would avoid expensive barriers among many threads). Further, there is a penalization when a buffer is packed among many threads across the targeted cache since there is an execution-time cost to bring the data together on all caches.

Algorithm: $[B] := \text{TRSM_LLN}(L, B)$

Partition $L \rightarrow \left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$ **where** L_{TL} is 0×0, B_B is $n \times 0$

while $m(L_{TL}) < m(L)$ **do**

Repartition

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

where L_{11} is $b \times b$, B_1 has b rows

$B_1 := L_{11}^{-1} B_1$ (`Trsm`)
$B_2 := B_2 - L_{21} B_1$ (`Gemm`)

Continue with

$$\left(\begin{array}{c|c} L_{TL} & L_{TR} \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} L_{00} & L_{01} & L_{02} \\ \hline L_{10} & L_{11} & L_{12} \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right)$$

endwhile

Figure 6.8: Variant of `Trsm`: left, lower, non-transposed and the DxT representation of the loop body.

Not all loops can be parallelized. For example, the k-dimension loop for `Gemm` cannot be parallelized except if locks are added. Further, some FLAME-derived algorithms cannot be parallelized at all because there is a dependency between one iteration's results (outputs) and the next iteration's operands (inputs). The `Trsm` `RLN` algorithm of Figure 6.8 cannot be parallelized since there is a dependency carried on the B_2 submatrix. DxTer could be augmented to perform dependency analysis to determine if a loop could be parallelized. This is a solved problem [41, 43] for DLA. As DxTer is meant to be research vehicle, we took an easier route.

With FLAME and the work of [41], we can easily determine if a given algorithm has independent iterations (meaning they can be parallelized). It takes less than a minute to prove or disprove independence of iterations formally by hand for any algorithm [41]. Therefore, we manually did this for each BLAS3 refinement and tagged loops if they could be parallelized. For now, this is a form of expert knowl-

edge we encoded manually. It was an analysis that a person would have performed manually if developing code by hand. Now it is encoded once and reused.

6.3.1 Quick Results

With these changes, DxTer-generated parallel `Gemm` code just as an expert developer. At the time of our work, the heuristic had been manually applied only to the four versions of `Gemm`, so BLIS did not include parallelization for any other BLAS3 operations. DxTer was able to parallelize all operations immediately, and two correctness bugs (in the DxTer knowledge base) were quickly discovered when testing parallelized output code.

For the first error, note the dependency between the two loop-body operations in Figure 6.8. The output code of DxTer was such that multiple threads were working on the first and second operations. Some threads might finish the first operation before others and start on the second with incomplete results. The solution was for DxTer to add a barrier in such situations. A *barrier* takes a communicator (in this case, the communicator over which the first operation is parallelized) and holds the communicator's threads in wait until all of the threads reach the barrier, meaning that all threads have completed the work before the barrier.

The second error was the result of some loops not being parallelizable. For some versions of `Trsm` (four of the eight), the n-dimension loop cannot be parallelized because the iterations are not independent. As desired, DxTer did not parallelize these loops, but the output code was still wrong. The implicit assumption was that all communicators in the hierarchy would be used. Therefore, the threads at the bottom of Figure 6.7 would each have different pieces of data. When the outer-most loop did not partition data among the `ProcComm` communicators, this assumption was

not valid. Within a processor, each thread had different data, but each processor was performing the same computation, which created race conditions. The solution was for DxTer to inspect the loops and primitives nested within the loops to make sure all communicators in the hierarchy were used. If they were not, then additional code was added to only have the *root* group perform the computation (e.g., only the first `ProcComm` would perform the computation).

These two errors did not happen with `Gemm` because it did not have similar dependencies. The errors were quickly discovered when testing DxTer output code and easily fixed in one place such that DxTer would then generate correct parallel code for all BLAS3 operations, not just the single operation tested and fixed. Thus, DxTer generated code that people did not yet have the time to implement by hand.

6.3.2 DxTer as a Productivity Enhancer

At this point, the heuristics for `Gemm` were not quite sufficient for high-performance implementations of some BLAS3 operations. When loops are not parallelized and only the root group performs computation, threads sit idle. New heuristics were needed to improve parallelism. In these cases, DxTer was used to experiment with new heuristics for parallelization. Each time new heuristics were added, all BLAS3 operations could be re-generated with the new heuristics (in less than a minute) to evaluate how they impacted all implementations. This is a form of performance validation while developing software (i.e., developing new parallelization schemes) and is a productivity enhancer

What were these new heuristics? First, since some loops could not be parallelized, a new communicator hierarchy was developed, shown in Figure 6.9. When the n-dimension loop is not parallelized, this hierarchy enables parallelization of the

m-dimension loop across all L2s instead of only parallelizing across the root processor's L2s. Also, when the m-dimension loop cannot be parallelized by `ProcComm` (i.e., across L2s), a macrokernel within is parallelized by `ProcComm` instead of by `L2Comm`. These heuristics applied to most of the 16 versions of `Trsm` and `Trmm`. When added, DxTer verified that this heuristic is no better than the `Gemm` heuristics on other BLAS3 operations because of associated data movement cost between processors (performance was re-validated for all operations).

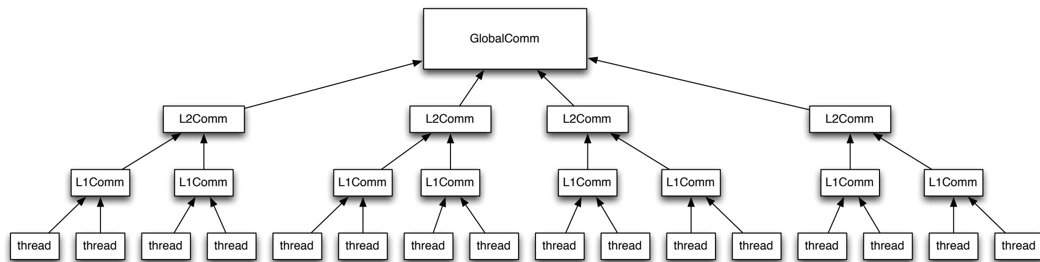

Figure 6.9: Alternative communicator layout that skips the `ProcComm` layer.

Some operations do not have the same amount of computation for all columns in the matrix. Symmetric rank-k update (`Syrk`), for example, updates a triangular matrix. A lower triangular matrix has more rows in the left-most columns than in the right-most columns. Partitioning such that all processors get an equal number of columns means one processor gets significantly more work. The solution is to partition work in non-equal portions for such operations. This is a simple optimization to implement, where a parallel loop is tagged to use non-equal partitioning for parallelism. This applied to the eight versions of `Syrk` and `Syr2k`.

Operation by operation, generated BLAS3 code was manually inspected. When an expert BLAS3 developer had new idea for optimizing code, it was added to DxTer. All BLAS3 code was then re-generated, so the idea would immediately be

evaluated in the context of all operations. Having optimizations immediately applied throughout all code was a great productivity enhancer and performance validator while developing novel code – DxTer allowed experts to explore implementation possibilities much faster than could have been done manually. Further, after the two initial bugs were fixed, all code was correct each time it was regenerated with new optimizations.

A side note on DxT – we found that decomposing implementations in terms of small transformations was useful. This explains how we improved existing implementations by adding new rules. There seems to be great pedagogical and practical value in doing so. For example, we explained multithreaded implementations via transformations in this chapter.

6.4 Performance Results

Our most important results come from how quickly we generated BLAS3 code and made far-reaching changes in that code by adding transformations to DxTer. As of this writing, BLIS still only has hand-coded, multithreaded implementations of the four `Gemm` versions. With DxTer, we were able to generate all other BLAS3 implementations quickly. Here, we present performance results of some of the real BLAS3 operations to confirm the quality of DxTer's output code.

The digitally enlargeable Figure 6.10 shows results with two Intel Xeon E5 (Sandy Bridge) octo-core processors. Each processor has one L3 cache, eight L2 caches, one L1 per L2, and one thread running on the single core attached to the L1. Each core has a peak performance of 21.6 GFLOPS, so the peak of all 16 is 345.6 GFLOPS.

The digitally enlargeable Figure 6.11 shows results from a system with four

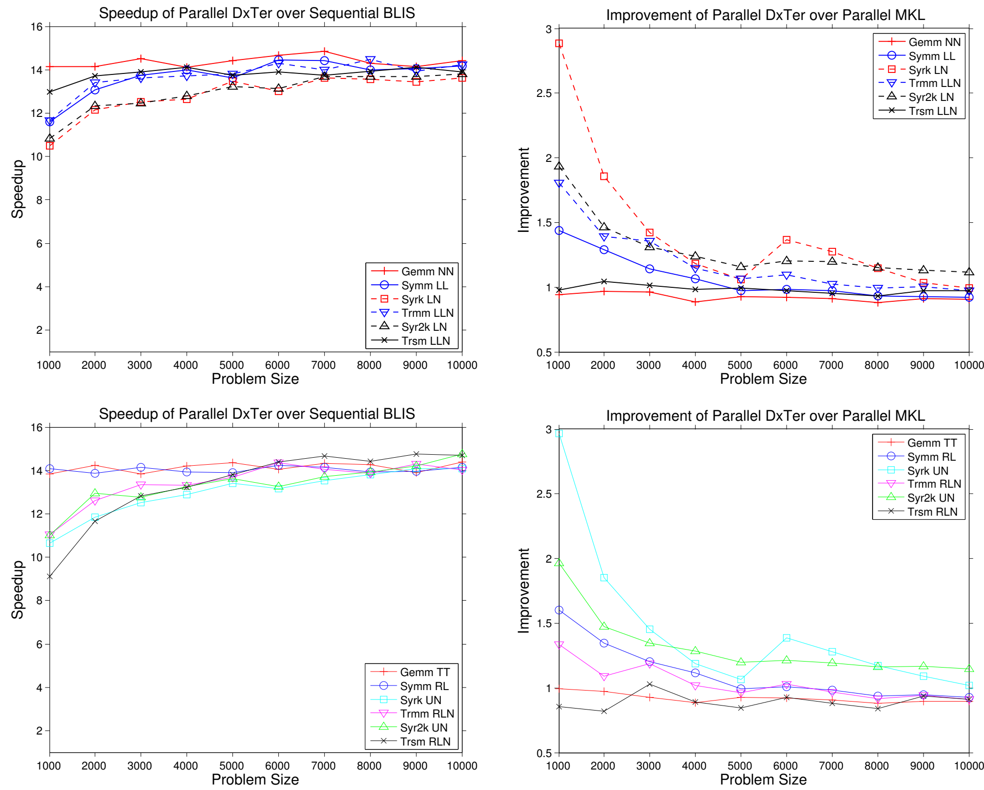

Figure 6.10: (Left) Speedup of DxTer-generated multithreaded code over sequential BLIS code. (Right) Improvement of DxTer-generated multithreaded code over MKL multithreaded code. The results are from 16 cores of Xeon E5.

Intel Xeon 7400 hexa-core processors. This contains four L3 caches, three L2 per L3, two L1 per L2, and one thread per L1. Each core has a peak performance of 10.6 GFLOPS, so the peak of all 24 is 254.4 GFLOPS.

We compare against Intel MKL [3] version 11.1, which is the trusted, high-performance, vendor-optimized BLAS library for these architectures. The two architectures, due to different memory hierarchies, require DxTer to generate different amounts of parallelism at various levels.

In the left-side graphs, we show the speedup running on all cores versus

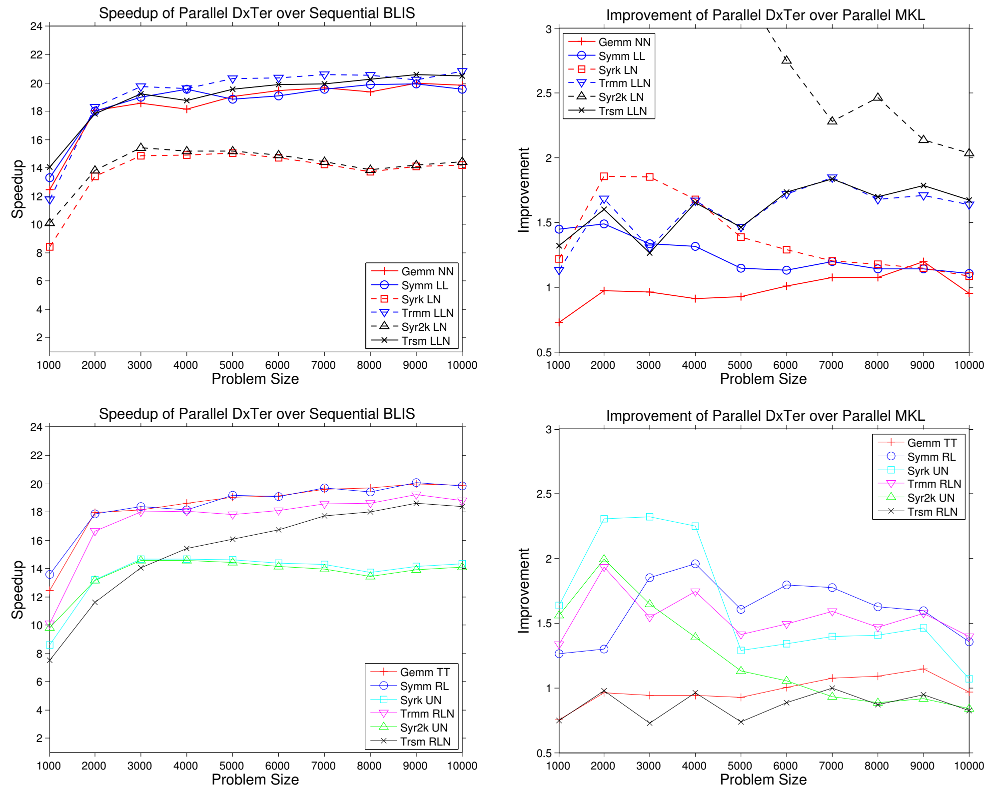

Figure 6.11: (Left) Speedup of DxTer-generated multithreaded code over sequential BLIS code. (Right) Improvement of DxTer-generated multithreaded code over MKL multithreaded code. The results are from 24 cores of Xeon 7400.

running on one for a sample of the real BLAS3 operations across a range of problem sizes. In the right-side graphs shows how that performance outperforms MKL (1 is the same performance and higher is better for DxTer). We only show two variants for each of the real-datatype BLAS3 operations because we had similar results for the others.

MKL performance of `Syr2k LN` in Figure 6.11 (top right) is very slow for small problem sizes, so DxTer's code's performance looks especially good by comparison. DxTer's `Syrk` and `Syr2k` code does not speed up well in Figure 6.11 (left). This is

likely a load-balancing issue that requires additional heuristics, which we will add to DxTer in the future. Still, these DxT implementations perform well compared to MKL.

The key here is that a developer added, piece by piece, new implementation knowledge (parallelizing optimizations) to DxTer and got parallelized code just as he would have created by hand. The difference is that the code is more trusted for correctness AND more trusted for performance since a system evaluated all of the implementation options for each operation.

6.5 Heuristics vs. Testing

For Elemental, we encoded all parallelization refinements, and DxTer used cost estimates to choose the best. For BLIS, we encoded heuristics that led to a limited number of parallelization options. Cost functions were also used, but they only had to differentiate between a small number of options. One might ask why did we do this?

First, there are simply fewer implementation options that an expert would consider. The heuristics developed with BLIS are based on expert decisions on what to consider. That is the knowledge we want to encode. There are valid implementation options that are known to be always bad, so they need not be encoded.

Further, the cost estimates for multithreaded implementations are not as good at rank ordering because smaller-order terms are more difficult to predict accurately. An expert often makes design decisions with a heuristic or simply justifies one implementation choice over another and applies it throughout a library of code. If improved cost functions are developed, they can be incorporated into DxTer and it

become even more powerful in searching through options with BLIS code generation.

In the future, we might encode all options. Then, the search space is larger and cost functions are less able to pick out the best implementation. Cost functions could still omit bad implementations (e.g., avoid idle cores or load imbalance). For the remainder of the search space, the code would have to be generated, compiled, and run for fitness evaluation. This is similar to auto-tuning approaches [63] or other code generation approaches [52] described in Section 1.5. Choosing an implementation this way is viable when a single run is short (unlike with distributed memory). There is still a benefit of using DxTer in this case since the expert would not have to create all implementations manually and he would trust the output code as functionally correct.

For now, though, the performance results and analysis of generated code by an expert developer show DxT is both powerful and practical as-is. In the future, we could explore these options for DLA or for other domains targeted for automation with DxT.

6.6 Summary

By exploiting layer-templatized refinements, algorithms encoded for distributed-memory targets were easily retargeted for sequential and shared-memory architectures. Adding some hardware-specific refinements, we could generate high-performance sequential BLAS3 code.

Then, we used DxTer as something of a high-level compiler for a software engineer developing parallelization schemes for shared-memory targets. Instead of requiring him to re-analyze and re-implement code manually with each new parallelization idea, the idea was encoded and added to DxTer as an optimization so

DxTer would perform the rote development work.

This study demonstrates the utility of code generation to aid a developer and the reusability of knowledge encoded in the DxT style.

Chapter 7

Conclusion

If I had asked people what they wanted, they would have said faster horses.

–Henry Ford

We have presented *Design by Transformation (DxT)* as a way to encode software-design knowledge. Our thesis was that *dense linear algebra (DLA)* algorithms can be encoded as graph transformations in the DxT style, which enables automatic code generation. We demonstrated this with distributed-memory and multithreaded targets for BLAS3 operations, among others. The main benefit, as we showed, is automatically generated code that is trusted for correctness and performance. Further, we demonstrated that transformations hold pedagogical value as we used them to explain the design decisions that lead to good code.

7.1 Contributions

We now summarize the main contributions of this dissertation.

7.1.1 A DLA Representation in DxT

We developed a representation of DLA algorithms and implementations in dataflow graphs. It is important to have a domain's software in a form that enables one to encode design knowledge. If making and implementing important design decisions in a DSL is entangled with minor details, the software is difficult to understand and encode. The DSLs we targeted do not have this problem. Beyond this, the representation of clean software and design knowledge about it in DxT is not obvious or trivial. The refined representation we presented is a significant contribution of this work.

Using templatized node types, we reuse basic domain algorithmic transformations across architectures. We also encoded architecture-specific transformations to target those algorithms to distributed-memory, sequential, and multithreaded architectures. This representation enabled us to generate code that either performed the same as or better than expert-developed code or to generate novel code that had not yet been developed.

We believe similar results can be shown for other architectures (e.g., GPUs) in the future by adding target-specific design knowledge and reusing much of the already-encoded knowledge.

7.1.2 A Prototype Generator

DxTer is a prototype to generate high-performance code. One inputs a graph representing functionality (e.g., a collection of domain-specific functions forming an algorithm) that DxTer is to implement in code for a particular architecture. Domain transformations (e.g., basic algorithms) and target-specific knowledge (e.g., parallelizing refinements and cost functions) are also input to DxTer to enable it to

implement and optimize desired functionality. DxTer enumerates a search space of implementations and rank-orders them based on performance estimates (runtime in the case of the DLA examples we show).

In our experiments with distributed-memory architectures, some operations led to massive search spaces that were too large for DxTer to generate given system memory constraints. We employed a number of tools to limit the search space intelligently using knowledge about the domain and the target architecture. These heuristics are reasonable when one considers how an expert searches the same space of implementations without exploring every single algorithm.

7.1.3 The Benefits of Encoding Design Knowledge

The biggest benefit we demonstrated by encoding design knowledge is better output code. In a number of cases, DxTer produced better-performing code than was hand-developed for Elemental. Further, it generated a range of functionality automatically when hand-coded versions did not yet exist.

In addition to these benefits that affect the user, there are aspects that should interest software engineers. DxTer-generated code is trusted for correctness (e.g., fixing a correctness bug in Elemental for one operation). This bug would have been discovered eventually via testing, but it is useful to have a system output highly trusted code automatically. Or when a library-wide bug fix needed to be made, it can be encoded in DxTer once and automatically applied across the library.

Further, by encoding designs in terms of transformations, we can better understand software. We explained algorithm implementations with Elemental and BLIS in terms of small, incremental steps. There are more intricate optimizations/changes made to the code in these libraries that were also encoded in DxTer

but not described here because the details are not important for this dissertation. By making them explicit in the DxTer knowledge base, though, they are now useful to non-experts wanting high performance code without becoming an expert and they are explicitly stored for posterity when a new engineer wants to become an expert.

This pedagogical aspect was especially useful when comparing two implementations and explaining why one is better than the other: comparing the transformations that yield each makes bad decisions easy to spot. We can see a future tool used by new developers to make implementation decisions with feedback from the tool explaining why particular choices are suboptimal.

Lastly, while encoding knowledge in terms of transformations, we required design decisions for the BLIS DSL to be justified. Some decisions led to inefficient code; these were identified by encoding the necessary transformations and questioning their effects or rationale. This led to a better designed DSL.

7.2 Future Work

DxT is not specific to DLA. It applies to domains representable as dataflow graphs. Nodes can have state, though they are stateless for DLA. Transformations can be very complicated if necessary, though simpler transformations are preferred. Search could be implemented with empirical testing instead of analytic estimates if cost functions are not sufficiently accurate. The key of DxT is to represent the starting algorithm and ending implementations as graphs and represent all design decisions in terms of transformations. Then, the derivation of an implementation is explicit, understandable, repeatable, and extensible.

It is this last piece that can lead to vast potential for DxT future work.

Adding new functionality to existing code is difficult, especially when it is unfamiliar and complicated code. If implementations are derived by an explicit series of understandable transformations, we believe extending functionality will be easier. Both the existing functionality will be more understandable, and changes and extensions to that functionality will be made in terms of transformations that are demonstrated (or proven) correct and are justifiable. This is more formal than standard practice: hacking an implementation until it does what you expect and it meets testing requirements.

DxT must be applied to many other domains to evaluate and demonstrate its generality. Tensors and fault tolerance in DLA software are domains related to what we have already done. The CombBLAS [15] are a collection of BLAS-like operations for sparse matrices that build the foundation for graph algorithms. They are structured similar to the BLAS, target multiple architectures, and have cost estimates, all of which are similar to properties that have made us successful in DLA.

For domains like DLA, we believe we have demonstrated great potential (given the right domain structure and DSLs). Not all domains have a similar structure with lots of functionality implemented with a relatively small knowledge based. Even for those domains, we believe there is potential. No longer will code be the result of hacking. Instead, it will be derived with pieces of trusted knowledge. That knowledge can be replayed, reused, and even extended.

7.3 Vision

We see the success of DxT applied to DLA as a predictor for future DxT success in other domains. We want to reach a point in software engineering where domain

and hardware knowledge is encoded explicitly instead of only storing the code that results from an expert tediously applying the knowledge over and over. We should have repositories of essential, expert software design knowledge ready for use just as an engineer has a catalogue of cogs to place in his machine.

The current approach of manually applying software design knowledge is error-prone, can lead to suboptimal code, and is often rote. We see automated code generation as a staple software engineering technology in the future. Further, we see encoding knowledge (à la DxT) as a means to making software engineering a more scientific endeavor. We should no longer hack software to accommodate changing architectures or algorithms. We should use a structured approach to modify and adapt code and we see DxT as a way to add such structure to software engineering.

Appendix A

Two-Sided Trmm

We now go through the transformations that produce a high-performance implementation of two-sided trmm, introduced in Section 1.1 and described in detail in Section 5.3.3. We start with the loop body of variant 4 (shown in Figure A.1), which is the highest performing variant (though all are searched as described in Section 5.3.3).

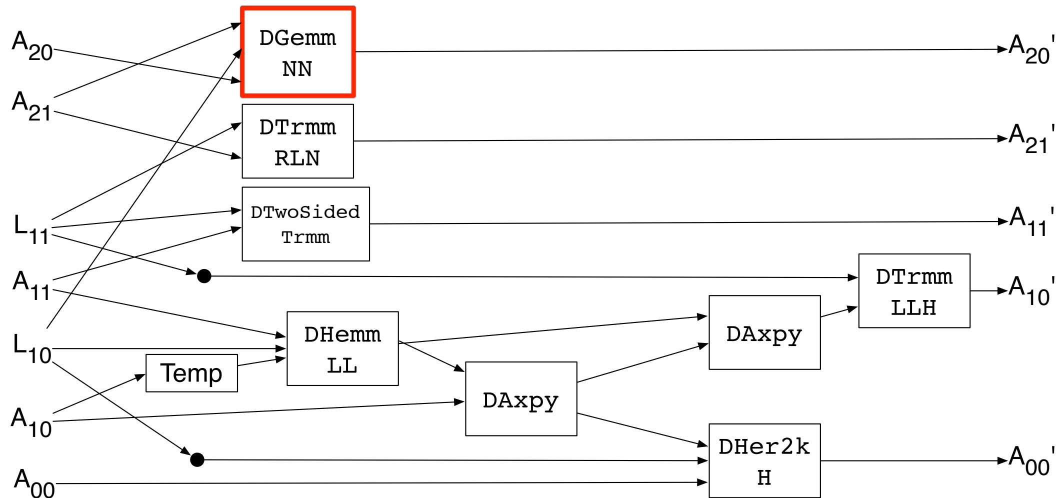

Figure A.1: Starting loop body of two-sided trmm.

In each figure of this appendix, we highlight with thick red borders the node(s) to which the next transformation applies. In this case, the first transformation is a refinement of `DGemm NN`, shown in Figure A.2, called "stationary C" because it does not redistribute the C matrix. The resulting graph is shown in Figure A.3. This refinement is one of those introduced in Section 5.1.2 to parallelize a `Trmm` algorithm.

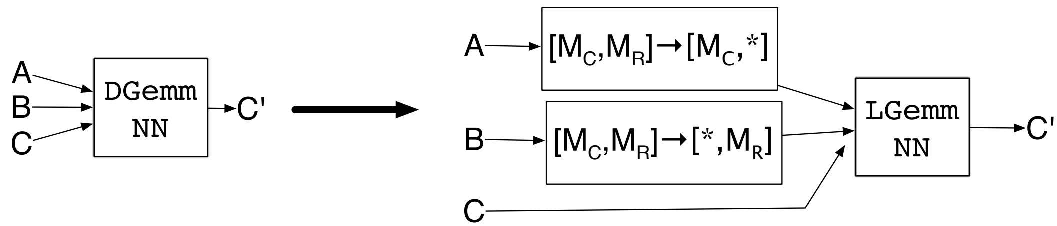

Figure A.2: Refinement of `DGemm NN`.

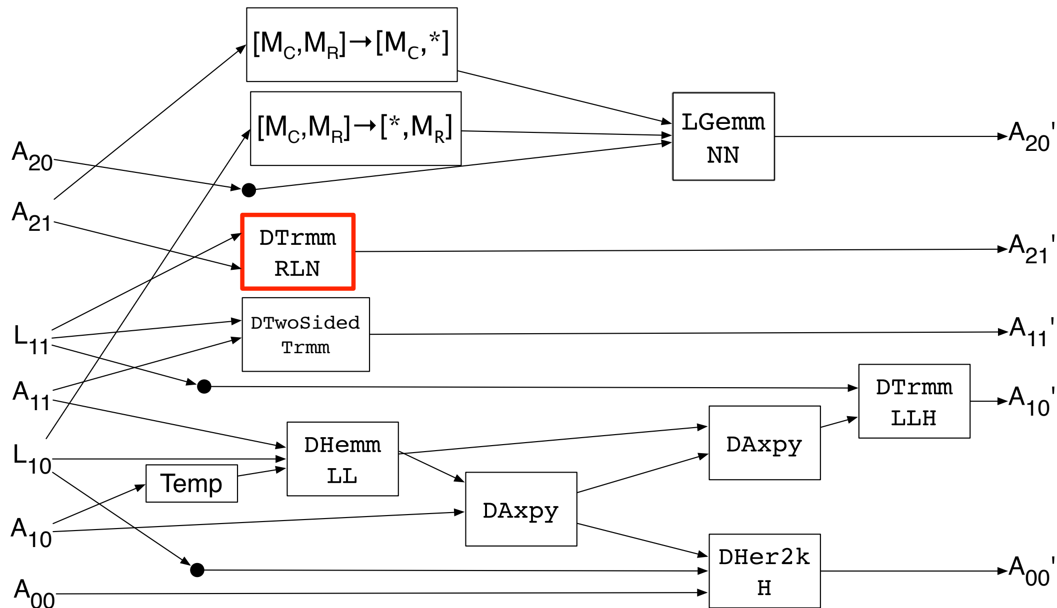

Figure A.3: Result of applying refinement of Figure A.2.

Next, the refinement of `DTrmm RLN` (Figure A.4) is applied to yield the graph of Figure A.5. This transformation was also used in the `Trmm` implementation of Section 5.1.2.

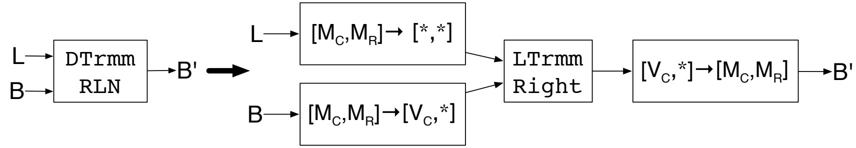

Figure A.4: Refinement of `DTrmm RLN`.

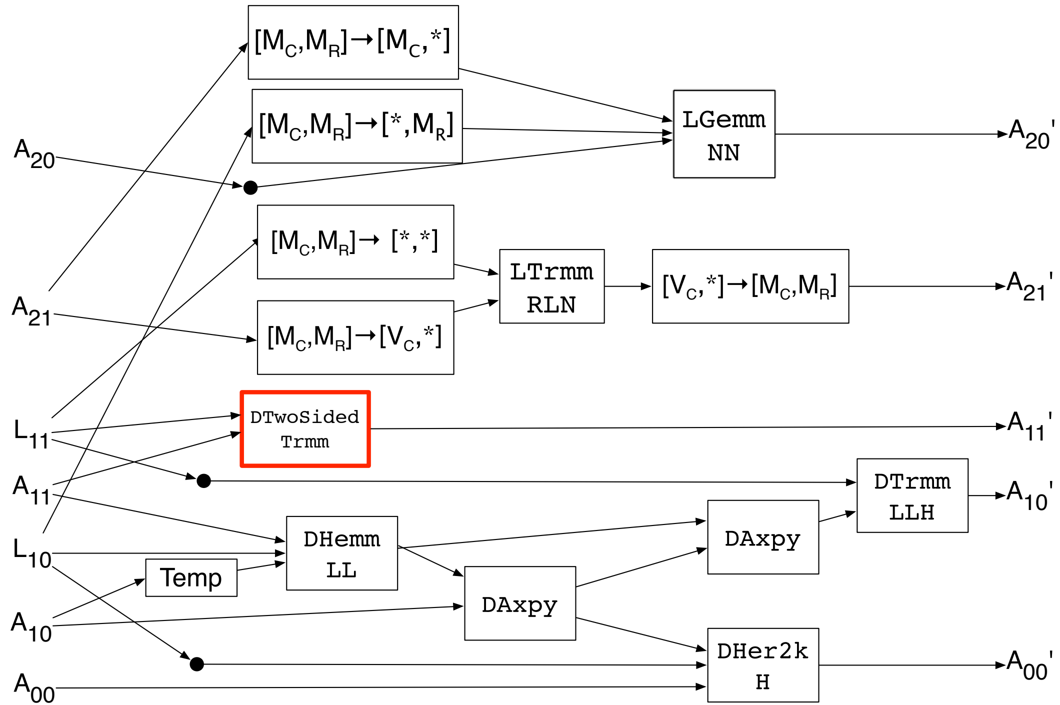

Figure A.5: Result of applying refinement from Figure A.4.

Now, we introduce a refinement in Figure A.6 that is unique to this operation. It is the same as the `DChol` refinement (see Section 5.3.1) with "`DChol`" replaced with "`DTwoSidedTrmm`." This gathers all of the data on all processes (via MPI `AllGather`) and performs the computation redundantly.

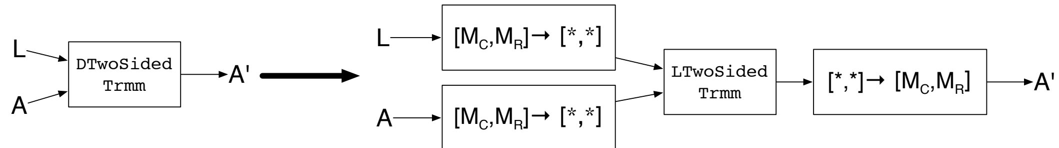

Figure A.6: Refinement of `DTwoSidedTrmm`.

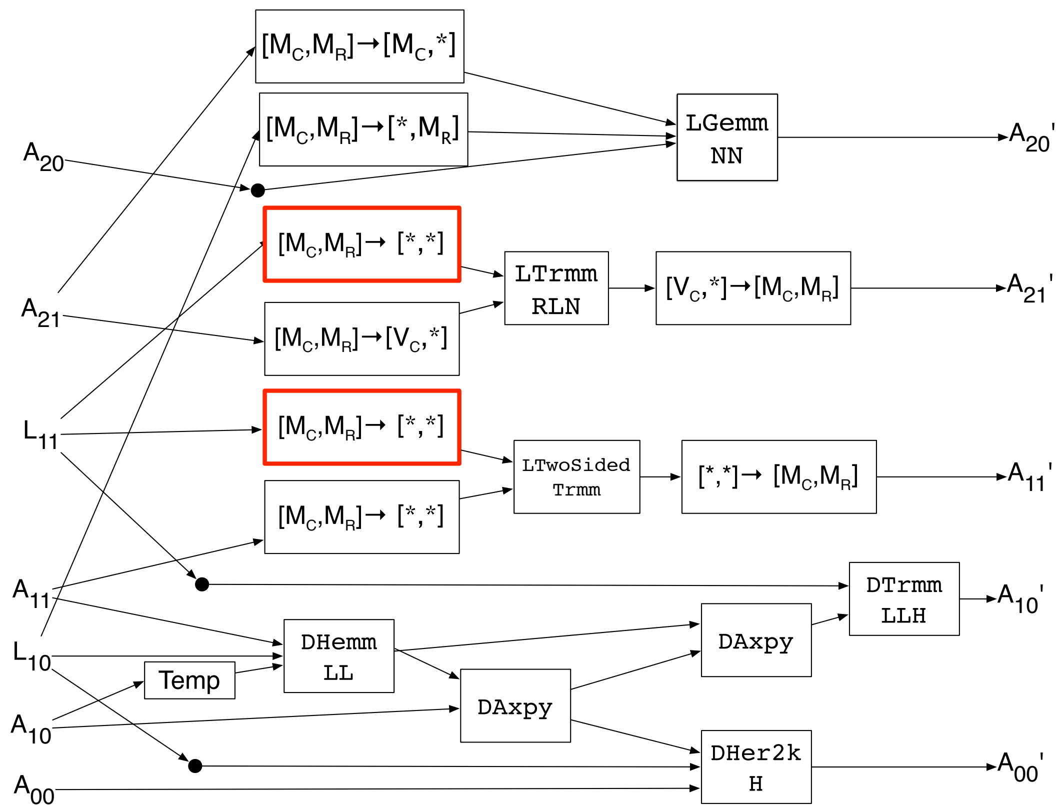

Figure A.7: Result of applying refinement from Figure A.6.

As highlighted in Figure A.7, the data of L_{11} is redistributed redundantly to $[*, *]$. We can apply the transformation of Figure A.8 to remove one of those expensive redistributions to yield the graph for Figure A.9.

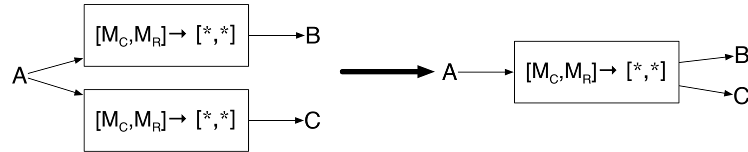

Figure A.8: Remove redundant redistribution.

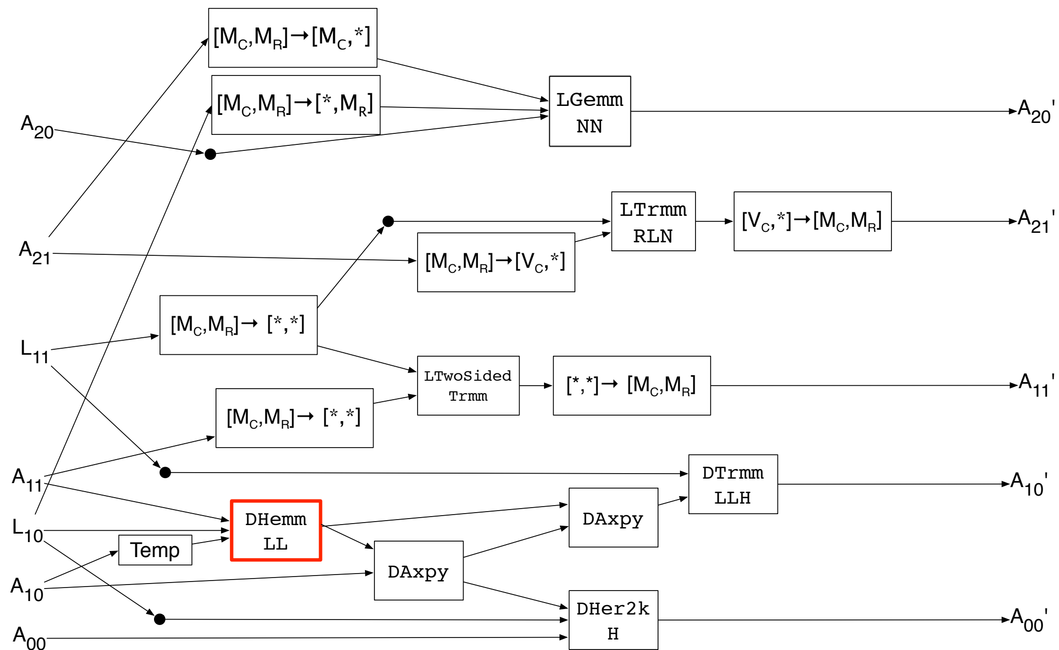

Figure A.9: Result of applying optimization from Figure A.8.

Now, we apply the refinement of Figure A.10 to `DHemm LL`. The result is shown in Figure A.11.

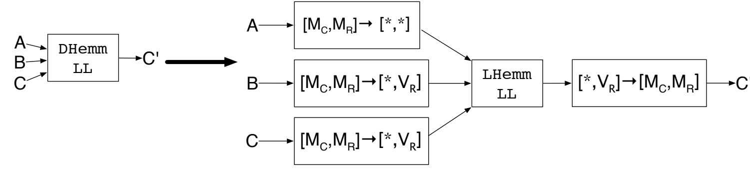

Figure A.10: Refinement of `DHemm LL`.

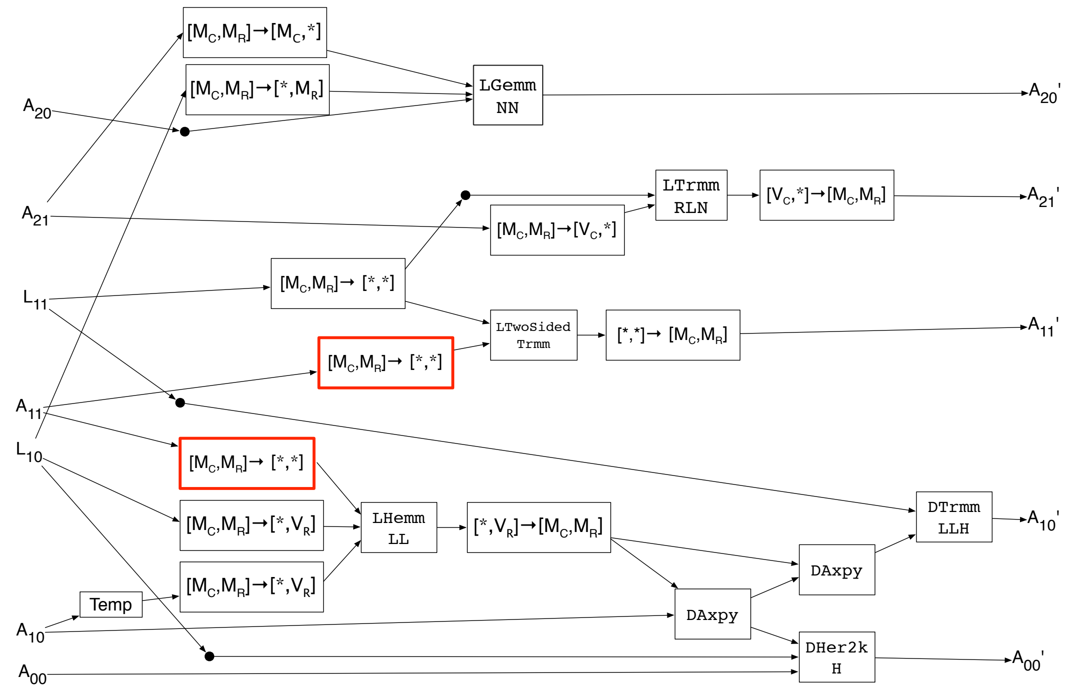

Figure A.11: Result of applying refinement from Figure A.10.

The graph in Figure A.11 has an inefficiency like that of Figure A.9, where data is redistributed to $[*, *]$ twice. Applying the optimization of Figure A.12 removes one of the redistribution to form the graph of Figure A.13.

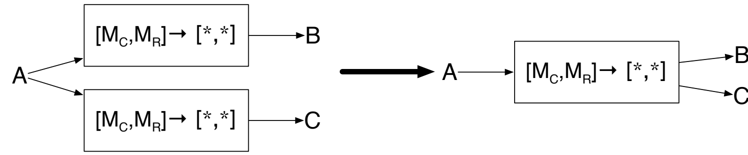

Figure A.12: Remove redundant redistribution.

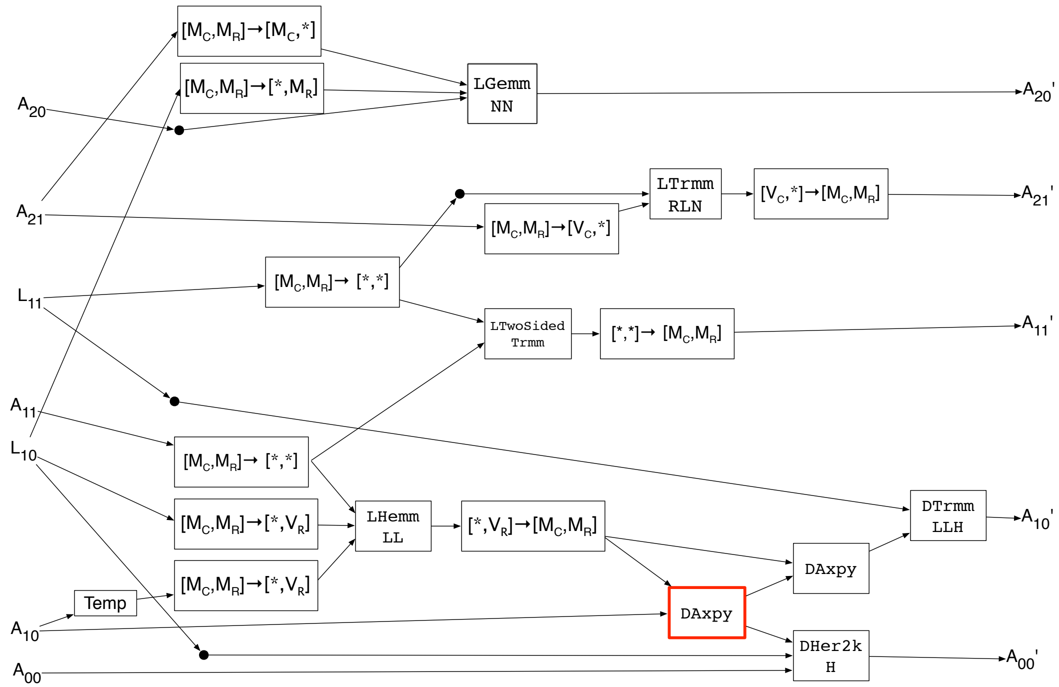

Figure A.13: Result of optimizing with transformation from Figure A.12.

Now, we need to refine `DAxpy`. We choose an option, shown in Figure A.14, that will enable many optimizations later. Figure A.15 shows the result.

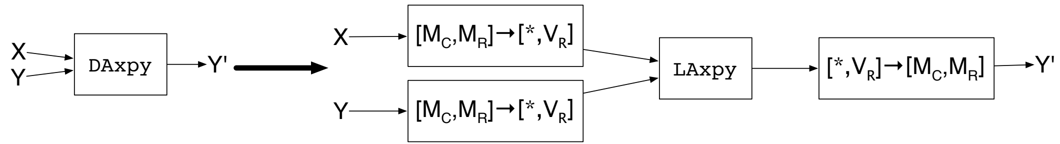

Figure A.14: `DAxpy` refinement.

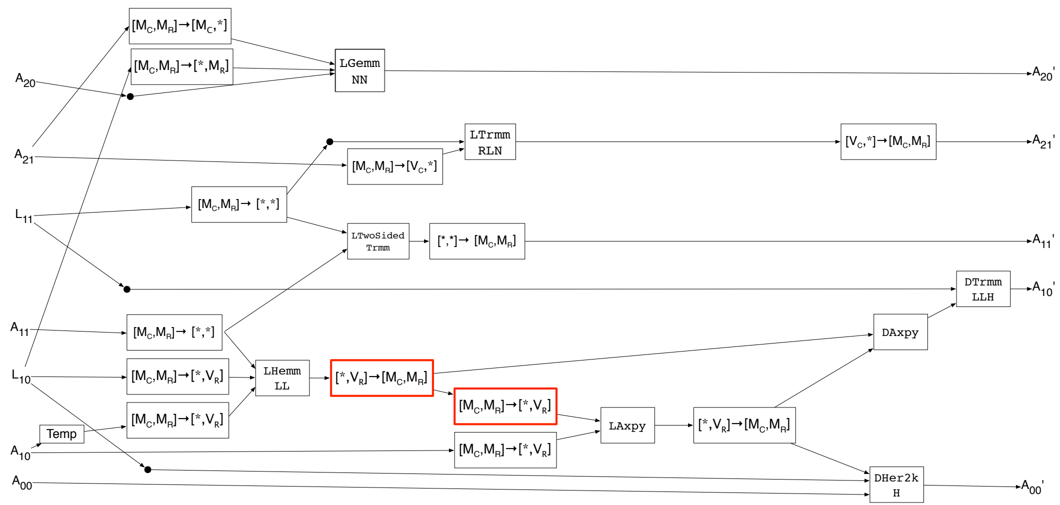

Figure A.15: Result of refining `DAxpy`.

Now, we have two redistributions that are inverses of each other. Data is redistributed from $[*, V_R]$ to $[M_C, M_R]$ and back to $[*, V_R]$. With the optimization of Figure A.16, we can remove the unnecessary redistribution back to $[*, V_R]$, which yields Figure A.17

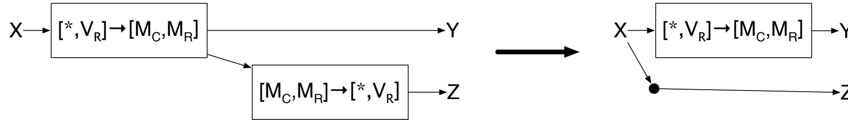

Figure A.16: Remove inverse redistribution.

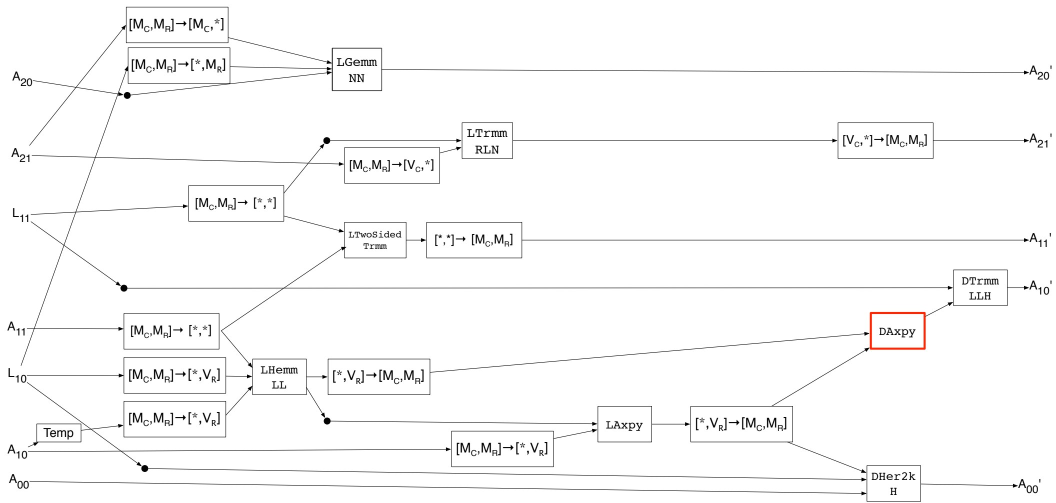

Figure A.17: Result of applying transformation of Figure A.16.

We can apply the same DAxpy refinement as before, shown again in Figure A.18, to form the graph in Figure A.19.

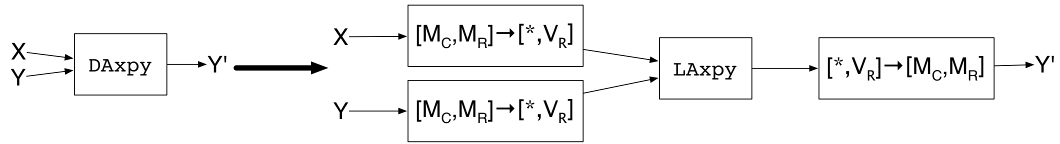

Figure A.18: DAxpy refinement.

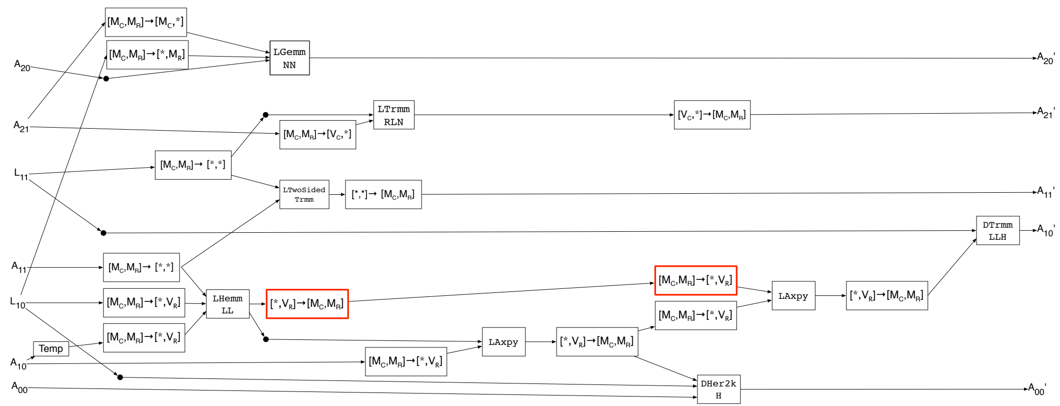

Figure A.19: Result of refining DAxpy.

We again have inverse redistribution as shown in Figure A.19. The difference from before is that the intermediate data (distributed as $[M_C, M_R]$) is not used, so Figure A.20 shows the optimization in this case. Figure A.21 shows the resulting graph.

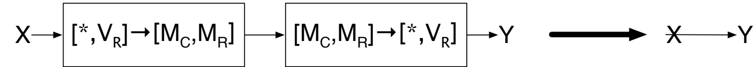

Figure A.20: Remove inverse redistribution.

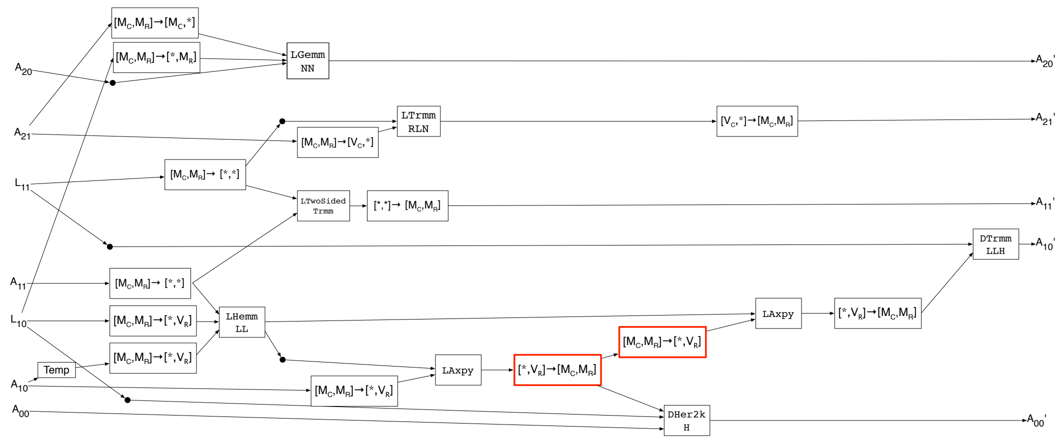

Figure A.21: Result of removing inverse redistribution.

And we have the same situation with inverse redistribution and use of the intermediate distribution. Reapplying the optimization, shown again in Figure A.22, yields the better implementation of Figure A.23

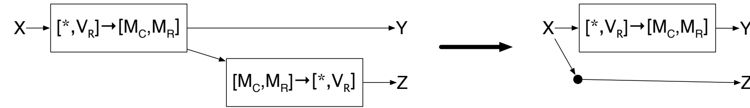

Figure A.22: Remove inverse redistribution.

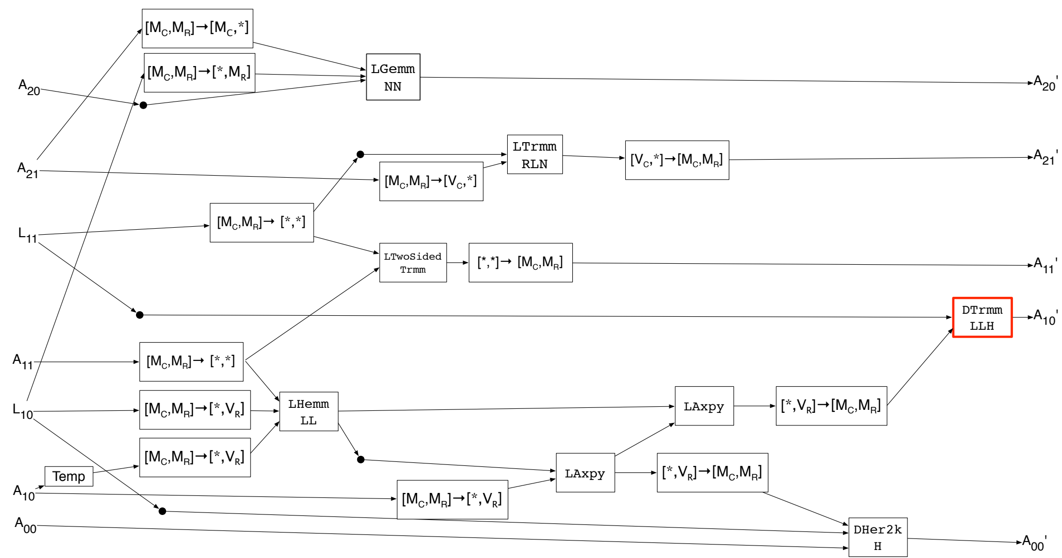

Figure A.23: Result of removing inverse redistribution.

Now, `DTrmm LLH` can be refined. The refinement is similar to that for `DTrmm RLN`, but now we use distribution $[*, V_R]$, as shown in Figure A.24. The result is in Figure A.25.

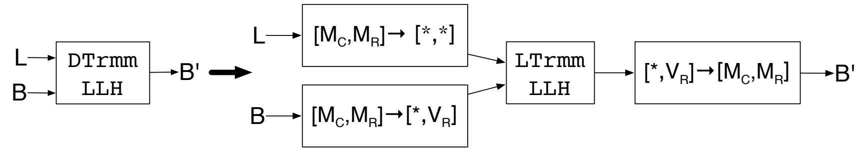

Figure A.24: `DTrmm LLN` refinement.

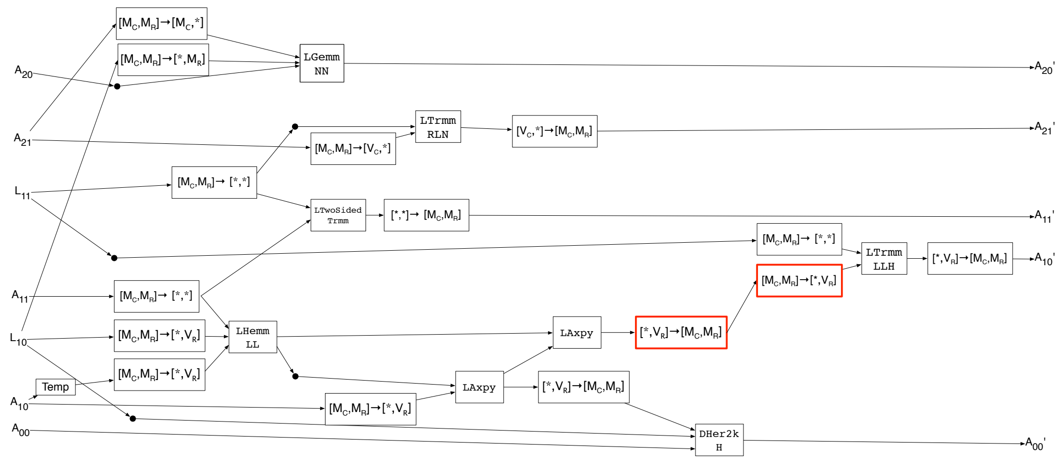

Figure A.25: Result of refining `DTrmm LLN`.

And once again we have inverse redistribution. Applying the optimization of Figure A.26, we get the graph of Figure A.27.

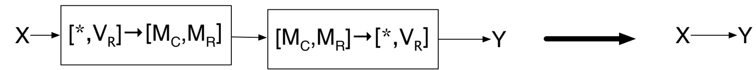

Figure A.26: Remove inverse redistribution.

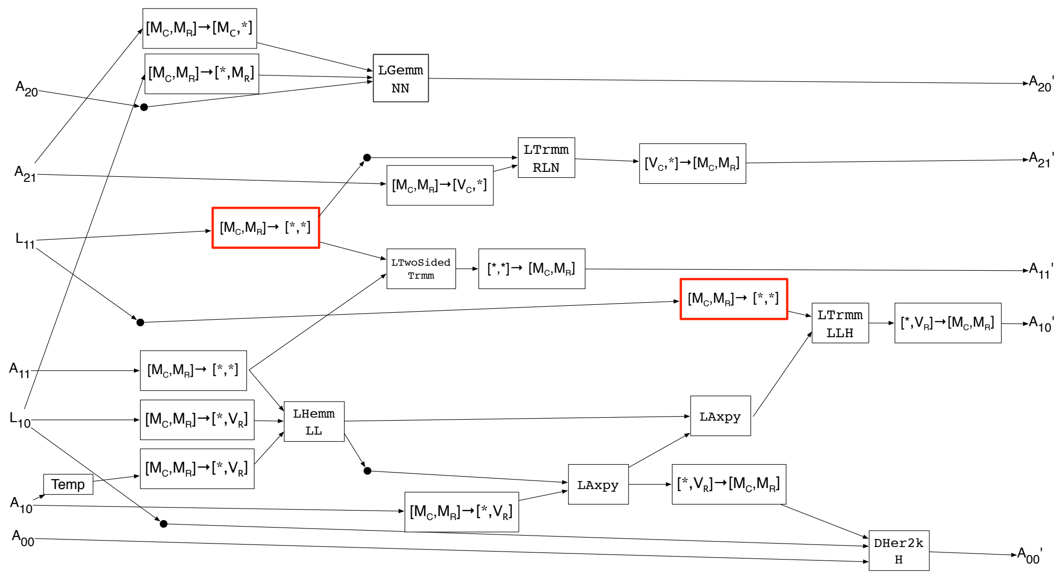

Figure A.27: Result of removing inverse redistribution.

Now we see L_{11} being redistributed twice to $[*, *]$. We can remove one of the redundant redistributions with the optimization of Figure A.28 to generate the better implementation of Figure A.29.

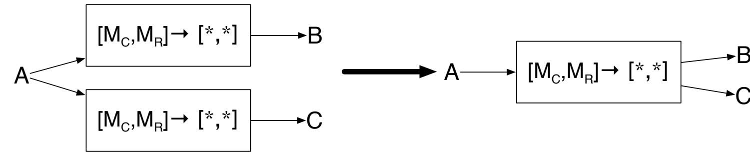

Figure A.28: Remove redundant redistribution.

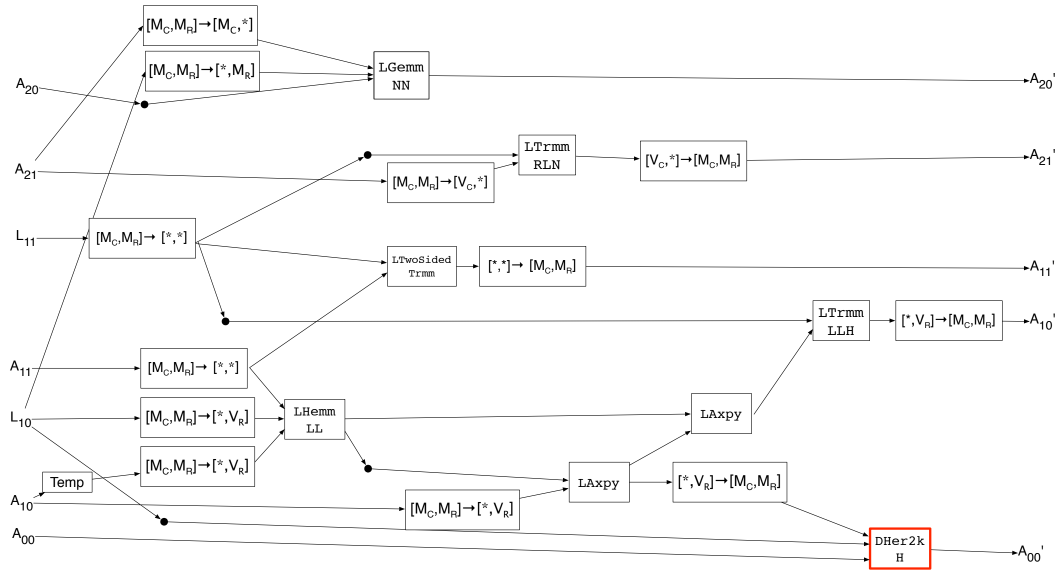

Figure A.29: Result of removing redundant redistribution.

Now we can refine `DHer2k H` with Figure A.30 to develop the graph of all primitives, shown in Figure A.31.

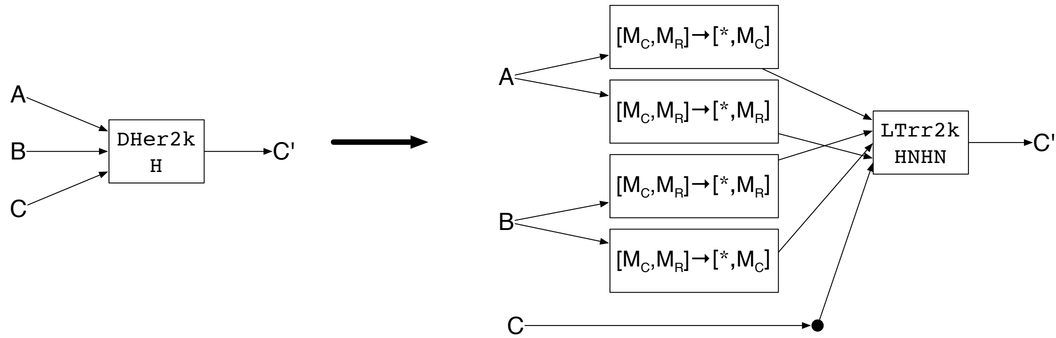

Figure A.30: `DHer2k H` refinement.

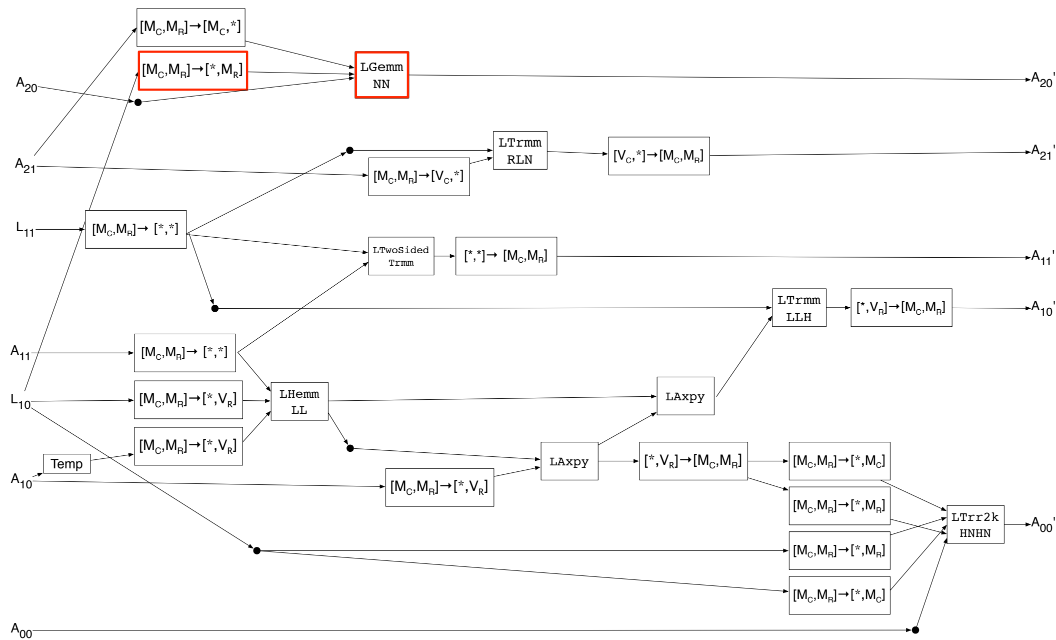

Figure A.31: Result of refining `DHer2k H`.

This implementation maps to Elemental code and was used in the Elemental library until DxTer found a better implementation that benefits from transposing data during communication and transposing again during computation. Figure A.32 shows an optimization to change the redistribution of the B input matrix to use what we call $[M_R, *]^H$, which is actually the $[M_R, *]$ distribution of B^H (i.e, the Hermitian-transposed data of B is distributed as $[M_R, *]$). Then, this data is Hermitian-transposed again in the `LGemm NH` operation to end with the same data as with $B[M_R, *]$. The difference is that $[M_C, M_R] \rightarrow [M_R, *]^H$ performs better due to improved data access.

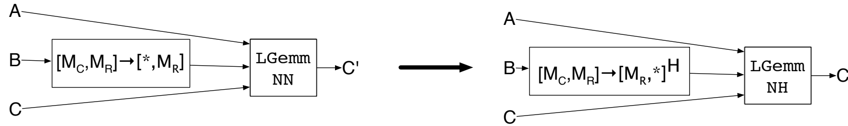

Figure A.32: Transpose redistribution of the B matrix used in an `LGemm` operation.

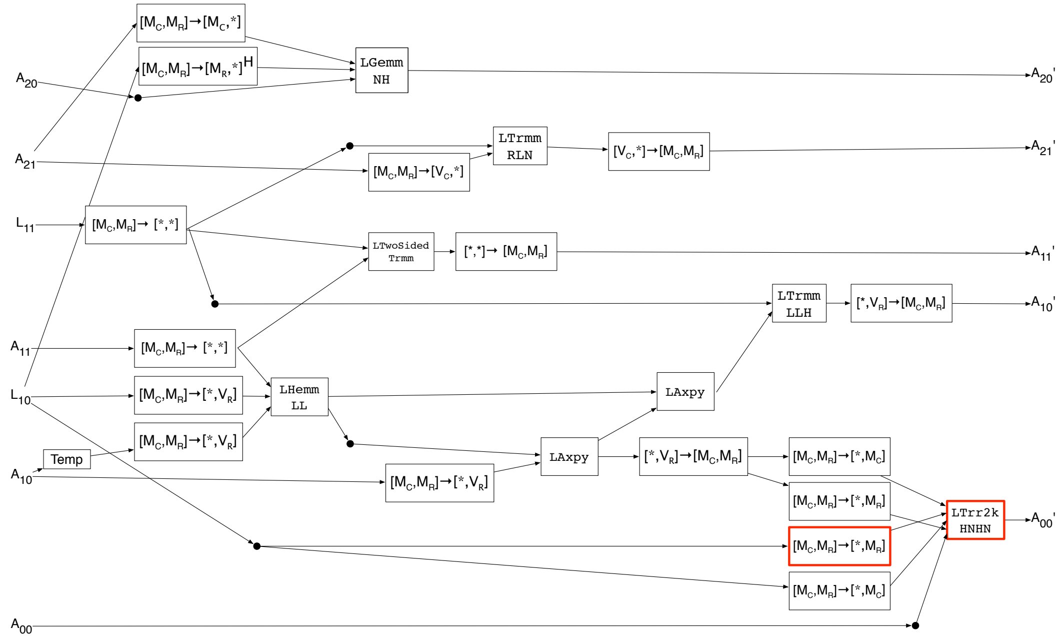

Figure A.33: Result of transposing one input to `LGemm`.

We can apply a similar optimization for `LTrr2k HNHN`, as shown in Figure A.34. The improved implementation is shown in Figure A.35.

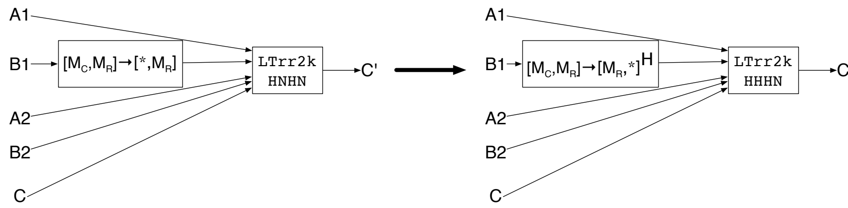

Figure A.34: Transpose redistribution of the $B1$ matrix input to `LTrr2k HNHN`.

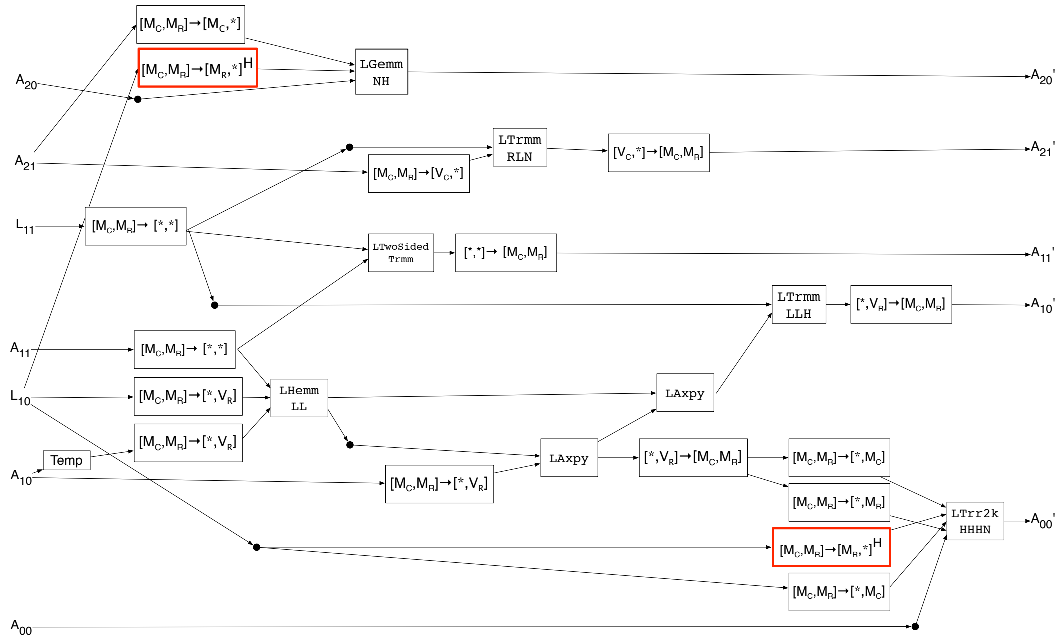

Figure A.35: Result of transposing one input to `LTrr2k HNHN`.

We have another redundant redistribution that can be optimized using the transformation in Figure A.36 to yield the implementation seen in Figure A.37.

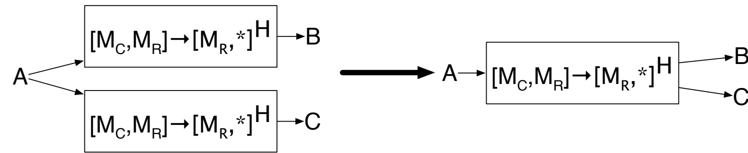

Figure A.36: Remove redundant redistribution.

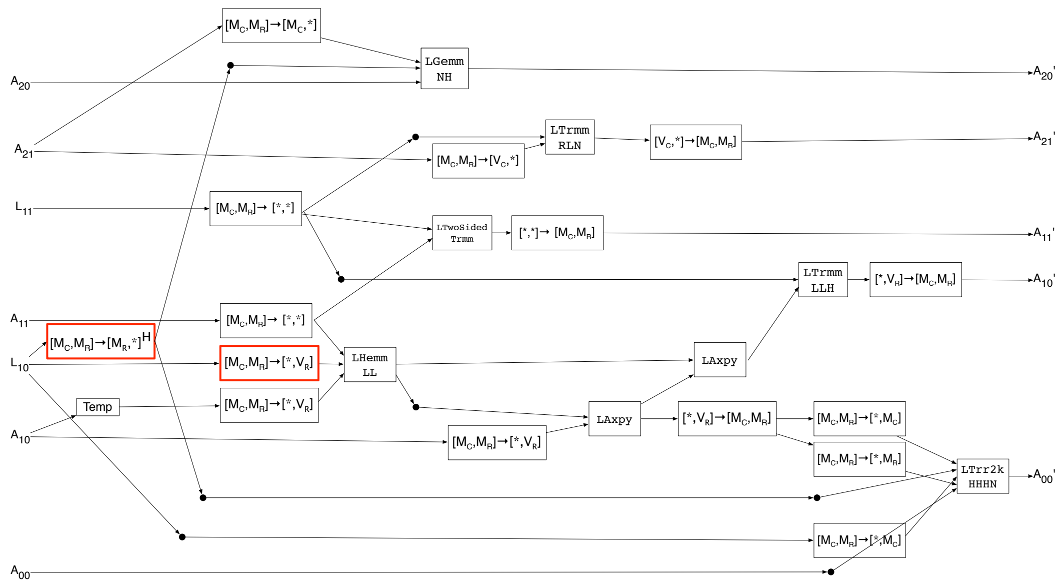

Figure A.37: Final, high-performance implementation.

Finally, we can implement the redistribution of $[M_C, M_R] \rightarrow [*, V_R]$ in terms of the intermediate distribution $[M_R, *]^H$ as shown in Figure A.38. $[M_C, M_R] \rightarrow [*, V_R]$ requires collective communication while $[M_R, *]^H \rightarrow [*, V_R]$ only requires locally data copying. Figure A.39 shows the final implementation. This maps to code that was better than that found in Elemental thanks to these four final optimization transformations. Since DxTer found it, Elemental's implementation has been updated to use this version.

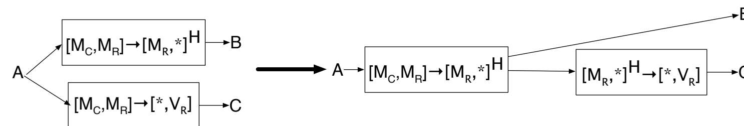

Figure A.38: Remove redundant redistribution.

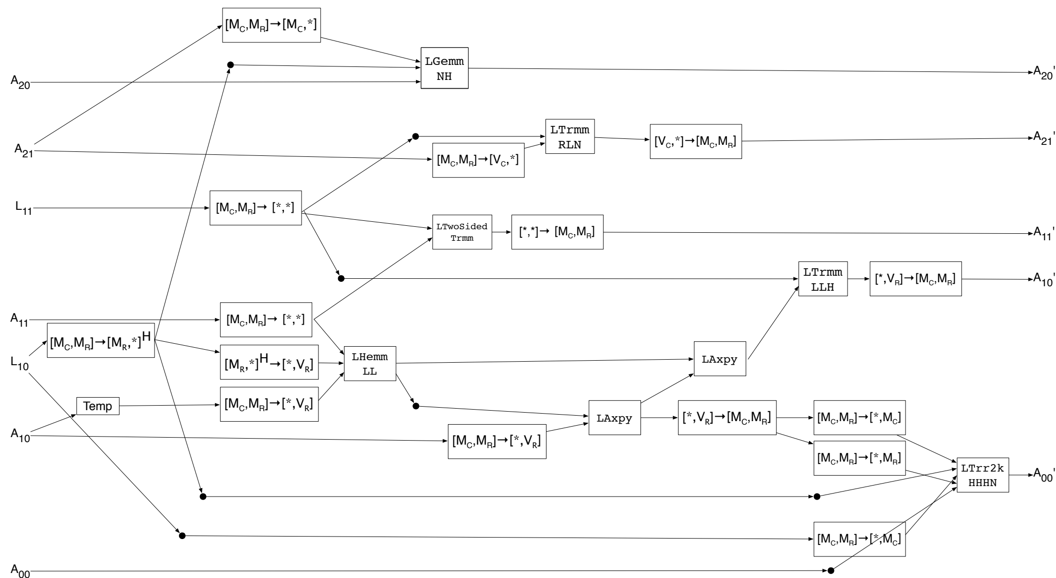

Figure A.39: Final, high-performance implementation.

Bibliography

[1] DxTer Source. `http://code.google.com/p/dxter/`.

[2] Elemental Source. `http://libelemental.org/`.

[3] Intel MKL. `http://software.intel.com/en-us/intel-mkl`.

[4] nVIDIA CUBLAS. `http://developer.nvidia.com/cublas`.

[5] Reference BLAS Source. `http://www.netlib.org/blas/`.

[6] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, 2004.

[7] 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.

[8] A. Auer, G. Baumgartner, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, S. Lam, Q. Lu,

M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Automatic code generation for many-body electronic structure methods: The Tensor Contraction Engine. *Molecular Physics*, 2005.

[9] D. S. Batory, R. Gonalves, B. Marker, and J. Siegmund. Dark knowledge and graph grammars in automated software design. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering (SLE)*, volume 8225 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[10] I. D. Baxter. Design Maintenance Systems. *CACM*, April 1992.

[11] G. Belter, E.R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[12] P. Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, August 2006.

[13] P. Bientinesi, B. Gunter, and R. A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*, 35(1):3:1–3:22, July 2008.

[14] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[15] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4), 2011.

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience: Research articles. *Concurrency and Computation: Practice & Experience*, 19(13):1749–1783, September 2007.

[17] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pages 123–132, New York, NY, USA, 2008. ACM.

[18] C.J. Date and H. Darwen. *A guide to the SQL standard: a user's guide to the standard database language SQL*. Addison-Wesley, 1997.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[20] E. W. Djikstra. *A discipline of programming*. Prentice Hall, 1976.

[21] J. Dongarra and P. Luszczek. Plasma. In *Encyclopedia of Parallel Computing*, pages 1568–1570. 2011.

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

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

[24] D. Fabregat-Traver and P. Bientinesi. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science – VECPAR 2010*, volume 7851 of *Lecture Notes in Computer Science*, pages 346–361, Heidelberg, 2013. Springer.

[25] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2003.

[26] R. C. Gonçalves, D. Batory, and J. Sobral. ReFlO: An interactive tool for pipe-and-filter domain specification and program generation. 2013.

[27] R. C. Gonçalves. *Parallel Programming by Transformation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2014 (To appear).

[28] K. Goto and R. van de Geijn. High-performance implementation of the level-3 blas. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, July 2008.

[29] K Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, 2008.

[30] J. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.

[31] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A flexible class of parallel matrix multiplication algorithms, 1998.

[32] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[33] J. A. Gunnels and R. A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.

[34] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE, Special issue on program generation, optimization and adaptation*, January-February 2005.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[36] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model-Driven Architecture*. Addison-Wesley, Boston, MA, 2003.

[37] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization*, 6(1):1:1–1:36, April 2009.

[38] B. Kgstrm, P. Ling, and C. Van Loan. Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, 1998.

[39] 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.

[40] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *ACM Special Interest Group on Management of Data (SIGMOD)*, 1988.

[41] T. M. Low. *A Calculus of Loop Invariants for Dense Linear Algebra Optimization*. PhD thesis, The University of Texas at Austin, 2013.

[42] T. M. Low, B Marker, and R. 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.

[43] T. M. Low, R. A. van de Geijn, and F. 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)*, pages 153–163, New York, NY, USA, 2005. ACM.

[44] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *ISMIS*, 1994.

[45] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18:674–704, 1992.

[46] B. Marker, D. Batory, and R. van de Geijn. A case study in mechanically deriving dense linear algebra code. *International Journal of High Performance Computing Applications*, 27(4):439–452, November 2013.

[47] B. Marker, D. S. Batory, and R. A. van de Geijn. Code generation and optimization of distributed-memory dense linear algebra kernels. In *ICCS*, 2013.

[48] B. Marker, J. Poulson, D. S. Batory, and R. A. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *VECPAR*, 2012.

[49] J. M. Neighbors and Bayfront Technologies. Draco: A method for engineering reusable software systems, 1987.

[50] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):13:1–13:24, February 2013.

[51] J. Poulson, R. A. van de Geijn, and J. Bennighof. (Parallel) Algorithms for Two-Sided Triangular Sovles and Matrix Multiplication. *ACM Transactions on Mathematical Software*, 2012. submitted.

[52] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.

[53] T. Riché, R. Goncalves, B. Marker, and D. Batory. Pushouts in Software Architecture Design. In *GPCE*, 2012.

[54] T. L. Riché, H. M. Vin, and D. Batory. Transformation-Based Parallelization of Requst-Processing Architectures. In *MODELS*, October 2010.

[55] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing Company, Incorporated, 1997.

[56] M. Schatz, J. Poulson, and R. van de Geijn. Parallel matrix multiplication: 2d and 3d. Technical report.

[57] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee. FLAME Working Note #71. Implementing level-3 BLAS with BLIS: Early experience. Technical Report TR-13-20, The University of Texas at Austin, Department of Computer Sciences, 2013.

[58] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

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

[60] R. A. van de Geijn and E. S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.

[61] F. G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[62] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for generating blas-like libraries. *ACM Transactions on Mathematical Software*. Accepted.

[63] R. Clint Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[64] K. Yotov, X. Li, G. Ren, Maria M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas?, 2005.

[65] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn. FLAME Working Note #69. Implementing level-3 BLAS with BLIS: Early experience. Technical Report TR-13-03, The University of Texas at Austin, Department of Computer Sciences, 2013.