Efficient Generation of Sequences of Dense Linear Algebra through Auto-Tuning

by

Geoffrey D. Belter

B.S., Clarkson University, 2003

M.E., University of Colorado at Boulder, 2012

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2012

This thesis entitled:
Efficient Generation of Sequences of Dense Linear Algebra through Auto-Tuning
written by Geoffrey D. Belter
has been approved for the Department of Electrical, Computer, and Energy Engineering

Jeremy Siek

Prof. Elizabeth Jessup

Prof. Dirk Grunwald

Prof. Manish Vachharajani

Dr. Boyana Norris

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Belter, Geoffrey D. (Ph.D., Electrical Engineering)

Efficient Generation of Sequences of Dense Linear Algebra through Auto-Tuning

Thesis directed by Prof. Jeremy Siek

It is rare for a programmer to solve a numerical problem with a single library call; most problems require a sequence of calls. In the case of linear algebra, programmers will chain a series of Basic Linear Algebra Subprogram (BLAS) library calls to achieve the desired result. When a sequence of BLAS calls is memory bound, a great deal of performance is missed because optimization has not occurred between library routines. It is not practical to create a library with every required sequence of linear algebra operations, but at the same time it is difficult for programmers to write their own high performance implementation. One solution is for programmers to use an auto-tuning tool capable of optimizing the sequence of operations that exactly suits their need. This thesis presents a matrix representation and type system that describes basic linear algebra operations, the loops required to implement those operations, and the legality of key optimizations. This is demonstrated in an auto-tuning tool which generates loops and performs data parallelism and loop fusion. Results show that this approach can match or exceed performance of vendor tuned BLAS libraries, general purpose optimizing compilers, and hand written code. Further, this approach is shown to be both portable and work with a range of dense matrix storage formats. All of this is achieved with search times in the range of several minutes to a few hours.

Dedication

To Sacha, for all of the support you provided. I could not have done it without you.

Acknowledgements

First I want to thank my wife Sacha for her tremendous support and encouragement throughout this process. Sacha kept me focussed during even the most difficult of times and took care of everything when I was too busy. I am also grateful to my parents for raising me with the determination and work ethic that was required to complete this degree. I am indebted to my cousins, Nick and Jake, who were there for myself and Sacha in this journey, always with no questions or expectations.

This thesis would not be possible without the support of my advisor, Professor Jeremy Siek. I am grateful for the motivation, knowledge and, guidance that Jeremy provided throughout my studies. Without Jeremy's support, my graduate career would not have included a PhD. I would also like to thank the other members of my committee, Liz Jessup, Boyana Norris, Dirk Grunwald, and Manish Vachharajani for their comments, ideas, and questions that helped to guide this work.

I am grateful to Ali for insisting on the completion of this degree and being flexible and supportive for the many years that I worked for him while pursuing this PhD.

I am indebted to my fellow lab mates, Ian Karlin, Thomas Nelson, and Arlen Cox to name a few. It was invaluable to have the advice and opinions of my fellow lab mates who were going through the same process. It was easier to handle the late nights, and weekend pushes when I could commiserate with the friends who were going through the same process.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

There is a wide range of computing applications that rely on linear algebra and in many of these applications, high performance linear algebra routines have a noticeable impact on overall system performance. This importance has spurred vast amounts of research on the topic of high performance linear algebra and has produced numerous implementations of linear algebra libraries and various tools to aid in the creation of these libraries. These high performance libraries are readily available, reliable and straightforward to use. Unfortunately, rapid hardware development quickly dates libraries, and the fixed set of routines in a library means important optimizations are often missed between library calls. This leaves the question; how can we efficiently produce high performance linear algebra so a library writer can create a more robust library, or a user can directly get a good implementation of their exact sequence of operations. One option may be auto-tuning, the process of automatically optimizing and evaluating a problem in order to find an ideal solution. A successful auto-tuning tool must incorporate a range of optimizations and have an efficient method for selecting the ideal set of optimizations.

This thesis discusses a fully automatic, linear algebra specific, auto-tuning tool using a generic matrix representation and type system that enables translating a high level linear algebra syntax to a set of uniquely optimized C routines. The best performing of these routines is selected by the tool with no interaction from the user. The matrix representation and type system support loop fusion, array contraction, shared memory parallelism, and cache tiling. The primary use case for the work presented here is optimizing sequences of memory bound basic linear algebra where

optimization across existing library calls is required for high performance. This thesis includes only brief details of search strategies used in auto-tuning, the focus of this thesis is the details of compiling and optimizing linear algebra in an auto-tuning environment.

1.1 Challenges of Generating High Performance Linear Algebra

There are several challenges that must be overcome to provide a consumer of linear algebra with highly optimized code. At a high level this can be broken into two problems: 1) distribution and 2) production of optimized code. Currently the main method for distributing linear algebra routines is through highly optimized libraries. This has many advantages and several disadvantages. The largest advantage to the producer is that there is a fixed problem that can be heavily optimized. This provides a good library for a consumer if their requirements fit into the problems solved by the library. However, in the event they do not, a consumer must piece together portions of the available library or try to write their own implementation. In either case, it is unlikely that they will achieve the highest possible performance.

The second challenge in providing a high performance linear algebra is the actual production of high performance code. Producing high performance code is made difficult by differences in computer architectures. For example, changes to an execution pipeline might require rescheduling instructions, while changes to the number of cores available might require selecting a completely different algorithm. Computer architectures vary from vendor to vendor and between generations of computers, forcing a producer of a high performance routine to choose between creating many versions of the routine, or accepting suboptimal performance on certain architectures. Given the cost and time to develop collection of linear algebra routines, neither solution is attractive.

Producing a high performance routine is complicated by architectural differences, and this requires a developer to be aware of current optimizations to deal with those difference. An experienced programmer may be able to narrow the set of optimizations that must be searched to find an ideal combination, but it is unlikely that the first set of optimizations tried will result in ideal performance. The process of discovering the best set of optimizations can be time consuming for

each given architecture. Additionally, as architectures grow more complex, the ideal combination of optimizations is likely to grow more complicated as well. More complex sets of optimizations will be more difficult to identify, and the process likely will be more time consuming.

1.2 Limitations of Existing Approaches

These production challenges have made packages that can automatically tune code more attractive. ATLAS is an example of a linear algebra auto-tuning package. ATLAS [7] empirically searches a combination of automatically generated code and hand written input to provide a complete Basic Linear Algebra Subprograms (BLAS) library [15]. This project exploits the fixed set of routines and the similarity between those routines to focus the empirical search on a handful of kernels that can be used to build the rest of the library. This focused empirical search allows for tuning times on the order of hours for the complete library.

ATLAS is a successful and portable package but is limited in a few ways. First, ATLAS has encoded certain decisions into the tuning process, for example ATLAS supports data repacking but only allows data to be repacked in a particular format. Second, ATLAS supports the BLAS routines and a few additional routines, but does not handle any routines outside of those. Certain sets of memory bound routines can be combined to provide significant performance improvements, and a fixed framework like ATLAS cannot exploit that. Even with these limitations, ATLAS is a standard for comparison and can be competitive with hand tuned routines.

The limitations of ATLAS described here have a significant real world benefit; they reduce the optimization search space. The optimization search space is the set of possible optimization combinations. Removing the encoded decisions of ATLAS and supporting routines outside of a fixed library exponentially grows the optimization search space. This can push auto-tuning times from hours to days, or in some cases longer.

Other auto-tuning tools have removed the limitation of only supporting BLAS routines, and have shown that search times can remain fast. Pluto is a general purpose loop restructuring compiler that operates on C code, and is capable of performing loop fusion and parallelism [18]. It does these

optimizations with a search time on the order of minutes. Pluto does well for certain dense linear algebra routines, however loop fusion enables array contraction, which is important in reducing overall memory traffic. In many cases, a source to source translator requires user annotations specifying which data structures are safe to eliminate in order to enable safe array contraction. Pluto does not support this optimization limiting performance for certain dense routines. Additionally, Pluto is built around the polyhedral model. This means it is limited in the matrix storage formats it can support, specifically it cannot support irregular storage formats such as sparse.

1.3 Outline of Contributions

Achieving something in the middle of ATLAS and Pluto is the goal of the ideas in this thesis. Specifically, to create an approach that will aid library writers but will also help users develop their own implementations that exploit optimization between existing library routines, primarily for sequences of memory bound linear algebra. This thesis presents a matrix representation and type system that can be utilized to relax some of the limitations imposed in ATLAS, namely, these ideas work with any user provided sequence of basic linear algebra operations, and optimizations are performed between these sequence of operations rather than individual operations. Additionally, this work eliminates the limitation of Pluto by utilizing a matrix representation that works with various matrix storage formats. Further, this allows the user to specify the sequence of operations at a level high enough that sufficient information is present to safely perform optimizations such as array contraction.

The presented matrix representation has been demonstrated to work with general, triangular, and triangular packed matrix formats. Furthermore, the type system has been shown to support loop fusion, array contraction, shared memory parallelism and cache tiling. An evaluation of generated C code shows that this approach is competitive with existing tools, libraries, and hand written code and that search times can remain on the order of hours or less. All of this has been demonstrated in the context of a prototype linear algebra auto-tuning tool named Build to Order BLAS.

An important aspect of auto-tuning is the search strategy. This topic is being actively researched by Thomas Nelson using the same auto-tuning tool. The merging of Nelson's search tool with the ideas for compiling linear algebra presented in this thesis show promising results. The methods for fast search are not the topic of this thesis and are only briefly discussed.

The primary contributions of this thesis include:

- A matrix representation that can express many matrix storage formats, data parallelism, and cache tilling.
- A type system using the generic matrix representation that describes the legality of linear algebra operations.
    * Provides a description of loops required to implement linear algebra operations.
    * Provides a method for automatically identifying data parallelism in a linear algebra operation.
    * Provides a description of legal loop fusion.
- Demonstration that the matrix representation and type system can provide a fully automatic approach to generating linear algebra with loop fusion, shared memory parallelism and cache tiling.
- Verification that the matrix representation and type system work as intended.

This thesis begins by discussing the problem and related work. Following this in Chapter 3 is a description of the Build to Order BLAS auto-tuning system which is the tool being worked with to demonstrate the ideas presented here. Chapter 4 formalizes the type system and discusses the verification of the type system using BTO. Chapter 5 explains the details of implementing automatic partitioning. This is followed by Chapter 6 which describes how various storage formats can be supported without effecting the loop generation and optimization portion of the remainder of the compiler. Chapters 7 and 8 examine the validation and evaluation of the code generated by BTO. Finally Chapter 10 summarizes the thesis and discusses future directions.

Chapter 2

Background

The ideal goal of implementing high performance linear algebra is to perform a given operation as quickly as possible on a particular computer. This computation time is going to be bound by portions of the target platform. This may be memory or floating point operation speed, or some combination of the two. Understanding when and what type of bottleneck a linear algebra (LA) implementation has hit is important to high performance LA. Even more important is being able to identify if that bottleneck is absolute for the given problem, or if there are optimization techniques or algorithm modifications that can overcome the bottleneck. Identifying the difference between an artificial and a true bottleneck requires knowledge of LA algorithms, optimizations techniques, and computer architecture and methodically evaluating combinations of optimizations to find the best set. The complexity of modern architectures has increased the time it takes to identify the ideal set of optimizations.

The practice of implementing high performance LA is further complicated by the interface between the programmer and the computer. A programmer must communicate a specific algorithm to a computer using a programming language and a compiler. Often the language is not expressive enough, or the compiler is not as knowledgeable as the programmer and cannot provide an ideal translation. This poor communication leaves programmers with the option of creating better languages, compilers, or expressing the algorithm directly in assembly.

Even if communication is perfect, modern hardware is complicated to the point that what a programmer thinks 'should' work, does not always work. For example, features of hardware

intended to improve performance such as out-of-order-execution, reorder buffers, and instruction micro-coding can mean the set of operations a programmer gives to a computer is not identical to the one it executes. There is often no control or no documentation describing some of these processor features meaning that to achieve the absolute best implementation on certain platforms may boil down to brute force searches over a range of optimizations.

Together these problems have driven research into tools ranging from making manual optimization faster to full blown auto-tuning tools. The rest of this section examines some of the important work that the development of a linear algebra specific auto-tuning tool can utilize.

2.1 Key Optimizations

An effective set of optimizations is required for an auto-tuning tool to be successful. This section covers some of the most important optimizations for memory bound linear algebra.

2.1.1 Loop Fusion

Combining two or more loops into a single loop is known as loop fusion. When there is shared data between the fused loops, this technique will reduce memory traffic which can lead to performance improvements [41, 54, 3, 79, 33, 2].

Loop fusion is legal when loops share common iterations, however it is not beneficial to fuse loops in all cases. In the absence of shared data there is an increase in the amount of data in caches and in registers. Depending on the architecture and operation, increasing register and/or cache pressure can have negative performance effects [49, 48]. This means, an auto-tuning tool must understand the target architecture.

2.1.2 Array Contraction

In the context of linear algebra, array contraction is the reduction of an array to a scalar value with the intention to reduce the overall memory traffic. This occurs in the presence of loop fusion, for example when one loop produces and array and the second consumes it, fusion can eliminate

the need the for the intermediate array, instead allowing the same flow of data using only one scalar value. Array contraction has been shown to be an important optimization in linear algebra [81, 58] and in scientific computing in general [65]. In functional programing language terminology this is known as deforestation [77].

2.1.3 Tiling for data locality

There are many levels of a modern memory system ranging from disk down to registers. Typically the closer to the processor a portion of the memory system lies the smaller its capacity, but the faster its response time. Utilizing the caches and translation look aside buffers efficiently is important to high performance linear algebra, however it is common for data structures to not fit completely within the smaller and faster levels of cache. Tiling is a technique which breaks data into tiles smaller than the original data structure which can fit in some portion of the memory system. Ideally the data in these tiles is reused to completion at which point a new tile can be fetched. Wolf shows that this technique can improve performance when used at the register and cache level [82]. Goto later discovered that tiling for the translation look aside buffer is important to certain linear algebra operations [35].

2.1.4 Parallelism

Trends in hardware design have made many-core computers common place. Linear algebra operations often can exploit these cores to achieve higher performance. Many shared memory linear algebra libraries exist today with parallel implementations including vendor tuned BLAS libraries from Intel [46], Apple [8], and IBM [42] as well as libraries for distributed memory systems such as PLAPACK [4].

2.2 Domain-specific Compilers

There are several examples of linear algebra specific compilers, however none have put together a complete enough set of optimizations to replace hand tuned libraries. The MaJIC and

FALCON compilers for MATLAB optimize matrix operations [70, 61], but they do not perform important optimizations such as loop fusion. Additionally these tools rely heavily on the user to identify portions of code for optimization potential.

Other approaches identify and call existing library routines. Broadway [38] optimizes calls to libraries such as PLAPACK [4]. FALCON [70] identifies BLAS [15] routines in MATLAB code. With rapid changes in architectures, current high performance libraries are not always available limiting the usefulness of tools that require them.

Arrays are a basic building block of many linear algebra operations. Several languages have explored optimizing array operations. High performance FORTAN provides array operations that compile to high performance code [51]. APL [20] is a language that was designed specifically for vector processors and has extensive array support. Much can be learned from these projects when working with dense linear algebra, however many linear algebra operations are performed with sparse data structures.

Domain-specific compilers are becoming important in many fields such as tensors [1] and optimizing user-defined abstractions [68]. SPIRAL is able to produce code that is competitive with hand tuned implementation in the DSP domain [67]. PATUS is a compiler designed for stencil computations and used in an auto-tuning environment which is shown to perform well on many-core architectures [24].

2.3 High-Performance Libraries and Auto-tuning

Several packages have been designed to optimize a specific library, these are called Active Libraries [25]. Automatically Tuned Linear Algebra Subprograms (ATLAS) is an example that is specific to the BLAS library [7]. ATLAS exploits similarities in the routines provided in BLAS to focus a search on key kernels that enable building a complete high performance library. ATLAS focuses on matrix-matrix and matrix-vector operations. CHiLL utilizes the polyhedral model with several custom extensions to automatically generate and evaluate several transformations including loop unrolling, tiling, and permutations [75]. CHiLL has been shown to generate code with

performance similar to or better than ATLAS.

Orio is an auto tuning tool that uses annotations to guide an automatic search [39] and can be competitive with hand tuned code. Orio is capable of code transformations such as loop unrolling, tiling and permutations. The annotations are language and architecture independent, allowing for interfacing with existing auto tuning tools. Orio in combination with PLUTO has improved performance beyond what either is capable of alone. This tool performs particularly well for linear algebra with small input sizes. Similarly POET is an annotation based parameterized search over similar optimizations [83]. These types of tools work well for architecture specific optimizations, such as selecting an ideal loop unrolling to fully exploit available registers.

The Formal Linear Algebra Methods (FLAME) provides programmers with syntax for expressing optimizations more quickly than writing loop based implementations [59]. The framework automatically generates an implementation based on the provided program description. This method allows a programmer to quickly try many optimizations, however it is not fully automated.

Similar to FLAME, Blitz++ and the Matrix Template Library provide methods for quickly expressing key optimizations such as loop fusion. These exploit the template feature of C++ [76, 72]. Along the same lines Hierarchically Tiled Arrays (HTA) [11] provide an abstraction for expressing tiled algorithms.

2.4 General Purpose Loop Restructuring Compilers

General purpose compilers often employ a similar set of optimizations as those required of high performance linear algebra, specifically the improvement of data locality. Lim and Lam use affine transformations to maximize parallelism while maintaining data locality [57]. PLUTO is a project based on the polyhedral model that restructures loops to express cache tiling, vectorization, and data parallelism [18]. The Intel [45] and Portland Group [66] compilers are state of the art compilers capable of performing vectorization, loop fusion, and data parallelism. These compilers can generate high performance code in many instances, however they miss the ideal case often enough that a more reliable approach is needed.

There are several disadvantages to general purpose compilers, one being that general purpose compilers require a loop based implementation of the desired program that is amenable to the analysis required to modify the loops. The program must be written in a form that provides enough information to safely allow loop manipulation, however this is complicated by common programming techniques such as the use of pointers. Fortunately, Netlib maintains a linear algebra repository with much of the code intended for reference only [15], which can provide code that can meet the requirements needed of safe analysis. The challenge is with strings of operations which often requires manual composition of routines which can be a nontrivial task. When appropriate starting sources are available dependence analysis is required [55]. This analysis can be time consuming and often must be conservative in determining dependencies which can allow for missed optimization potential.

2.5 Performance Analysis

Performance analysis is as important to auto-tuning as a complete set of optimizations. Analyzing performance can be analytic, empirical, or some combination of the two.

2.5.1 Analytic Methods

With analytic modeling there are trade offs between the level of accuracy and evaluation times. As the details of analytic modeling are not a primary topic of discussion, this section only briefly covers the topic. There are several examples of highly accurate memory models which require runtimes that present little to no savings over empirical testing [34, 69]. In the context of auto-tuning software, this solution presents no benefit. Ferrante provides a faster solution, however this model is limited to estimating the number of cache lines accessed for perfectly nested loops [31]. In the presence of loop fusion, linear algebra operations can form imperfectly nested loops.

2.5.2 Empirical Methods

An early use of empirical testing was demonstrated in the tuning of matrix-matrix multiply. Parameterized code generators, which generated Portable, High-Performance, ANSI C, which is known as PHiPAC produced many variants of the matrix-matrix multiply operation. These variants were empirically tested to identify the best performing which was shown to be competitive with hand tuned implementations [13].

Many of the ideas from PHiPAC were adopted by ATLAS [7] which was one of the first auto-tuning libraries to make use of empirical evaluation. This is the only performance analysis method in the ATLAS framework and has provided high performance implementations on a range of platforms. ATLAS exploits the relatively small sized search space to allow a complete empirical search on the order of hours.

Zhao et al [86] focus on loop fusion and use exhaustive search and empirical testing to identify the best combination of loop fusions. Qasem shows that SSE vectorization, strength reduction, loop unrolling, and prefetching can be searched empirically [84]. This required trying several search strategies as exhaustive testing was not possible.

The history of empirical testing started with manageable search spaces and was modified over the years to handle large sets of optimizations. It is clear that a strictly empirical search with the optimizations required for a modern architecture is too time consuming.

2.5.3 Hybrid Methods

Trends in auto-tuning tools have been toward a combination of empirical and analytic evaluation. This approach has been shown to work with large sets of optimizations including combinations of loop permutation, unrolling, register and loop tiling, copy optimization, and prefetching [23]. This approach narrows the set of optimizations and performs a find grain search with empirical testing. Similar work [85] advocates using analytic methods to perform coarse evaluation and then use empirical search to fine-tune performance.

More sophisticated systems use empirical results in combination with machine learning tools to adapt analytic models to particular problems and computers [30]. This approach shows promising results, however, it requires a time consuming training process.

2.5.4 Search Strategies

The complexity of the optimization search space has driven research into a range of search strategies. Balaprakash utilizes random search, genetic algorithms, and Nelder-Mead simplex to find a best performing implementation when searching loop unrolling, parallelization, vectorization and register tiling [9]. The Nelder-Mead method was the most successful.

Seymour et al. compare simplex, genetic algorithm simulated annealing, particle swarm, orthogonal and random search of loop unrolling, compiler flags and blocking of several linear algebra operations [71]. The findings were that particle swarm had a slight advantage with low numbers of search dimensions (1-5), but with high numbers of dimensions there was little separation between any of the search methods.

Kisuki explores tiling and loop unrolling with genetic algorithm, simulated annealing, pyramid, and random search on linear algebra operations [53]. They found that all of these search methods were nearly as efficient as one another and the performance results were generally within five percent.

Chen et al. exploit characteristics of permutation, tiling, fusion, and others to perform an orthogonal search, which searches over optimizations one at a time [22]. This technique allows their CHiLL framework to be competitive with hand tuned linear algebra libraries and ATLAS.

This sample of work examining search strategies all had a similar theme: there is no strategy that is a clear winner with high dimensional search spaces and a range of input programs.

2.6 Summary

This sample of work discusses several topics that must come together to make a successful linear algebra auto-tuning tool. The work discussed tries one of three approaches; optimize fixed

sets of operations with specific code generators, optimize existing source code, or identify specific linear algebra operations and optimize them. Code generators are limited in the scope of problem they can address. Tools that optimize existing code tend to be overly conservative or require extensive user input to perform well. Identification and optimization of linear algebra operations limits optimization between operations. No approach is ideal for sequences of memory bound linear algebra.

None of the approaches create a type system that describes linear algebras, legal optimizations, and represents matrices. The rest of this thesis describes such a system and demonstrates the ideas work with a prototype tool which combines the type system with ideas from the auto-tuning literature. The next section outlines the details of the overall tool.

Chapter 3

Build to Order Linear Algebra

The Build to Order Linear Algebra (BTO) tool is a prototype auto-tuning system designed specifically for memory bound linear algebra. BTO takes a high level description of matrix algebra using a subset of MATLAB and with a combination of analytic modeling and empirical testing produces a high performance C implementation. Currently the primary focus of BTO is optimizing memory bound operations that have loop fusion potential. The optimizations utilized by BTO are loop fusion, array contraction and data partitioning. Data partitioning is used to express data parallelism and cache tiling but could be extended to express loop unrolling and vector operations as well. BTO is the auto-tuning tool used to the matrix representation and type system presented in this thesis.

3.1 What is BTO

The BTO tool consists of a linear algebra specific compiler, and a search tool. The search tool guides a search for the best performing implementation by providing optimization specifications that the compiler uses to generate optimized C code. Figure 3.1 shows a high level diagram of the auto-tuning process. The user begins by selecting a search strategy (i.e. exhaustive, genetic), and providing the input program to optimize. The compiler begins by creating a data flow graph and inferring any missing types. The type inference phase also performs a type check. The search tool then precisely describes an optimized program. Using this description, the compiler introduces partitioning, introduces loops, and performs loop fusion, finally outputting a C implementation.

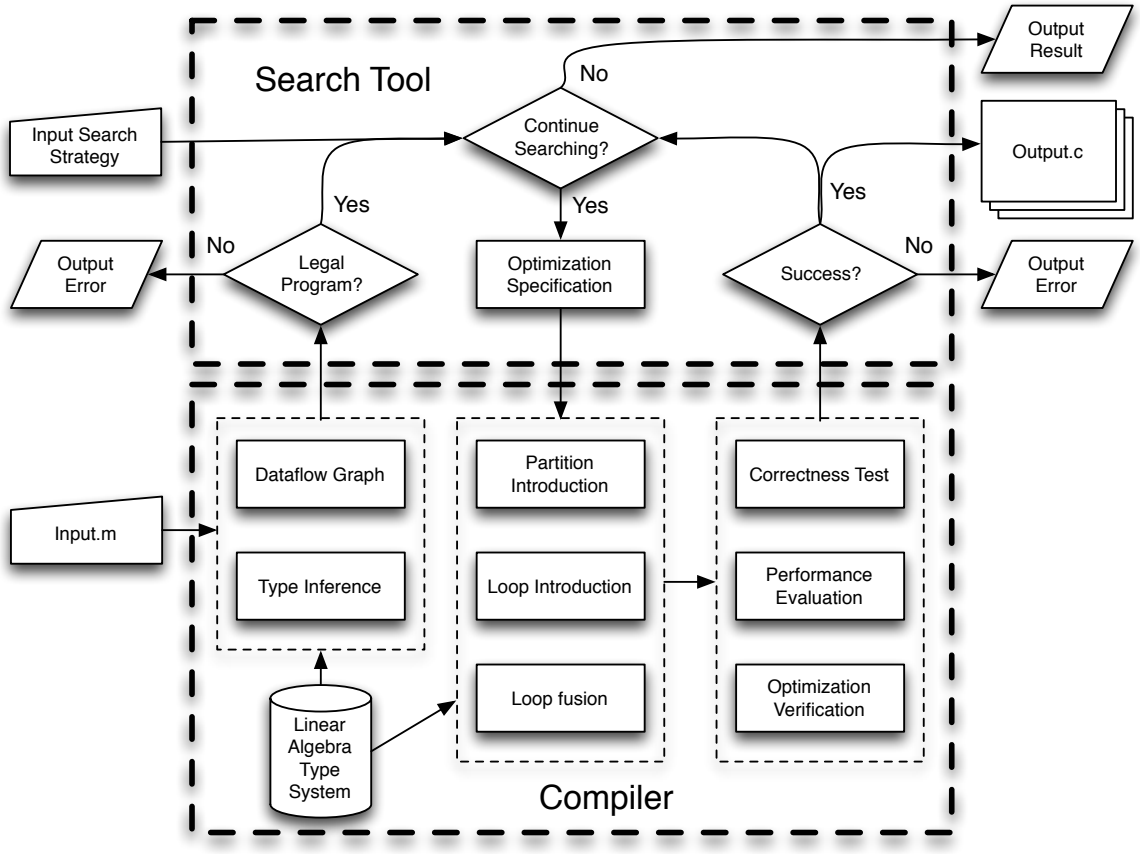


Figure 3.1: Overview of the BTO auto-tuning tool. The overall process consists of two primary tools, the search tool, and the compiler.

This code is checked for correctness, performance, and to ensure that it meets the description provided by the search tool. These results are fed back to the search tool which can decide to continue the search or not. The focus of this thesis is on the compiler portion of the search tool. The rest of this section provides a high level overview of the compiler with the rest of the Chapter describing the details of each step.

The input to BTO is matrix and vector operations (add, subtract, multiply) using syntax similar to MATLAB. For example to express a matrix-vector multiply, the body of the input would be $y = A \times x$. Prior to the body, is a section for declaring variables as matrices, vectors and scalars, where each is declared as an input, output, or inout. A variable may be undeclared implying that

```
GEMV
in A : column matrix, x : vector
out y : vector {
    y = A * x
}
```

Listing 3.1: Compiler input for the GEMV kernel.

```
void GEMV (double *A, int A_nrows,
           int A_ncols, double *x,
           double *y) {
  int i, j;

  for (i = 0; i < A_nrows; ++i) {
    y[i] = 0.0;
  }

  for (i = 0; i < A_ncols; ++i) {
    double tmpX = x[i];
    double *tmpA = A + i * A_nrows;
    for (j = 0; j < A_nrows; ++j) {
      y[j] += tmpA[j] * tmpX;
    }
  }
}
```

Listing 3.2: Compiler output for the GEMV kernel.

the variable is an optional temporary. Both matrices and vectors can take an orientation description (column or row) which is required for the type system but for matrices also describes the storage in memory. See Listing 3.1 for a sample of the matrix-vector multiply and Listing 3.2 for typical C code output. Listing 3.2 represents one of many implementations produced by the compiler.

There are several points worth noting. The first is the level and detail of the input. The input describes the operation, but not an implementation of the operation, leaving the compiler and search tool the ability to directly select an implementation. This saves analysis required to determine if a transformation of existing loops is safe, a process that is error prone and often overly conservative. The second point is that a variable in the program is either marked with input/output/inout, or can be left undeclared. When a variable is undeclared, the compiler will remove that variable when possible, often providing a significant memory and performance savings. Further, the programmer need not know the type of the temporary, the compiler is able to infer the correct type.

Compilation utilizes two separate but related pieces of information, the type system, and a

data flow graph which is derived from the input program. The type system drives type inference, introduction of loops, introduction of partitions, and loop fusion. The data flow graph ensures data dependencies are respected which is required for loop fusion and array contraction. The data flow graph is annotated with types, which allow the type system to understand which types are involved in any given operation. Using these two pieces of information, the compiler can determine the legality of a program and optimizations.

The evaluation portion of the compiler has three responsibilities; to ensure that generated code is correct, to ensure that the optimizations were implemented as intended, and finally to evaluate the performance. Correctness and ensuring implemented optimizations match the expected optimizations is somewhat optional, though these checks provide confidence that the tool is working as intended. The performance evaluation is required to identify the best generated routine. The current performance evaluator can use both analytic modeling and empirical testing. The analytic model predicts data movement in and out of the various levels of the memory system. From these predictions a cost is derived which allows for comparison amongst the versions produced by the compiler. This allows only those versions the model ranks as best performing to be empirically tested. In practice this method has proven both successful and fast.

The BTO auto-tuning system is fully automated, the user provides the input program to the tool and is returned the best C implementation BTO can find. The user does not have to identify optimizations or areas in need of optimization. This allows a wide range of linear algebra users to make use of BTO.

3.1.1 What is BTO trying to achieve

The goal of BTO is to provide a high level interface with few implementation decisions encoded in the input. This level of input provides the auto-tuning tool the opportunity to generate any implementation it deems ideal. This simplifies the analysis when performing operations like loop reordering, loop fusing and cache tiling.

In addition to allowing the tuning tool more freedom, the high level input provides a simple

interface that is not platform specific. For example, reference BLAS routines can be found at the Netlib repository [15], these routines are implemented in Fortran using loops. The Netlib routines are intended for functionality and reference and are not designed to be high performance. However they still represent a specific loop ordering. If these routines are the base for an auto-tuning tool, the tool must work backward to understand what the operation is doing and how to best implement that, likely by reordering or adding additional tiling loops. The high level input used by BTO provides the operation only and no implementation details, saving analysis and allowing BTO to directly create the ideal combination of loops.

BTO strives to produce the highest possible performance. Ideally this is competitive with the best hand tuned routines. In practice, this represents a difficult challenge. This requires both finding the ideal algorithm to most efficiently utilize the memory system, while also finding the ideal set of instruction and instruction scheduling for a target computer. BTO currently only focuses on algorithmic modification to improve memory efficiency and relies completely on general purpose compilers for the low level computer implementation details. Low-level optimization are planned as future work, however given the class of problem (memory bound), there are more significant wins from the algorithmic modifications. Unfortunately relying on a general purpose compiler with less information than BTO has provides the potential for performance to be missed.

3.1.2 What problems does BTO solve

With the current set of optimizations, BTO is best suited for matrix-vector and vector-vector operations, known as Level 2 and Level 1 operations respectively. Because BTO focusses on memory efficiency optimizations, it will not be competitive for all Level 1 and 2 operations. For example, **dscal** is a BLAS routine that scales a vector. This operation has little opportunity for memory efficiency improvement and is going to rely primarily on good instruction scheduling.

The class of Level 1 and 2 operations that BTO is suited for are those that have loop fusion potential. For example operations like AA^Tx provides both reuse of the matrix A and the potential to fuse two matrix vector operations. Not all loop fusions are going to provide observable

performance increases, for example an operation such as $Ax + y + z$ has the potential to fuse the $y + z$ reducing memory traffic by order N, however the complete operations is order N^2 and a reduction of N may have little observable effect. Generally for loop fusion to be successful, it must reduce memory traffic by an amount close to the order of the over all operation, or it must enable additional optimizations such as array contraction. When these cases are present, BTO will perform well.

The set of programs BTO will perform well on is larger than it appears. The specific types of loop fusion required for BTO to perform well appear often when sequences of Level 1 and 2 BLAS calls are used. This occurs frequently enough that an addition to the BLAS standard has been accepted which would introduce many routines into BLAS [16] that BTO does well with.

3.1.3 What problems does it not solve

BTO is not competitive on matrix-matrix operations, known as Level 3. To compete with hand tuned implementations of this type, BTO would require additional optimizations. Data repacking is often required to utilize caches efficiently while loop unrolling of multiple loops is required to increase data reuse at the register level enabling ideal short vector unit utilization.

The addition of these two optimizations would make the optimization search space extremely large. Search times would be measured in days. Before BTO can take on these additional optimizations, an improved search strategy that understands these optimizations is required.

3.2 BTO Compiler Technical Details

The BTO compiler is based on two systems; recursive data containers and a graph-based representation of the flow of data. With these two systems, the BTO compiler can express a wide range of implementations for a single program and for a range of matrix storage formats.

3.2.1 Data Representation

All data in a program is given a type with the following grammar:

$$\begin{array}{lrcl} \text{orientations} & O & ::= & C \mid R \\ \text{types} & \tau & ::= & O\texttt{<}\tau\texttt{>} \mid S. \end{array}$$

The type S is for scalars, and represents the base type, for example float or double in the C language. A type of the form $O\texttt{<}\tau\texttt{>}$ describes a container with element type τ and orientation O. The orientation is either C for column or R for row. Orientation describes the shape of the node, e.g., $C\texttt{<}S\texttt{>}$ is a column vector, and it describes the layout in physical memory. For example, $C\texttt{<}R\texttt{<}S\texttt{>>}$ describes a matrix whose rows are stored in contiguous memory whereas $R\texttt{<}C\texttt{<}S\texttt{>>}$ describes a matrix whose columns are stored in contiguous memory. Orientations may be transposed as follow: $R^T = C, C^T = R$.

These types describe generic matrix storage formats which can map to general, triangular, sparse, etc. The underlying storage format plays a part in loop generation described later, however for the most part, rules describing characteristics of these containers is free of the underlying storage format. The rest of this Chapter discusses these types as general matrices, Chapter 6 describes the details required to support additional formats.

3.2.2 Data flow graph

The intermediate representation of the compiler is a data flow graph. Each of the vertices represent an operation and a data type. The edges between these vertices describe the order of operations, data dependencies and data flow. The data flow graph begins with a vertex for each input, output and operation specified in an input program. An example input program for $y = AA^Tx$ is shown in Listing 3.3 with the initial data flow graph shown in Figure 3.2. On the left of the graph are the two input values A and x and on the right of the graph is the output value y. There are three operations, two multiplies and one transpose, each with their own vertex. The edges show A^Tx must occur first and the result of this is multiplied by A and stored into y.

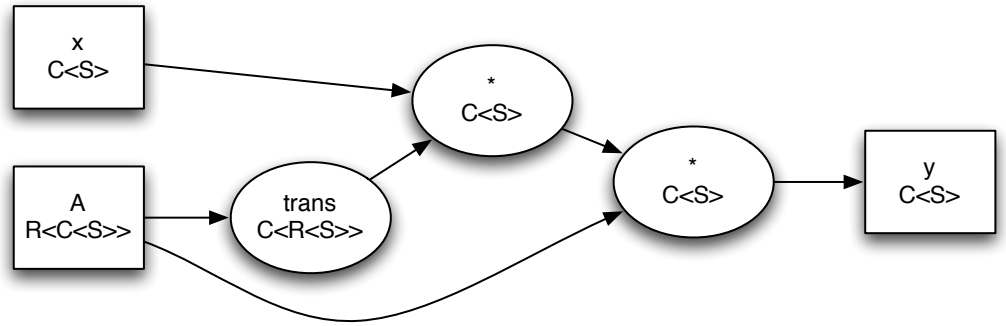


Figure 3.2: Example AATX data flow graph ($y = AA^T x$).

```
AATX
in A : column matrix, x : column vector
out y : column vector {
    y = A* (A' * x)
}
```

Listing 3.3: Compiler input for the AATX kernel.

3.2.3 Type Inference

The type assigned to each of the input and output vertices matches the declared type in the input program. A type inference phase assigns a type to any temporary data structures that may not have been declared and all operations. The type assigned to operations is the result type of the operation. For example in Figure 3.2, the transpose operation will initially be untyped. The transpose is operating on the input A which has a known value, so the output type of the transpose can be inferred. By understanding the rules of linear algebra (encoded as a type system), missing types can be determined by enumerating the legal options are based on known types of the operation. The type inference pass iterates, narrowing down the set of legal types for each vertex, until all unknown vertices have a single possible type. This works in most cases, however, in rare cases there is the potential for a program to be typed more than one way. For example, given the

Algo	Op and Operands	Result Type	Pipe
`add`	$O\texttt{<}\tau_l\texttt{>} + O\texttt{<}\tau_r\texttt{>}$	$O\texttt{<}\tau_l + \tau_r\texttt{>}$	yes
`s-add`	$S + S$	S	no
`trans`	$O\texttt{<}\tau\texttt{>}^T$	$O^T\texttt{<}\tau^T\texttt{>}$	yes
`s-mult`	$S \times S$	S	no
`rr-mult`	$R\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$R\texttt{<}R\texttt{<}\tau_l\texttt{>} \times \tau_r\texttt{>}$	yes
`cc-mult`	$C\texttt{<}\tau_l\texttt{>} \times C\texttt{<}\tau_r\texttt{>}$	$C\texttt{<}\tau_l \times C\texttt{<}\tau_r\texttt{>>}$	yes
`dot`	$R\texttt{<}\tau_l\texttt{>} \times C\texttt{<}\tau_r\texttt{>}$	$\sum(\tau_l \times \tau_r)$	no
`outer1`	$C\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$C\texttt{<}\tau_l \times R\texttt{<}\tau_r\texttt{>>}$	yes
`outer2`	$C\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$R\texttt{<}C\texttt{<}\tau_l\texttt{>} \times \tau_r\texttt{>}$	yes
`r-scale`	$S \times O\texttt{<}\tau\texttt{>}$	$O\texttt{<}S \times \tau\texttt{>}$	yes
`l-scale`	$O\texttt{<}\tau\texttt{>} \times S$	$O\texttt{<}\tau \times S\texttt{>}$	yes

Table 3.1: The linear algebra knowledge base.

two operands of an outer product, there is not enough information to determine if the result is a row or column major matrix. Currently when a program can be typed in more than one way, an arbitrary decision is made. Future work will enable evaluating the performance of all correctly typed versions.

The rules of linear algebra required by the type inference phase are provided in Table 3.1. The table describes the rules of linear algebra in terms of the recursive types used by the compiler. In this Table, there is one row that describes each matrix algebra operation. Each of these rules can provide a valid type estimate if the operator (e.g., + or ×) matches the operation label on the node and the partial type information known about the operands and result type do not exclude a match. If after reaching a steady state any node has zero inferred types, the program is not a correct linear algebra operation and the compiler reports an error. For example $y = A \times x$ where y is a row vector; regardless of x and A's types, the multiply node cannot have a type which makes this a correct matrix algebra operation.

The notation in use for result types in Figure 3.1 deserves some explanation. The notation includes the use of the + and × operators within the type. What this means is that the type inference algorithm is applied recursively to obtain the result type. For example, consider the `add` algorithm whose result type is specified as $O\texttt{<}\tau_l + \tau_r\texttt{>}$. Suppose the operands of a node labeled

with the operation + both have type C<S>. To compute the result type we recursively compute the result type for $S + S$. The only algorithm that applies is `s-add`, so the inner result type is S and therefore the outer result type is C<S>.

As an example, consider the multiply node corresponding to $trans * x$ in Figure 3.2. The matrix A^T has type C<R<S>> and the vector x has type C<S>. Thus, the algorithm `cc-mult` is a match for this multiplication node and the result type begins by taking the C from the left operand with an inner type of the rest of the left operand type multiplied by the complete right operand type,

$$C< \quad R<S> \times C<S> \quad >.$$

To determine the next level of the type, we find $R \times C$ in the table and find `dot` to be a match. This describes a dot product like operation which is summing the product of corresponding elements of the two operands, in this case the elements are S and $S \times S$ is simply S. The complete type result type is found using the three steps together as follows,

$$C<R<S> \times C<S>> = C<S>.$$

The table presented in this section is used to determine the type system that describes linear algebra. It is the underlying rules for loop generation and partitioning. This table is formalized in Chapter 4, along with a more detailed explanation of the type inference algorithm.

3.2.4 Introduction of Loops

Once types have been inferred, the data flow graph is modified to represent specific algorithmic implementations, for example with the introduction of loops which can be translated to C code. For the sake of discussion, the modifications to the graph are said to introduce loops, however the changes are more abstract and robust. The changes can represent loops, vector units, and parallel portions of computation to name a few. Generally, the modifications represent repetitive tasks that can map to a range of hardware features using a range of language features. This process of graph

```
VADD
in x : column vector, z : column vector
out y : column vector {
    y = z + x
}

.
```

Listing 3.4: Compiler input for the VADD kernel.

```
void VADD (double *y, int y_ncols, double *z,
        int z_ncols, double *x, int x_ncols) {
  int i;

  for (i = 0; i < y_ncols; ++i) {
     y[i] = x[i] + z[i];
  }
}
```

Listing 3.5: Compiler output for the VADD kernel.

modification is called lowering and a graph is completely lowered when all operations in the graph have scalar operands and produce scalar results.

The process of lowering is driven by the same linear algebra table used for type inference (Table 3.1) and, as in type inference, the process functions by modifying the operation nodes in the graph. A program is completely typed before this process begins, so an algorithm is a valid implementation for an operation node in the graph if 1) the algorithm's operator (e.g., + or ×) matches the operation label on the node, 2) the operand types match the types of the operands, and 3) the result type matches the type of the node.

The introduction of loops requires the addition of vertices for *get* and *store* operations. A detailed explanation of these operations is provided in Chapter 6, here the high level concept is discussed. Consider vector addition shown in Listing 3.4 as the BTO input and Listing 3.5 as C code output. In the C code we see a loop where a single iteration processes a single element of the result, $y[i] = z[i] + z[i]$. This requires accessing a single element from each of the vectors and these accesses are expressed in the graph as *get* and *store* operations.

From the initial **vadd** graph, Figure 3.3, we see there is the single + operation. The linear algebra rules table has a single algorithm named `add` that matches the addition operation. The operation matches, as do the operand and the result types. The introduction of loops, *get* and *store* operations is driven by the information in the **Result Type** column of the linear algebra rules table, in this case that says $C < S + S >$. This is interpreted as meaning we have two column containers whose elements are scalars and we need to sum the corresponding elements from each

container. This is represented in the graph by introducing a loop for the $C <>$ portion with size and iteration details provided by the type details of the column container (loop details discussed in detail in Chapter 6). The loop body, derived from the $< S + S >$ portion must get elements from each operand vector, sum them, and store them in the result vector.

Loops are represented in the data flow graph as subgraphs. Each subgraph has associated with it the size of the iteration space and the method for traversing through that iteration space, for example, iterate over every element or iterate over blocks of n elements. The introduction of the subgraph for the **vadd** operation can be seen in Figure 3.4. The dashed box represents the subgraph, with iteration details of $y\_ncols : 1$, meaning the size of the iteration space is y_ncols and the step size is a single element. This corresponds directly with the C loop in Listing 3.5.

Three new operations have been introduced: two *get* operations for the operands and a *store* operation for the result vector. The types associated with all of these new nodes is derived from the lowering algorithm, which in this case says get an element of the container. For **vadd** that is S, but if the type of an operand is more complex, i.e. $C < R < S >>$, the type of the *get* operation still gets only the element of the outer most container, $R < S >$. The type of the addition node takes the type of the next recursive step as described in the knowledge base, in this case $S + S$ which is S. This process continues on a single operation until the operand node has a result type of S. This occurs for all operations in the graph.

The linear algebra rules table (Table 3.1) has a final column labeled **Pipe**. This column describes the state of partial results which controls the use of completed portions of data structures before the operation has ended. Continuing with the vector addition example, once the first element has been summed, that value is complete and can be used by a consumer before the remaining elements are computed. The `add` algorithm has **yes** in the pipe column. In contrast consider the `dot` algorithm which takes the dot product of two containers. The result is not complete until the sum of all products has been found. There is no partial result that may be used by a consumer. In this case the **Pipe** column has listed **no**.

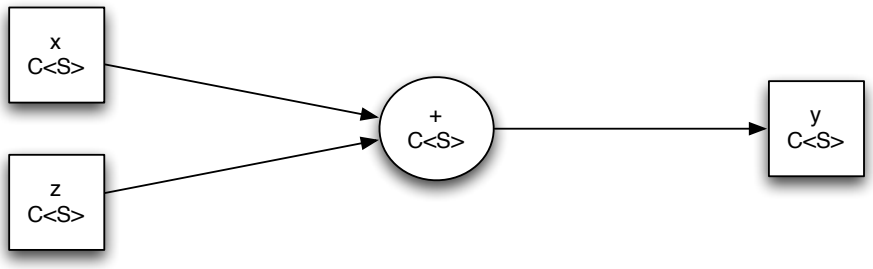


Figure 3.3: Example vadd data flow graph ($y = x + z$).

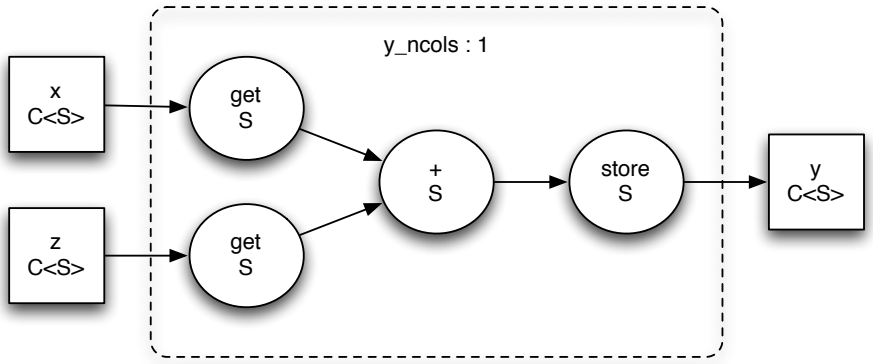


Figure 3.4: Example vadd data flow graph lowered ($y = x + z$).

3.2.5 Optimizations

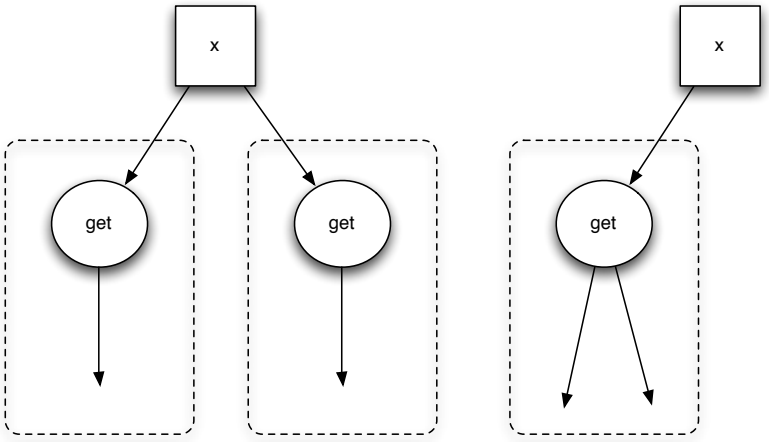
Optimizations are the key to BTO's success. It is important that the underlying program representation be general enough to represent a range of optimizations. The graph and type representation used by BTO succeeds in allowing many optimizations to be expressed. Equally as important is to not encode the set of optimizations deeply within the compiler framework so that adding or removing optimizations is difficult. BTO handles this by making optimizations data driven, controlled by interchangeable sets of optimizations. There are two separate systems for optimizing in BTO, one to modify graph structure, and one to modify types, both are data driven but each operates under a separate set of rules. The system for modifying types allows for the introduction of data partitioning which is discussed in more detail in Section 5.

Graph modification as discussed here is also known as graph rewriting or graph-grammars, and is the process of applying graph rewriting rules to an original graph to replace some subgraph with another [17, 29]. There is a large body of literature concerning graph rewriting on a wide range of topics. The use in BTO is not novel, just another application of the topic.

Optimizations enabled by graph modification include loop fusion, array contraction, operation ordering, pipelining, and any other optimization that will modify subgraphs, number of nodes, or edges between nodes. Currently BTO only implements loop fusion and array contraction. These graph modifications begin by identifying components of a graph that match the optimization's requirements. This may be based on edges, operation, data types, subgraph information, or node location to name a few. For example one type of loop fusion that is implemented identifies a data structure that is consumed by more than one loop. An example is shown in Figure 3.5(a). The requirement to match a component of the graph to this optimization is the existence of a single piece of data with two out edges which enter two separate subgraphs and that those two subgraphs are in no way dependent on one another. For example if the subgraph on the left produced a result consumed by the subgraph on the right, this optimization would not match.

When an optimization finds a matching portion of the graph, the optimizations corresponding graph modification rules are used to modify the graph to take the shape of optimization. The target shape for this optimizations is shown in Figure 3.5(b). This optimization must combine the contents of the two subgraphs into a single subgraph. Then the two *get* operations must be combined which requires removing one of the operations and updating the edges so that the existing *get* operation has two output edges.

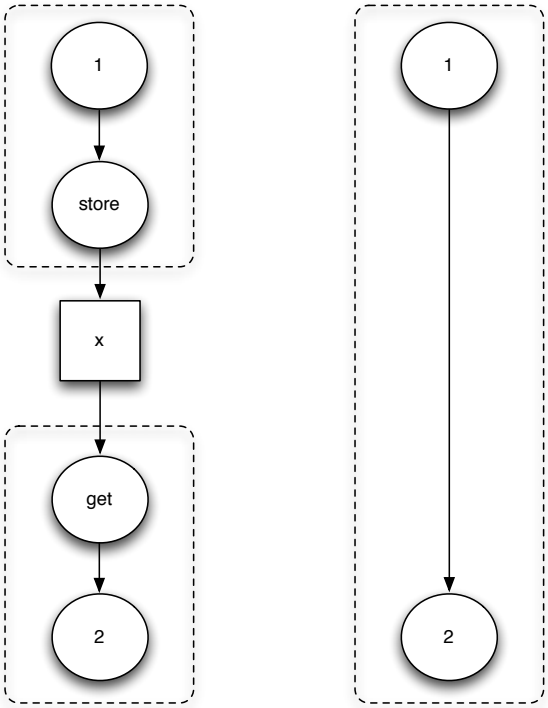
Many optimizations can be expressed with this type of graph modification (requiring a graph component match, and corresponding modification rules) and can be plugged into BTO at any time. In addition to the optimization shown in Figure 3.5, BTO currently has one other prototype optimization for array contraction. This fuses two loops when an intermediate data structure in between the two loops is not required. Figure 3.6 shows the required matching components and the modified graph resulting from the optimization.



(a) Section of graph matching loop fusion optimization.

(b) Section of graph after loop fusion optimization.

Figure 3.5: Loop fusion example.



(a) Section of graph matching loop fusion with array contraction optimization.

(b) Section of graph after loop fusion with array contraction optimization.

Figure 3.6: Loop fusion with array contraction example.

BTO currently has a limited understanding and check for legality of loop fusion. This understanding is beginning to improve and Appendix B discusses thoughts on moving toward a more formal approach to verifying legality of loop fusion. This is left to future work.

3.2.6 Code Generation

Once a graph has been completely lowered and optimizations applied, the graph moves onto a code generation phase where BTO generates C code output. This step converts each node into scalar expressions and each subgraph into loops. Because nodes and loops can represent different hardware and language features, the code generator can be extended to support a range of other hardware and functionality. For example by changing the code generated by a subgraph, a program could map to a GPU or a vector unit. The code generation pass is set up to allow generation of various types of code for a single routine, for example certain subgraphs can be translated to pthread dispatch loops, while inner loops can be translated to cache tiling code and finally the inner most loops to scalar C syntax. This design makes the backend flexible, portable, and able to accommodate future architectures and language features.

Figure 3.7 shows how the subgraphs map to C loops. For clarity, additional detail is added to the *get* and *store* operations in each subgraph showing how these operations translate directly into C syntax.

3.2.7 Search Strategy

Efficient search of the optimization space created by loop fusion, and partitioning for cache tiling and data parallelism is a large topic and is being actively researched. A high level description of the fast and reliable search strategy currently utilized by the auto-tuning tool is discussed here, however for additional technical details see related work concerning the creation and verification of the search method [10]. The decisions being made by the search tool include, when to fuse, how to partition, and what size a partition should be.

The primary challenge of the search space is that the space is discrete, for example, performing

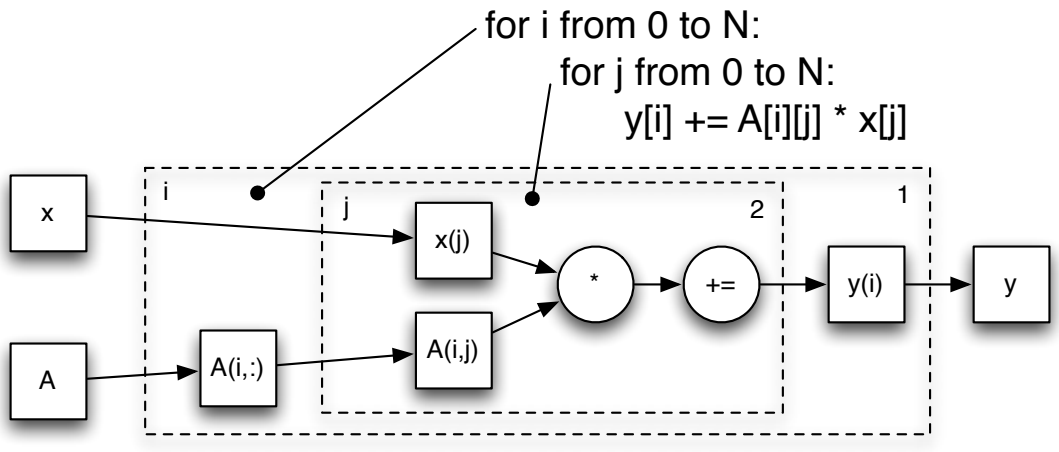


Figure 3.7: Example data flow graph of the operation $y \leftarrow Ax$ with loops expressed as subgraphs.

some optimization may have no effect on performance but may be required to enable another optimization that can significantly improve performance. Further, there is interaction between the optimizations that make it difficult to define the size and shape of the search space. For example, fusion can change the legal ways for partitioning a program. Finally, generic search representations utilized by many general purpose search tools (i.e. sequences of values that can be manipulated to describe a set of optimizations), describe a search space where only a very small fraction of points are legal programs.

The current search approach utilizes a custom representation that ensures that the majority of points in the search space are legal. Additionally, the search method uses the observation that fully fused programs tend to perform well. Unfortunately, fully fused programs are not always the best implementation [49, 48] and so a method for backing away from full fusion is required.

The search method begins by identifying as much fusion possible. A greedy fusion strategy is utilized by the compiler to return this highly fused program. Using this program as a seed, a genetic algorithm mutates the amount of fusion, which operations to partition and how to partition them, and also the size of the partitions. The custom representation utilized by the search tool describes characteristics which disallow the mutations to create many illegal programs and provides

enough information for the compiler to quickly determine the legality in the remaining instances.

This approach has been shown to be fast, portable, and to reliably find either the best, or very near the best, performance possible by BTO for a representative set of programs. See related work for additional performance details [10].

3.3 Advantages of BTO's program representation

One of the goals of this work is to automatically partition linear algebra operations. The task of partitioning data is regularly done in the context of optimizing compilers. One common approach to solve this problem is the use of the polyhedral model. General purpose tools such as the polyhedral model require loops or iteration spaces. In contrast, BTO starts with no loops, taking a high level description of linear algebra, for example $y = A \times x$. There are two motivations for this level of input. First this allows BTO to optimally generate loops as it sees fit and second this enables high level optimizations that would be difficult to identify in loop format. For example given the sequence of statement $y = A \times A^T \times x$ it is clear from the high level input that the ordering $A \times (A^T \times x)$ will require significantly less operations than $(A \times A^T) \times x$. Identifying this in loop form requires complex analysis.

BTO could perform a round of optimizations at the high level, produce loops and then utilize existing polyhedral frameworks to implement both loop fusion and data partitioning. However there are advantages to the custom method that make it worthwhile. Polyhedral tools are excellent at understanding and manipulating iteration spaces, but they do not have the ability to understand the linear algebra details being optimized. For example, polyhedral tools can fuse two loops successfully, but can not have enough information to determine that the fusion eliminated the need of a temporary data structure. Using the data flow graph approach, identifying this is a matter of checking the out edges of the temporary data structure in question to determine if there are other uses that require maintaining the data structure.

Another advantage of the representation used in BTO is that it can with various matrix storage formats. Specifically, all of the methods employed in BTO can easily be extended to handle

sparse storage formats that have irregular iteration spaces. The polyhedral model is limited to dense, well defined iteration spaces.

By keeping BTO specific to linear algebra and keeping the level of information high throughout the process, BTO has an advantage over general purpose tools. The one advantage to using existing tools would be larger development and research resources as well as additional testing and verification from increased use. However, we feel the advantages gained from the high level information and linear algebra specificity are worth while.

3.4 Correctness Testing

Any complex piece of code needs to be rigorously tested and the compiler portion of the BTO tool is no exception. This is done by ensuring the correctess of the code produced by BTO. Writing a test for each program that is fed into the compiler is time consuming, difficult, and error prone. To deal with this, an automated correctness tester has been implemented. The correctness tester works by comparing the output produced by BTO routines to a known correct output. This requires two steps; generating a routine that will provide the correct output, and generating code to compare the outputs of BTO and the correct routine.

3.4.1 Generating a known correct routine

The generation of a routine that is known to be correct relies on a correct BLAS library [15]. All of the operations that BTO understands are supported by BLAS. To convert to BLAS calls, each operation in the input program must be mapped to a BLAS call and the calls placed in the correct order. The data flow graph presented in Section 3.2.2 provides an ideal data structure for this. The data flow graph provides all of the ordering information and all of the information about type and size of data structures.

In the order imposed by the data flow graph, a BLAS call is created for each operation. For example consider the operation $y = alpha \times A \times x + beta \times y$, for which the data flow graph is show in figure 3.8. A call will be created for each of the four operations. The four BLAS

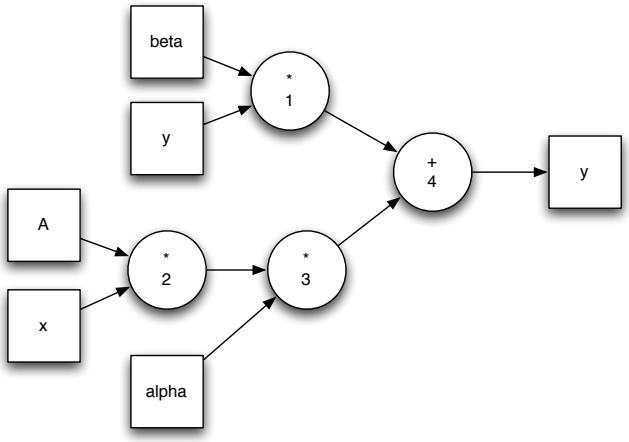


Figure 3.8: Example dgemv data flow graph ($y = alpha \times A \times x + beta \times y$).

```
BLAS calls
--------------------------
dscal   // tmp1 = beta * y
dgemv   // tmp2 = A * x
dscal   // tmp3 = alpha * tmp2
daxpy   // y = tmp2 + tmp3
```

Listing 3.6: BLAS calls generated for the dgemv kernel.

calls used for this routine are shown in Listing 3.6, requiring **dscal** ($y \leftarrow y \times alpha$), **dgemv** ($y \leftarrow A \times x \times alpha + beta \times y$), and **daxpy** ($y \leftarrow alpha \times x + y$). It is worth noting that this routine could be tested with a single call to **dgemv**. The reason for separating out each call is to simplify the conversion from graph to BLAS calls. The analysis required to identify that a particular set of vertices within a graph matches a BLAS call is considerably more complex than the method in use here which matches a single vertex. The advantage to the less complex analysis and use of BLAS is that the likelihood of an error in the translation is reduced. Also the use of BLAS this way will not dramatically effect the accuracy of the results for use in comparison. Finally, mapping single vertices to BLAS calls enables a known bound on error for each operation. This is required for comparison and is described more in section 3.4.2.

The downsides to this method are speed and efficiency. This method can require multiple reads of data structures that could be avoided with a better use of BLAS. Additionally this method often requires the creation of temporary data structures that would not be required with better use of BLAS routines. In practice, the speed and memory issues have not been a problem.

Each operation node has the appropriate details from the data types of the operands and the result to determine how to construct the BLAS call. Most of the information, such as data

structures, sizes, increments, and leading dimensions are simply translated directly into the call. The only source of complication is that BLAS calls are written to handle multiple cases, such as row or column major, and transpose or no transpose. All of the information required to generate the call in any of these cases is present in the type information, however each case must be treated appropriately to ensure correct results as well as correct error bounds.

3.4.2 Comparing BTO outputs to the known correct routine

The nature of floating point representation and rounding during operations makes comparison more difficult that just comparing two numbers. Fortunately, given the types of operations that BTO currently supports, it is possible to bound the acceptable error by a function of the number of operations and details of the base data type (i.e. double or float in C). This method relies on data that will not provide floating point problems such as subtractive cancelation or gross mismatches in magnitude which can accumulatively lead to large losses. This method also relies on the IEEE floating point standard which bounds error for basic arithmetic operations [43].

The correctness tester compares element wise rather than using norms for comparison. Given the nature of the code being generated, for example tiled and partitioned into parallel components, it is possible for distinct sub-portions of a data structure to have been computed using separate code paths. Minor code generation errors, for example swapping two stores (want to store C[i] = tmp0 and C[i+1] = tmp1 but generate C[i] = tmp1 and C[i+1] = tmp0) can result in errors localized in a specific sub-portion of the output data structure. Element wise comparison provides more information in locating issues such as those described here.

The error bounds are calculated while the data flow graph is being converted to BLAS calls. When each operation in the graph is converted to a BLAS call, the error bounds for the single output value can be calculated based on the number of operations for each element. This bound is known for each possible BLAS call. A single BLAS call is straightforword, however in a series of calls, error will propagate and this must be taken into account.

In practice the method that has worked for propagation of error bounds has been to propagate

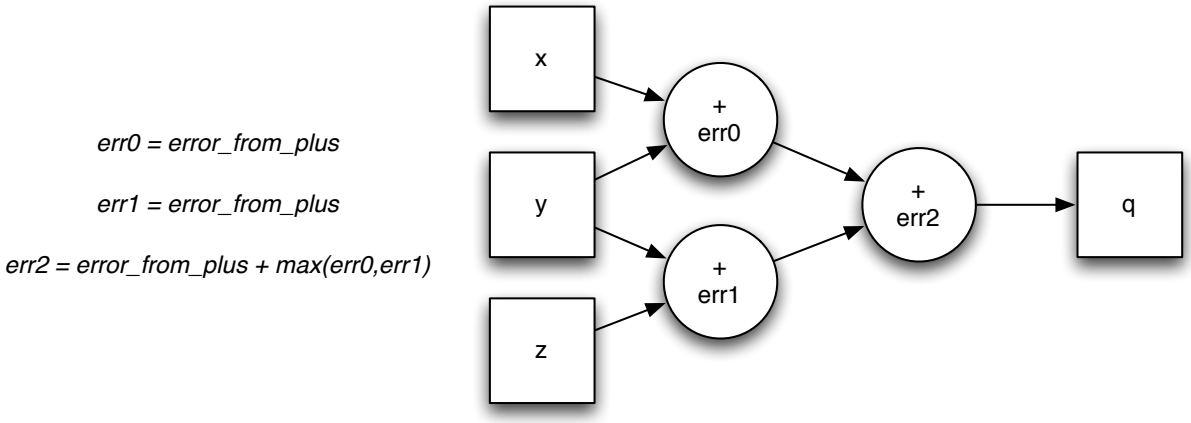


Figure 3.9: Error propagation example using $q = (x + y) + (y + z)$.

the largest error. All input data has zero error because the same input values are used for the reference routine as for the BTO routine. An operation with two input data structures for its operands will only have the error provided by the operation. An operation with error from one or two operands will have the error produced by the current operation plus the maximum error of the operands. Consider the case shown in Figure 3.9. The summation of vectors x and y will produce $err0$ from only the add operation because the input values have zero error. Similarly, the summation of y and z will produce err1 from only the add operation. When these two vectors are summed, the error could be as high as $err2 = error\_from\_plus + err0 + err1$. In practice however, bounding there error by $err2 = error\_from\_plus + max(err0, err1)$ keeps the bounds lower, still identifies errors and does not produce false positives.

3.5 Empirical Testing

The generation of an empirical tester is necessary to determine which set of optimizations performs the best on the given computer. An empirical tester is a simplification of the correctness tester. It needs only identify input and output data structures, fill them with valid data and place a call to the BTO routine. There must be a balance between providing reliable timing results and

performing quickly enough to be of use to the auto-tuning process. A single run of the routine may not fall within the timer resolution, or may not be completely representative because of other system noise. Some repetition is required to ensure valid data. The number of repetition in practice for data sizes on the order of fitting in the largest level of cache have been under 10. This provides consistent results while keeping the time of empirical testing relatively low.

3.6 Summary

This section has presented an overview of the BTO auto-tuning tool. Additional details were presented for the compiler portion of the tool, including a look at how matrices are represented, loops introduced, and how optimizations are performed. Additionally details of evaluating performance and correctness were discussed. The matrix representation and linear algebra rules presented are the rules that create the type system responsible for correctly compiled and optimized linear algebra. The next section presents these rules as a type system and discusses verification of these rules.

Chapter 4

Formalization of the Type System

The type system is the underpinnings of translating linear algebra from high level Matlab syntax to C as well as enabling the partitioning features. It is an extension of linear algebra rules presented in Table 3.1. If there is a flaw in this system, the approach used to generating loops and the partitioning of linear algebra operations described in this thesis break down. Aside from implementing the type system in BTO and thoroughly testing by comparing the output produced to BLAS versions of many routines, an additional and more rigorous approach was taken to formalize and verify the type system. An understanding of the details of the intermediate representation of BTO, specifically the data flow graph and the data representation as described in Section 3.2 is required to understand this chapter. This chapter begins by formalizing the type system and then showing two practical approaches to verifying that the type system works.

4.1 Type System

A data element in BTO is considered an abstract container where that container is described by a recursive type τ. Each data element is represented by a node in a programs data flow graph. A node is written as n_ℓ where ℓ is the unique identifier assigned to a node. The type environment Γ maps each of these nodes to their type. The type environment of a data flow graph is shown in Figure 4.1, where C is column, R is row, S is scalar.

Linear algebra consists of several operations, in this section $+$, $-$, $\times$, and transpose are discussed. Each of these operators must handle both overloading and recursion. The type of the

$O ::= C \mid R$ $\qquad\qquad$ $\Gamma ::= [n_\ell \mapsto \tau]$

$\tau ::= O\texttt{<}\tau\texttt{>} \mid S$ $\qquad\qquad$ $n_\ell \in$ Nodes of a program

Figure 4.1: The type environment of a BTO program where C is column, R is row, S is scalar and n_ℓ is the node given the unique identifier ℓ.

operands involved in the operation specify which version of the overloaded operator is correct. This is complicated by type recursion, because the overloaded instance of the operator does not have to remain the same for different levels of the type (think row $\times$ row vs. column $\times$ column rather than $+$ vs. $\times$). The correct instance of an operator requires the name of the operator, along with the type of the operands. Four type rules are presented in Figure 4.2, *Inputs* and *Outputs* for the input and output nodes, and *Oper1* for unary operators and *Oper2* for binary operators.

The *Inputs* and *Outputs* rules state that input and output nodes have the type τ from the type environment (these are known from the input program and entered into Γ initially). *Oper2* says that the type for an operation node n_3 is described by the name of the operator, the type of the left operand n_1 and the type of the right operand n_2. Table 4.1 describes how to determine this result type for the $+$ and $-$ operators. The correct instance of the operator is selected based on the operand types, where the left column specifies the left operand and the top row specifies the right operand. The center of the table represents the return type, for example, if the type of both operands are S, the top left corner of the table shows that the resulting type will also be S. An example of recursion is $R\texttt{<}\tau_1\texttt{>} + R\texttt{<}\tau_2\texttt{>}$, which says that the return type is $R\texttt{<}\tau_3\texttt{>}$ where τ_3 can be either τ_1 or τ_2, or in english, the type of the element of either the left or right operand.

Table 4.2 shows the same table for the $\times$ operator. In this table the left and right operand types are distinct and so they are annotated with $_l$ and $_r$ to distinguish the two. The table is similar to the $+$ and $-$ table, however it is worth noting that C$\texttt{<}\tau_1\texttt{>}$ * R$\texttt{<}\tau_2\texttt{>}$ can not be completely determined with just the operand types, the resulting type is also required.

Tables 4.1 and 4.2 describe the resulting type given the type of one or more operand, but

$$[Inputs] \qquad \Gamma \vdash n_\ell : \tau \quad \text{if } \Gamma(n_\ell) = \tau$$

$$[Outputs] \qquad \Gamma \vdash n_\ell : \tau \quad \text{if } \Gamma(n_\ell) = \tau$$

$$[Oper_2] \qquad \frac{\Gamma \vdash \tau_1 = \tau(\text{op}(n_3), \Gamma(n_1), \Gamma(n_2)) \qquad \Gamma \vdash \Gamma(n_3) = \tau_1}{\Gamma \vdash n_3 : \tau_1}$$
$$\text{where } [n_1, n_2] := \text{in\_edges}(n_3)$$

$$[Oper_1] \qquad \frac{\Gamma \vdash \tau_1 = \tau(\text{op}(n_2), \Gamma(n_1)) \qquad \Gamma \vdash \Gamma(n_2) = \tau_1}{\Gamma \vdash n_2 : \tau_1}$$
$$\text{where } [n_1] := \text{in\_edges}(n_2)$$

Figure 4.2: The type rules describing correct typing for inputs, outputs, and operation nodes in a data flow graph.

τ op$_{\{+,-\}}$	S	R<τ_2>	C<τ_2>	unknown
S	S	-	-	S
R<τ_1>	-	R<τ_3>	-	R<τ_1>
C<τ_1>	-	-	C<τ_3>	C<τ_1>
unknown	S	R<τ_2>	C<τ_2>	unknown

Table 4.1: The resulting types for op$_+$ and op$_-$ shown in the center of the table. The upper row represents the type of the right operand and the first column represents the type if the left operand. The - represents an illegal operation. τ_3 can be either of τ_1 or τ_2

τ op$_*$	S	R$_r$<τ_2>	C$_r$<τ_2>	unknown
S	S	R$_r$<τ_2>	C$_r$<τ_2>	unknown
R$_l$<τ_1>	R$_l$<τ_1>	R$_r$< R$_l$<τ_1>*τ_2 >	$\tau_1 * \tau_2$	unknown
C$_l$<τ_1>	C$_l$<τ_1>	{R$_r$< C$_l$<τ_1>*τ_2 >, C$_l$< τ_1*R$_r$<τ_2> >}	C$_l$< τ_1*C$_r$<τ_2> >	unknown
unknown	unknown	unknown	unknown	unknown

Table 4.2: The resulting types for op$_*$ shown in the center of the table. The upper rows represents the type of the right operand and the first column represents the type of the left operand. The operand types are annotated with $_l$ and $_r$ to distinguishthe left and right operands.

τ_{**}	Possible Operands as (left,right)
S	(S, S)
	(R<S>, C<S>)
R<τ>	(S, R<τ>)
	(R<τ>, S)
	(R<τ_1>, R<τ_2>)
	(R<τ_1>, C<τ_2>)
	(C<τ_1>, R<τ_2>)
C<τ>	(S, C<τ>)
	(C<τ>, S)
	(C<τ_1>, C<τ_2>)
	(R<τ_1>, C<τ_2>)
	(C<τ_1>, R<τ_2>)

Table 4.3: For each return type in the left column, the possible operand types are listed in the right column as (left,right) pairs (left $\times$ right).

determining the operand types based on a given resulting type is more complicated. Two additional tables are added to list the set of operand types that could have generated the given result type. Table 4.3 shows the set of possible pairs as (left operand type $\times$ right operand type) that could result from a known return type. For example, a scalar result type (S) shown in the first row of the table can be the result of either $S \times S$, or $R\langle S\rangle \times C\langle S\rangle$ (dot product). Each of $R\langle\tau\rangle$ and $C\langle\tau\rangle$ shown in Table 4.3 can result from five possible sets of operand types.

The corresponding table for addition and subtraction is show in Table 4.4. These operations are simpler; given the type of the result, the type of both operands must be identical to the result. This yields not a set of operand types, but the only possible types for the operands.

4.2 Constraint Based Implementation of Type System

The type rules and corresponding tables listed in Section 4.1 can be written as a set of boolean equations. When the conjunction of these constraints is satisfiable, the program is type safe. The constraints as boolean equations are shown in Figure 4.3, where $\tau ::= S \mid O\langle t\rangle$ and t is the type of an element of the container $O\langle t\rangle$ (t is used as convenient notation to express element, rather than uniquely identifying each element type which would unnecessarily explode the constraints).

$\tau_{++,--}$	Possible Operands as (left,right)
S	(S, S)
R<τ>	(R<τ>, R<τ>)
C<τ>	(C<τ>, C<τ>)

Table 4.4: For each return type in the left column, the possible operand types are listed in the right column as (left,right) pairs (left $\times$ right).

The *Result* annotation on a type means the type from the result operation, $Left$ and *Right* are used to annotate the types of the left and right type operands of a binary operation, and I is the predecessors type in the case of unary operations.

Consider the *Output* constraint, this says that the type of the result, τ_{Result} must match the type of the predecessor, τ_I. This makes logical sense, consider an operation of which the result is an output in the program, the result of the operation directly creates the output value. Recursion is seen in the $Multiply$ constraint. Consider scaling a column vector, where the left operand is the scalar and the right operand is the vector. Logically, the right operand and the result types must match, both being column vectors. The second line of the multiply constraint begins by requiring that the left operand have a type of scalar ($\tau_{Left} = S$) and ($\wedge$), that the outer level types of the right and result types match ($O_{Right} = O_{Result}$), in this case both C. These both hold, but the rules goes on to say the multiply constraint must hold true for the complete left type, and the elements of the right operand and result types ($Cons[*, t_{Result}, \tau_{Left}, t_{Right}]$). In this case the recursion would be checking if $S = S \times S$ which is true, so the overall operation is legal.

These boolean constraints have been written in the syntax of Yices [28], an SMT solver. This allows more than type checking an input program, it also allows inferring the types of operations so users do not need to specify these types. This is an advantage as loops are generated from the type assigned to each operation and BTO is attempting to do this in a memory friendly manner. BTO can generate these Yices constraints for a program and provide the type of only the input and output nodes. The set of values that satisfies the constraints contains the inferred type information for a correctly typed program. In the absence of a satisfiable set of values, the program cannot be

$$
\begin{array}{ll}
[Input] & Cons[\tau_{Result}] := \emptyset \\
[Output] & Cons[\tau_{Result}, \tau_I] := (\tau_{Result} = \tau_I) \\
[Addition] & Cons[+, \tau_{Result}, \tau_{Left}, \tau_{Right}] := (\tau_{Left} = \tau_{Right} \wedge \tau_{Left} = \tau_{Result}) \\
[Subtraction] & Cons[-, \tau_{Result}, \tau_{Left}, \tau_{Right}] := (\tau_{Left} = \tau_{Right} \wedge \tau_{Left} = \tau_{Result}) \\
[Transpose] & Cons[', \tau_{Result}, \tau_I] := ((\tau_{Result} = S \wedge \tau_I = S) \\
 & \quad \vee (O_{Result} \neq O_I \wedge Cons[', t_{Result}, t_I])) \\
[Multiply] & Cons[*, \tau_{Result}, \tau_{Left}, \tau_{Right}] := ((\tau_{Left} = S \wedge \tau_{Right} = S \wedge \tau_{Result} = S) \\
 & \quad \vee \ (\tau_{Left} = S \wedge O_{Right} = O_{Result} \wedge Cons[*, t_{Result}, \tau_{Left}, t_{Right}]) \\
 & \quad \vee \ (\tau_{Right} = S \wedge O_{Left} = O_{Result} \wedge Cons[*, t_{Result}, t_{Left}, \tau_{Right}]) \\
 & \quad \vee \ (O_{Left} = R \wedge O_{Right} = R \wedge O_{Result} = O_{Right} \wedge Cons[*, t_{Result}, \tau_{Left}, t_{Right}]) \\
 & \quad \vee \ (O_{Left} = C \wedge O_{Right} = C \wedge O_{Result} = O_{Left} \wedge Cons[*, t_{Result}, t_{Left}, \tau_{Right}]) \\
 & \quad \vee \ (O_{Left} = R \wedge O_{Right} = C \wedge Cons[*, \tau_{Result}, t_{Left}, t_{Right}]) \\
 & \quad \vee \ (O_{Left} = C \wedge O_{Right} = R \wedge O_{Result} = O_{Left} \wedge Cons[*, t_{Result}, t_{Left}, \tau_{Right}]) \\
 & \quad \vee \ (O_{Left} = C \wedge O_{Right} = R \wedge O_{Result} = O_{Right} \wedge Cons[*, t_{Result}, \tau_{Left}, t_{Right}]))
\end{array}
$$

Figure 4.3: Constraints expressing the type system with recursion and operation overloading.

correctly typed. Though not formally proven correct, extensive testing has shown this to be correct. Part of the verification is comparison to a second type inference method implemented before the types system was formalized. This is discussed in Section 4.3.

There are a few draw backs to using an SMT solver. The primarily challenge is that it is possible to write a program with this type system that can be typed multiple ways. BTO has the knowledge to select the typing for a program that maximizes memory efficiency, however Yices would return only one solution in these cases which may not be the most efficient. BTO could identify this case and re-query with more type information provided to force the remaining correctly typed programs, however this essentially requires creating a custom type inference in BTO to recognize these cases. This is the primary reason the formal constraints are not used for type inference, but as a method for checking a custom written type inference tool in BTO.

4.3 BTO Custom Type Inference Algorithm

This section describes the type inference method utilized by BTO using the type system and corresponding tables in Section 4.1. The results have been thoroughly compared to the formal constraints discussed in Section 4.2

The algorithm takes as input the data flow graph, the input types, the output types and the tables from Section 4.1 in their functional form. These functions return one or more type, where τ_{op} returns a set of possible types for the result of the operation and $\tau_{**,++,--}$ returns a list of pairs as described above. The algorithm returns Γ where Γ holds sets of types for each node in the graph. The algorithm continues iterating until all sets are of size one, or a fixed point is reached, meaning an incorrect input program. Programs that can be typed more than one way are handled by this algorithm. The algorithm is listed in Figure 4.4.

The algorithm begins in *Step 1* by assigning the supplied types of input and output nodes. All other nodes begin as an empty set. In then continues to *Step 2* which checks each node for one of three cases; 1) current node and all nodes directly preceding the current node have a single type, 2) the current type is unknown but the functional tables can determine some set of types for the current node, 3) the operation type is known, but one or more operand type is unknown and the functional tables can provide a list of types for both operands. In this last scenario, some node may have an existing set of possible types known already, and now it has a second set of types that is determined from the result type function. The intersection of these two sets becomes the possible types for the node.

This process continues until either all nodes have a single type assigned and the program is typed, or a fixed point is reached. When a fixed point is reached the algorithm looks for outer product operations, the only way ambiguity can exist in the type system. If one of these is present, and two types exist in the set of possible types for that node, one is selected based on performance traits and iteration continues. If an outer product is not present, the program contains some error in the types of the input or output nodes. For example column vector $\times$ column vector = column

vector would reach steady state with a set of operations for the operation node, this program is illegal.

4.4 Summary

This chapter has presented a formalization of the type system in use by BTO which describes correct linear algebra based on the recursive containers used to represent matrices and vectors. The type system has been implanted and tested rather than formally proven. Two separate implementations have been shown to match and testing has shown that loops generated from these types create correct linear algebra (discussed further in Chapter 7).

This section discusses these rules in terms of unpartitioned matrices and operations. It is worth pointing out that partitioning is an extension to the type system as described here. For example, if one wanted to partition an operation, an additional layer could be added to any type in the operation, and the tables presented here would describe the possible matching types for the remaining two nodes of the operation. This is discussed in further detail in the next Chapter.

INPUT: Inputs, Outputs, τ_{op}, τ_{**}
OUTPUT: Γ
METHOD: Step 1: Initialization

```
Γ = [ ]
for n in Nodes
    if n ∈ Inputs
        Γ[n] := {Inputs[n]}
    else if n ∈ Outputs
        Γ[n] := {Outputs[n]}
    else
        Γ[n] := { }
```

Step 2: Iteration

```
last := 0
while True do
    for n in Nodes
        if Γ[n] = {τ_n} ∧ ∀x in in_edges(n) : Γ[x] = {τ_x}
            continue
        if Γ[n] ≠ {τ_n}
            [l,r] := in_edges(n)
            Γ[n] = τ(op(n), Γ[l][0], Γ][r][0])
        else
            L := τ_*(op(n), Γ[n][0])
            update_types(Γ,in_edges(n),L)
    typed := have_types(Γ)
    if typed = num_nodes
        end
    if typed = last
        make_arbitrary(Γ)
    last := typed
end while
```

USING: have_types(Γ) is

```
cnt := 0
for n in Nodes
    if Γ[n] = {τ_n}
        cnt += 1
return cnt
```

update_types(Γ, e, L) is

```
;;L is a list of (τ_l × τ_r) and e := [l, r];;
for m in L
    for n in range(e)
        if Γ[e[n]] ≠ { } ∧ m[n] ∉ Γ[e[n]]
            L.erase(m)
            break
for n in range(e)
    Γ[e[n]] := {m[n] | m ∈ T}
```

make_arbitrary : It is possible to have an arbitrary decision.
In this case select based on performance estimations.
in_edges : node → ordered list of nodes
op : node → operation name
Inputs ::= $[n_\ell \mapsto \tau_*]$, Outputs ::= $[n_\ell \mapsto \tau_*]$
$n_\ell \in$ Nodes and τ_* is a concrete type.

Figure 4.4: Type inference algorithm utilized by BTO.

Chapter 5

Data Partitioning

Cache tiling, loop unrolling, vectorization, and data parallelism are all important optimizations in linear algebra. Identification and introduction of each of these optimizations can be generalized with an abstract representation of data partitioning. This representation enables BTO to understand only general data partitioning while being capable of expressing a range of optimizations. This section starts with a discussion of how data partitioning can represent important optimizations, discusses some of the rules required to automate partitioning, and finally covers problems associated with automated partitioning.

5.1 Optimizations enabled with data partitioning

There are several optimizations important to linear algebra that can be expressed with data partitioning. This includes extracting data parallelism to utilize multiple cores or vector units and modifying memory access patterns to improve data reuse.

Parallelism Parallelism is a necessary optimization as hardware trends show increasing core counts within a computer and systems being built with increasing compute nodes. Data parallelism in linear algebra can have several advantages, including increased computational power, increased cache and memory systems, and increased memory bandwidth. This feature is required to take advantage of any modern supercomputer and most personal computers. Most linear algebra libraries take advantage of speedups from data parallelism.

Cache Tiling Cache tiling is a primary tool when improving the utilization of the cache in linear algebra operations. A program is a good candidate for cache tiling when there is high amounts of data reuse and significant portions of the data structures cannot reside in cache. For example, matrix-matrix operations where reuse is often order N for each element and matrices can fall out of low levels of cache with orders as low as 64 elements. With matrix-vector operations, the only reuse occurs with the vector, but vector sizes must be much larger to not fit into caches which often reduces the effectiveness of cache tiling. Cache tiling works by modifying the access pattern of data structures, enabling a reused piece of data to be used multiple times once fetched to registers or memory. Beyond performance, cache tiling can reduce floating point error [6].

Loop Unrolling and Vectorization These two optimizations are closely related, and both are essential to high performance linear algebra. Loop unrolling enables vectorization when vector units are present; when vector units are not present this can reduce loop overhead. These optimizations are fundamental in vector-vector and matrix-vector operations where parallelism and cache blocking have less effect.

5.2 BTO Partitioning Technical Details

Using BTO containers to represent matrices and vectors enables partitioning of these structures without significant changes to the remaining BTO framework. Additionally, the nature of containers provides an abstract method for introducing data partitions. However, more than the ability to partition data is required to safely partition a program. Program partitioning requires the ability to partition operations. Operation partitioning is an extension of the type system presented in Chapter 4.

5.2.1 Terminology

Data partitions can be imposed on sets of linear algebra operations to enable a range of optimizations important to high performance numerical code. Partitioning can occur on both matrices and vectors, and for consistency, terms used throughout section will be described here.

This includes implementation details as these details are important in high performance code.

A scalar value is the smallest building block of the linear algebra data types. In memory a scalar value occupies $sizeof(type)$ where $type$ is a machine representation of numbers, for example 32 and 64 bit floating point representations as described in the IEEE floating point standard [43].

A vector of size n, has n elements. A vector has an increment associated with it which describes physical memory layout. For example an increment of 1 means the vector is contiguous in memory, while an increment of 2 means that the elements are spaced every other memory location. A vector of size n with increment 2 contains n elements, but requires $2 \times sizeof(type) \times n$ memory. A sub-vector is a portion of a vector, and has a size less than or equal to the size of its parent vector. Any sub-vector has the same increment as its parent vector.

A matrix of m rows by n columns has $m \times n$ elements. A matrix may be stored in memory with either rows (Row Major) or columns (Column Major) contiguous in memory. A row major matrix will have the first n elements stored contiguously in memory, follow by the next n elements. The contiguously stored dimension of the matrix is called the leading dimension. The number of memory locations consumed by the leading dimension can be greater than or equal to the number of elements in the given dimension. Given a row major matrix with n elements in each row and a $n + k$ ($k >= 0$) leading dimension there are k unused memory locations between any row and the next. The matrix will consume $sizeof(type) \times m \times (n + k))$ memory.

A sub-matrix is a portion of a parent matrix with size less than or equal to the parent matrix. Given a $m \times n$ matrix with leading dimension k, an $a \times b$ sub-matrix can be of size $0 <= a <= m$ and $0 <= b <= n$. The leading dimension of the sub-matrix will always match that of the parent matrix unless the sub-matrix is copied to some other memory location.

5.2.2 Data Partitoning

To explain partitioning, we introduce a new notation for a **type list** given as,

$$[O_n, \ldots, O_1] \text{ given the type } O_n\texttt{<}\ldots\texttt{<}O_1 < S\texttt{>>}, \tag{5.1}$$

where the list describes the ordered set of containers in a given type. Our notation for appending type lists is a comma and is overloaded to handle appending a single container or scalar type to a type list. In these examples

$$\begin{aligned} R, [C] \quad & \text{gives} \quad [R, C] \\ [R], [C] \quad & \text{gives} \quad [R, C] \end{aligned} \tag{5.2}$$

we demonstrate the creation of column major matrices.

Using this notation we can describe the single rule required to determine legal data partitioning of an individual container. Given some type, if the type list l contains a container with orientation O, it is legal to append a container with the same orientation to the type list, written as O, l. Consider a row vector $[R]$, the only legal method for partitioning the vector is to create a row of rows $[R, R]$. Figure 5.1 helps to visualize this partitioning, showing on the bottom a partitioned row vector. We can see from Figure 5.1 that we have introduced an additional row container to represent the partition. On the right, Listing 5.1 shows the loops and the memory access pattern of x for both the unpartitioned and partitioned cases.

In terms of the data structure represented by the container, we have modified many of the original details. The original vector (call this the outer vector now) has sub-vectors as elements and its increment will be in terms of these sub-vectors, so an increment of one will point to the next sub-vector. Each sub-vector will have an increment equal to that of the original vectors increment. BTO forces even partitioning, so any two sub-vectors will have the same size, with the exception of potentially one odd sized vector at the end. This sub-vector size can be determined one of two ways, either a fixed number of elements, i.e. 128, or a certain number of partitions, i.e. if the original vector size was n and we want 4 partitions, the sub-vector size will be $n/4$.

Matrices may be partitioned using the same set of rules. A desired matrix partition can be created based on where a partition is inserted in the type list and the order of partitioning the row or column dimensions. The data structure of a matrix partitioned this way will have its details modified similar to a vector. For example consider a matrix of type C<R<S>> partitioned as follows: R<C<R<S>>>. This partition can only effect the existing R container in the type. The new

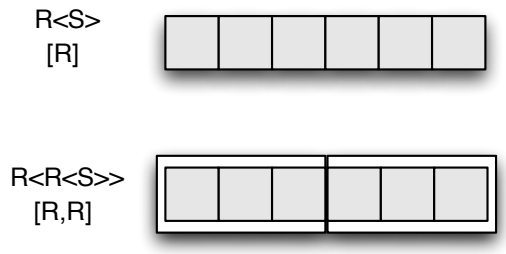


Figure 5.1: A row vector on the top and a partitioned row vector on the bottom.

```
for (i := 1 to n)
   x[i]

for (k := 1 to numBlocks)
    for (i := 1 to blockSize)
      x[k*blockSize + i]
```

Listing 5.1: Corresponding loops and memory accesses

outer row container will have sub-matrices as elements. The number of sub-matrices are based on how the partition is used as with vectors; partition to fixed size, or partition as $n/num\_partitions$. Similarly, the size of the inner row container will be determined by the method of partitioning. The leading dimension will never change with any partitioning because partitioning does not effect layout in memory, which is what leading dimension describes.

Using this method of adding containers to the recursive type, any possible partitioning is possible. The number and location in the type list of added containers will specify the number and dimensionality of the partition.

Currently BTO only allows the addition of the new container to the outside of the type, i.e. O_{new}, l, even though it is legal to place the new container anywhere higher than the container being partitioned. For example, one could partition $l_{upper}, O_{old}, l_{lower}$ as $l_{upper}, O_{new}, O_{old}, l_{lower}$. This inserts the new partition inside the type list rather than as the outermost type.

Of the two requirements, the first requirement says the new container must be higher than the old one in the type list, i.e. $[O_{new}, O_{old}]$ and not $[O_{old}, O_{new}]$. Consider this with the example of a vector $[C]$ that is partitioned to $[C_{new}, C_{old}]$. The only logical difference between this order and the reverse is that the attributes (size, step, etc) of the new and old types are swapped. The physical partition described remains the same, so this requirement does not limit expressiveness of partitioning, but has the benefit of halving the possible methods of partitioning that need to be considered.

The second requirement is the tradeoff between allowing an arbitrary placement above the old container versus only allowing the placement to be on the outside of the existing type list, i.e. O, l and not l_{upper}, O, l_{lower}. What is lost with the outside placement is the possibility to partition a matrix of $[C, R]$ to $[C, R_{new}, R]$, where R_{new} can be one or more desired partition (same for $[R, C_{new}, C]$). Given the correct order of introducing partitions, any other partitioning is expressible with the outside placement. Consider partitioning the C direction of the matrix $[R, R, C]$. By disallowing an arbitrary placement of the new C container the partitioning $[R, C, R, C]$ cannot be expressed. However consider building this matrix by a different order, starting with $[R, C]$, then adding $C, [R, C]$, then adding $R, [C, R, C]$ we can achieve the desired partitioning. This outer placement restriction places a significant cap on the possible ways to partitioning, and loses only a small amount of expressiveness, specifically two possible ways to partition a matrix. Certainly when a stronger method for program enumeration and evaluation exists, this tradeoff could be removed to enable expressing the last two ways of partitioning a matrix.

5.2.3 Operation Partitioning

BTO represents linear algebra operations as a sequence of binary and unary container operations. An individual operation is where the type system can be utilized to determine legal partitioning. Consider the case of a binary vector operation, $y \leftarrow x + z$. If BTO decides to only partition x, we can think of this as needing two loops to access x, but only a single loop to access y and z. There are ways that this could be written with loops, however this would represent a bizarre loop nest with no benefits and additional loop overhead. BTO does not allow this situation to occur by making data partitioning decisions based on operations.

Deciding how to partition data based on operations ensures that the operation remains correctly typed. A modification to the linear algebra rules presented in Table 3.1 creates a set of rules that describes how a linear algebra operation can be partitioned. These partitioning rules describe how to express a partition by modifying the types involved in the operation. Table 5.1 shows the partition rules. The first row shows how to partition addition and subtraction operations. First,

Algo	Operation	Type	List	Partition
`add/sub`	$\tau = \tau + \tau$	τ	u, O, l	O, u, O, l
`mult-bc`	$\tau_c = \tau_a \times \tau_b$	τ_b	u_b, R, l_b	R, u_b, R, l_b
		τ_c	u_c, R, l_c	R, u_c, R, l_c
`mult-ac`	$\tau_c = \tau_a \times \tau_b$	τ_a	u_a, C, l_a	C, u_a, C, l_a
		τ_c	u_c, C, l_c	C, u_c, C, l_c
`mult-ab*`	$\tau_c = \tau_a \times \tau_b$	τ_a	u_a, R, l_a	R, u_a, R, l_a
		τ_b	u_b, C, l_b	C, u_b, C, l_b
`scale`	$\tau = S \times \tau$ $\tau = \tau \times S$	τ	u, O, l	O, u, O, l
`trans`	$\tau_t = \tau_n$	τ_t	u_t, O_t, l_t	O_t, u_t, O_t, l_t
		τ_n	u_n, O_n, l_n	O_n, u_n, O_n, l_n
			where $O_t \neq O_n$	

Table 5.1: Linear algebra partitioning rules where algorithms marked with '*' will require a reduction operation.

the operation must match the description given in the **Operation** column; for addition this states that both operand and return types all must match. The remaining three columns describe how each type can be updated. For the `add/sub` algorithm there is only one unique type τ; and if that can be written as the set of appended lists and container described in the **List** column, then the operation can be partitioned. The **Partition** column describes the modification to the given type. In this example we can add a new container with orientation O to the outermost level of the type.

This table is based on the linear algebra rules in Table 3.1, and by extension, a rewrite of type rule Tables 4.2, 4.1, 4.3, and 4.4 which describe the constraints of a correctly typed linear algebra operation. Another way to think about the partitioning rules described here, would be to begin with a correctly typed linear algebra operation and randomly partition any of the containers in the operation based on the rules of container partitioning. The operation will remain legal when the remaining types involved in the operation are updated per the matching entry in the Tables. For example consider column vector addition, the type rule Table 4.1 shows that given a left operand of C<τ> and a right operand of C<τ>, the result must be C<τ>. Consider the unpartitioned vector addition is on types τ and a new C is added to the left operand. The table shows that both the right operand and the result type must also be updated with that same C type.

Partitioning an operation in isolation from the remaining program provides the potential for

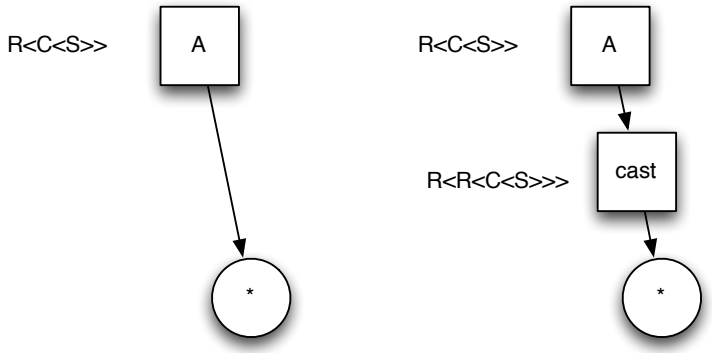


Figure 5.2: Example of introduction of cast node.

incorrect loop generation and can limit optimization potential. This is handled by inserting cast nodes each time a partition is introduced.

5.2.4 Cast Nodes and Partitioning

Cast nodes add or remove a level of partitioning from a given type. These nodes are required to represent certain partitioned operations but also enable more expressive ways of partitioning a complete program. These nodes do not have an effect on code generation, they simply describe partitioning details internally.

When a node is to be partitioned, a cast node is created with the same type as the node and is placed in between that node and the following node. Then the type of the cast node can be modified to express the desired partition. Figure 5.2 shows on the left the initial node to be cast (A) and on the right, the introduction of the cast node and the partition.

For Correctness Cast nodes are required to handle certain partitioned operations correctly. Consider an outer product using the same vector; $A \leftarrow x \times x^T$. If BTO wants to add a single partition to this operation, i.e. partition the left x vector and the rows of A, a cast node is required to separate x so that one operand can be partitioned and the other not. Figure 5.3 shows on the left the graph that represents the pre-partitioned operation and on the right the operation as partitioned with a cast node. From this figure it is clear to see that the partitioning cannot be

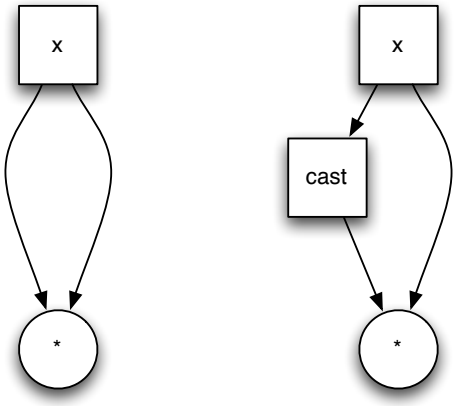


Figure 5.3: Example of case where cast node is required to express partitioning ($A \leftarrow x \times x^T$).

placed directly on x an still achieving the desired partitioning of the operation (if the type of x is changed directly, the type of the operation would need to be partitioned in both the row and column orientation to remain correct).

For expressiveness In the absence of cast nodes, when a shared node is partitioned, the only option is to force all connected operations to update based on the partitioning decisioning made for a single operation. Consider the following set of operations:

$$\begin{aligned} w &\leftarrow Az \\ y &\leftarrow Ax. \end{aligned} \tag{5.3}$$

The graph for this set of operations is shown in Figure 5.4 with lines through A and w representing a partitioning decision made on the left operation. Listing 5.2 shows that loops for partitioning only this left operation are possible to write. However, in BTO, if the type of A is modified directly as shown in this graph, the right operation is no longer correctly typed. The right operation can only remain correct if the partition decision is propagated through that operation. By introducing a cast node for the partitioning of the left operation, the right operation remains correctly typed. One can imagine a series of operations where forcing the same partitioning onto a data structure is not ideal. For example consider partitioning a shared vector which is used in a matrix vector multiply ($O(n^2)$) and is also used in a vector addition ($O(n)$). It may be beneficial

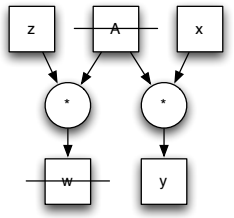


Figure 5.4: Example data flow graph of the operation $y \leftarrow Ax$ and $w \leftarrow Az$. A partition of the operation $w \leftarrow Az$ is represented with lines through A and w.

```
for (k := 1 to numRowBlocks)
    for (j := 1 to numColumns)
        for (i := 1 to rowBlockSize)
            w[k*rowBlockSize + i] +=
                    A[j*lda + k*rowBlockSize + i] * z[j]

for (j := 1 to numColumns)
    for (i := 1 to numRows)
        y[i] += A[j*lda + i]*x[j]
```

.

Listing 5.2: Loops shown when part of $y \leftarrow Ax$ and $w \leftarrow Az$ is partitioned. A has two distinct access patterns.

to partition the matrix vector operation for parallelism while not partitioning the vector addition at all. Cast nodes are required to express this type of program partitioning.

5.3 Multidimensional Partitioning

Multidimensional partitioning is required for two-dimensional matrix partitioning, or for expressing multiple features, such as both parallelism and cache tiling. Multidimensional partitioning requires no extension to the partitioning or type system rules, or the matrix representation. The recursive nature of these rules allows any number of partitions to be introduced to an operation and remain the correct linear algebra operation and still work with the loop introduction, and optimization phases of the compiler. Testing has shown this to work for up to four levels of partitioning, this is discussed further in Chapter 8.

The primary challenge with multidimensional partitioning is the size of the search space. With only a handful of operations, and two levels of partitioning, search times can increase significantly. Imagine a program with two operations that each can be partitioned in two ways. There are three versions to consider for each operation, no partitioning and the two ways. This means there are nine ways the program can be partitioned. If a second level of partitioning is introduced, there is un-partitioned, two ways for the first level, and for each of those, two more ways for the second

level, for a total of seven unique ways to partition one of the operations. This brings the program total to 49 unique ways to partition the program, a significant jump from nine. It is clear that for even moderately sized programs, multi-dimensional partitioning presents a search challenge.

BTO has been shown to be correct for multidimensional partitioning, but has yet to find an application where this is worth the significant increase in search time. Cache tiling and shared memory parallelism has, in certain cases, shown performance increases of a few percent. Distributed memory and shared memory parallelism is expected provide more significant performance gains, but this is left to future work.

5.4 Search of Fusion and Partitioning

Much of the pressure of searching partitioning decisions falls on the search tool portion of BTO. The search tool precisely describes how to partition each operation in a program. For each operation, this includes the numbers of partitions to introduce, and for each level of partitioning, the way to partition and the size of the partition. This limits the BTO compiler to only being responsible for ensuring the correctness of the described partitioning. The type system naturally covers this because of the recursive nature of the system.

The search tool also describes loop fusion, and BTO must be able to fuse the partitions that it has introduced. In the presence of cast nodes, BTO must have a method specifying that two casts can represent the same piece of data, and therefore are safe to fuse. The first requirement for this is that each level of the recursive type of two cast nodes have the same orientation. The second is that the symbolic size assigned to each of the containers can represent the same physical size. This is a book keeping issue, there are symbolic sizes that are created from program input, for example the number of rows of a matrix will be a user input, and those that are introduced by BTO. For example when a partition is introduced, a symbolic size is assigned that represents the number of partitions. It is safe for BTO to merge any of these sizes that it has introduced, while it is not safe for it to merge any of the sizes derived at program creation. Using these rules, BTO can determine when two cast nodes can be made the same cast node, allowing loop fusion.

5.5 Summary

Partitioning linear algebra is an extension of the type system presented in Chapter 4. In theory this works to any dimensionality of partitioning because of the recursive nature of the matrix representation and type system. In practice this has been shown to work up to four dimensions. In addition to this method of partitioning working with the type system and therefore loop generation, there are only minor changes required to support loop fusion of partitioned operations. This approach enables BTO to express nearly any program partitioning and utilize those partitions for cache tiling and share memory parallelism.

Further, this process works for various matrix storage formats as it is largely disconnected from the implementation details of the storage format. The next chapter explains the details of supporting various matrix storage formats and how that works with the partitioning ideas presented here.

Chapter 6

Matrix Storage Formats

A general matrix is a rectangular or square matrix with all elements stored in memory. For example a row major matrix is stored such that rows are contiguous in memory and then those rows are in memory at some fixed interval (the leading dimension). This format can represent any matrix, however there are several advantages to using other formats. The primary reason for additional formats is there are many common matrices that consist of regular patterns of large amounts of zeros, for example triangular and banded. Performance gains can be achieved by not using the zero values in the computation. Secondly, performance gains can come from never accessing the memory of large portions of a matrix that is zero or redundant. For example, symmetric matrices can be loaded with nearly half of the memory accesses as there are elements in the matrix. Finally, it can be beneficial to use less memory in storing the matrix, for example triangular packed allows only the non zero elements of a triangular matrix to be stored, utilizing nearly half the memory as compared to a general matrix. Throughout this section loops are used in discussions, often with C loops as examples, however the ideas presented here work for parallel loops, cache tiling loops, etc.

6.1 Representation of Loops and Memory Accesses is BTO

BTO supports several matrix storage formats. To understand how this works, the details of general matrices, their loop representation, and memory accesses are discussed. Understanding the general storage format enables discussion of triangular and triangular packed.

```
1  for (i = 0; i < M; i += 1) {
2    double *row = A + i * LD;
3    for (j = 0; j < N; j += 1) {
4        double element = row[j];
5    }
6  }
```

Listing 6.1: Loops to access row major matrix A of dimension MxN with leading dimension LD.

6.1.1 Details of General Matrices in BTO

A general matrix is described in BTO by the number of rows (M), the number of columns (N), and the leading dimension (LD) which describes how to move from one row or column to the next depending on row or column major. Consider writing two loops to access each element of a row major matrix. The memory friendly way is to iterate over the rows of the matrix in the outer loop, and then for each row, the inner loop iterates over the columns of the matrix. Listing 6.1 shows the corresponding C loops for some matrix A. To explain this example, loops are discussed separately from the mapping of iteration variables to memory locations.

6.1.2 Representation of Loops

Each loop is responsible for iterating over some portion of the underlying data structure, in this example, matrix A. BTO maintains three pieces of information for each loop; *initializations*, *conditions*, and *updates*. Listing 6.2 shows how this information maps to a C loop.

```
for ( initializations ; conditions ; updates )
```

Listing 6.2: Three pieces of information maintained by BTO and its mapping to a C loop.

The *initialization* portion of the loop for a general matrix is straightforward, the iteration variable is always set to zero, in the C example, this presents as $i = 0$. The *condition* for a general matrix states that the iteration variable is always bound by the number of elements in the

orientation of the current iteration, for example, if iterating over the rows of a matrix, this bound is the number of rows in the matrix. This presents in the C example as $i < M$. The final piece, the *update*, is simply adding a value of one to the iteration variable, in the example shown as $i \mathrel{+}= 1$.

There are alternatives to this approach, for example a loop iterating over the rows of a row major matrix could have the constraint of $i < M * LD$ and the update portion be $i \mathrel{+}= LD$ (M is number of rows in the matrix, LD is leading dimension). The motivation for the approach used in BTO is that it enables multiple containers (matrices and vectors) to determine their desired memory access all from the same iteration variable. The loop does not need to have information about the containers utilizing the iteration variable, it only needs to maintain the information about the dimensional or spatial bounds of the containers. This both simplifies the loop, and enables BTO to generically create loops with no knowledge of how the underlying containers actually traverse memory. As discussed further in Section 6.1.3, this allows any storage format that can map a spatial index into a memory location to work in BTO.

6.1.3 Mapping from iteration variable to memory

Referring back to Listing 6.1, the second part of accessing the elements of a general matrix is the pointer updates and the element accesses (i.e. $double * row = A + i * LD$). BTO provides an iteration variable that describes the spatial location in the current iteration space, where the iteration variable contains no information about where in memory any element is stored. spatial location describes the location of a particular element in the matrix as the number of elements in each dimension from some starting location. For example, consider accessing the memory element shown in grey in Figure 6.1. If the pointer A points to location 0, 0, then from this starting location, we can specify the location of the gray element, as $A[2, 3]$. Or in english, that is two rows down and 3 columns across. In general, this has nothing to do with how the matrix is stored in memory, however in this general matrix example it happens that there is a strong relationship between memory location and spatial index.

Each format supported by BTO then must map this spatial index into a memory location. For

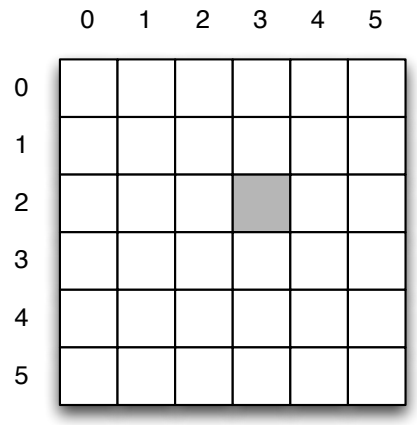


Figure 6.1: General matrix where each box represent and element and row and column indexes shown.

a general matrix, this map is shown in Table 6.1. In this table, *itr* represents the iteration variable, and LD is the leading dimension. For a row major matrix, to get to the i^{th} row, the map says the memory location from $ptr[0,0]$ is $i * LD$. To get the the j^{th} column, the map says the memory location is just j. Therefore, to access some element at row i and column j, BTO builds the pointer updates and accesses as: $ptr + i * LD + j$. In Listing 6.1, we see this as $double * row = A + i * LD$ and $double\ element = row[j]$.

6.1.4 Loops of Partitioned General Matrices in BTO

In terms of loops and mapping from iteration variable to memory, partitioned matrices behave no differently than unpartitioned matrices. Consider taking a 2-dimensional partition of a matrix where the block size in the row direction is $blkM$ and the block size in the column direction is $blkN$. Listing 6.3 shows the C code for one way this matrix could be accessed. Lines 1-4 represent the additions for the partitioning. We see that the loops are quite similar to the unpartitioned version, however in this case the update portion of the loop is not by 1, but rather by the block size. The pointer generation in lines 1-4 also use the same mapping, and by line 4, $blkCol$ is pointing to the 2-dimensional sub-matrix representing a single tile of the partitioned matrix.

Direction	Row Major	Column Major
Row	$itr * LD$	itr
Column	itr	$itr * LD$

Table 6.1: Mapping from iteration variable to memory location where itr is the iteration variable and LD is the leading dimension.

```
1   for (i = 0; i < M; i += blkM) {
2     double *blkRow = A + i * LD;
3     for (j = 0; j < N; j += blkN) {
4       double *blkCol = blkRow + j;
5       for (k = 0; k < blkM; k += 1) {
6         double *row = blkCol + k * LD;
7         for (w = 0; w < blkN; w += 1) {
8           double element = row[w];
9         }
10      }
11    }
12  }
```

Listing 6.3: Loops to access 2-dimensional partitioned row major matrix A of dimension MxN with leading dimension LD. The row block size is blkM and the column block size is blkN. Ignoring cleanup for clarity.

Lines 5-8 are nearly identical to the unpartitioned matrix example, they just happen to to have a new starting reference location that is a sub-matrix, specifically the pointer *blkCol* rather than A. As far as BTO is concerned, the loop generation for this sub matrix is the same as an unpartitioned matrix and the same mapping from iterator to memory applies.

The separation of loops and mapping from spatial location to memory location enables BTO to introduce loops, perform loop fusion and data partitioning free of the implementation details of the storage format. With this loop framework in place, supporting additional storage formats is a

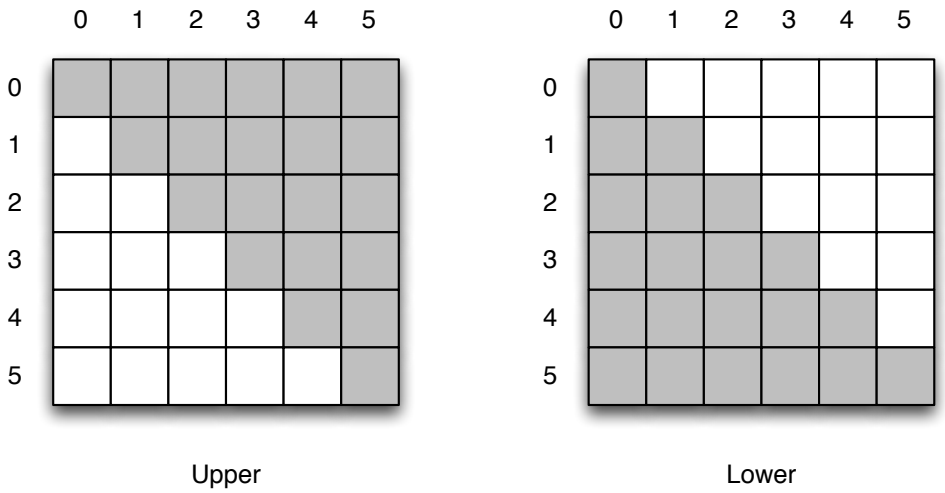


Figure 6.2: Upper and Lower triangular matrices where each box represent and element in memory and the gray boxes represent those that are non-zero.

```
1  for (i = 0; i < M; i += 1) {
2    double *row = A + i * LD;
3    for (j = i; j < N; j += 1) {
4        double element = row[j];
5    }
6  }
```

Listing 6.4: Loops to access row major **upper** triangular matrix A of dimension NxN with leading dimension LD.

```
1  for (i = 0; i < M; i += 1) {
2    double *row = A + i * LD;
3    for (j = 0; j <= i; j += 1) {
4        double element = row[j];
5    }
6  }
```

Listing 6.5: Loops to access row major **lower** triangular matrix A of dimension NxN with leading dimension LD.

matter of mapping the spatial indexing to memory locations.

6.2 Triangular

To demonstrate the flexibility of the loop system in BTO, support for triangular matrices was added. Triangular matrices consume a full NxN elements of memory, however only the specified (upper or lower) portion is ever accessed. Figure 6.2 shows both the upper and lower triangular matrices, where each box represents and element, but only the gray elements are every accessed.

Because triangular matrices consume NxN elements in memory and utilize a leading dimension to specify row or column location in memory, the map from a spatial iteration variable to a memory location is no different than a general matrix. The difference is in the generation of the loops. Consider accessing each element of a row major upper triangular matrix. The C loops for one approach are shown in Listing 6.4. In this unpartitioned example, the only difference over the general matrix format is on line 3, specifically, the initialization portion of the loop no longer begins with zero, it is now controlled by the current overall spatial location which it dictated by any loops above the current loop. In this case, just the loop on line 1, which provides the current location through iteration variable i.

Listing 6.5 shows the C loops for a row major lower triangular matrix. In this example we see the only difference as compared to the general format on line 3, this time, instead of changing the initialization portion of the loop, the condition portion of the loop is bound by the overall location, again determined by all outer loops (line 1, iteration variable i).

The unpartitioned cased is relatively straight forward, now consider a partitioned example. Listing 6.6 shows a 2-dimensional partitioning of a row major upper triangular matrix. The partitioning loops on lines 1 and 3 are similar to the unpartitioned case, only differing by the update portion of the loop (block size updates instead of 1). The loops on lines 5 and 7 require some explanation.

For the tiled loops on line 5 and 7, the bounds are listed in Table 6.2. There are two bounds to respect for both the beginning and the ending value of the iteration variable. These are the bounds that keeps the iterator value spatially within the local block, and the bounds that keep the iterator within the global requirements which in this case are the dimensions of the matrix and the diagonal. Consider the line 5 loop in Table 6.2, in this case three of the four required bounds are listed. However the global starting bound is implied by the local starting bound, for example BTO has guaranteed that the tile can never start at a location less than $A[0,0]$ so if the tile requires a start of 0, the global requirement of starting at 0 is also satisfied. Similarly, in the case of line 7, the global end bound is implied by the end of the local block. One might observe that bounding

```
1   for (i = 0; i < M; i += blkM) {
2     double *blkRow = A + i * LD;
3     for (j = i; j < N; j += blkN) {
4       double *blkCol = blkRow + j;
5       for (k = 0; k < blkM && k+i <= j+blkN; k += 1) {
6         double *row = blkCol + k * LD;
7         for (w = max(i+k-j,0); w < blkN; w += 1) {
8           double element = row[w];
9         }
10      }
11    }
12  }
```

Listing 6.6: Loops to access 2-dimensional partitioned row major upper triangular matrix A of dimension MxN with leading dimension LD. The row block size is blkM and the column block size is blkN. Ignoring cleanup for clarity.

the loop on line 7 by $blkN$ is incorrect if $blkN$ is not a multiple of N. This is presented this way for simplicity of explanation, in practice BTO determines the current tile size, correctly respecting local and global bounds.

Local bounds are determined as if the the local block was an unpartitioned version of the matrix storage format. Global bounds require tracking the overall current location, for example, from Table 6.2 (row titled line 7), $(i+k)-(j+w)$ is calculating the relative global location of the diagonal. This is found by summing all iteration variables from the row orientation and separately summing all of the iteration variables from the column orientation and finding the difference. When the tile is below the diagonal, relative diagonal location will dictate the ending global bound, and when the tile is above the diagonal, this diagonal location will dictate the global starting bound.

These bound requirements can be reduced to general rules which are shown in Table 6.3. In this table, **Direction** specifies if the loop is iterating across a row or column of a triangular

	Bound requirements of iteration variable	**Purpose**
line 5 (k)	$k >= 0$ $k < blkM$ $k + i <= j + blkN$	start is bound locally by block end is bound locally block end is bound globally by diagonal
line 7 (w)	$w <= 0$ $(i + k) - (j + w) >= 0$ $w < blkN$	start is bound locally by block start is bound globally by diagonal end is bound locally by block

Table 6.2: Bound requirement and reason for requirement of iteration variables for line 5 and 7 of Listing 6.6.

matrix. The **Relation to Diagonal** column specifies the currents blocks relation to the diagonal, a lower triangular matrix will be below the diagonal and an upper will be above. The **Constrained Value** column lists start when the initialization portion of a loop will be determined and end when the condition portion of a loop will be determined. Finally the **Constraint** column lists the requirements for *itr* to remain within the spatial bounds of the triangular matrix where *itr* is the iteration variable of the current loop.

Rows 1 and 2 apply in all cases to any type of matrix, ensuring that the current iterator remains within the bounds of the matrix and the current block. Row 3 states that given a lower triangular matrix, the condition (end bound) of the iteration is going to additionally be bound by its relation to the diagonal. There are no additional requirements on start, so it will just default to zero from the rule in row 1. The ending bound is determined by finding the difference of the overall row location and the overall column location, where in this case *itr* is contributing to the column location.

These rules can be made more generic to include an upper and lower bandwidth. Table 6.4 lists these rules (the spatial bounds of 0 and N are omitted, rows 1 and 2 of Table 6.3). This table lists the start and ending requirements for both row and column traversal based on the current spatial location and the bandwidth of the matrix. For example, the first row of the table shows that when traveling across a row, the global location must be within KL elements to the left of the diagonal and within KU elements to the right of the diagonal. The value in the **Loop Values**

	Direction	Relation to Diagonal	Constrained Value	Constraint
1	any	-	start	$itr > 0$
2	any	-	end	$itr < localSize$
3	across a row	below	end	$curRow - (curCol + itr) >= 0$
4	across a row	above	start	$(curCol + itr) - currRow >= 0$
5	across a column	below	start	$(curRow + itr) - currCol >= 0$
6	across a column	above	end	$(curCol + itr) - curRow >= 0$

Table 6.3: Rules controlling generation of triangular matrices. Constrained value lists start for determining an initialization value and end for determining a condition value for the BTO loop.

column specifies what BTO must generate for the *initialization* and *condition* portion of a loop to correctly control the iteration variable.

The rules in Table 6.4 control the loop creation of various matrix formats, specifically this will work with general, triangular, banded, and symmetric matrices, where general will set $KL = KU = N$ and triangular will set $KU = N$ for an upper triangular and $KL = N$ for a lower triangular matrix.

6.3 Triangular Packed

The discussion of loop creation for a triangular matrix demonstrated the ability of BTO to handle matrices that have elements that should not be used in computation, but were otherwise laid out in memory as if they were general matrices. To demonstrate the flexibility in the memory addressing of BTO, a triangular packed matrix format is utilized.

A triangular packed matrix stores a triangular matrix using only memory for the elements accessed, specifically $(N \times (N-1))/2 + N$ elements in memory. Figure 6.3 shows and example of how a row major packed upper triangular matrix is stored in memory. Where as a non-packed format can access an element by $i*LD+j$ where i is current row iterator, j current column iterator, and LD is leading dimension, the packed format is not as straight forward. The against the storage format traversal no longer has the regular leading dimension, meaning the $i * LD$ portion of the address calculation must be more sophisticated.

	Constrained Value		Constraint	Loop Values
1	row	start	$curRow - curCol <= KL$	$itr = max(curRow - curCol - KL, 0)$
		end	$curCol - curRow <= KU$	$itr <= KU - curCol + curRow$
2	column	start	$curCol - curRow <= KU$	$itr = max(curCol - curRow - KU, 0)$
		end	$curRow - curCol <= KL$	$itr <= KL - curRow + curCol$

Table 6.4: Rules controlling the generation of several matrix formats (the spatial bounds of 0 and N are omitted).

BTO handles this problem by requiring a mapping from a spatial location to a memory address for any supported storage format. In the general storage format this is straight forward. For example, as discussed in Section 6.1.3, for a row major general matrix, to get the i^{th} row, one must multiply i by the leading dimension, while to get the j^{th} column, one must add j. These mappings are not responsible for checking bounds, only providing the pointer to the correct element in memory. This means that a fully stored triangular format has the same mapping as a general matrix.

Consider again Figure 6.3, a row major upper triangular packed matrix. In order to access the diagonal element of the 4^{th} row, element numbered 15 in the figure, BTO uses the spatial row location $i = 3$ and finds the location as if the matrix were general, $i \times N$ or 18 in this example. It then subtracts off the un-stored portion of the matrix using $i \times N - (i \times (i - 1))/2$. This provides memory location 15 which is the diagonal element of the fourth row. Once in a given row, a spatial column location can determine the offset from the diagonal by subtracting the current row location from the current column location.

This approach allows for completely random access of the triangular packed structure based on the spatial location and the current overall location in the matrix. These are both pieces of information that BTO maintains. Because random access is supported, the triangular packed storage format can be partitioned in any orientation and in any number of dimensions.

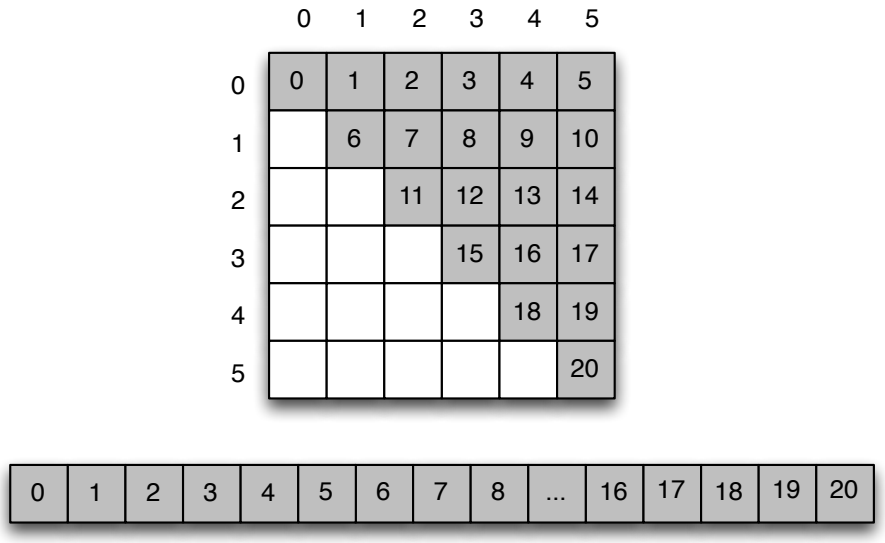


Figure 6.3: Example of how an row major upper triangular packed matrix is stored in memory.

6.4 Discussion of Extending Memory Mappings for Sparse Formats

A future goal of BTO is to support sparse storage formats. The framework put in place provides a strong foundation for this to occur. The primary difference will be that BTO needs to limit random access in certain cases. Consider a sparse format such as compressed sparse row, where at a high level, an index array provides memory offsets to the start of each row in some other element array. BTO can currently handle this storage format given the description of the mapping and loop details (initializations, conditions, etc). However, BTO will incorrectly try to partition in the column orientation where a fairly significant amount of work (as compared to existing pointer updates) is required to determine the offset and size of the block.

One approach is for BTO to extend its mapping from only supporting random access to also support a *get-next* mapping. This *get-next* would only allow for moving to the next element (be it element, row, column). If the sparse format described above only had a mapping for the column orientation that allowed for getting the next element, BTO would not be able to partition in that orientation because it could not map the random column spatial location to a memory location.

This change should be sufficient to allow BTO to generate correctly partitioned code for sparse formats.

6.5 Fusion of Triangular and Triangular Packed

The current high-level workflow of BTO is to partition each individual operation, then lower each individual operation and finally perform loop fusion. This approach has led BTO to make fusion decisions based on generic subgraph types (along with dependence analysis described earlier) which contain the information required to create a C loop. This means BTO only needs to check two subgraphs for compatibility to determine if it is legal to fuse. The matrix format details have already imposed their restrictions during the lowering phase so the fusion pass can remain completely general and ignorant of the format of the matrices accessed by any given loop.

This approach significantly limits the requirements of introducing a new storage format to BTO. Consider the alternative by trying to fuse a triangular matrix vector product with a general matrix vector product. There are certain loops that will be legal to fuse, specifically the loop over the triangular matrix that is across the memory layout. But others loops will be illegal, the loop with the memory layout over the triangular matrix will not be able to fuse with the general matrix in any way as the bounds are not compatible. In this example one could enumerate the legal fusions, however imagine the user determined they needed banded, general, and triangular matrix vector products fused. The user would be responsible for enumerating the fusability of each supported storage format with all others. The likelihood of getting this enumeration correct is low.

6.6 Summary

The separation of the loop generation from the memory address calculation allows BTO to optimize and generate C code for a range of matrix storage formats. One of the primary benefits of this approach is that loop generation can be driven by general information that describes the constraints of a matrix storage format. Additionally, a new storage format can be added to BTO with limited interaction with the majority of BTO's code base. Specifically, a programmer will

not be required to modify any of the optimization process which is the most precarious portion of BTO. Another benefit is that this approach is capable of extending to sparse storage formats.

Up to this point, a matrix representation capable of supporting many matrix storage formats and a type system that describes linear algebra and data partitioning has been presented. This has all been demonstrated in a compiler that can additionally perform loop fusion. The next chapter discusses the methods and approach for evaluating the compiler and by extension the ideas presented here.

Chapter 7

Validation

There are several ideas presented in this thesis that need to be validated, first, do the ideas presented in this thesis correctly translate a high level description of a sequence of linear algebra operations to a lower level implementation of the same sequence of operations. Provided the lower level implementation remained the correct sequence of linear algebra operations, the next piece to verify is whether or not the claimed code transformations exist in the implementation. Both of these topics have been validated in the context of the BTO compiler. In this Chapter, Section 7.1 discusses validating the correctness of the the translation from high level to low level code and Section 7.2 discusses how to ensure that a code transformation was correctly applied.

7.1 Validating Correctness of Translation from High Level to Low Level

The goal of the ideas in this thesis boil down to providing a method for a user of linear algebra to express their computation at a high level without regard for the details of implementing a fast program on some computer. This requires a compiler capable of understanding linear algebra at that high level well enough to produce a high performing low level implementation. A significant portion of the BTO auto-tuning tool is a compiler designed to translate from the high level to the low level (specifically C) and it must generate code that is correct. Due to the complexity of verifying that a translation is correct, compilers are often validated rather than proven correct. For example clang and gcc come with large test suites which are continuously growing and are the primary method of validation [56, 74]. The validation through testing approach is utilized for BTO,

which also has a continuously growing test suite.

7.1.1 Test Suite

The BTO test suite is a collection of routines added by several users with several different motivations. Many of the routines have been added because they represent real world use cases, while others have been added to demonstrate various performance characteristics. Finally, and most importantly, are those tests that have been added to stress test BTO.

This group of tests has been designed to exploit difficulties encountered by BTO at various stages of development. Stress tests have been added for type inference, loop fusion and data flow analysis, and robustness of the type system in general.

Type inference tests were added to ensure that any input program could be typed. In most cases the types in a program can be determined relatively straight forward. With most operations, given the types of any two of three (two operands and one result) containers in an operation the third type can be determined. This allows type information to travel from the known input and output types to the majority of operations in any direction (i.e. down the data flow graph from the inputs and up the data flow graph the outputs). There is however one case where an ambiguity can arise. This is related to the outer product operation, which requires the type of the result matrix to be known in order for an ideal loop ordering to be generated. This essentially means that type information is required to come from the result node and travel up the data flow graph for programs with outer products to type correctly. Several sequences of outer products strung together can expose this, putting pressure on the type inference algorithm. Many of these test cases are present in the test suit.

Loop fusion tests have been added as a more complete understanding of loop fusion as represented by data flow graph and the type system arise. The primarily challenge with loop fusion has not been the legality of fusion because of types, it has been data flow analysis. Several tests consisting of complex data flow graphs have been created to ensure that data decencies are always respected.

Robustness of the type system is one of the main ideas of this thesis and a great deal of tests have been generated specifically to exercise the possible type combinations one may encounter when writing any basic linear algebra program. Nearly exhaustive unit tests have been created to ensure that addition/subtraction and multiplication all work as intended. For addition this consists of scalar, both orientations (row/column) of vector-vector operations and both orientations of matrix-matrix operations, and for each of these, their transpose. Multiplication has far more possible combinations, all un-transposed cases of matrix-matrix multiply exist in the test suite. This covers matrix-vector operations, which are a simplification of matrix-matrix multiply. For matrix-matrix, the four cases transposing the two operand matrices are also included in the test suite. In addition to the matrix-matrix tests, further transpose cases exist for matrix-vector operations. This includes the transposing of both operands and the result as well as left and right multiply by the matrix. Scaling of matrices and vectors is also included in the test suite. There are too many cases of multiplication to enumerate all tests, however, the various transpose cases become redundant with the un-transposed cases.

Matrix formats are all tested in the same manner. The tests described in the robustness of type system have been duplicated for general, triangular and triangular packed matrices. Similarly, many of the loop fusion tests have been duplicated for the three listed storage formats. Type inference is not dependent on storage format, and duplicating tests has not been required.

7.1.2 Partitioning

The verification of partitioning is an extension of the targeted testing in the previous section. Because the type system is recursive by nature, the majority of the challenges remain the same in the presence of partitioning. The primary difference is the additional degree of freedom; the amount of partitioning requested by the user.

Exhaustively testing partitioning is time consuming, to the point that for certain routines, it becomes impossible to complete a test in any usable time frame for extensive partitioning. Exhaustive testing of smaller routines has been successfully completed for up to four levels of

partitioning. Exhaustive testing of two levels of partitioning has been completed for nearly the entire test suite. Testing using faster search strategies has been completed for two levels of partitioning for the entire test suite.

7.1.3 Comparing to BLAS

A test suite either has to be self verifying, or have something to compare to. One advantage in this application is that any program that BTO can generate can written in a sequence of BLAS calls. BTO automatically generates a sequence of BLAS calls with the primary intention of testing. The details of the test generation and floating point error analysis are covered in Section 3.4.

There are two notable drawbacks to this approach for verification; 1) how can one know the BLAS library is correct, and 2) how can one know the test generator is correct. The first point is relatively easy to deal with. Many implementations of the BLAS library exist. Testing has been performed with Netlib, Apple's Accelerate, Intel's MKL, and AMD's ACML. As further evidence this approach works, existing bugs in two of these libraries have been discovered forcing the use of other BLAS libraries.

The second drawback, how can one know the test generator is working, is more difficult. BTO has been through many incarnations, and for the first many years of its existence, correctness testing was performed with hand written tests comparing to hand written implementations of a corresponding BLAS version. This collection of initial tests provided confidence that at the time of writing BTO's test generator, BTO was correctly generating code for the current test suite (or so it was thought). It turned out that the automatically generated BLAS version caught many problems with BTO and the new approach was more rigorous than hand writing individual tests. Even so, this is one area that must be handled lightly, and it is recommend that any new features (i.e. storage formats) are hand tested initially as well.

7.2 Validating Optimizations

Claims are made that using the type system and data representation, loop fusion, array contraction, and partitioning for parallelism can be performed. It is important to show that those transformations are indeed happening as intended. BTO implements these optimizations and shows that the various routines generated remain the correct linear algebra operation. Two questions remain; 1) how to ensure that optimizations are indeed being applied when possible, and 2) when BTO claims an optimization was performed, how can one be sure it was?

7.2.1 Optimization Coverage

Ensuring that optimizations are being applied in all legal cases is one of the most challenging aspects of developing the BTO compiler. No automated method has been discovered to ensure this is happening, however hand comparisons have been made for several representative routines.

For a period of time, BTO employed an analytic model that made performance predictions based on loop ordering and loop fusion [50]. In the process of interfacing the analytic model with BTO, a significant by-hand comparison was performed of expected fusion results from the models standalone tool to those results that were generated by BTO. In this process, several missing loop fusion opportunities were discovered and resolved. The standalone tool would generate a predicted runtime that was close to the actual runtime. In the event that there was a significant mismatch between predicted and actual, further investigation would be performed by looking at the operation and the C loops to determine if fusion potential was missed. This approach was able to catch missing loop fusion and array contraction. This comparison was beneficial, however was limited in scope as it was primarily focussed around loop fusions that were expected to be beneficial.

Further comparisons were made by hand using targeted tests of difficult to fuse sequences of routines. These comparison were done be comparing a data flow graph output by BTO which depicts loops, fusion, and partitioning to expected combinations of fusion. This comparison consists of determining maximal fusion and array contraction, and ensuring that particular routine was

generated. Then other fusion combinations could be compared by extrapolating back the various combinations of fusion that were possible to lead up to the maximum fusion. The graph's generated by BTO provide a quick way to evaluate if the fusion was performed. Because these graphs directly dictate code generation, if the fusion was present in the graph, the C code was guaranteed to have the fusion. The draw back here is the process is time consuming and in more complex cases, it was possible for BTO to find fusion that unexpected, though legal.

When possible, regression testing was performed. This consists of comparing the number of versions generated to archived counts of expected versions. Any decrease in the number of versions signaled a possible missing optimization. Unfortunately, BTO is a research compiler and significant changes capable of modifying the expected number of routines generated was common. For example, partitioning, introduction of cast nodes, and relaxation of optimization requirements would all cause a change in the number of generated routines that made comparing to historic values difficult.

The most likely way to rigorously verify that optimizations are not being missed is through a formalization of the optimization rules similar to those presented in Chapter 4 for the type system. The understanding of the details required to formalize loop fusion and array contraction are just starting to come together. With a fusion formalization and a SAT solver it may be possible to enumerate all legal fusion, partitioning, and array contraction programs to have a point of comparison. This is left to future work, however Appendix B presents some work toward the formalization.

7.2.2 Generating Expected Optimizations

Determining if BTO actually generates the set of optimizations it believes it did is an easier problem. BTO has been interfaced with a general purpose search tool. Under these conditions, BTO is given an exact specification of a routine. This specification describes which operations need to be fused, how they need to be fused, and also how to partition each of the operations. The representation is a sequence of digits that are interpreted to describe the routine. The details

of interpretation are not significant here, however the sequence of digits is simple enough that enumeration is possible, so coverage is thorough.

This provided description of a routine enables an automated check to take place after the optimization phase and before code generation using the data flow graph. The data flow graph after the optimization pass exactly describes any optimizations that exist. For example if fusion has been performed with operations 1 and 2, a sub-graph nest will exist containing both operations 1 and 2. If partitioning is present, the types associated with the partitioned operation must reflect that partitioning in both depth of type and orientation as described in the description.

Discrepancies between graph and specification raise an error in the compilation process. At first, there were several instances when the compiler thought that an optimization had been performed, however the graph rewrites had failed and the optimization was not present in the data flow graph. This demonstrated that the method is effective. Comparing the data flow graph to a specification is fast, therefore this check is always performed. This has improved the soundness of the optimization pass and inspires great confidence that the expected optimizations are indeed present in the generated routine.

7.3 Summary

This section has described the test based approach to verifying that BTO is correctly translating a high level linear algebra specification to C and that optimizations are applied when BTO believes them to be. The testing is thorough and instills confidence in the correctness of the transformation and the presence of expected optimizations. The verification that is lacking is that of optimization coverage. The understanding of the optimizations is approaching a point where a more formal verification or test can be created to ensure full coverage. At present state, hand verification has instilled confidence that fully fused and well partitioned versions of routines are being generated. Hand searches and other tools have demonstrated that these tend to be the best performing optimizations, so if coverage is not complete, it appears not to be a significant problem performance wise for the current test suite.

This chapter discusses testing that shows that in the majority of cases BTO is performing the correct translation and applying the optimizations it claims. The next chapter evaluates that output from BTO to ensure that the ideas create code competitive with existing approaches.

Chapter 8

Evaluation

Any auto-tuning tool must meet two requirements to be successful; 1) the performance of generated code is fast enough to warrant using the tool, and 2) the search times are fast enough that a programmer can realistically use the tool. This section presents data that shows BTO is capable of meeting both of these requirements.

8.1 Test Environment and Kernels

The results in this section demonstrate BTO's ability to optimize the kernels shown in Table 8.1. Some of these kernels respond well to loop fusion and data parallelism while others do not. Some of these kernels are in the updated BLAS [14] but have not been adopted by vendor-tuned BLAS libraries. These kernels also represent various uses of the BLAS. For example, the DGEMV kernel maps directly to a BLAS call while others are equivalent to multiple BLAS calls. As an example, Listing 8.1 shows the sequence of BLAS calls that implement the BICGK kernel.

The computers used for testing include recent AMD and Intel multicore architectures which are described in Table 8.2.

```
// q = A * p
dgemv('N', A_nrows, A_ncols, 1.0, A, lda, p, 1, 0.0, q, 1);
// s = A' * r
dgemv('T', A_nrows, A_ncols, 1.0, A, lda, r, 1, 0.0, s, 1);
```

Figure 8.1: Example sequence of BLAS calls that implement BICGK.

Table 8.1: Kernel specifications.

Kernel	Operation
AXPYDOT	$z \leftarrow w - \alpha v$ $\beta \leftarrow z^T u$
VADD	$x \leftarrow w + y + z$
WAXPBY	$w \leftarrow \alpha x + \beta y$
ATAX	$y \leftarrow A^T A x$
BICGK	$q \leftarrow Ap$ $s \leftarrow A^T r$
DGEMV	$z \leftarrow \alpha Ax + \beta y$
DGEMVT	$x \leftarrow \beta A^T y + z$ $w \leftarrow \alpha Ax$
GEMVER	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$ $x \leftarrow \beta B^T y + z$ $w \leftarrow \alpha Bx$
GESUMMV	$y \leftarrow \alpha Ax + \beta Bx$

8.2 General Matrix Performance Evaluation

Much of the focus of this work has been on general matrix storage formats. This section presents the evaluation of the performance of the general storage format only. This section compares the performance of BTO-generated routines and several state-of-the-art tools and libraries that perform similar sets of optimizations, as well as hand-optimized code. The BTO compiler generates code that is between 16% slower and 39% faster than hand-optimized code. The other automated tools and libraries achieve comparable performance to BTO and hand-optimized code for only a few of the kernels.

8.2.1 Details of Setup, Tools and Tested Code

To place BTO performance results in context, the results are compared with several state-of-the-art tools and libraries. BTO performs loop fusion and array contraction and makes use of data parallelism. However for more architectural specific optimizations, such as loop unrolling and vectorization, BTO relies on the native C compiler. The two compilers used for this comparison are the Intel C Compiler (ICC) [45] and the PGI C Compiler (PGCC) [66]. Both ICC and PGCC

Table 8.2: Specifications of the test machines.

Processor	**Cores**	**Speed (GHz)**	**L1 (KB)**	**L2 (KB)**	**L3 (MB)**
Intel Westmere	24	2.66	12 x 32	12 x 256	2 x 12
AMD Phenom II X6	6	3.3	6 x 64	6 x 512	1 x 6
AMD Interlagos	64	2.2	64 x 16	16 x 2048	8 x 8

unroll loops and vectorize. They also identify and exploit data parallelism and perform loop fusion. Through examination of the compiler output as well as the generated assembly, it was clear that both ICC and PGCC did a good job at generating vectorized code in nearly all routines tested. Vectorization is an important optimization in the routines being evaluated, but one that BTO relies on the native compilers to perform. In the absence of a good vectorizing compiler, BTO would need to generate vectorized code to be competitive with existing libraries for certain routines.

The first two comparisons compile C loops with ICC and PGCC, which represent the best commercial compilers for the targeted processors. The third comparison is using Pluto [18], a source-to-source translator capable of performing loop fusion and identifying data parallelism. The fourth comparison is using Intel's Math Kernel Library (MKL) [45] which is a vendor-tuned BLAS implementation targeting Intel CPUs and AMD's Core Math Library (ACML) [5] targeting AMD CPUs. The fifth is a hand-tuned implementation (applying loop fusion, array contraction, and data parallelism) created by an expert in performance tuning who works in the performance library group at a major vendor.

The input for ICC, PGCC is a translation of the BLAS calls to C loops. Where appropriate certain portions of the BLAS call would be removed. For example if a call to DGEMV is used without the scaling by *alpha* or *beta*, then that part of DGEMV will not be written. As an example, the C code for BICGK is shown in Listing A.1 in Appendix A. Pluto requires an input without any pointer updates. The C code shown is Listing A.1 is modified for Pluto as shown in Listing A.2 in Appendix A. Pluto also requires perfectly nested loops, so in certain situations the

	Compiler Flags
ICC	**-O3 -mkl -fno-alias**
PGCC	**-O4 -fast -Mipa=fast -Mconcur -Mvect=fuse -Msafeptr**

Table 8.3: Compiler flags. (Msafeptr not used on Interlagos).

zeroing of accumulators needed to be hoisted out of loops for Pluto to work correctly. An example of this can be seen with the zeroing of the vector s in Listing A.2. The output generated by Pluto is shown in Listing A.3. For comparison, the BTO and hand tuned versions are also shown in Appendix A in Listings A.4, A.5, A.6, and A.7.

The compiler flags for ICC and PGCC are shown in Table 8.3, experimentation showed these to be the best combinations for the routines being evaluated. These flags enable vectorization as well as automatic parallelism. Data parallelism is exploited by ICC, PGCC, Pluto, and MKL by using OpenMP [26]. When Pluto generated OpenMP code, ICC and PGCC would not try to add an additional layer of parallelism. BTO and the hand-tuned versions utilize Pthreads [63]. In most cases, experimentation showed a slight advantage to using OpenMP.

8.2.2 BTO Compared with Existing Tools for General Matrices

A detailed comparison of BTO and the five other approaches for generating high performance code on the Intel Westmere are presented. Due to similarities in results, only a brief summary of results on the AMD Phenom and Interlagos are presented.

Figure 8.2 shows the speedup relative to ICC on the y-axis for several linear algebra kernels. (ICC performance is 1). The raw data is shown in Table A.1 in Appendix A. On the left are the three vector-vector kernels, and on the right are the six matrix-vector kernels, all from Table 8.1.

PGCC tends to do slightly better than ICC, with speedups ranging from 1.1 to 1.5 times faster. Examining the output of PGCC shows that all but GESUMMV and GEMVER were parallelized. However, PGCC's ability to perform loop fusion was mixed; it fused the appropriate loops in AXPYDOT, VADD, and WAXPBY but complained of a "complex flow graph" on the remaining

kernels and only achieved limited fusion.

The MKL BLAS outperforms ICC by factors ranging from 1.4 to 4.2. The calls to BLAS routines prevent loop fusion, so significant speedups, such as those observed in AXPYDOT and GESUMMV, can instead be attributed to parallelism and well tuned vector implementations of the individual operations. Unfortunately there is no way determine why the BLAS version performs so well for AXPYDOT as the sources are not public. Surprisingly, the BLAS DGEMV does not perform as well as Pluto and BTO. Given the lack of fusion potential in DGEMV, it is likely that differences in the parallel implementations are the cause; evidence suggests that MKL may not be parallel for DGEMV at these sizes.

The Pluto results show speedups ranging from 0.7 to 5.7 times faster than ICC. The worst-performing kernels are AXPYDOT, ATAX, and DGEMVT. These three kernels represent the only cases where Pluto did not introduce data parallelism. For the remaining two vector-vector kernels, VADD and WAXPBY, Pluto created the best-performing result, slightly better than the BTO and hand-tuned versions. Inspection shows that the only difference between Pluto, hand-tuned, and BTO in these cases was the use of OpenMP for Pluto and Pthreads for hand-tuned and BTO. The fusion was otherwise identical and the difference in thread count had little effect. For the matrix-vector operations, if we enabled fusion but not parallelization with Pluto's flags, then Pluto matched BTO with respect to fusion. However, with both fusion and parallelization enabled, Pluto sometimes misses fusion and/or parallelization opportunities. For example BICGK was parallelized but not fused. The GEMVER results depend on the loop ordering in the input file. For GEMVER, Pluto performed either complete fusion with no parallelism or incomplete fusion with parallelism; the latter provided the best performance and is shown in Figure 8.2.

The hand-tuned implementation is intended as a check to ensure expected results were achieved by each of these tools. For the vector-vector operations, the hand-tuned version is within a few percent of the best implementation. Typically, the fusion in both the hand tuned and the best tool based version are identical with the primary difference being either thread count or what appears to be a difference between Pthreads and OpenMP performance. In the case of the matrix-

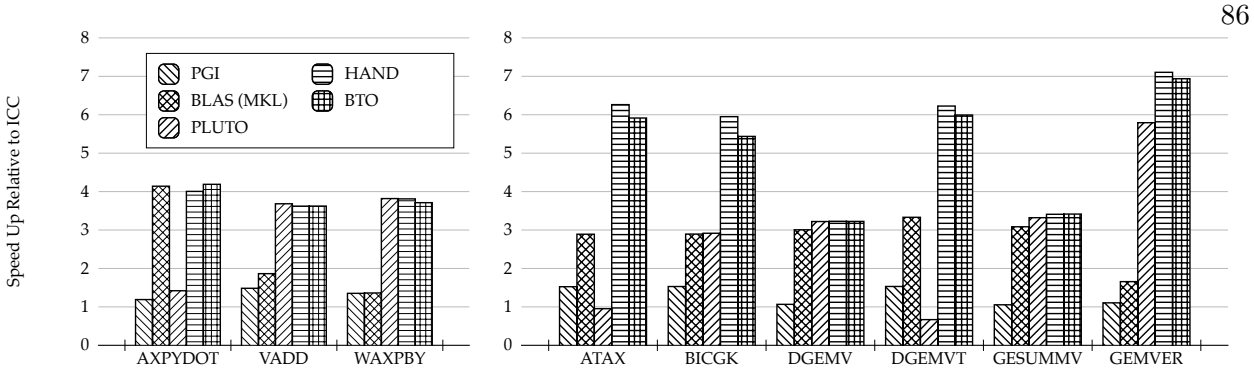


Figure 8.2: Performance data for Intel Westmere. Speedups relative to unfused loops compiled with ICC (ICC performance is 1 and not shown). The left three kernels are vector-vector while the right six are matrix-vector operations. In all cases, BTO generates code that is between 16% slower and 39% faster than hand-optimized code and significantly faster than library and compiler-optimized versions.

vector operations, the hand-tuned version is the best for all but DGEMV and GESUMMV, where it is equal to the best.

The BTO performance results show speedups ranging from 3.2 to 6.9 times faster than ICC. For the vector-vector operations, the performance is similar to the hand-tuned version in all cases. Inspection shows that for AXPYDOT, BTO was slightly faster than the hand-tuned version because BTO did not fuse the inner loop while the hand-tuned version did. BTO performed slightly worse than hand-tuned on WAXPBY because of a difference in thread count. Similarly, the performance of the matrix-vector operations is close but slightly lower than that of the hand-tuned version. BTO fused the same as hand-tuned for BICGK, GEMVER and DGEMVT with the only difference being in thread count. For ATAX, both BTO and hand-tuned fused the same and selected the same number of threads, but BTO was slightly slower because of where it zeroed out a data structure. In the hand-tuned version the zeroing occurred in the threads, while in BTO's case it occurred in the main thread.

Similar results on an AMD Phenom and AMD Interlagos are shown in Figures 8.3 and 8.4 respectively (Raw data shown in Tables A.2 and Table A.3). The Pluto-generated code for the matrix-vector operations tended to perform worse than that produced for the other methods evaluated. On these AMD based computers, achieving full fusion while maintaining parallelism

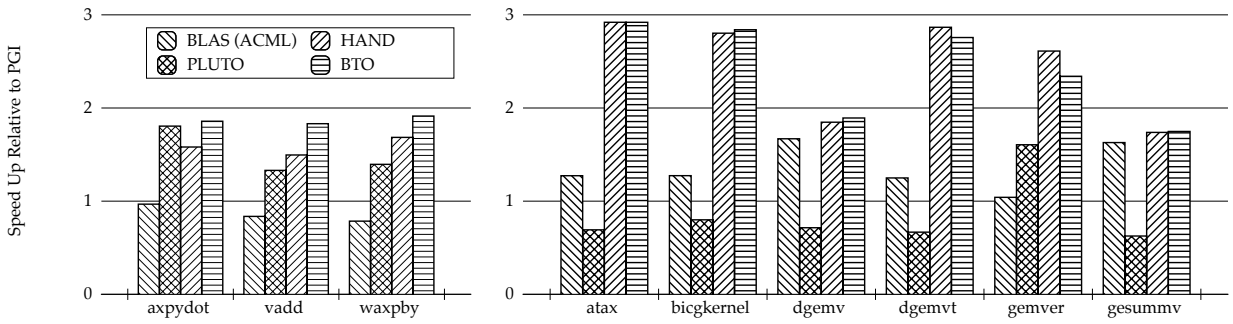


Figure 8.3: Performance data for AMD Phenom. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). The left three kernels are vector-vector while the right six are matrix-vector operations. In all cases, BTO generates code that is between 10% slower and 38% faster than hand-optimized code and significantly faster than library and compiler-optimized versions.

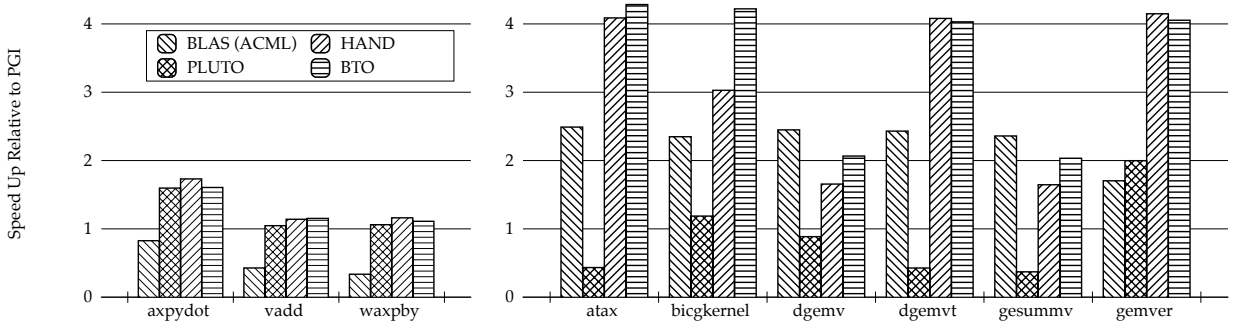


Figure 8.4: Performance data for AMD Interlagos. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). The left three kernels are vector-vector while the right six are matrix-vector operations. In all but two cases, BTO generates code that is between 7% slower and 28% faster than hand-optimized code and significantly faster than library and compiler-optimized versions.

is of great importance. As previously discussed, Pluto tended to achieve fusion or parallelism but struggled with the combination. These results demonstrate the difficulty of portable high-performance code generation even under auto-tuning scenarios.

The number of threads selected by BTO for Figures 8.3, 8.3, and 8.4 are shown in Section 8.3, Table 8.4. The 24 core Westmere had threads counts of between 8 and 24, the 6 core Phenom between 2 and 6, while the 64 core Interlagos has between 10 and 24 threads selected. This variability in thread count within each architecture demonstrates the difficulty in tuning these

routines and further demonstrates the importance of auto-tuning.

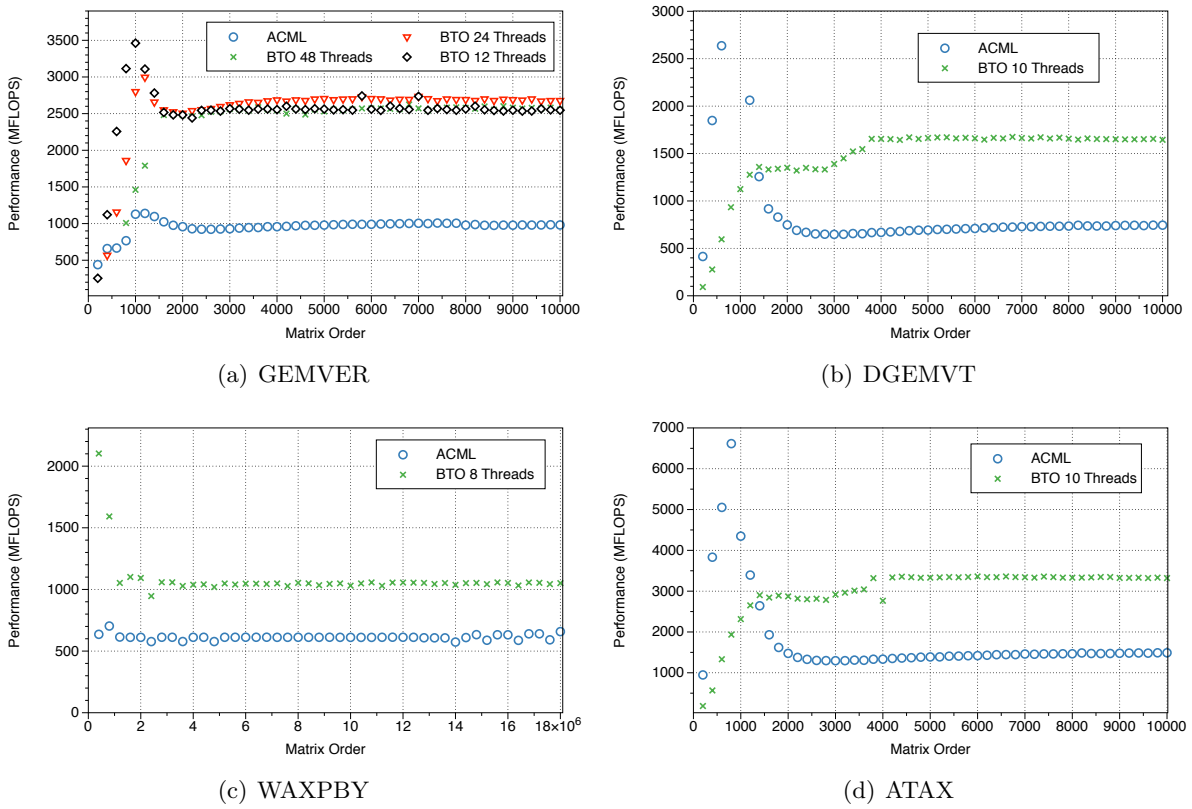
8.2.3 Discussion of Matrix Size

The set of optimization used for the highest performing implementation may vary for a given routine based on the size of the input matrices or vectors. Generating a routine with various versions for different ranges of input sizes is important for real world use cases. Consider common operations such as matrix factorization where the same routines are used over and over again with sub-matrices of decreasing sizes. In cases like this size specialization can have a significant impact on performance.

Figure 8.5 shows the performance for a range of input matrix sizes for GEMVER, DGEMVT, WAXPBY, and ATAX. These results demonstrate a common trend, beyond a certain matrix size, there is little variation in performance. Examination of BTO output of versions for the sizes with little performance change shows that the same set of optimizations is always selected once this steady state is reached. This point of no change generally correlates to some level of the memory system. For example, once a matrix falls out of cache, it becomes common to always find the version that implements full loop fusion. Similarly Figure 8.5(a) shows various thread counts for GEMVER. Once in the steady state, the same thread count tends to perform the best for all sizes.

However, for smaller sized inputs the ideal implementation tends to vary. Consider Figure 8.5(b) and 8.5(d). In these graphs, we see that for small sizes ACML has a significantly better performance than BTO. In these instances loop fusion and parallelism become less important and optimizations such as vectorization and loop unrolling become increasingly important.

BTO currently does not do specialization for matrix sizes, however it is possible for a user to run BTO for several matrix sizes and piece together a size specialized routine. Automating this process is left to future work. An ideal solution will likely interface with a tool such as Orio [39] to exploit optimizations important for small sized matrices such as loop unrolling and vectorization.



(a) GEMVER

(b) DGEMVT

(c) WAXPBY

(d) ATAX

Figure 8.5: Performance comparison of BTO and Parallel ACML on a 48-core AMD Opteron.

8.2.4 Discussion of Portability

BTO has been compared to vendor BLAS and similar tools over a range of years. The computers have included PowerPC, AMD and Intel and have included two through 64 core computers. The results are typical to those presented in this section; an advantage exists when loop fusion can be exploited. This optimization is quite portable. Similarly, if more cores are available and there is enough computation, it is worthwhile using them, regardless of the computer.

The place that BTO generated routines break down is with vectorization. Exploiting single instruction multiple data operations on modern processors can provide significant performance advantages. With AVX this can be a theoretical performance boost of four times faster for double and eight times faster for float types [44]. All results shown in the section have utilized good vectorizing compilers. However if a good vectorization compiler is not available on a target platform,

the results may suffer as compared to a vendor tuned library which is likely leveraging the vector units.

8.2.5 Summary of General Matrix Performance Evaluation

Compared with the best alternative approach for generating a given kernel, BTO performance ranges from 16% slower to 39% faster. Excluding hand-written versions, BTO performs between 14% worse and 229% better. Pluto, ICC, PGCC, and BLAS all achieve near-best performance for only a few points; however, BTO's performance is most consistent across kernels and computers. Excluding the hand-optimized results, BTO finds the best version for 7 of 9 kernels on the Intel Westmere, all 9 kernels on the AMD Phenom, and 7 of 9 kernels on the AMD Interlagos. Surprisingly, on the AMD Phenom, BTO surpassed the hand-optimized code for 7 of the 9 kernels and tied for one kernel.

8.3 Parallel Performance Evaluation

Parallelism is an increasingly important optimization. For BTO, parallelism comes at the cost of a significantly increased search space. However the benefits are worth while. Section 8.2.2 showed that BTO finds optimized routines that are competitive with the best tools and hand tuned implementations. This section examines the effects of the parallelism in that performance.

Figures 8.6, 8.7, and 8.8 compare the best performing serial implementation to the best performing parallel implementation. The results are shown as speed up achieved by parallelism relative to the serial case, so numbers larger than one show parallel outperforming the serial implementation. On the Intel Westmere and AMD Interlagos parallelism can improve performance by nearly seven fold. On average, for the Westmere the improvement is four fold, on the Phenom the average improvement is 2 fold, and on the Interlagos the average improvement is 2 fold. The performance improvements from parallelism alone can be significant.

Unfortunately, the performance improvements are nowhere near linear speedup. Table 8.4 shows the number of threads selected for the best performing parallel implementation for each

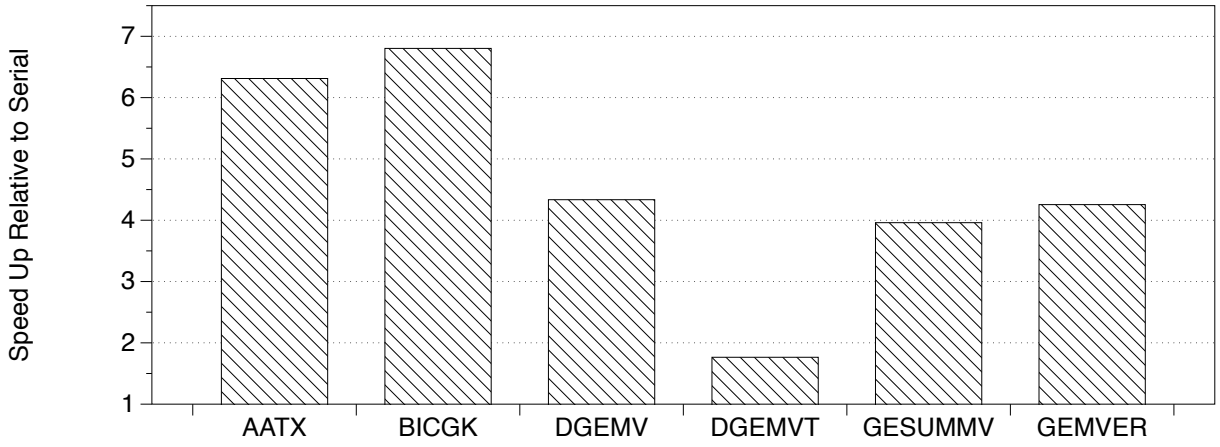


Figure 8.6: Speedup of parallel implementation over serial implementation for the Intel Westmere. Thread counts for selected routines shown in Table 8.4.

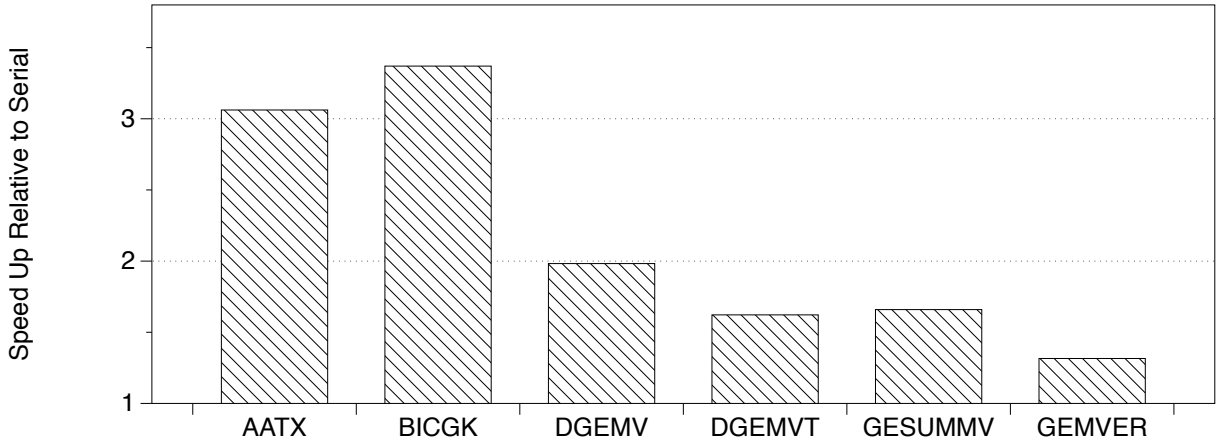


Figure 8.7: Speedup of parallel implementation over serial implementation for the AMD Phenom. Thread counts for selected routines shown in Table 8.4.

of the Westmere, Phenom, and Interlagos architectures. Consider the GESUMMV performance improvement of four times on the Westmere shown in Figure 8.6 as compared to the 24 threads used to achieve that performance shown in Table 8.4. There is clearly a discrepancy in the increase in resources as compared to the performance improvement.

Currently BTO only searches for best performance. There may be cases where, for example halving the number of threads has minor performance impacts. This tradeoff between resources consumed versus the performance improvement is going to be problem specific. It may be necessary

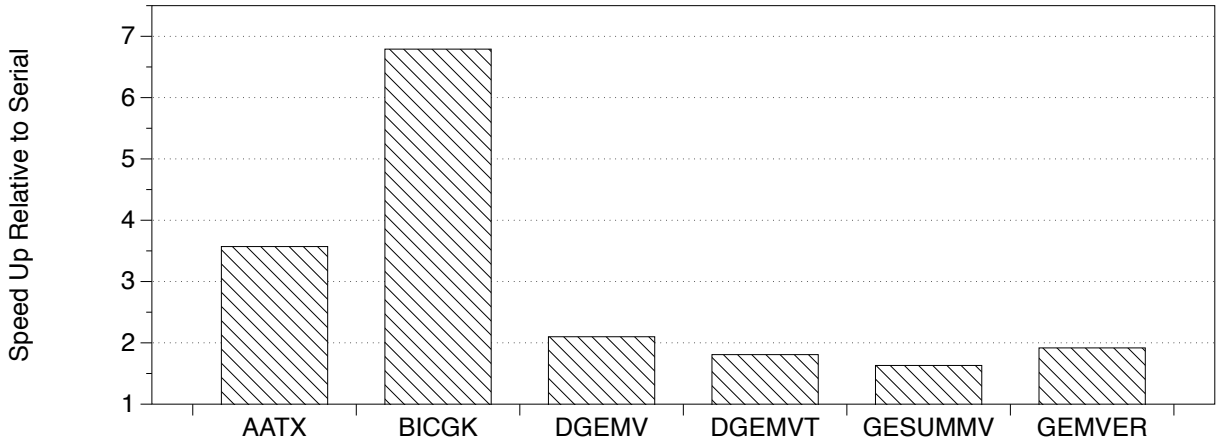


Figure 8.8: Speedup of parallel implementation over serial implementation for the AMD Interlagos. Thread counts for selected routines shown in Table 8.4.

Kernel	**Westmere (24)**	**Phenom (6)**	**Interlagos (64)**
AXPYDOT	10	2	24
VADD	16	3	12
WAXPBY	8	6	12
ATAX	12	6	12
BICGK	20	3	10
DGEMV	10	3	12
DGEMVT	12	3	12
GEMVER	12	3	12
GESUMMV	24	3	20

Table 8.4: Numbers of threads selected by BTO for the performance data shown in Sections 8.2.2 and 8.3. Number of logical cores shown in heading in parenthesis.

in future work to modify the thread search strategy to handle these scenarios. However, the results show that BTO can successfully introduce parallelism to achieve high performance.

8.4 Triangular and Triangular Packed Performance Evaluation

Throughout this section, the matrix kernels listed in Table 8.1 are used to present performance results for the triangular and triangular packed storage formats. The naming is unchanged to keep comparison straight forward, however when discussing the BLAS results, the correct BLAS routines are used. For example, a DGEMV call is replaced by either a DTRMV (triangular) or DTPMV

(triangular packed) call.

Figure 8.9 shows the speedup of the BTO version for triangular and triangular packed as compared to a BLAS implementation using Intel MKL on the Intel Westmere. Values greater than one show BTO outperforming BLAS. In all routines evaluated, BTO was able to outperform the BLAS library with performance improving by up to 66% for triangular formats and up to 62% for triangular packed formats. BTO is able to outperform BLAS in the case of the DGEMV equivalent routines, DTRMV and DTPMV. This is caused by the triangular equivalent routines mapping to multiple BLAS calls to handle the scaling present in DGEMV, but which are not supported with the triangular equivalent.

The observation that DGEMV maps to multiple BLAS calls in the triangular case is worth further discussion. There is certain amount of sequencing of linear algebra operations present in BLAS, for example DGEMV performs $y = \alpha Ax + \beta y$. This is primarily present for the general formats. The triangular equivalents do not offer the same sequence of operations, for example both DTRMV (triangular) and DTPMV (triangular packed) perform only $x = Ax$. This gives BTO an additional advantage as compared to the BLAS routines when working with formats other than general.

This comparison shows that BTO is competitive with BLAS libraries, however the benefit of using a triangular or triangular packed matrix format is an increase in performance caused by a reduction in operations and memory accesses. This reduction should be approximately half as compared to the general format. Provided an application is memory bound, this provides a performance improvement bound of two times faster. Figure 8.10 shows the speedup for triangular and triangular packed formats over the same operation with general matrix formats on an Intel Westmere. In nearly all cases, the performance improvement over the general format is two times faster as expected. This trend holds across architectures. It is not clear why the triangular format of ATAX does not follow the two fold trend.

It is worth pointing out that there were a few minor differences in the fusion and partitioning that was selected for the various formats within a given operation. For the DGEMV and ATAX

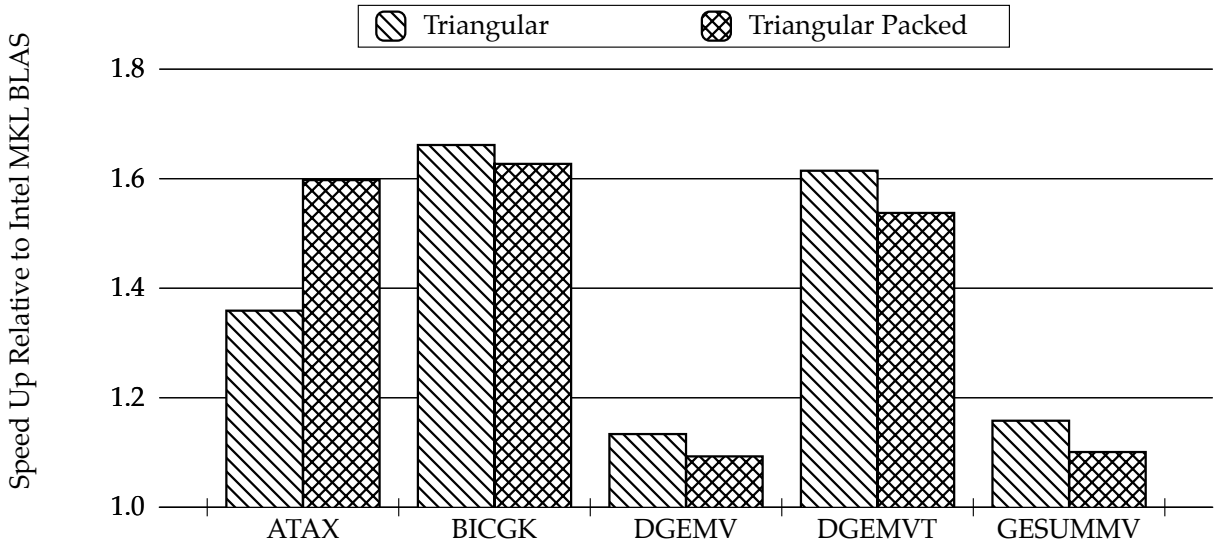


Figure 8.9: Performance comparison of triangular and triangular packed storage formats between the BTO versions and the BLAS versions. Values above show BTO performing better than BLAS. Data from Intel Westmere using Intel MKL BLAS. Speedups between 16% and 66% faster triangular and 9% to 62% for the triangular packed format.

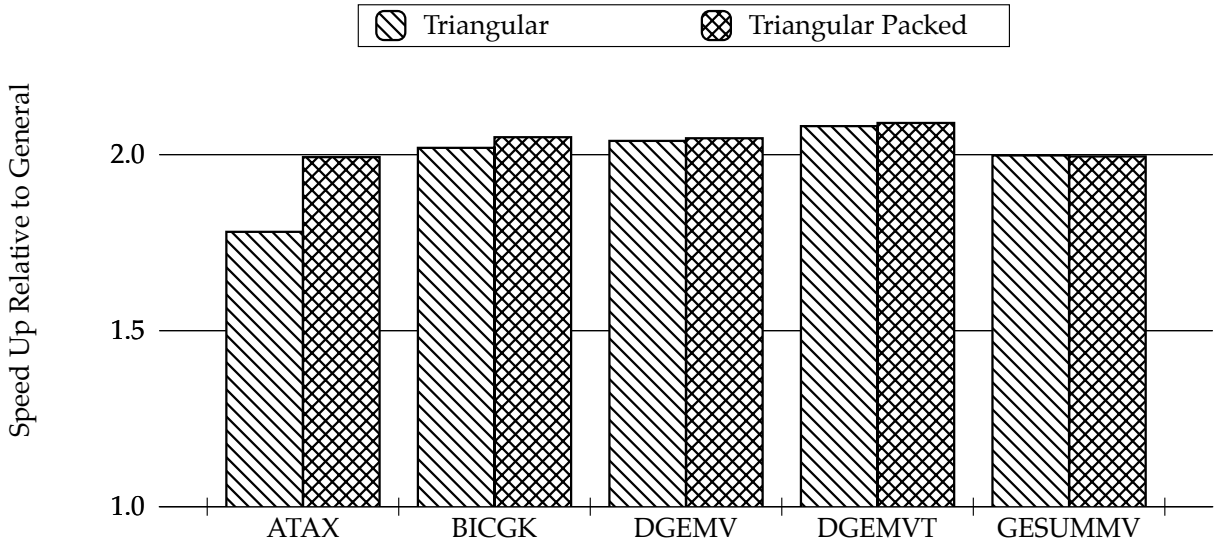


Figure 8.10: Performance comparison between general, triangular, and triangular packed storage formats for Intel Westmere. Speedups relative the general format, so numbers greater than one show the triangular formats performing better. Speedups between 170% and 200% faster for the triangular formats.

routines, BTO selected the same partitioning and loop fusion for all three storage formats. For

BICGK, the optimizations decisions were nearly identical, the only difference being the general format chose not to fuse the inner most loops while the triangular and triangular packed formats did. The DGEMVT kernel performed loop fusion and partitioned the the fused loop the same way for all storage formats. There is one loop, a scaling of a vector, that cannot be fused due to data dependencies. The triangular packed chose not the make this scaling parallel, while general and triangular did. The GESUMMV kernel had the most variation. The general format partitioned across the rows of the matrices and fully fused all the way to the inner loop. The triangular packed version partitioned across the columns of the matrices and only fused the parallel loop, leaving the thread body completely unfused. The triangular format partitioned the same as the packed format, however did not fuse any loops.

These differences in selected optimizations between storage formats are all minor and within expectations. Performance for the GESUMMV kernel varies little with loop fusion because the fusion is reducing order N memory accesses and leaving the matrix accesses the same at N^2. Similarly the small performance variances between a serial and parallel vector scaling in DGEMVT is expected because this operations is order N as compared to the rest of the operation at N^2. The difference between fusing and not fusing the inner loops of BICGK is most difficult to explain. Further examination shows that the performance difference in the general format between fusing and not fusing these loops is small and variable. This is likely caused by the reused row of the matrix remaining in cache, making accesses fast enough that loop fusion has little effect. It is therefore not unexpected to see differences in this depth of fusion.

The results presented in this section show that BTO can successfully optimize triangular and triangular packed matrix formats. Performance results are shown to beat BLAS routines by up to 60% and improve performance over general formats by up to 200%. Additionally, no modification were required to the existing search strategies.

8.5 BTO's Effect on Routine Development Time

Assuming an auto-tuning tool can produce performance at the desired level, the next biggest question is how long does it take to get those results. To place the auto-tuning time of BTO in context, the time taken for hand written implementations is compared to the time take for BTO to generate its best implementation. The search strategy used by BTO for this section is a genetic search beginning with a seed of maximally fused loops as discussed in Chapter 3. This search strategy has been shown to reliably generate high performance routines with a fast search time [10]. Search methods are not the topic of this thesis, so only this reliable search strategy is presented in these results. There are a wide range of use case scenarios for a tool such as BTO, but the author is most familiar with library creation. In this scenario, the goal is to tune a given routine for a specific platform given some available set of optimizations. This requires three steps; 1) writing the routine, 2) verifying correctness, and 3) evaluating the performance. This section looks primarily at the time taken to write the routine, with a brief discussion of verification and performance evaluation.

8.5.1 Time to Write a Routine

Several routines were hand written by the author. The specification for each routine is listed in Table 8.1 and a performance comparison with BTO is shown in Section 8.2.2. The optimizations used included only those that BTO considers, specifically loop fusion, array contraction, and shared memory data parallelism. Time was recorded for three steps of this process; 1) the time to get a correct serial version, 2) the time the make that serial version parallel, and 3) the time taken to perform a search for the correct thread size for a single matrix input size. Table 8.5 shows the time in minutes for each of the routines. The time shown includes time spent verifying the routine correct, but not the time taken to write the correctness test.

In most cases, writing a fused serial version was less than five minutes, the most complex, GEMVER, took 19 minutes. It generally took more than 20 minutes to make the serial version

Kernel	Serial	Parallel	Thread Search	Total
ATAX	10	33	17	40
AXPYDOT	4	20	16	40
BICGK	5	18	16	39
DGEVM	3	21	18	42
DGEMVT	6	32	16	54
GESUMMV	4	25	16	45
GEMVER	19	57	20	96
VADD	3	15	18	36
WAXPBY	3	16	17	36

Table 8.5: The time in minutes for the author to write a version comparable to BTO for each of the listed kernels. Includes verifying correct, but not time to write correctness tests.

parallel. Often debugging consumed a fair amount of this time. The thread search was performed manually (this could be scripted but thats what BTO is for). The process for thread search was to change the thread count in the C file, re-compile, run and log the result. Of 24 available threads, only 12 different thread counts were considered. The total time ranged from 36 to 96 minutes and on average less than five implementations were considered, in most cases only the fully fused serial, and fully fused parallel were written. BTO in a similar amount of time can evaluate thousands of implementations.

Table 8.6 shows three columns, the total time for hand writing from Table 8.5, that same time plus 60 minutes for generation of a correctness and empirical test, and the third column shows the time taken for BTO to perform a search. The 60 minutes added for test generation is based off the time taken to write one set of tests from scratch. This number would vary greatly depending on application, and often existing test harnesses can be modified in a matter of minutes. Excluding time for test generation, speedups range from 8 to 40 times faster for BTO.

8.5.2 Writing a Verification and Performance Evaluation Harness

In the previous section, the time taken to write a correctness and performance evaluation test harness was largely neglected. This is going to be a somewhat variable task depending on what a programmer has available to them. In many cases a programmer may have libraries for comparison

Kernel	Hand Written	Hand Written + Test Generation	BTO Search
ATAX	40	100	1
AXPYDOT	40	100	1
BICGK	39	99	1
DGEVM	42	102	5
DGEMVT	54	114	4
GESUMMV	45	105	20
GEMVER	96	156	10
VADD	36	96	1
WAXPBY	36	96	1

Table 8.6: Comparison of total time (in minutes) to write by hand and that same time plus time to write a correctness and empirical test as compared to BTO search time.

of floating point number and timing routines that will help to speed this process.

The time consuming portion of this task will likely be the floating point error analysis. For a routine such as GEMVER, determining the error bounds for each output vector and matrix can be difficult. For the individual BLAS calls these error bounds are more straightforward to determine, but in the case of GEMVER one must account for the propagation of output with error from one BLAS call to the input of another BLAS call.

Due to the variability and potentially complexity nature of the task, it is realistic to expect this portion of writing a high performance routine to take a significant portion of development time. BTO can generate these correctness and performance test in seconds and this has been shown to work for a range of input routines and input sizes. Given very rigorous requirements additional correctness tests may be required beyond the one generated by BTI, but in most use cases, BTO has a significant speed advantage with its ability to generate these tests automatically.

8.5.3 Engineering Time vs. Computer Time

BTO has an additional time saving advantage. An engineer can only perform one task at a time. If it takes that engineer 90 minutes to write GEMVER, that is all they will get done during that time. BTO on the other hand can be started in a few minutes and left to run on a computer

un-attended while that engineer is free to do other work. In a real world setting, computer time is significantly less expensive than engineering time.

8.6 Summary

The results presented in this chapter show that BTO portably generates high performance routines that are competitive with existing tools, BLAS libraries and hand written code. It demonstrates that BTO can achieve these results while supporting various matrix storage formats.

Additionally, results are presented that demonstrate that BTO can achieve this high performance code in a reasonable amount of time. Optimization times were generally significantly faster than writing a version by hand, and fast enough that a programmer should have no trouble replacing a BLAS library based approach.

Chapter 9

Future Work

With any project of this scope the list of future work always seems longer than the list of accomplishments. There are two obvious topics that will be discussed briefly here.

The first is improving the verification of the optimization coverage. As discussed in Chapter 7, there is currently no automated approach to ensuring that all optimization combinations that are legal are actually being generated by the compiler. A hand comparison has been performed, however this is limiting, because the number of optimizations is in the hundreds of thousands for certain routines, and a hand comparison is not practical and quite error prone.

Currently BTO performs loop fusion based on analysis of the data dependencies in the data flow graph and subgraph describing the loops. The understanding of the type's and type system is nearly to the point where a constraint based approach could be created that describes the legality of fusion of operations based on the types involved in the operation. This in connection with a data dependence analysis on an un-lowered graph would be much less error prone. Currently, dependence analysis occurs on a lowered graph, which is complicated by the number of nodes describing the get and store operations between any two arithmetic operations. It is believed that a constraint based approach could be written in a standalone tool, allowing automatic comparison to the existing BTO implementation. Given the number of versions generated, an automatic comparison is required for a sound verification of loop fusion coverage. Appendix B presents a hypothesis of the rules governing loop fusion based on the data flow graph and type system.

The second topic for future exploration is sparse storage formats. Many scientific applications

work with matrices that are too large to represent with dense matrices. Many of the existing linear algebra and general purpose optimization tools break down when sparse storage formats are used because of the indirect addressing of matrix elements and irregular iteration spaces. The framework put in place to support triangular packed matrices provides a good foundation for supporting sparse formats. The triangular packed matrix format has an irregular memory access pattern when an operation is partitioned. To support irregular formats, BTO includes get and store operation nodes that are specific to difficult memory accesses. To support basic sparse operations, it appears that only a description of indirect memory accesses specific to a sparse format needs to be implemented. Partitioning should extend without modification, however loop fusion requires further examination.

Chapter 10

Conclusions

This thesis has presented a matrix representation and corresponding type system that enables compilation and optimization of a high level syntax to optimized C. The approach works with general, triangular, and triangular packed matrix storage formats. The generated C code is automatically generated with no input from the user. Performance experiments have shown the results to be competitive with existing tools, vendor tuned BLAS libraries and hand written code on a range of modern computers. Search times have been shown to be reasonable, typically on the order of minutes.

The matrix representation using generic recursive containers enables BTO to introduce loops in a manner that provides optimal memory access patterns. These containers allow optimization and loop creation to occur with out complete knowledge of the underlying storage format. This concept is not new, however a novel type system that describes legal linear algebra in terms of these containers has been presented which enables BTO to work with any sequence of basic linear algebra. The type system ensures that operation sequences are legal, describes how to introduce loops, and describes how linear algebra operations may be partitioned.

The ability to partition linear algebra operations is important to performing well on a modern computer. The type system presented here describes the legality of any data partitioning of a basic linear algebra operation. It does so in a manner that is general enough to support an assortment of matrix storage formats and is general enough to support various hardware and software features. For example, this allows BTO to utilize partitioning for data parallelism, cache tiling and in theory

vectorization. For a limited time, a GPU backend existed that demonstrated this partitioning worked for mapping a problem to GPU's.

The ability to generate loops for an assortment of storage formats is unique. Most existing tools only work with general matrix storage formats, some work with triangular formats, but none have been found that work with triangular packed formats. Additionally, with formats outside of general, the BLAS library routines limit the sequences of operations supported, making a tool capable of automatic loop fusion more important for formats such as triangular and triangular packed. More importantly, no tool can work with sparse storage formats. BTO does not yet support sparse matrices, however the framework put in place for triangular and triangular packed sets a strong foundation for extending BTO to support sparse. By implementing support for irregular memory access patterns, BTO lays the foundation for extending to sparse formats.

All of the ideas presented here have been evaluated with thorough testing, the common approach to verifying a compiler. Many of the ideas presented here were implemented in a manner that allows BTO to self check the output for correctness and completeness of optimization.

Bibliography

[1] A. Allam, J. Ramanujam, G. Baumgartner, and P. Sadayappan. Memory minimization for tensor contractions using integer linear programming. 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), April 2006.

[2] F.E. Allen and J. Cocke. A catalogue of optimising transformations. In Randall Rustin, editor, Design and Optimization of Compilers, Prentice-Hall series in automatic computation, pages 1–30. Englewood Cliffs, N.J. : Prentice-Hall, 1972.

[3] John Randal Allen. Dependence analysis for subscripted variables and its application to program transformations. PhD thesis, Houston, TX, USA, 1983. AAI8314916.

[4] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: parallel linear algebra package design overview. In Supercomputing '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM), pages 1–16, New York, NY, USA, 1997. ACM.

[5] AMD. Amd core math library.

[6] Anthony T. Chronopoulos Anthony M. Castaldo, R. Clint Whaley. Reducing floating point error in dot product using the superblock family of algorithms. In SIAM Journal on Scientific Computing, volume 31, pages 1156–1174. SIAM, December 2008.

[7] Clint Whaley Antoine, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. Parallel Computing, 27:2001, 2000.

[8] Apple. Apple accelerate framework. http://developer.apple.com/library/mac/documentation/Accelerate.

[9] P. Balaprakash, S. Wild, and P. Hovland. Can search algorithms save large-scale automatic performance tuning? conditionally accepted for Sixth international Workshop on Automatic Performance Tuning (iWAPT 2011), July 2011.

[10] Geoffrey Belter, Elizabeth R. Jessup, Thomas Nelson, Boyana Norris, and Jeremy G. Siek. Reliable generation of high-performance matrix algebra. CoRR, abs/1205.1098, 2012.

[11] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Maria J. Garzaran, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 48–57, New York, NY, USA, 2006. ACM Press.

[12] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 48–57, New York, NY, USA, 2006. ACM.

[13] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In Proceedings of the 11th international conference on Supercomputing, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[14] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software, 28(2):135–151, June 2002.

[15] Netlib blas. http://www.netlib.org/blas/index.html.

[16] Blas technical forum standard. http://www.netlib.org/blas/blast-forum/, August 2001.

[17] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical use of graph rewriting, 1995.

[18] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not., 43(6):101–113, 2008.

[19] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. Commun. ACM, 16:575–577, September 1973.

[20] Timothy A. Budd. An APL compiler for a vector processor. ACM Trans. Program. Lang. Syst., 6(3):297–313, 1984.

[21] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 123–132, New York, NY, USA, 2008. ACM.

[22] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, Department of Computer Science, University of Southern California, June 2008.

[23] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In CGO '05: Proceedings of the International Symposium on Code Generation and Optimization, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[25] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In Selected Papers from the International Seminar on Generic Programming, pages 25–39, London, UK, 2000. Springer-Verlag.

[26] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng., 5(1):46–55, January 1998.

[27] L DeRose, K Gallivan, E Gallopoulos, B A Marsolf, and D A Padua. Falcon: A matlab interactive restructuring compiler. In In Proc. 8th International Workshop on Languages and Compilers for Parallel Computing, pages 269–288, 1995.

[28] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006.

[29] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on, pages 167–180, 1973.

[30] Arkady Epshteyn, Maria Garzaran, Gerald DeJong, David Padua, Gang Ren, Xiaoming Li, Kamen Yotov, and Keshav Pingali. Analytical models and empirical search: A hybrid approach to code optimization. In 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 2005.

[31] Jeanne Ferrante, Vivek Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, pages 328–343, London, UK, 1992. Springer-Verlag.

[32] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 315–326, New York, NY, USA, 2005. ACM.

[33] G Gao, R Olsen, V Sarkar, and R Thekkath. Collective loop fusion for array contraction. In Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing, pages 281–295. Springer-Verlag, 1992.

[34] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems, 21:703–746, 1999.

[35] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw., 34:12:1–12:25, May 2008.

[36] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In ICS '07: Proceedings of the 21st annual international conference on Supercomputing, pages 116–125, New York, NY, USA, 2007. ACM.

[37] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, Maria J. Garzaran, and David Padua. Programming with tiles. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 111–122, New York, NY, USA, 2008. ACM.

[38] Samuel Z. Guyer and Calvin Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. Proceedings of the IEEE, 93(2):342–357, February 2005.

[39] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio.

[40] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Gary W. Howell, James W. Demmel, Charles T. Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software, 34(3):14:1–14:33, 2008.

[42] IBM. Ibm's engineering and scientific subroutine library. www.ibm.com/systems/software/essl.

[43] Ieee floating point standard. IEEE, January 2011.

[44] Intel. Intel advanced vector extensions programming reference. Technical report, Intel, 2011.

[45] Intel. Intel Composer. http://software.intel.com/en-us/articles/intel-compilers, April 2012.

[46] Intel mkl. Intel Website, 2011.

[47] F. Irigoin and R. Triolet. Supernode partitioning. In POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 319–329, New York, NY, USA, 1988. ACM.

[48] Elizabeth R. Jessup, Ian Karlin, Erik Silkensen, Geoffrey Belter, and Jeremy Siek. Understanding memory effects in the automated generation of optimized matrix algebra kernels. Procedia Computer Science, 1(1):1873 – 1881, 2010. ICCS 2010.

[49] I. Karlin. Memory analysis and tuning of composed linear algebra kernels. In Colorado Celebration of Women in Computing, pages 1–5, Boulder, CO, April 2008.

[50] Ian Karlin. Runtime Prediction of Fused Linear Algebra in a Compiler Framework. PhD thesis, University of Colorado at Boulder, 2011.

[51] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pages 7–1–7–22, New York, NY, USA, 2007. ACM Press.

[52] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pages 1–12, New York, NY, USA, 2007. ACM.

[53] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation, 2000.

[54] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[55] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 207–218, New York, NY, USA, 1981. ACM Press.

[56] Chris Lattner. Testing clang. http://clang.llvm.org/hacking.html, October 2012.

[57] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 201–214, New York, NY, USA, 1997. ACM Press.

[58] Amy W. Lim, Shih wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. pages 103–112. ACM Press, 2001.

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

[60] B A Marker, F G Van Zee, K Goto, G Quintana-Ort?, and R A van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In In Euro-Par ?07: Proceedings of the Thirteenth International European Conference on Parallel and Distributed Computing, 2007.

[61] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In ICS '99: Proceedings of the 13th International Conference on Supercomputing, pages 434–443, New York, NY, USA, 1999. ACM Press.

[62] Frank Mueller. Pthreads library interface, 1994.

[63] Frank Mueller. Pthreads library interface. Technical report, Florida State University, 1999.

[64] Openmp 3.0 standard. http://www.openmp.org/mp-documents/spec30.pdf, May 2008.

[65] Geoff Pike and Paul N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In SC'02, pages 1–12, 2002.

[66] Portland Group. Portland group compiler. http://www.pgroup.com, April 2012.

[67] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. 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, 93(2):232–275, Feb. 2005.

[68] Dan Quinlan, Markus Schordan, Richard Vuduc, and Qing Yi. Annotating user-defined abstractions for optimization. In Workshop on Performance Optimization for High-Level Languages and Libraries, 2006.

[69] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM), page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[70] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, Bret A. Marsolf, and David A. Padua. FALCON: A MATLAB interactive restructuring compiler. In LCPC '95: Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing, pages 269–288, London, UK, 1996. Springer-Verlag.

[71] K. Seymour, Haihang You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In Cluster Computing, 2008 IEEE International Conference on, pages 421 –429, 29 2008-oct. 1 2008.

[72] J G Siek and A Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In In ECOOP ?98: Workshop on Object-Oriented Technology, pages 466–467. SpringerVerlag, 1998.

[73] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference, volume 1. The MIT Press, 2nd edition, 1998.

[74] GCC Team. Gcc testing efforts. http://gcc.gnu.org/testing/, October 2012.

[75] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[76] Todd L. Veldhuizen. Arrays in Blitz++. In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Lecture Notes in Computer Science, pages 223–230. Springer-Verlag, 1998.

[77] Philip Wadler. Deforestation: transforming programs to eliminate trees. In Proceedings of the 2nd European Symposium on Programming, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

[78] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 75–84, New York, NY, USA, 2009. ACM.

[79] J Warren. A hierarchical basis for reordering transformations. In In: POPL '84, ACM, 1984.

[80] David B. Whalley. Tuning high performance kernels through empirical compilation. In ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing, pages 89–98, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. pages 30–44, 1991.

[82] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pages 30–44, New York, NY, USA, 1991. ACM Press.

[83] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–8. IEEE, 2007.

[84] Qing Yi and Apan Qasem. Exploring the optimization space of dense linear algebra kernels. In Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers, pages 343–355, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing, pages 141–150, New York, NY, USA, 2005. ACM.

[86] Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, December 2005.

Appendix A

Extended Evaluation Details

```
void BICGK(int A_nrows, int A_ncols, double* A, int p_nrows,
           double* p, int r_nrows, double* r, int q_nrows,
           double* q, int s_nrows, double* s) {

    int i, j;

    // q = A * p
    //cblas_dgemv(CblasColMajor,CblasNoTrans,A_nrows,A_ncols,1.0,
    //              A,A_nrows,p,1,0.0,q,1);
    for (j = 0; j < A_nrows; ++j) {
        q[j] = 0.0;
    }
    for (j = 0; j < A_ncols; ++j) {
        double *Aj = A + j*A_nrows;
        double pj = p[j];
        for (i = 0; i < A_nrows; ++i) {
            q[i] += Aj[i] * pj;
        }
    }

    // s = A' * r
    //cblas_dgemv(CblasColMajor,CblasTrans,A_nrows,A_ncols,1.0,
    //              A,A_nrows,r,1,0.0,s,1);
    for (i = 0; i < A_ncols; ++i) {
        double *Ai = A + i*A_nrows;
        double accum = 0.0;
        for (j = 0; j < A_nrows; ++j) {
            accum += Ai[j] * r[j];
        }
        s[i] = accum;
    }
}
```

Listing A.1: Example sequence of BLAS calls that implement BICGK translated to C loops for input to ICC and PGCC.

```
void BICG(int A_nrows, int A_ncols, double A[A_nrows][A_ncols], int p_nrows,
          double* p, int r_nrows, double* r, int q_nrows,
          double* q, int s_nrows, double* s) {

    int i, j;

    // q = A * p
    //cblas_dgemv(CblasColMajor,CblasNoTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,p,1,0.0,q,1);

    // s = A' * r
    //cblas_dgemv(CblasColMajor,CblasTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,r,1,0.0,s,1);

    /* pluto start (A_ncols, A_nrows) */
#pragma scop
    for (j = 0; j < A_nrows; ++j) {
        q[j] = 0.0;
    }
    for (j = 0; j < A_ncols; ++j) {
        for (i = 0; i < A_nrows; ++i) {
            q[i] += A[j][i] * p[j];
        }
    }

    for (j = 0; j < A_ncols; ++j) {
        s[j] = 0.0;
    }
    for (j = 0; j < A_ncols; ++j) {
        for (i = 0; i < A_nrows; ++i) {
            s[j] += A[j][i] * r[i];
        }
    }
#pragma endscop
    /* pluto end */
}
```

Listing A.2: Example sequence of BLAS calls that implement BICGK translated to C loops for Pluto input.

```
void BICG(int A_nrows, int A_ncols, double A[A_nrows][A_ncols], int p_nrows,
          double* p, int r_nrows, double* r, int q_nrows,
          double* q, int s_nrows, double* s) {

    int i, j;

    // q = A * p
    //cblas_dgemv(CblasColMajor,CblasNoTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,p,1,0.0,q,1);

    // s = A' * r
    //cblas_dgemv(CblasColMajor,CblasTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,r,1,0.0,s,1);

    /* pluto start (A_ncols, A_nrows) */
  int t1, t2, t3;
 register int lb, ub, lb1, ub1, lb2, ub2;
/* Generated from PLUTO-produced CLooG file by CLooG 0.16.3-UNKNOWN gmp bits in 0.01s. */
 lb1=0;
 ub1=A_ncols-1;
#pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3)
 for (t2=lb1; t2<=ub1; t2++) {
  s[t2]=0.0;;
}
if (A_nrows >= 1) {
 lb1=0;
 ub1=A_ncols-1;
#pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3)
 for (t2=lb1; t2<=ub1; t2++) {
    for (t3=0;t3<=A_nrows-1;t3++) {
      s[t2]+=A[t2][t3]*r[t3];;
    }
  }
}
 lb1=0;
 ub1=A_nrows-1;
#pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3)
 for (t2=lb1; t2<=ub1; t2++) {
  q[t2]=0.0;;
}
if (A_ncols >= 1) {
 lb1=0;
 ub1=A_nrows-1;
#pragma omp parallel for shared(t1,lb1,ub1) private(t2,t3)
 for (t2=lb1; t2<=ub1; t2++) {
    for (t3=0;t3<=A_ncols-1;t3++) {
      q[t2]+=A[t3][t2]*p[t3];;
    }
  }
}
/* End of CLooG code */
    /* pluto end */
}
```

Listing A.3: Example sequence of BLAS calls that implement BICGK after run through Pluto.

```
typedef struct {
    double *t24;
    double *t14;
    double *t18;
    double *r;
    int A_nrows;
    double *t13;
    int __m2;
} BICG_0_msg_t;

void *BICG_body_0(void *mesg) {
    int ii,j,k;
    BICG_0_msg_t *msg = (BICG_0_msg_t*)mesg;
    double *t24 = msg->t24;
    double *t14 = msg->t14;
    double *t18 = msg->t18;
    double *r = msg->r;
    int A_nrows = msg->A_nrows;
    double *t13 = msg->t13;
    int __m2 = msg->__m2;

    for (j = 0; j < __m2; j+=1) {
        // 2_2
        double *t21 = t13 + j*A_nrows;
        t18[j] = 0.0;
        for (k = 0; k < A_nrows; k+=1) {
            // 2_3
            t18[j] += (t21[k]*r[k]);
            t24[k] += (t21[k]*t14[j]);
        }
    }
    return NULL;
}
```

Listing A.4: Message structure and thread function of best BICGK routine generated by BTO.

```
void BICG
(int A_nrows, int A_ncols, double* A, int p_nrows, double* p, int r_nrows,
 double* r, int q_nrows, double* q, int s_nrows, double* s)
{
    int disp, ii, __ns, __np,i;
    int nparts__s2 = 20;
    int __s2;
    if (nparts__s2 > 1 && s_nrows > nparts__s2) {
        __s2 = s_nrows/nparts__s2;
        if (s_nrows%nparts__s2)
            ++__s2;
    } else {
        __s2 = s_nrows;
        nparts__s2 = 1;
    }

    double *t16 = malloc(sizeof(double)*nparts__s2*A_nrows);
    double *t36 = q;
    for (ii = 0; ii < nparts__s2*A_nrows; ++ii) t16[ii] = 0.0;
    BICG_0_msg_t *BICG_0 = malloc(sizeof(BICG_0_msg_t)*nparts__s2);
    pthread_t *threads_0 = malloc(sizeof(pthread_t)*nparts__s2);
    disp = 0;
    for (i = 0; i < s_nrows; i+=__s2) {
        // 2_1
        int __m2 = i + __s2 > s_nrows ? s_nrows - i : __s2;
        double *t18 = s + i;
        double *t24 = t16 + disp*A_nrows;
        double *t13 = A + i*A_nrows;
        double *t14 = p + i;
        BICG_0[disp].t24 = t24;      BICG_0[disp].t14 = t14;
        BICG_0[disp].t18 = t18;      BICG_0[disp].r = r;
        BICG_0[disp].A_nrows = A_nrows;
        BICG_0[disp].t13 = t13;      BICG_0[disp].__m2 = __m2;

        pthread_create(&threads_0[disp], NULL, BICG_body_0, (void *)(BICG_0+disp));
        ++disp;
    }
    for (disp = 0; disp < nparts__s2; ++disp) {
        pthread_join(threads_0[disp],NULL);
    }

    free(threads_0);
    free(BICG_0);
    for (__ns = 0; __ns < A_nrows; ++__ns)
        t36[__ns] = t16[__ns];
    for (__np = 1; __np < nparts__s2; ++__np) {
        for (__ns = 0; __ns < A_nrows; ++__ns) {
            t36[__ns] += t16[__np*A_nrows+__ns];
        }
    }
    free(t16);
}
```

Listing A.5: Entry point and parallel driver of best BICGK routine generated by BTO.

```
typedef struct {
    int A_ncols;
    int A_nrows;
    int A_ld;
    double *A;
    double *p;
    double *q;
    double *r;
    double *s;
} bicg_t;

void *BICG_body(void *msg) {

    bicg_t *lmsg = (bicg_t*)msg;
    int A_nrows = lmsg->A_nrows;
    int A_ncols = lmsg->A_ncols;
    int A_ld = lmsg->A_ld;
    double *A = lmsg->A;
    double *p = lmsg->p;
    double *q = lmsg->q;
    double *r = lmsg->r;
    double *s = lmsg->s;

    int i, j;

    // q = A * p
    //cblas_dgemv(CblasColMajor,CblasNoTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,p,1,0.0,q,1);
    // s = A' * r
    //cblas_dgemv(CblasColMajor,CblasTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,r,1,0.0,s,1);

    for (j = 0; j < A_nrows; ++j) {
        q[j] = 0.0;
    }

    for (j = 0; j < A_ncols; ++j) {
        double *Aj = A + j*A_ld;
        double pj = p[j];
        double accum = 0.0;
        for (i = 0; i < A_nrows; ++i) {
            q[i] += Aj[i] * pj;
            accum += Aj[i] * r[i];
        }
        s[j] = accum;
    }
    return NULL;
}
```

Listing A.6: Message structure and thread function of hand written BICGK routine.

```
void BICG(int A_nrows, int A_ncols, double* A, int p_nrows,
          double* p, int r_nrows, double* r, int q_nrows,
          double* q, int s_nrows, double* s) {
    // q = A * p
    //cblas_dgemv(CblasColMajor,CblasNoTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,p,1,0.0,q,1);
    // s = A' * r
    //cblas_dgemv(CblasColMajor,CblasTrans,A_nrows,A_ncols,1.0,
    //            A,A_nrows,r,1,0.0,s,1);
    int i, j;
    int numThreads = 10;
    int blk = A_nrows / numThreads;
    if (blk < 64) {
        blk = 64;
        numThreads = A_nrows/blk;
        if (numThreads*blk != A_nrows)
            ++numThreads;
    } else {
        if (blk*numThreads != A_nrows)
            ++blk;
    }

    double *stmp = malloc(sizeof(double)*A_ncols*(numThreads-1));
    bicg_t *msg = malloc(sizeof(bicg_t)*numThreads);
    pthread_t *threads = malloc(sizeof(pthread_t)*numThreads);

    for (i = 0; i < numThreads-1; ++i) {
        msg[i].A_nrows = blk;    msg[i].A_ncols = A_ncols;
        msg[i].A_ld = A_nrows;   msg[i].A = A + i*blk;
        msg[i].p = p;            msg[i].q = q + i*blk;
        msg[i].r = r + i*blk;    msg[i].s = stmp + i*A_ncols;

        pthread_create(threads+i, NULL, BICG_body, (void*)(msg+i));
    }

    int mBlk = i*blk+blk > A_nrows ? A_nrows - i*blk : blk;
    msg[i].A_nrows = mBlk; msg[i].A_ncols = A_ncols;
    msg[i].A_ld = A_nrows;  msg[i].A = A + i*blk;
    msg[i].p = p;           msg[i].q = q + i*blk;
    msg[i].r = r + i*blk;   msg[i].s = s;

    BICG_body((void*)(msg+i));

    for (i = 0; i < numThreads-1; ++i) {
        pthread_join(*(threads+i), NULL);
        for (j = 0; j < A_ncols; ++j) {
            s[j] += stmp[i*A_ncols+j];
        }
    }
    free(stmp);
    free(msg);
    free(threads);
}
```

Listing A.7: Entry point and parallel driver of hand written BICGK routine.

Table A.1: Performance data for Intel Westmere. BLAS numbers from Intel's MKL. Speedups relative to unfused loops compiled with ICC (ICC performance is 1 and not shown). Best performing version in bold.

Kernel	**BLAS**	**Pluto**	**HAND**	**BTO**
AXPYDOT	4.14	1.41	4.00	**4.18**
VADD	1.86	3.68	3.62	**3.63**
WAXPBY	1.36	3.81	**3.81**	3.71
ATAX	2.89	0.95	**6.26**	5.91
BICGK	2.89	2.91	**5.95**	5.43
DGEMV	3.00	3.22	3.22	**3.23**
DGEMVT	3.33	0.66	**6.22**	5.99
GEMVER	3.08	3.32	3.40	**3.41**
GESUMMV	1.65	5.79	**7.10**	6.93

Table A.2: Performance data for AMD Phenom. BLAS numbers from AMD's ACML. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). Best performing version in bold.

Kernel	**BLAS**	**Pluto**	**HAND**	**BTO**
AXPYDOT	0.97	1.81	1.58	**1.86**
VADD	0.84	1.33	1.50	**1.83**
WAXPBY	0.79	1.40	1.68	**1.91**
ATAX	1.27	0.69	**2.92**	**2.92**
BICGK	1.27	0.80	2.80	**2.84**
DGEMV	1.67	0.71	1.85	**1.89**
DGEMVT	1.67	0.71	1.85	**1.89**
GEMVER	1.04	1.61	**2.61**	2.34
GESUMMV	1.63	0.63	1.74	**1.75**

Table A.3: Performance data for AMD Interlagos. BLAS numbers from AMD's ACML. Speedups relative to unfused loops compiled with PGCC (PGCC performance is 1 and not shown). Best performing version in bold.

Kernel	**BLAS**	**Pluto**	**HAND**	**BTO**
AXPYDOT	0.82	1.60	**1.73**	1.61
VADD	0.43	1.05	1.14	**1.15**
WAXPBY	0.34	1.06	**1.16**	1.11
ATAX	2.49	0.43	4.09	**4.28**
BICGK	2.35	1.60	3.03	**4.22**
DGEMV	**2.45**	0.89	1.66	2.07
DGEMVT	2.43	0.43	**4.08**	4.03
GEMVER	1.70	2.00	**4.15**	4.05
GESUMMV	**2.36**	0.37	1.65	2.03

Appendix B

Toward Formalizing Loop Fusion

B.1 General Fusion Rules

Two requirements for legal fusion, 1) types involved in the creation of loops for the two operations in question generate fusible loops, and 2) all data dependencies are respected.

One new categorization of the linear algebra type rules is introduced to describe the rules presented here. A linear algebra multiplication rule will fall into one of three categories; 1) a *dot* product operation, 2) an *l-scale* operation, and 3) an *r-scale* operation. A *dot* operation is a rule that performs a reduction. The two scaling rules generalize the form of non-dot multiplication. A multiplication is either scaling the right operand by the complete type of the left operand (*r-scale*) or scaling the left operand by the complete type of the right operand (*l-scale*). Table B.1 shows the linear algebra rules with categories introduced. Beyond this categorization, no changes to the linear algebra rules and type system are required.

B.1.1 Rules Describing Legal Fusion Based on Types

Two operations may be fused when they each have loops that share a common iteration, generally that means they have the same initialization, condition, and update values. In BTO, this can be determined by the types involved in the two operations. Given the following two operations,

$$\tau_1 = \tau_2 \; op_a \; \tau_3$$

$$\tau_4 = \tau_5 \; op_b \; \tau_6,$$

Category	Algo	Op and Operands	Result Type	Pipe
-	`add`	$O\texttt{<}\tau_l\texttt{>} + O\texttt{<}\tau_r\texttt{>}$	$O\texttt{<}\tau_l + \tau_r\texttt{>}$	yes
-	`s-add`	$S + S$	S	no
-	`trans`	$O\texttt{<}\tau\texttt{>}^T$	$O^T\texttt{<}\tau^T\texttt{>}$	yes
-	`s-mult`	$S \times S$	S	no
dot	`dot`	$R\texttt{<}\tau_l\texttt{>} \times C\texttt{<}\tau_r\texttt{>}$	$\sum(\tau_l \times \tau_r)$	no
r-scale	`rr-mult`	$R\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$R\texttt{<}R\texttt{<}\tau_l\texttt{>} \times \tau_r\texttt{>}$	yes
	`outer2`	$C\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$R\texttt{<}C\texttt{<}\tau_l\texttt{>} \times \tau_r\texttt{>}$	yes
	`r-scale`	$S \times O\texttt{<}\tau\texttt{>}$	$O\texttt{<}S \times \tau\texttt{>}$	yes
l-scale	`cc-mult`	$C\texttt{<}\tau_l\texttt{>} \times C\texttt{<}\tau_r\texttt{>}$	$C\texttt{<}\tau_l \times C\texttt{<}\tau_r\texttt{>>}$	yes
	`outer1`	$C\texttt{<}\tau_l\texttt{>} \times R\texttt{<}\tau_r\texttt{>}$	$C\texttt{<}\tau_l \times R\texttt{<}\tau_r\texttt{>>}$	yes
	`l-scale`	$O\texttt{<}\tau\texttt{>} \times S$	$O\texttt{<}\tau \times S\texttt{>}$	yes

Table B.1: The linear algebra knowledge base categorized for fusion rules.

the first step is to determine which types are involved in the creation of the loop. These types are determined from the linear algebra lowering rules. Each loop of an operation is derived from one or more of types involved in the operation. For multiplication this is either the left operand and the result type (*r-scale*), the right operand and result type, (*l-scale*), or the left and right operand types (*dot*). The function *nodesForLoop* in Listing B.1 simplifies this process by returning the set of nodes numbers of the types that are used to create the loop.

Any node in a set returned by *nodesForLoop* will have an identical outer type, so only a single node is needed, the rest of the set will be explained later. Therefore, fusion is legal to occur for the two operations listed above when the two sets from *nodesForLoop*,

$$setOp1 = \text{nodesForLoop}(op_a, \tau_2, \tau_3, \tau_1)$$

$$setOp2 = \text{nodesForLoop}(op_b, \tau_5, \tau_6, \tau_4),$$

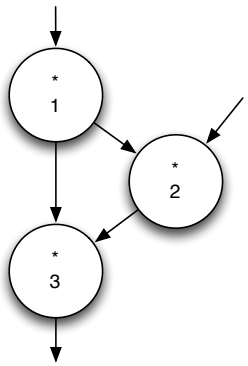
are both non-empty sets and the loop generated by the type of any node in *setOp1* is fusible with the loop generated by the type of any node in *setOp2*. Again, fusible loops require compatible initializations, conditions, and updates.

```
isLScale(TL,TR,TU)
        if (TL == scalar && TR != scalar && TU != scalar)
                return true
        if (TL == row && TR == row && TU == row)
                return true
        if (TL == column && TR == row && TU == row)
                return true
        return false

isRScale(TL,TR,TU)
        if (TL != scalar && TR == scalar && TU != scalar)
                return true
        if (TL == colin && TR == column && TU == column)
                return true
        if (TL == column && TR == row && TU == column)
                return true
        return false

isDot(TL,TR,TU)
        if (TL == row && TR == column)
                return true
        return false

nodesForLoop(op,TL,TR,TU)
        // TL is type of left operand which is node number L
        // TR is type of right operand which is node number R
        // TU is type of result operand which is node number U

        if (op == multiply)
                if (isLScale(TL,TR,TU))
                        return {R,U}
                else if (isRScale(TL,TR,TU))
                        return {L,U}
                else if (isDot(TL,TR,TU))
                        return {L,R}
        else if (op == add || op == subtract)
                return {L,R,U}

        return {}
```

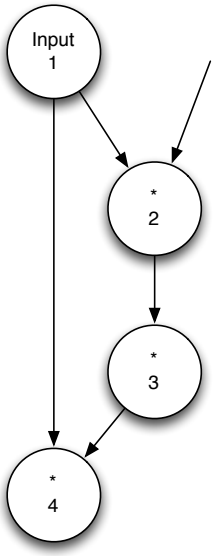
Listing B.1: Function to determine which type of an operation generates the outer loop.

B.1.2 Data dependencies

This describes the legality of fusion based only on the types of operations involved in an operation. However, data dependencies must be observed. Data dependencies can be expressed in



(a) Dataflow graph showing dependence restricting pipeline fusion of nodes 1 and 3.



(b) Dataflow graph showing dependence restricting shared data fusion of nodes 2 and 4.

terms of the data flow graph. If any set of edges connect operation a and operation b and contains a third operation c the two operations a and b may not be fused because either a or b depend on c. Figures B.1(a) and B.1(b) show examples of these dependencies.

Additionally if operation a precedes operation b, i.e. a directed edged from a to b exists, dependencies exists, but in certain cases fusion can break the dependence. If the loop generated by operation a is created by the *dot* rule, (*nodesForLoop* returns $\{L, R\}$), then there exists a reduction and fusion may not occur. In other cases, fusion becomes legal only when the requirement described in the pipeline section are met.

The two requirements above (fusible loops, and respect of data dependences) describe legal fusion. BTO is more specific in what is fused for profitability reasons. When shared data exists between the two loops, it means fusion will reduce memory traffic. BTO limits itself to this type of loop fusion, performance analysis has shown that loop fusion in the absence of limiting memory traffic has little to no effect on performance. BTO performs two types of shared access fusion, 1) shared data access, and 2) pipelines.

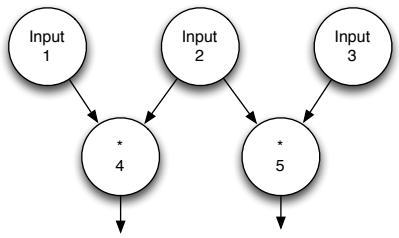


Figure B.1: Dataflow graph showing fusion potential.

B.1.3 Shared Data Access

Shared data access looks like the sequence of operations in Figure B.1. Given the following types:

$$\tau_4 = \tau_1 \; op_a \; \tau_2$$

$$\tau_5 = \tau_2 \; op_b \; \tau_3,$$

and the set of operation a's operands which is $\{1, 2\}$ and the set of operation b's operands which is $\{2, 3\}$, this fusion is deemed profitable when the intersection of these two sets is non-empty. Here this intersection returns $\{2\}$ meaning that this fusion is profitable.

Note that if *nodesForLoop* returns $\{1, 4\}$ for operation a and $\{3, 5\}$ for operation b, and the profitability check returns $\{2\}$, this fusion may still be profitable. What this case describes is the case when the complete un-lowered type of node 2 is passed through the outer fused loop, which is still a reduction in the overall access of the data described by node 2.

B.1.4 Pipeline

Shared data access by pipelining looks like the sequence of operation in Figure B.2. As described in the dependence analysis section, this fusion will only be legal in certain situations. To begin, the set of operands for the second operation (b) is intersected with the set returned by *nodesForLoop* from the second operation (b). Secondly, this set is intersected with the set of the result operand from the first operand (a). When this set is non-emtpy, fusion is legal and will be

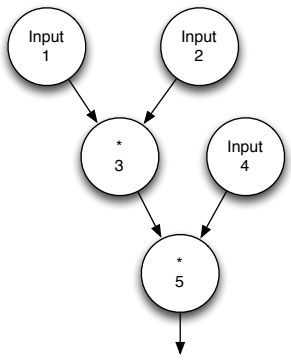


Figure B.2: Dataflow graph showing fusion potential.

profitable.

As an example, the following types are given from Figure B.2:

$$\tau_3 = \tau_1 \; op_a \; \tau_2$$

$$\tau_5 = \tau_3 \; op_b \; \tau_4.$$

In this example, if *nodesForLoop* returns $\{3, 4\}$ (*dot*) or $\{3, 5\}$ (*r-scale*) for operation b, the possible intersections will be:

$$\{3, 4\} \cap \{3, 4\} \cap \{3\}$$

$$\{3, 4\} \cap \{3, 5\} \cap \{3\},$$

which both result in the set $\{3\}$ meaning this pipeline fusion will be profitable and legal. On the other hand if *nodesForLoop* returns $\{4, 5\}$ (*r-scale*) for operation b, the intersection becomes:

$$\{3, 4\} \cap \{4, 5\} \cap \{3\},$$

resulting in the empty set. In this case fusion will not be legal. This fails because the type resulting from operation a can never match the shared operand type of operand b.

Finally recall that the data dependence analysis will fail if operation 3 is lowered with a *dot* rule.