Spring 1-1-2015

DSLs and Search for Linear Algebra Performance Optimization

Thomas Harrison Nelson
University of Colorado at Boulder, thomasharrisonnelson@gmail.com

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds
Part of the Numerical Analysis and Scientific Computing Commons, and the Programming Languages and Compilers Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/109

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
DSLs and Search for Linear Algebra Performance Optimization

by

Thomas Nelson

B.S. University of Texas, 2007

M.S. University of Colorado, 2010

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 Computer Science

2015
This thesis entitled:
DSLs and Search for Linear Algebra Performance Optimization
written by Thomas Nelson
has been approved for the Department of Computer Science

Prof. Elizabeth Jessup

Prof. Jeremy Siek

Prof. Xiao-Chuan Cai

Prof. Jed Brown

Prof. Pavol Cerny

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.
Linear algebra and tensor algebra computations lie on the critical path of many scientific applications. These numerical problems have a mathematical structure that allows for complex transformations of the code. This thesis presents search strategies for two Domain-Specific Languages (DSLs) focusing on high performance numerical computing domains. The first DSL is Build-to-Order BLAS (BTO), a language for matrix computation. Scientific programmers often turn to vendor-tuned Basic Linear Algebra Subprograms (BLAS) to obtain portable high performance. However, many numerical algorithms require several BLAS calls in sequence, and those successive calls do not achieve optimal performance. The entire sequence needs to be optimized in concert. Instead of vendor-tuned BLAS, a programmer could start with source code in Fortran or C and use a state-of-the-art optimizing compiler. However, experiments in this thesis show that optimizing compilers often attain only one-quarter of the performance of hand-optimized code. I present a scalable search algorithm for BTO that reliably achieves high performance by choosing the best combination of loop fusion, array contraction, and multithreading for data parallelism.

The second DSL is the Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI). Tensor computations are characterized by arrays with numerous dimensions, inherent parallelism, and moderate data reuse. The best-performing implementation is heavily dependent on the tensor dimensionality and the target architecture. This optimization problem is especially challenging when the computation requires many iterations with tensors of small dimensions. I created a high-level search representation and input language as part of a toolchain to solve these problems. In this thesis, OCTOPI maps such tensor computations to GPUs, starting with a high-level tensor input language and producing efficient CUDA code as output. This approach combines tensor-specific mathematical transformations with a GPU decision algorithm and autotuning of a large parameter space. Generated code shows significant performance gains over sequential and OpenMP parallel code, and a comparison with OpenACC shows the importance of autotuning and other optimizations for achieving efficient results.
To Katie. Thank you for sharing this adventure with me.
Acknowledgements

First I want to thank Katie, for carrying me through graduate school with her love and support. She kept me on task, helped every way she knew how, and kept me smiling. I also want to thank my parents for encouraging me throughout school, my brother Joe for listening to me ramble on about programming and the meaning of life, and my whole family for believing that I could do this.

I want to thank my advisor, Liz Jessup. Her patience and guidance brought me to finally finish this huge undertaking. Thank you also to Jeremy Siek, Boyana Norris, Paul Hovland, Prasanna Balaprakesh, and Mary Hall, for teaching me how to do research and giving valuable feedback and insight throughout the PhD process. Thank you to Xiao-Chuan Cai, Jed Brown, and Pavol Cerny, for being on my thesis committee and reading this document.

I also want to thank Geoff Belter, Ian Karlin, and Axel Rivera for sharing the hard work of writing code, debugging, running experiments, and making graphs. Both of these projects were collaborations and they worked very hard with me to make them a success. Thanks to all the other grad students who I shared lab space and reading groups with. They gave me a community and taught me how to be a grad student without going (too) crazy.
2.4.3 Search and Autotuning ........................................... 23
2.5 Conclusion .......................................................... 24

3 BTO .......................................................... 26

3.1 Overview of the BTO Compiler ................................. 26
3.1.1 Lowering to Loops over Scalar Operations .............. 28
3.1.2 Applying Legal Transformations .......................... 29

3.2 Representing and Searching the Transformation Space .......... 32
3.2.1 A Straightforward but Inefficient Representation of the Search Space .......... 33
3.2.2 An Efficient Representation of the Search Space ........... 35
3.2.3 Incremental Type-Pruning .................................. 37

3.3 Genetic/Greedy Search Strategy .................................. 39
3.3.1 The Max-Fuse Greedy Search ............................... 40
3.3.2 Mutation ......................................................... 41
3.3.3 Selection and Crossover .................................. 41
3.3.4 Search for Number of Threads .............................. 43

3.4 Performance Results ............................................. 43
3.4.1 Test Environment and Kernels ............................. 44
3.4.2 Comparison with Similar Tools ............................. 45
3.4.3 MFGA Compared with Exhaustive Searches .............. 48
3.4.4 Evaluation of Search Methods ............................. 49

4 OCTOPI ................................................... 54

4.1 Overview .......................................................... 54
4.1.1 Code Generation ............................................. 56

4.2 OCTOPI Input and Tensor Transformation ...................... 57
4.2.1 Transformation Example .................................. 58
4.2.2 Addition and Multiplication ................................ 61
4.3 TCR and Generating the Search Space ........................................ 61
4.4 Search Space Exploration ....................................................... 63
4.5 Performance Measurements .................................................... 65
  4.5.1 Individual Tensor-Contraction Computations .......................... 66
  4.5.2 GPU Code Generation Strategies in Context .......................... 66
  4.5.3 Comparison with Manual OpenMP Code ............................... 68
  4.5.4 Performance Scaling with Problem Size ............................... 69

5 Conclusions and Future Work .................................................. 71
  5.1 Conclusions and Future Work ................................................ 71

Bibliography ................................................................. 73
Chapter 1

Introduction

My research is about search for empirical autotuning. I worked on two different Domain-Specific Language (DSL) projects, BTO and OCTOPI. Each language targeted different input problems and different output problems.

Traditionally, scientific programmers use linear algebra libraries such as the Basic Linear Algebra Subprograms (BLAS) [25, 26, 42] and the Linear Algebra PACKage (LAPACK) [3] to perform their linear algebra calculations. A programmer links an application to vendor-tuned or autotuned implementations of these libraries to achieve efficiency and portability. For programs that rely on kernels with high computational intensity, such as matrix-matrix multiply, this approach can achieve near-optimal performance [29, 71]. However, memory bandwidth, not computational capacity, limits the performance of many scientific applications [3], with data movement dominating performance for the foreseeable future [2].

A tuned BLAS library can perform loop fusion to optimize memory traffic only in the limited scope of a single BLAS function, which performs a small number of mathematical operations. Moreover, separately compiled functions limit the scope of parallelization on modern parallel architectures. Each BLAS call spawns threads and must synchronize before returning, but much of this synchronization is unnecessary when considering the entire sequence of matrix algebra operations. The BLAS Technical Forum identified several new routines that combine sequences of BLAS, thereby enabling a larger scope for optimization [14, 35]. However, the number of useful BLAS combinations is too large to implement and tune for each new architecture. Increasing the number of BLAS adds a correspondingly larger burden on library maintainers, optimizers and users.
1.1 BTO

Instead of using vendor-optimized BLAS, a scientific programmer can start with source code in Fortran or C, perhaps based on the Netlib BLAS [50], and then use a state-of-the-art optimizing compiler to tune the code for the architecture of interest. However, our experiments with two industrial compilers (Intel and Portland Group) and one research compiler (Pluto [15]) show that, in many cases, these compilers achieve only one-quarter of the performance of hand-optimized code (see Section 3.4.2). This result is surprising because the benchmark programs we tested are sequences of nested loops with affine array accesses and the optimizations that we applied by hand (loop fusion, array contraction, and multithreading for data parallelism) are well-established. Nevertheless, for some benchmarks, the compilers fail to recognize that an optimization is legal. For other benchmarks, they miscalculate the profitability of choosing one combination of optimizations over another combination.

These observations demonstrate that achieving reliable, automatic generation of high-performance matrix algebra is nontrivial. In particular, the three main challenges are (1) recognizing whether an optimization is legal, (2) accurately assessing the profitability of optimizations and their parameters, and (3) efficiently searching a large, discontinuous space of optimization choices and parameters. In this dissertation, I present my recent improvements to the Build to Order BLAS (BTO) compiler. To my knowledge, it is the first compiler that solves all three challenges in the domain of dense matrix algebra.

BTO accepts as input a sequence of matrix and vector operations in a subset of MATLAB, together with a specification of the storage formats for the inputs and outputs, and produces an optimized kernel in C. This input language helps solve the problem of determining whether an optimization is legal: it makes all data dependencies explicit, so there is no difficulty recognizing whether an optimization is semantics-preserving. Further, BTO uses a carefully designed internal representation for transformation choices that rules out many illegal transformations while at the same time succinctly representing all the legal choices. To accurately assess profitability, the BTO compiler relies on a hybrid approach that we have presented in our prior work [10]: BTO uses an analytic model for coarse-grained pruning and empirical timing to make the ultimate decisions. Early prototypes of BTO are described in several papers [10,11,38,63]. My dissertation
considers more optimizations than in our prior work and describes a new search algorithm that is scalable with respect to the number of optimizations and their parameters. In particular, I present a special-purpose genetic algorithm whose initial population is the result of a greedy, heuristic search. This search strategy quickly finds a near-optimal combination of code transformations compared to hand-written code, in an otherwise intractable search space.

With respect to storage formats, BTO currently supports row-major and column-major dense matrices, and symmetric, triangular, and banded matrices. Our first prototype of the BTO system fused loops at every opportunity [63]. The next refinement of the system added the ability to explore all possible fusions and used a hybrid search strategy that combined analytic modeling with empirical performance testing [10]. Later improvements to BTO included the ability to produce shared memory parallel code [11] and analytically model shared memory parallel systems [38]. The following are the technical contributions of my work on BTO.

(1) I present an internal representation for optimization choices that is complete (includes all legal combinations of loop fusion, array contraction, and multithreading for data parallelism) but that inherently rules out many illegal combinations, thereby greatly reducing the search space (Section 3.2).

(2) I present a scalable and effective search strategy: a genetic algorithm with an initial population seeded by a greedy search. We describe this strategy in Section 3.3 and show in Section 3.4.2 that it produces code that is between 16% slower and 39% faster than hand-optimized code.

(3) I compare this genetic/greedy search strategy with several other strategies to reveal experimentally why this strategy works well. (Section 3.4.4).

1.1.1 A Typical Use Scenario of the BTO Compiler

BTO is meant to be used by long-running, performance-critical numerical applications with a significant linear algebra component. Therefore, the BTO compiler spends more time than does a typical general-purpose compiler to achieve performance that is on par with hand-tuned code. For typical examples, the BTO compiler takes less than 2 minutes to produce an optimized subprogram, which is significantly faster
than manual tuning. Here we demonstrate the workflow of using BTO by optimizing the bidiagonalization algorithm of [35]. The pseudocode for a portion of this algorithm is given in Fig. 1.1.

\[
\begin{align*}
(s, u, k) &\leftarrow \text{DLARFG}(A_{i:m,j}) \\
\alpha &\leftarrow -u_1 \\
\hat{A} &\leftarrow \hat{A} - \hat{u}z^T - \hat{w}v^T \\
v^T &\leftarrow v^T + \alpha u^T \hat{A} \\
w &\leftarrow \beta \hat{A}v \\
(s, v, k) &\leftarrow \text{DLARFG}(v) \\
w &\leftarrow (sA_{i+1} + w)/k \\
z &\leftarrow \bar{u} - \bar{w}v^Tv
\end{align*}
\]

Figure 1.1: Excerpt of the bidiagonalization algorithm of [35]. The DLARFG subroutine takes a vector and and returns a scalar and vector whose outer product form a Householder reflector.

The first step in using BTO is to identify which parts of an algorithm can be expressed in terms of matrix and vector operations and, ideally, which of them are most time-critical for the overall execution. Those portions of the code are then written as BTO kernel input files. The sequence of three statements starting with the assignment to $\hat{A}$ in Fig. 1.1 can be expressed as the kernel given in Fig. 1.2. This is the GEMVER kernel, as described by Howell et al. [35] and added to the BLAS standard [14]. The BTO compiler takes the specification in Fig. 1.2 and outputs a highly tuned C function. The user then compiles the C function using their native C compiler, such as the Intel C compiler [36].

The second step is to insert a call to the BTO-generated C function in the appropriate place within the enclosing algorithm. It is straightforward to call and link to C functions from most languages that are popular in scientific computing (especially Fortran, C, and C++). If needed, BTO could easily be modified to generate Fortran instead.

One advantage of the BTO process, in contrast to the standard BLAS-based process, is ease of readability and maintainability. By expressing complex mathematical algorithms in terms of high-level linear algebra, the programmer creates highly legible and modifiable code. The programmer does not need
to match the desired linear algebra computations against the list of existing BLAS functions. Moreover, this readability is accomplished without sacrificing performance; in Section 3.4 we show improved performance over BLAS and hand-tuned approaches. The BTO compiler is particularly effective at optimizing sequences of matrix-vector operations (like level 2 BLAS) on large matrices and vectors, where big gains can be obtained by reducing memory traffic through loop fusion.

Figure 1.3 shows the performance results of using a BLAS-based approach compared with using BTO to generate the GEMVER kernel that is used within the bidiagonalization algorithm. These results are on a 12-core Intel Westmere, doing a full bidiagonalization (the DGEBRD LAPACK routine) on matrices of order 16 to 4096. The bidiagonalization algorithm repeatedly invokes GEMVER on smaller and smaller submatrices.

The line labeled MKL in Figure 1.3 is a BLAS-based implementation of GEMVER based on Howell’s code. We link to Intel’s MKL implementation of the BLAS. The line labeled BTO replaces the Howell GEMVER implementation with two BTO-generated kernels. For matrices smaller than 1024x1024 the BTO version calls a kernel autotuned for small matrix sizes. For matrices larger than 1024x1024, the BTO version dispatches to a BTO kernel autotuned for larger matrices. The results show a 13% overall improvement in performance by using the BTO-generated kernel compared with the MKL BLAS-based kernel. The improvement on GEMVER alone is four times the performance of the MKL BLAS for matrices of order 2048 or greater. For smaller matrices, MKL outperforms BTO, most likely because BTO does not introduce vectorization or register tiling. These optimizations could be added in future versions of BTO, but current work has focused on optimizing the larger matrix orders. The drop in performance for BTO between $2^{10}$
and $2^{11}$ appears to be due to cache effects, but more exploration is needed.

The bidiagonalization algorithm is an example of a typical application where BTO can improve performance even over vendor-tuned libraries. In addition, the high-level input language is easy to use and understand.

![Bidiagonalization Performance: BTO vs. naive MKL BLAS](image)

**Figure 1.3:** Performance results for bidiagonalization.

1.2 OCTOPI

Chapter 4 describes OCTOPI, the Optimizing Compiler with Tensor OPeration Intelligence, and a code generation system that uses it. The overall motivation for this project came from applying some of the key ideas in BTO to tensors, which are a multidimensional generalization of matrix-matrix multiplication. Originally I considered extending BTO to work on tensors. I rejected that plan for two reasons: (1) substantially different optimizations needed, and (2) the codebase itself was somewhat old and not maintained.

BTO targeted multithreaded CPUs for optimization. That is an oversimplification of the architecture space. Computer architecture is undergoing a significant period of exploration to find new ways for continued
performance gains while maintaining energy efficiency and reliability. The result is a diverse landscape of architectures that incorporate features such as massive socket-level parallelism, accelerators, and deep memory hierarchies. The developers of high-performance computational applications are thus faced with the challenge of maintaining performance portability in the face of rapidly increasing architectural diversity.

My research is about search for empirical autotuning. I am pursuing an approach to performance portability that uses a domain-specific language (DSL) to specify high-level semantics, a transformation and code generation framework to map the DSL to an architecture, and an autotuning engine to search among many possible code variants. I worked on two different Domain-Specific Language (DSL) projects, BTO and OCTOPI. Each language targeted different input problems and different output problems.

In Chapter 4, I describe the DSL and modular optimization framework for GPU-based computation of tensor contractions. Such computations arise frequently in numerical applications. I focus on specific instances from computational fluid dynamics using the spectral element method, and electronic structure modeling using coupled cluster theory. My contributions center on the domain-specific language OCTOPI and associated domain-specific optimizations; I also describe how this is used in the modular optimization framework, a new decision algorithm for generating an autotuning search space for GPUs, and a search algorithm based on state-of-the-art machine learning and statistics tools.

As compared to other DSLs for tensor contraction [8, 64], this work focuses on tensor computations with small dimensions; in such cases, mapping the problem to use highly-tuned linear algebra libraries will not achieve high performance as these libraries are optimized for large matrices. Further, it targets GPU architectures. My approach was driven by a desire to improve specific tensor problems not solved by current tools, but I also view this work as an example of developing highly-tuned applications specialized for individual architectures starting with a mathematical representation of the problem in a DSL.

Section 4.1 provides an overview of the problem domain and describes the DSL and the modular optimization framework. Section 4.2 describes the first of the two new modules: Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI), a DSL for tensor contractions with tensor-specific optimizations. Section 4.3 describes the second new module: Tensor Contraction Representation (TCR) performs code generation from a tensor-specific representation, and applies a decision algorithm to encode the autotuning
search space. This tool relies on CUDA-CHiLL for code generation and Orio for navigating the search space. Section 4.4 presents the machine learning algorithm to search the space of code transformations. Section 4.5 describes the experimental design and results on two NVIDIA GPUs, the Tesla C2050 and the Tesla K20.

1.3 My Contributions

For the BTO project project I was in charge of the genetic algorithm for search. The initial concept and type system was based on Jeremy Siek’s Matrix Template Library [62]. Geoff Belter extended the BTO implementation and added parallelism and cache tiling support [10]. In previous versions of BTO, only exhaustive search was implemented. My research involved studying possible search strategies, and representations, and ultimately choosing to implement a genetic algorithm. The problem representation work is described in Section 3.2, and the Genetic Algorithm is described in Section 3.3.

On the OCTOPI project, I created the OCTOPI language. This involved choosing an appropriate input language, something that represented the tensors clearly and explicitly. After choosing the input, I had to implement the tensor transformations OCTOPI performs, and create output that worked with the SURF tool [58]. The OCTOPI language is described in Chapter 4.2.
Chapter 2

Background

This thesis builds on work from four research areas: numerical linear algebra, compilers, domain-specific languages, and search or function optimization. The key to a successful autotuning matrix (or tensor) compiler is to combine the work from these separate areas into a new useful tool. Much work has already been done, especially in trying to make the interaction of compilers and linear algebra work more smoothly, and some work on DSLs for numerical problems has already shown good results. This chapter reviews the literature in the field and talks about how that work relates to the work in this thesis.

2.1 Introduction

High-performance computing is both more powerful and more technically complex than at any time in the past. Supercomputing clusters require dedicated power sources and budgeting processing power is increasingly important. This trend creates a pressure to ensure that all code running on such a machine runs as fast as possible.

At the same time, trends in micro-architecture design have added to the challenges a programmer must overcome to achieve top performance. As processor speed has increased, the performance of the memory hierarchy has become increasingly important, with a single cache miss taking 10-100 times as long as a multiplication [55].

Manually tuning numerical programs for high-performance computing is extremely time-consuming and error-prone, so modern research has turned towards auto-tuning. Auto-tuning is the process of automatically choosing one of many possible code transformations for performance. This thesis focuses on empirical
search: searching the space of possible code variants by compiling and running them on test kernels. There is
an interesting body of research on modeling and auto-tuning as well, where the performance of different vari-
ants is estimated rather than found by experiment, but it is outside of the scope of this thesis. Auto-tuning
has many possible applications, but this review focuses on numeric computation, which is both practically
important and home to important research in the field.

In the context of numeric libraries, auto-tuning means optimizing a specific set of functions for a
given architecture. For domain-specific languages (DSLs), the situation is more complex; the compiler has
to auto-tune any computation the language can express. This means the potential exists for more powerful
auto-tuning, but limits are placed on the practicality of the search.

Empirical auto-tuning is time-consuming, and not all applications require the level of performance
it provides. A general-purpose compiler needs to keep compile times low, so such auto-tuning isn’t usually
practical. Auto-tuning provides the most value in cases when a program will be run for a much longer time
than it will be compiled. Numerical computation provides problems with very long runtimes, so it is a good
target for auto-tuning [59].

Section 2.1.1 compares the target domain of auto-tuning tools. Then sections 2.2.1 and 2.2.2 briefly
describe some important serial and parallel optimizations that are frequently used in auto-tuning. Section
2.3 gives examples of currently used search strategies and describes simplex search in detail, one of the more
popular search strategies. Section 2.4 examines the test kernels and experimental evaluations used by the
auto-tuning tools and looks at some of the results they describe.

2.1.1 Target Domains

Some of the first successful auto-tuning solutions were for Basic Linear Algebra Subprograms (BLAS)
[14]. BLAS are a set of linear algebra programs (sometimes called kernels) that can be tuned and portably
used in a wide variety of scientific and numeric contexts. The first BLAS specifications were created in 1979
and have been gradually updated and expanded, most recently in 2001. Tuned BLAS have greatly improved
the ease with which scientists can write high-performance programs. However, because that performance de-
pends greatly on optimizations for specific architectures, attempts have been made to automatically generate
high-performance BLAS, for example ATLAS [71].

Even auto-tuned libraries have issues, however. One problem is scope: a library of statically defined cannot possibly handle every situation. The BLAS standard already contains over 150 subprograms, which threatens to hinder usability. Several projects have tried to expand on that functionality by providing a auto-tuning DSL that can generate a superset of BLAS routines, including CHILL [66], POET [73], and PLUTO [17]. Other projects take a similar approach to other domains, as SPIRAL [55] does for discrete signal processing, or the work of Kamil et al. [37] in stencil kernels.

The key tradeoff in moving to a more general-purpose language is the increased difficulty of code analysis: source-to-source translators such as CHILL or PLUTO, for example, attempt to analyze Fortran, a general-purpose language, which makes preserving correctness more difficult. This in turn limits how aggressively the tool can transform the code, as this thesis discusses in section 3.4. On the plus side, working with Fortran as an input language means less work for a user whose software is already written, since most scientific computation is currently done in C or Fortran.

2.2 Optimizations

Even more than the target domain, an auto-tuning tool is defined by the code transformations it optimizes. There are a wide variety of such optimizations, and to describe all of them and their effectiveness is beyond the scope of this chapter. Instead I focus on optimizations common to several tools and give examples of a few key optimization types.

2.2.1 Serial Optimizations

There are many possible optimizations to serial code, but most numerical auto-tuners tend to focus on a few high-reward code transformations that are commonly missed by general-purpose compilers, particularly loop fusion, tiling, and loop unrolling. Each of the auto-tuning tools discussed in this chapter shows some serial optimization improvements that are not found by general-purpose compilers [17,37,55,60,66,73]; some of these results are compared in Section 3.4.

Two aspects of optimizing for the memory hierarchy make it difficult to do by hand and so create
tempting targets for empirical tuning. First, for best performance, programs should use cache and register tiling, which are non-trivial code transformations with a range of possible parameters. Second, the best tile sizes depend heavily on machine architecture, in ways that can be difficult to predict. The number of levels of cache hierarchy, the size of each cache in the hierarchy, and cache associativity and eviction policies can all affect the best tiling strategy, and these aspects often differ from chip to chip, even among chips with a similar overall design.

Loop unrolling is a combination of copying the body of a loop and increasing the iteration step size. Doing so reduces the number of branches and exposes opportunities for constant propagation and common subexpression elimination, at the risk of increasing code size. Consider the following simple stencil-like computation for $u_i \leftarrow a_{i-1} + xa_i + a_{i+1}$:

```c
for (i=0; i < N; ++i) {
    u[i] = a[i-1] + x*a[i] + a[i+1];
}
```

Unrolling the loop by 1 gives the following:

```c
for (i=0; i < N; i+= 2) {
    u[i] = a[i-1] + x*a[i] + a[i+1];
    u[i+1] = a[i-1+1] + x*a[i+1] + a[i+1+1];
}
```

A general purpose compiler can usually transform the second version to this optimized version:

```c
for (i=0; i < N; i+= 2) {
    float t1 = a[i];
    float t2 = a[i+1];
    u[i] = a[i-1] + x*t1 + t2;
    u[i+1] = t1 + x*t2 + a[i+2];
}
```
In the example above, \(a[i]\) and \(a[i+1]\) are accessed once per loop iteration, as in the pre-transformation example, but the loop is executed only half as many times. Unrolling saves two memory lookups per iteration of the loop, a significant speedup for memory-bound kernels. In combination with these techniques, optimizations such as constant propagation and common subexpression elimination (which is actually being done above in moving from the second to the third variant) are commonly combined with unrolling and tiling to reduce computation.

Although simple to describe and implement, loop unrolling and cache tiling interact nonlinearly, and finding the optimal amount of unrolling and tile sizes is very difficult without empirical search [66]. The question for researchers is how best to search that space; although some progress has been made, no definitive answer has emerged yet. A common search strategy for these parameters is simplex search, which is described in section 2.3.1.

Megiddo and Sarkar [45] study the problem of deciding which loops to fuse in a context where parallelization choices have already been made (such as an OpenMP program). They model this problem as a weighted graph whose nodes are loops and whose edges are labeled with the runtime cost savings resulting from loop fusion. Because the parallelization choices are fixed prior to the fusion choices, their approach sometimes misses the optimal combination of parallelization and fusion decisions.

Darte and Huard [23], on the other hand, study the space of all fusion decisions followed by parallelization decisions. Pouchet et al. [54] take a similar approach, using an orthogonal method that exhaustively searches over fusion decisions, then using the polyhedral model with analytic models to make tiling and parallelization decisions. These approaches roughly correspond to the orthogonal search technique that I compare to BTO in Section 3.4.4.1.

### 2.2.2 Parallel Optimizations

In addition to dealing with the memory hierarchy, most modern chips are multi-core, requiring programs to take advantage of shared-memory parallelism. Efficient parallelism requires the compiler to choose not only how to split up the work but also how many threads should be operating. Different chips can have very different architectures, requiring different code to perform optimally: compare the Opteron X2 and the
Intel Clovertown shown in figure 2.1.

Increasing the thread count can allow greater throughput for a program, but because the cores share levels of the memory hierarchy, sometimes a lower thread count is necessary to prevent bus contention or cache conflicts. When two or more cores attempt to access the same memory at the same time, typically one is forced to wait on the other.

Most auto-tuners that target shared-memory parallelism use either pthreads, a low-level unix threading primitive, or openMP, a slightly higher-level and more portable library that uses pragmas to direct threading.

Multi-core involves multiple threads running independently; by contrast, one can also parallelize computation using Single Instruction Multiple Data (SIMD) hardware. Most commonly SIMD involves either graphics-processing units or vector instructions on the CPU. In either case, some sub-vector of elements in an array are processed simultaneously. GPUs require moving data from the CPU to the GPU, doing the computation, and moving it back; this means that programs with a low computation-I/O ratio will not benefit from GPU usage.

Automatic parallelization of general-purpose code is an active and difficult research topic [59]. Code analysis for numeric auto-tuning is often made easier by restricting the domain to either a specific set of functions or a DSL, which gives information at a higher level about what optimizations are possible. For example, Williams et al. [60] performed auto-tuning of register, cache, and thread tiling for sparse matrix-vector product. They found that GPU code on an STI Cell blade significantly outperformed CPU versions for that kernel. Because they limited the experiment to a single algorithm, they performed a wider search.

Similarly, SPIRAL uses its domain knowledge of signal-processing transforms to include streaming SIMD extensions (SSE) in its output [55]. Spiral has a set of rules for generating vector intrinsics based on the mathematical formulas it is generating code for; using this technique the authors find roughly 1.5x speedup over gcc and icc automatic vectorization [55].

2.2.3 Combining Optimizations

Different optimizations can interact with one another in complex ways, resulting in an increasingly difficult search problem as more optimization types are added. For example, shared memory parallelism
Figure 2.1: Overview comparison of an Opteron X2 and quad-core Intel Clovertown.
<table>
<thead>
<tr>
<th>System</th>
<th>Unrolling</th>
<th>Tiling</th>
<th>Permutation</th>
<th>multicore</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SPIRAL</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pluto</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Chill</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>POET</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2.1: Some optimizations performed by auto-tuning tools. Some tools perform optimizations not in this chart.

changes the working set of an algorithm and requires different cache tile sizes to reflect the different memory usage.

Table 2.1 shows some possible auto-tuning optimizations and which projects use them. Notice that Pluto, Chill, and POET target general-purpose languages, and they also have the least emphasis on parallelism, suggesting that perhaps parallelism is easier to achieve in domain-specific languages. Unrolling and cache tiling are commonly performed optimizations, and they interact.

The approach to partitioning matrix computations described in this thesis is inspired by the notion of a blocked matrix view in the Matrix Template Library [62]. Several researchers have subsequently proposed similar abstractions, such as the hierarchically tiled arrays of [1] and the support for matrix partitioning in FLAME [30].

Bilmes et al. [12] and Whaley et al. [71] autotune matrix multiplication using empirical evaluation to determine the profitability of optimizations. Zhao et al. [74] use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Qing et al. [72] apply empirical search to determine the profitability of optimizations for register reuse, SSE vectorization, strength reduction, loop unrolling, and prefetching. Their framework is parameterized with respect to the search algorithm and includes numerous search strategies.

2.3 Search Strategies

This section provides brief overviews of several search algorithms used in auto-tuning, followed by a more in-depth explanation of simplex search as an example of a popular tuning strategy. Search strategies are necessarily somewhat domain specific: the optimization space being searched over determines what types
of searches are feasible to implement and efficient in practice.

For some search spaces, exhaustive search is possible, and many auto-tuning tools provide the option to search the given space exhaustively. However, for larger spaces, some other form of search is needed. One very simple search strategy is random: randomly testing points in the space for a fixed number of iterations. Random search makes a good baseline strategy and sometimes gives surprising results: in Yi et al. [73] found that random search outperformed their simplex search on the GEMVT kernel.

Pluto [15] employs the heuristic of maximally fusing loops. Loop fusion is generally beneficial, but too much can be detrimental because it can put too much pressure on registers and cache [39]. Bondhugula et al. [16] develop an analytic model for predicting the profitability of fusion and parallelization and show speedups relative to other heuristics such as always fuse and never fuse. However, they do not validate their model against the entire search space as we do where possible here.

Orthogonal search begins by varying a single parameter, holding all others fixed. After finding the best value for that parameter, it searches along the next parameter for the new best value. This process can be repeated until a fixed point is reached. Pseudocode for a single pass to minimize a function $f$ having $N$ parameters with possible values in $(1, M)$ is given below:

```plaintext
for i = 1 to N:
    best = inf
    bestarg = 0
    for j = 1 to M:
        x[i] = j
        test = f(x)
        if test < best:
            best = test
            bestarg = j
        x[i] = bestarg
```

Note that the equivalent real-valued search is over $x + \alpha e_i$ maximizing $\alpha$. 
2.3.1 Simplex Search

The simplex method, also known as Nelder-Mead [48], is a hill-climbing algorithm originally designed for real vector-valued problems. It has good practical performance leading researchers to extend its use to integer domains like loop unrolling and cache tiling.

The basic N-simplex method begins with N arbitrary points in the space. The N points are evaluated, and then, at each iteration, the worst of the N points is reflected, contracted, or expanded depending on the values found and the details of the algorithm. For example, a reflection is defined by:

\[ x_r = x_o + \alpha(x_o - x_n) \]

Here, \( x_n \) is the worst point in the simplex, \( x_o \) is the center of gravity of the simplex, and \( \alpha \) is a constant controlling step size. Figure 2.2 shows visual examples of a 3-simplex in 2-dimensional space, showing reflections, contractions, and expansions. Typically a reflection is tested first (b): if the reflection is better than the other simplex points, an expansion is tried (d). If the reflection is worse than the other points, a contraction is performed (c). Thus the simplex grows when the local minimum is outside of the simplex and shrinks when the local minimum is inside, providing an automatic scaling of search step.

For auto-tuning, simplex is most commonly used for unrolling and tiling parameters, which have integer parameters that allow for this notion of search. However, even in integers, many points suggested by the simplex are not valid points, for example non-integer points or tiling values outside of acceptable range. For this reason, some sort of near-legal neighbor must computed. For example, CHILL [66] uses an approximate nearest neighbor algorithm in conjunction with simplex search for its tiling and loop unrolling search. Details of this function’s implementation affect the speed and consistency of the search.

2.3.2 Search for Autotuning

In practice, search strategies are often tailored for the specific numerical problem, taking advantage of whatever is known about compute resources and problem space to reduce or eliminate search time. This approach can be combined with a machine model that estimates program performance based on a simplified
model of the target hardware. The following examples are successful applications of this kind of domain-
limited search.

Vuduc et al. [70] study the optimization space of applying register tiling, loop unrolling, software
pipelining, and software prefetching to matrix multiplication. They show that this search space is difficult
(a very small number of combinations achieve good performance), and they present a statistical method for
determining when a search has found a point that is close enough to the best.

Balaprakash et al. [4] study the effectiveness of several search algorithms (random search, genetic
algorithms, Nelder-Mead simplex) to find the best combination of optimization decisions from among loop
unrolling, scalar replacement, loop parallelization, vectorization, and register tiling as implemented in the
Orio autotuning framework [33]. They conclude that the modified Nelder-Mead method is effective for their
search problem. The genetic algorithm they employ uses a vector-based approach similar to the inefficient
representation described in Chapter 3, which doesn’t translate well to this search problem.

SPIRAL [55] uses a genetic algorithm as one search strategy for autotuning discrete signal processing.
They translate their search space into trees of rules for breaking discrete transforms into simpler units. They
develop a unique crossover and mutation scheme for these ruletrees based on swapping and manipulating
subtrees. SPIRAL has many similarities to the BTO approach. However, the difference in domain requires
different search and code generation. In signal processing transforms are usually created by calling smaller
blocks that handle subproblems, and the primary search challenge is which of these transform decompositions
to apply at each stage. In BTO, the problem is not decomposing matrix operations, but rather applying
transformations to them.

The Tensor Contraction Engine [8] considers loop fusion and parallel code generation. TCE takes an
analytic approach, using a memory model to reduce traffic and exploit space-time tradeoffs. Unlike BTO,
TCE does not need to consider dependencies across multiple operations, as it focuses on optimizing a single
very large contraction. For similar reasons, the parallelism in TCE is distributed-memory parallelism that
applies a generalization of Cannon’s algorithm to reduce internode communication.

CHiLL [20] develops a framework for empirical search over many loop optimizations such as permu-
tation, tiling, unroll-and-jam, data copying, and fusion. They employ an orthogonal search strategy, first
searching over unrolling factors, then tiling sizes, and so on. [66] describe an autotuning framework that combines ActiveHarmony’s parallel search backend with the CHiLL transformation framework. CHiLL is used to process OCTOPI output in the results from Chapter 4.

Looptool [56] and AutoLoopTune [57] support loop fusion, unroll-and-jam, and array contraction. AutoLoopTune also supports tiling. POET [73] also supports a number of loop transformations.

The DxT project [44] uses a cost model based on operation count and communication costs to estimate the performance of many possible implementations of distributed-memory dense linear algebra, by composing each algorithm mostly out of Level 3 BLAS subroutines. They use a similar style of search heuristics to narrow the space, focusing on transformations likely to be helpful. The fact that even a slightly different domain (distributed-memory level 3 linear algebra vs. shared memory level 2.5 linear algebra) requires a very different set of code transformations and search heuristics highlights the difficulty of the problem and the amount of domain-specific knowledge that needs to be encoded in these solutions. When non-integer parameters are being tuned, other search strategies are necessary. Tools that wish to combine unrolling and tiling with other optimizations such as SIMD-ization or loop fusion need another search strategy, or need to combine simplex with some other method such as orthogonal search. More research needs to be done in this area.

Zhao et al. [74] use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Qing et al. [72] apply empirical search to determine the profitability of optimizations for register reuse, SSE vectorization, strength reduction, loop unrolling, and prefetching. Their framework is parameterized with respect to the search algorithm and includes numerous search strategies.

2.4 Measuring Success: Empirical Comparisons

The most common and practical baseline is to compare a naive implementation compiled by a general-purpose compiler (for example gcc, icc, or ifort). Comparing to this baseline establishes the performance improvements the tool provides over general purpose compilers.

Auto-tuning tools can also compare to existing hand-tuned or auto-tuned libraries if the test kernels exist for those libraries. SPIRAL compares its Discrete Fourier Transform generation to four powerful
<table>
<thead>
<tr>
<th>Tool</th>
<th>test kernels</th>
<th>performance comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUTO</td>
<td>1d stencil, LU factorization, $y = ABx$</td>
<td>previous work (Griebl, Lim &amp; Lam)</td>
</tr>
<tr>
<td>Williams '10</td>
<td>3d stencils: Laplacian, Divergence, Gradient</td>
<td>self-compare, explaining contributions of optimizations</td>
</tr>
<tr>
<td>SPIRAL</td>
<td>DFT, DCT-2, WHT</td>
<td>IPP, MKL, fftw, GNU sci. lib.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(DFT only)</td>
</tr>
<tr>
<td>CHILL</td>
<td>GEMM, TRSM, 3d stencil</td>
<td>ATLAS for BLAS</td>
</tr>
<tr>
<td>SparseMV</td>
<td>GEMV with 14 sparse matrices from various applications</td>
<td>self-compare, breakdown of optimization effects</td>
</tr>
<tr>
<td>POET</td>
<td>GEMM, GEMV</td>
<td>ATLAS</td>
</tr>
</tbody>
</table>

Table 2.2: Overview of performance comparisons for various auto-tuning tools

libraries: IPP, MKL, fftw, and GNU scientific library. It is competitive with (although slightly worse than) these hand-tuned libraries, and the authors also note that SPIRAL generates a wide range of signal transforms for which no highly-tuned libraries are available. Similarly CHILL compares its performance on matrix-matrix product and triangular solve (GEMM and TRSM) to ATLAS, the auto-tuned BLAS library. The work from Williams et al. compares its sparse matrix-vector product to the auto-tuned OSKI library.

Self-comparison can be used to isolate and explain the effects of particular transformations or search strategies. Self-comparison is also helpful when test kernels don’t have easily available tuned versions. For example Chill compares to ATLAS for BLAS kernels, but uses self-comparison for its stencil operation.

POET and PLUTO are somewhat unusual in that they present performance on compile time, in addition to runtime. POET compares its compile time performance to the LoopProcessor tool.

OCTOPI combines compiler optimizations, scientific computation, and search algorithms. Some past research has dealt with tensor computation specifically, and a large body of work has used some form of search to improve code performance, either using a domain-specific language as we do here, or searching over an optimization space for a general purpose language like C or Fortran.

2.4.1 Optimizing tensor computations

Many tools and libraries have been developed for optimizing tensor contraction computations. Among these are the Tensor Contraction Engine (TCE) [8], the Super Instruction Assembly Language (SIAL) [61], and Cyclops [64]. These efforts have focused primarily on contractions involving very large tensors, possibly
distributed across a large parallel computer. Performance optimization focuses on reducing the number of operations performed by exploiting symmetry and redundant subexpressions. Often, tensors are transposed so that a high-performance matrix-matrix multiplication can be used. Our focus is on the small tensors that arise in the spectral element method and can be used as a building block for computations involving large tensors. Thus, different optimization priorities apply.

OCTOPI builds from ideas in the similar TCE project which also implements a DSL for tensors. In particular, its fusion and strength reduction algorithm is the same one described by the TCE papers for the single thread case. TCE takes an analytic approach, using a memory model to reduce traffic and exploit space-time tradeoffs [31]. Due to the large tensor problems it solves, the parallelism in TCE is distributed-memory parallelism that applies a generalization of Cannon’s algorithm to reduce internode communication [43].

Several efforts have sought to optimize the performance of the Nekbone proxy application on GPUs. The CESAR codesign center reports tensor contraction performance of 100–200 GFlops on a Fermi GTX 590 GPU for tensors of size $8 \times 8 \times 8$ to $12 \times 12 \times 12$ using hand-coded OpenCL kernels [65]. The CRESTA project ported Nekbone to a multi-GPU system and reported a speedup of 1.59x using 512 NVIDIA Kepler K20x GPUs versus a CPU-only implementation (512 nodes with 8192 cores) [21]. Although direct comparisons are difficult, our speedup of 1.3x versus OpenMP is encouraging, especially since our results include the time to transfer data back and forth between CPU and device memory.

2.4.2 Domain Specific Languages for Performance Optimization

In addition to the tensor languages described above, other projects have used a domain-specific language to focus on high-level optimization and search opportunities.

SPIRAL [55] is a domain-specific language for discrete signal processing. It uses a genetic algorithm as a search strategy for autotuning. SPIRAL translates the search space into trees of rules for breaking discrete transforms into simpler units. The authors develop a unique crossover and mutation scheme for these ruletrees based on swapping and manipulating subtrees. SPIRAL has similarities to our approach, however the difference in domain requires different search and code generation. Signal processing transforms
are usually created by calling smaller blocks which handle subproblems, and the primary search challenge is which of these transform decompositions to apply at each stage.

The DxT project [44] is a DSL for distributed-memory dense linear algebra that uses a cost model based on operation count and communication costs to estimate the performance of many possible implementations. It composes each high-level algorithm mostly out of Level 3 BLAS subroutines. The authors use a similar style of search heuristics to narrow the space, focusing on transformations likely to be helpful.

### 2.4.3 Search and Autotuning

Vuduc, Demmel and Bilmes [70] study the optimization space of applying register tiling, loop unrolling, software pipelining, and software prefetching to matrix multiplication. They show that this search space is difficult (a very small number of combinations achieve good performance), and they present a statistical method for determining when a search has found a point that is close enough to the best.

Tiwari et al [66] describe an autotuning framework that combines ActiveHarmony’s parallel search backend with the CHiLL transformation framework. Looptool [56] and AutoLoopTune [57] support loop fusion, unroll-and-jam, and array contraction. AutoLoopTune supports tiling. POET [73] supports a number of loop transformations.

Integer search parameters, such as loop unroll factors or tile sizes, can take advantage of the integer space in the search strategy. Optimizations like loop fusion or adding SIMD instructions are not easily represented in a integer search parameter, so most search strategies do not apply. Zhao et al [74] use exhaustive search and empirical testing to select the best combination of loop fusion decisions. Qing and Qasem [72] apply empirical search to determine the profitability of optimizations for register reuse, SSE vectorization, strength reduction, loop unrolling, and prefetching. Their framework is parameterized with respect to the search algorithm and includes numerous search strategies.
2.5 Conclusion

Computer architectures are more complex now than ever before, and optimizing code for them is an increasingly daunting task. The rapid introduction of new architectures worsens the problem, as does the high power cost of modern supercomputing. Auto-tuning offers a possible solution: the potential for producing faster code overall and reducing human effort spent tuning kernels. This chapter compares several modern auto-tuning tools, highlights key issues any such tool must address, and shows similarities and differences for each issue.

There are still unresolved questions in many areas of auto-tuning compilers. First, there are major differences in the approaches of each tool to input specification and target domain. While there is room for useful differentiation and specialization at the input level, the evidence seems to point away from auto-tuning libraries and towards DSLs. Another large area open to research is how best to search the optimization space. While some methods have been used successfully by multiple tools (simplex for example), we lack a clear guideline for how to choose auto-tuning strategies that are scalable and cover large search spaces. Most tools still compromise on both the generality of the target domain and the types of optimizations performed. Future work should move towards addressing these concerns.
Figure 2.2: Explanation of simplex
Chapter 3

BTO

3.1 Overview of the BTO Compiler

This section gives an overview of the BTO compiler, emphasizing the pieces most relevant for understanding the new search algorithm. Throughout this section we use the example kernel BATAK, which performs $y \leftarrow \beta A^T Ax$ for matrix $A$, vectors $x$ and $y$, and scalar $\beta$. The BTO specification for BATAK is given in Fig. 3.1.

```
BATAK
in x : vector(column), beta : scalar, A : matrix(row)
out y : vector(column)
{
  y = beta * A' * (A * x)
}
```

Figure 3.1: The BTO kernel input file for $y \leftarrow \beta A^T Ax$.

The compiler has four components: parsing, lowering, search, and transformation. The parser produces a high-level dataflow representation of the program. The lowering phase then compiles to a sequence of loops and scalar operations, represented as a hierarchical dataflow graph. The search component takes the lowered dataflow graph and tries to determine the best combination of transformations. For each point that the search algorithm empirically evaluates, it requests a set of code transformations, such as applying loop fusion or adding data parallelism to some of the loops. The transformation component performs those changes and then BTO compiles and executes the specified code variants and reports the performance results back to the search algorithm.
Just prior to lowering, the parser creates a dataflow graph. For example, Fig. 3.2 shows the dataflow graph for the BATAK kernel. The square boxes correspond to the input and output matrices and vectors, and the circles correspond to the operations. The operations are labeled with numbers, which we use to identify the operations in the remainder of the paper. The transpose (labeled “T”) is not numbered because it doesn’t require any work in the code output. Instead, BTO interprets the second matrix-vector product as occurring on the transpose of $A$, changing the loops without moving any data.

![Dataflow graph for $y \leftarrow \beta A^T Ax$.](image)

In addition to the dataflow graph, the BTO compiler uses a type system based on a container abstraction to describe the iteration space of the matrices and vectors. Containers may be oriented horizontally or vertically and can be nested. We assume that moving from one element to the next in a container is a constant-time operation and good for spatial locality, but we place no other restrictions on what memory layouts can be viewed as containers. The types are defined by the following grammar, in which $R$ designates row (horizontal traversal), $C$ designates column (vertical traversal), and $S$ designates scalar.

\[
\text{orientations} \quad O \ ::= \quad C \mid R \\
\text{types} \quad T \ ::= \quad O<T> \mid S
\]

During the creation of the dataflow graph, each node is assigned a type. The input and outputs are assigned types derived from the input file specification, whereas the types associated with intermediate results are inferred by the BTO compiler.

Figure 3.3 shows several types with a corresponding diagram depicting the container shapes: a row container with scalar elements (upper left), a nested container for a row-major matrix (right), and a partitioned row container (lower left). Partitions are a general type construct used to introduce data parallelism into the program, as we discuss shortly.
3.1.1 Lowering to Loops over Scalar Operations

BTO lowers matrix and vector operations to vector and then to scalar operations by repeatedly examining the types of the containers and then introducing loops and appropriate lower-level operations. Table 3.1 shows some of the lowering rules for BTO. More details about these rules are given in [10]. Consider operation 1 (A * x) of the BATAX kernel. The matrix A is row-major, and the vector x is a column-vector, so we have the following types:

\[ A : C<\tau> \quad \text{and} \quad x : C<\tau>. \]

Thus, the only match in Table 3.1 is the second r-scale rule. This rule introduces a loop (with index i) that iterates down the rows of A, as shown in the sequence of transformations at the top of Fig. 3.4. We use MATLAB-style notation, so A(i,:) is row i of matrix A, and A(:,j) is column j. Inside the i loop, we have the product of a row-vector (type R<\tau>) and a column-vector (type C<\tau>), which is lowered by rule dot to another loop (with index j) that computes the inner product.

Suppose the matrix A had instead been specified in Fig. 3.1 as column-major. Then the type of A

Table 3.1: Linear algebra lowering rules.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Op and Operands</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans</td>
<td>O&lt;\tau&gt;'</td>
<td>O'&lt;\tau&gt;'</td>
</tr>
<tr>
<td>s-add</td>
<td>S + S</td>
<td>S</td>
</tr>
<tr>
<td>add</td>
<td>O&lt;\eta&gt; + O&lt;\tau&gt;</td>
<td>O&lt;\eta&gt; + O&lt;\tau&gt;</td>
</tr>
<tr>
<td>s-mult</td>
<td>S x S</td>
<td>S</td>
</tr>
<tr>
<td>dot</td>
<td>R&lt;\eta&gt; x C&lt;\tau&gt;</td>
<td>\sum(\eta x \tau)</td>
</tr>
<tr>
<td>l-scale</td>
<td>S x O&lt;\tau&gt;</td>
<td>O&lt;\tau&gt; x S</td>
</tr>
<tr>
<td>r-scale</td>
<td>O&lt;\tau&gt; x S</td>
<td>O&lt;\tau&gt; x S</td>
</tr>
<tr>
<td></td>
<td>C&lt;\eta&gt; x C&lt;\tau&gt;</td>
<td>C&lt;\eta&gt; x C&lt;\tau&gt;</td>
</tr>
<tr>
<td></td>
<td>C&lt;\eta&gt; x R&lt;\tau&gt;</td>
<td>C&lt;\eta&gt; x R&lt;\tau&gt;</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
t_0 &= A \ast x \quad \Rightarrow \quad \text{for } i = 1:m \\
t_0(i) &= A(i,:) \ast x \\
&\quad \Rightarrow \quad \text{for } i = 1:m \\
&\quad \text{for } j = 1:n \\
&\quad t_0(i) += A(i,j) \ast x(j) \\
t_0 &= A \ast x \quad \Rightarrow \quad \text{for } j = 1:n \\
t_0 &= A(:,j) \ast x(j) \\
&\quad \Rightarrow \quad \text{for } j = 1:n \\
&\quad \text{for } i = 1:m \\
&\quad t_0 += A(i,j) \ast x(j)
\end{align*}
\]

Figure 3.4: Lowering \( t_0 \leftarrow Ax \) to loops and scalar operations. The pseudo-code uses MATLAB notation.

would be \( R\langle C\langle S\rangle \rangle \) and the lowering would instead proceed as shown in the bottom sequence of Fig. 3.4.

First, the rule \texttt{dot} applies, which introduces a loop that goes across the columns of \( A \). The inner operation is then the product of a column-vector and a scalar, so the \texttt{r-scale} rule applies, and \( \text{BTO} \) introduces a loop that goes down each column of \( A \), scaling it by \( x(j) \).

The loops generated during this lowering phase are abstract loops, with operation, index, and dependency information. These abstract loops can be fused and manipulated during the search and transformation phase. During code generation, these loops can be concretized as either thread dispatch loops, tiling loops, or sequential C loops, based on the specification of the \( \text{BTO} \) user.

### 3.1.2 Applying Legal Transformations

The main transformations we focus on in this article are loop fusion, array contraction, and data parallelism. Each code transformation must be checked to ensure legality: the onus of preserving correctness lies on the transformation code, not the search algorithm. This approach allows the search strategies to overapproximate the legal space safely. After correctness is verified, the code transformations are applied to the lowered loops so that the final code variant can be tested.

#### 3.1.2.1 Loop Fusion and Array Contraction

Figure 3.5 shows an example of applying fusion to the \( \text{BATA}X \) kernel. As we saw above, the first matrix-vector product (operation 1) is lowered to a sequence of inner products. On the other hand, the second matrix-vector product (operation 2) operates on \( A^T \), which has type, \( R\langle C\langle S\rangle \rangle \), so it is lowered into a linear combination of columns of \( A^T \) (similar to the example at the bottom of Fig. 3.4). Now, both outer
loops of operations 1 and 2 iterate over the rows of $A$, so they can be fused as shown in Fig. 3.5. The final transformation, also shown in Fig. 3.5, is array contraction, replacing the temporary array $t_0$ with the scalar $\alpha$. Operation 3 cannot be fused with operations 1 and 2 because it depends on the temporary vector $t_1$, and all of $t_1$ is updated in each iteration of the $i$ loop. Thus, the $i$ loop must finish before the $j$ loop may begin.

BTO breaks fusion legality into two main tests: data dependency tests and iteration space tests. Data dependency tests are based on the dataflow graph. Two operations cannot fuse if there is another node in a path between them. Figure 3.6 shows a case where nodes 1 and 3 cannot fuse because of the path from 1 to 3 through 2. Also, if the two nodes are in a pipeline (the output of one operation is an input to another operation), the first operation’s output cannot be a summation (i.e., a dot rule). The second test is the iteration space test: two loops can fuse only if both loops are iterating over the same iteration space. This situation is determined by examining the types and seeing which lowering rule generates the loop. From the lowering rule we can determine which data elements are being iterated over and in what order, and before fusion we ensure that those iterations are the same for both loops. As a heuristic, we additionally reject fusions of loops with no shared data; this cuts down the search space without losing useful loop fusions.

3.1.2.2 Data Parallelism

In addition to loop fusion, BTO introduces data parallelism by partitioning the data. From the point of view of types, a partition is an additional nesting of containers. A partition splits one of the existing dimensions of the data. For example, the row-major matrix $A$ (of type $C_{R<R>S>}$) can be partitioned with

Figure 3.5: Lowering $y \leftarrow \beta A^T x$, fusing two loops, and contracting the temporary array $t_0$. 

horizontal cuts by adding an outer $C$ container,

$$C<C<R>S>>,$$

or it can be partitioned with vertical cuts by adding an outer $R$ container,

$$R<C<R>S>>.$$  

We refer to the choice of a $C$ or $R$ partitioning as the axis of partitioning.

Figure 3.7 shows in pseudocode how the matrix-vector product $Ax$ can be partitioned in two ways, corresponding to the two partitions of $A$ described above. Recall that the container abstraction is not tied to the actual memory layout and, in the case of partitioning, does not change the physical layout of the data. Because of the extra container, lowering creates an additional abstract loop, in this case the outermost loop on the left and right of Fig. 3.7. During code generation, this abstract loop is transformed into code that spawns threads. In this example $b$ is the block size, that is, the number of iterations given to each thread. That block size is an additional search parameter for each parallel container in the empirical search.

\[
\begin{align*}
&\text{for } k=1:b:n \\
&\quad \text{for } i=k:k+b-1 \\
&\quad \quad \text{for } j=1:m \\
&\quad \quad \quad t(i) += A(i,j) \times x(j) \\
&\text{for } k=1:b:m \\
&\quad \text{for } i=1:n \\
&\quad \quad \text{for } j=k:k+b-1 \\
&\quad \quad \quad t(i) += A(i,j) \times x(j)
\end{align*}
\]

Figure 3.7: Two ways to partition $t = Ax$.

BTO uses the pthreads library for shared-memory parallelism and ensures that it introduces threading only where no threads will write to the same memory. When a summation must be done in parallel, BTO
generates code to introduce a temporary vector or matrix to gather the results of each thread and perform the reduction after joining.

The legality of every partitioning must also be checked for each operation. In the absence of fusion, doing so is simply a matter of checking the type of each operand and the result of a given operation. We introduce a partition to the data in an operation by choosing which of the lowering rules in Table 3.1 will generate the code for that operation. We then add either a row or column partition to whichever of the data types are involved in the partition operation. For example, the `dot` rule iterates over a row of the left argument and column of the right argument, so to apply `dot` partition to operation 1, BTO makes the following transformation by adding a row container on the left and a column container on the right:

$$C^{<R<S>>} \times C^{<S>} = C^{<S>}$$

$$\Downarrow$$

$$R^{<C^{<R<S>>}} \times C^{<C<S>>> = C^{<S>}}$$

The partitioning above generates the code in the right of Figure 3.7. Similarly the code in the left of Figure 3.7 is generated by applying an `r-scale` partition, which adds a column type to the left operand and result:

$$C^{<R<S>>} \times C^{<S>} = C^{<S>}$$

$$\Downarrow$$

$$C^{<C^{<R<S>>}} \times C^{<S>} = C^{<C<S>>>$$

### 3.2 Representing and Searching the Transformation Space

As we saw in the previous section, the BTO compiler can apply several transformations: loop fusion, array contraction, and data parallelism through multithreading. However, these transformations may be applied in many different and conflicting ways, and BTO's job is to find which combination of code transformations results in the best performance. This is nontrivial to achieve because there are many trade-offs, such as register pressure versus maximizing data locality for cache. We refer to all possible combinations of transformation choices as the `search space` for a given BTO Kernel. This search space is sparse, consisting of a
high ratio of illegal to legal programs. Further, within the legal programs, only a handful achieve good performance. The search space is also discrete because performance tends to cluster with no continuity between clusters. Efficiently searching this space is the goal, and doing so requires a well-designed representation of the search space. This section describes the search space and the challenges with regard to representing the space efficiently. We present a domain-specific representation that enables BTO to eliminate many illegal points without spending any search time on them. This section also sets up the discussion of the new search algorithm in Section 3.3.

The optimization search space we consider here has three dimensions: loop fusion, axes of data partitioning, and number of threads for a parallel partition. Even considering only these three dimensions, there is a combinatorial explosion of combinations that BTO considers. The search space could have included more choices, but some optimizations are almost always profitable. For example, BTO always applies array contraction in the process of fusing loops; the resulting reduction in memory traffic almost always improves performance. Similarly, BTO always chooses a lowering of operations that traverses matrices along contiguous memory. (For example the inner loop over a column-major matrix always traverses over a column.) If no such lowering is possible, BTO reports an error to the user.

3.2.1 A Straightforward but Inefficient Representation of the Search Space

To demonstrate the challenge of developing an efficient representation of the search space, we start by presenting a straightforward representation and explain why it results in an inefficient search. Loop fusion can be described as a graph, with loop nests as nodes and edges labeled with the amount of fusion between the two loop nests. Each node can have a Boolean describing whether or not it is partitioned and two integer parameters: one for the number of threads and one for the axis of partitioning. This representation can be easily encoded in a vector of bits or integers, which can then be input to off-the-shelf search algorithms.

However, this approach fails to capture important information about the search space, and, as a result, contains a large number of illegal points. For N loops, there are $N(N - 1)/2$ possible loop fusions, which means $2^{N(N-1)/2}$ points in the loop fusion space at each level of loop nesting. For a kernel such as GEMVER with 9 operations, that creates 68.7 billion points without even considering data partitioning or number of
threads. The problem with this representation is that it does not capture the interactions between choices in fusion and partitioning. In our experiments, the search time using this representation was dominated by discarding illegal points. We now summarize two important features that this representation does not encode.

**Fusion is an equivalence relation**

Consider the first line of the GEMVER kernel, which consists of two outer products and two matrix subtractions, each of which we have labeled with a subscript.

\[ A \leftarrow A - d \cdot u \cdot a^T - c \cdot w \cdot v^T \]

The subscripts in these and the following equations represent the loops associated with those operations. We can represent the fusion possibilities for the above with an adjacency matrix \( M \), where \( M(i,j) \) shows the depth of fusion between the loop nests of \( i \) and \( j \): an entry of 1 means that the outer loops are fused whereas an entry of 2 means that both the outer and inner loops are fused. Below, we show a valid fusion choice on the left and an invalid fusion choice on the right.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The matrix on the left describes fusing the outer loop of all four operations; but only \( a, b, \) and \( c \) have the inner loop fused. The matrix on the right indicates fusing the inner loop of \( a \) with \( b \) and \( b \) with \( c \), but not \( a \) with \( c \), which of course is impossible. We can describe these constraints as forcing the relation specified in the adjacency matrix to be an equivalence relation at every depth. That is, if operation \( i \) is fused with operation \( j \) and \( j \) is fused with \( k \), then \( k \) must be fused with \( i \).

**Fused operations must use the same number of threads**

Consider a fuse graph that specifies a fusion of operations \( a \) and \( b \) but then a partition that specifies that \( a \) use 4 threads and \( b \) use 6 threads. Partitioning the two operations with different thread counts
prevents these two operations from being fused.

With a maximum thread count of 8, taking these two restrictions into account brings the number of points in the search space down to just over 1,000, or less than one-tenth of a percent of the number of points without these restrictions.

### 3.2.2 An Efficient Representation of the Search Space

Designing a representation that respects the restrictions discussed in Section 3.2.1 requires domain knowledge. At the expense of having to custom-build the search algorithm, we designed a representation that disallows, with no search time required, a large number of illegal points.

Loop fusion is represented by **fuse-sets**. Each operation (node in the dataflow graph) is given a unique identifier, and each abstract loop is represented with curly braces `{}`.

A single loop operation is represented as `{ID}`, where ID is a number identifying an operation node in the dataflow graph. For example, operation 3 of BATA, scaling a vector, is represented as `{3}`. A two-loop operation, such as a matrix-vector product, is represented as `{ID}`. When discussing a specific loop, we annotate it using a subscript after the first curly brace, as in `{i1}`, where i describes an axis of the iteration space. We use i to describe the iteration over rows of a matrix and j for columns of a matrix. A complete iteration space for a matrix can be described as `{i{j}} or `{j{i}}. For example, operation 1 of BATA, the matrix-vector product shown in Fig 3.4, can be described as `{i{j1}} (for row-major A) or `{j{i1}} (for column-major A). The fuse-set representation of all three operations of BATA is

\[ \{i\{j1\}\{i\{j2\}\}\{j3\}. \]

Fusion is described by putting two operations within the `{}`. For example, fusing the outer loops of operations 1 and 2 of BATA changes the fuse-set representation to

\[ \{i\{j1\}\{j2\}\}\{j3\}. \]

This notation encodes the equivalence relation of loop fusion, disallowing a large number of illegal fusion combinations.
In BTO, fuse-sets are more general than described so far. They also represent the partitioning loops used for data parallelism. The matrix-vector operation of \( \{i, j_1\} \) can be partitioned as \( \{p(i), \{i, j_1\}\} \) or \( \{p(j), \{i, j_1\}\} \), where the \( \{ \)s annotated with \( p(i) \) and \( p(j) \) describe the new iteration axis and the existing \( i \) or \( j \) loop variables that the partition affects. The search tool must specify which existing loop is being modified and how many threads should be used. Fuse-sets can represent any level of nesting this way and describe both sequential loops and data parallelism. The following is the fuse-set description of the BATA\(X \) kernel with operations 1 and 2 fused and with partitioning added to both loops:

\[
\{p(i), \{i, j_1\}\}\{j_2\}\{p(j), \{j_3\}\}.
\]

By extending the fuse-set representation to partitioning, thread counts can be assigned to each partition fuse-set, eliminating the consideration of points with mismatched thread counts within a fused operation. BTO uses this representation to enumerate or manipulate the fuse-sets and to generate the search space. This approach allows BTO to avoid touching the majority of the illegal points one would encounter using the straightforward representation.

This representation is complete in the sense that any combination of loop fusions can be represented by a set containing each operation in the fused loops, and additional data partitionings can be represented by sets containing the partitioned operations. The remaining parameters, partition dimension and size, can be represented by annotations on those fuse sets. Thus any code transformation that BTO can perform is represented by a single fuse set.

In the fuse-set representation, every configuration of fuse-sets represents a possible fusion, unlike with a graph-based representation. In addition, by annotating the partition fuse-sets with the relevant data for that partitioning, the effective partition search-space is reduced, throwing out nonsensical combinations of partition parameters. Figure 3.8 shows a graphical representation of an overly general search space and what area of that search space BTO currently searches. The gray areas represent illegal programs. This area is large, and spending time in it makes search times intractable. This section describes a representation that allows BTO to spend time only on the section labeled BTO Considered Search Space, which contains many fewer illegal points. To further improve search times, within the legal space, BTO prunes points it
deems unlikely to be unprofitable, as we discuss in the next section.

![Figure 3.8: Visualization of the search space, showing how BTO avoids searching a large portion of illegal points.](image)

### 3.2.3 Incremental Type-Pruning

Although the representation used by BTO greatly reduces the illegal points in the search space, a significant number of illegal points remain. Identifying them as early as possible is key to a fast search. The fuse-set representation says nothing about the data dependencies of the operations: BTO must check transformation legality using the type information and dataflow graph as described in Section 3.1.

One of the most difficult parts of the search is finding data partitionings that allow operations to be fused. To facilitate pruning illegal fusion-partitioning combinations, we added an incremental type-pruning approach to make it easy for any search to request legal transformations.

Before a search begins, BTO determines every legal partitioning for each operation in the kernel. Each operation has at most three possible axes of partitioning. Each of these partitioning axes has a corresponding lowering rule in Table 3.1: \texttt{dot}, \texttt{l-scale}, or \texttt{r-scale}. For example, the two partitionings added in Fig. 3.7 correspond to the \texttt{l-scale} and \texttt{dot} lowering rules, respectively. In each case the lowering rule uniquely determines how the types change for the partitionings.

During the search, the search algorithm can ask for all the possible ways to partition a specific fuse-set while maintaining fusion. The type-checker looks at the pregenerated partitionings for each operation in the fuse-set and eliminates any partitioning combinations whose data-types conflict or whose partitionings create
a reduction that prevents pipeline fusion. The type-checker returns a list of all combinations that are not eliminated in this way. This list may consist of zero to many combinations that work for a fuse-set, but all will be legal. This approach quickly rules out the illegal combinations, leaving only the legal points to consider.

To illustrate these ideas, we consider the partitioning and fusion choices in the BATAX kernel. Below we show two ways to partition operation 1 of BATAX, \( t_0 = A \times x \). We use MATLAB’s colon notation for a complete iteration and \( k \) for the subblock on which to operate in parallel. On the right is the representation as a fuse set. Partitioning (X) cuts the rows of \( A \) and vector \( t_0 \), while partitioning (Y) cuts the columns of \( A \) and the vector \( x \). Partitioning (Y) leads to a reduction at the parallel level, so \( t_0 \) is not available for use until after a join.

\[
\begin{align*}
\text{(X)} & \quad t_0(k) = A(k,:) \times x & \{p(i)\{i\{j1\}\}\} \\
\text{(Y)} & \quad t_0 += A(:,k) \times x(k) & \{p(j)\{i\{j1\}\}\}
\end{align*}
\]

Operation 2 of BATAX, \( t_1 = A' \times t_0 \), can be partitioned in the following ways.

\[
\begin{align*}
\text{(Z)} & \quad t_1(k) = A(k,:) \times t_0 & \{p(j)\{i\{j2\}\}\} \\
\text{(W)} & \quad t_1 += A(:,k) \times t_0(k) & \{p(i)\{i\{j2\}\}\}
\end{align*}
\]

BTO generates all four of these partitioning possibilities before search begins. Then, at some point during the search, the search algorithm may ask how to partition operations 1 and 2 while keeping them in the same fuse-set. Data dependence analysis says that partition (Y) of operation 1 will cause fusion to fail, because (Y) introduces a reduction, so operation 1 must be partitioned by using method (X). However, this limits the options for operation 2. Since matrix \( A \) is shared, in order to achieve fusion after partitioning, \( A \) needs to be accessed the same way in both partition loops. From partitioning (X) we see that \( A \) is accessed as \( A(k,:) \). Because operation 2 accesses the transpose of \( A \), we must select partitioning (W), accessing \( A^T \) as \( A(:,k) \). In terms of the types, both (X) and (W) convert \( A \) from type \( C\langle R\langle S\rangle\rangle \) to type \( C\langle C\langle R\langle S\rangle\rangle\rangle \). This can also be expressed in the fuse-set notation: the partitions introduced in (X) and (W) both generate \( \{p(i)\} \). In large fuse-sets, the likelihood of finding a correct set of operation partitions randomly is small. BTO uses this more intelligent approach to quickly work through the type constraints. In this example, BTO successfully prunes the space and chooses (X) and (W) without having to individually
Search(program, N, num_generations) =
organism1 = Max-Fuse(program)
population = N randomly chosen mutations of organism1
for j = 1 to num_generations do
    empirically evaluate the population
    new_population = {Best Organism Found}
    for i = 1 to N do
        parent1 = tournament_select(population)
        parent2 = tournament_select(population)
        new_population = new_population ∪ {crossover(parent1, parent2)}
        mutate every organism within new_population
        population = new_population
    return the best performing organism within the population

Figure 3.9: Pseudocode for the MFGA search algorithm.

3.3 Genetic/Greedy Search Strategy

This section describes the BTO search strategy, which is based on a genetic algorithm whose initial population is determined by a greedy search that tries to maximally fuse loops. We refer to this search strategy as MFGA, for Maximal Fusion followed by a Genetic Algorithm. Section 3.4 presents empirical results that support this design compared to some alternatives.

Genetic algorithms are a category of global optimization metaheuristics inspired by biological evolution [46]. In our setting, a set of transformation choices creates a code variant, which plays the role of an organism in the genetic algorithm. A genetic algorithm operates on a population of organisms. At each generation, the worst organisms are removed from the population and are replaced with newly generated organisms. Genetic algorithms do not always provide simple off-the-shelf solutions because the results are highly sensitive to (1) a representation for organisms (which we accomplished in Section 3.2), (2) a definition of mutation, and (3) a definition of crossover. We present the mutation and crossover operations in this section. A high-level outline of the MFGA algorithm is given in Figure 3.9.

The user can run the genetic algorithm for as much or as little time as desired by specifying the number of generations. We also include an option to terminate the search if there has been no improvement after a user-defined period of time. The GEMVER kernel is representative of the largest kernels that the
BTO is intended to be used on, and it takes less than 2 minutes to find the best optimizations for it by using MFGA. This shows that a user can run BTO search and get an answer quickly.

### 3.3.1 The Max-Fuse Greedy Search

The search begins with a greedy Max-Fuse (MF) heuristic: we attempt to fuse as many of the loops as possible to the greatest depth possible, using the representation described in Section 3.2. The MF search starts from unfused but partitioned variants of the kernel. Continuing with the BATAK example, the following fuse-sets represent the unfused but partitioned kernel. $X$, $Y$, and $Z$ are unknowns that represent the partitioning axis and are found during the MF search.

{\{X_i\{j_1\}\}\{Y_i\{j_2\}\}\{Z_j\}}

The MF search attempts to fuse loops in a depth-first manner, starting with outermost loops and moving toward the innermost loops. The search is greedy in that it immediately fuses two loops when doing so is legal, even though that fusion might disable another opportunity for fusion somewhere else in the program. The MF search uses the dataflow graph and type constraints to check legality. Suppose MF can fuse the loops labeled $X$ and $Y$. To fuse these loops, we need $X = Y$, so we proceed with the fusion and constrain ourselves to $X = Y$.

{\{X_i\{j_1\}\}\{X_i\{j_2\}\}\{Z_j\}}

$\Rightarrow$ {\{X_i\{j_1\}\}\{i\{j_2\}\}\{Z_j\}}

Recall from Section 3.2.3 that only one partitioning axis allows fusion of the two operations. Referring to the dataflow graph and type constraints, MF identifies that $X$ must be $p(i)$ because the alternative, $p(j)$, would mean that the necessary results from operation 1 would not be available for operation 2. After fusing outermost loops, MF proceeds to fuse loops at the next level down. In this example, MF fuses the $i$ loops surrounding operations 1 and 2:

{\{p(i)\{i\{j_1\}\}\{i\{j_2\}\}\{Z_j\}}

$\Rightarrow$ {\{p(i)\{i\{j_1\}\}\{j_2\}\}\{Z_j\}}
Going down one more level, MF examines fusing the \( j \) loops surrounding operations 1 and 2; but as we discussed in Section 3.1.2, \( \{j,2\} \) depends on the result of \( \{j,1\} \), and \( \{j,1\} \) is an inner product, so the loop’s result is not ready until the loop is complete, and therefore it cannot be fused with \( \{j,2\} \).

Popping back to the outermost loops, MF next considers whether the \( p(i) \) loop can be fused with \( Z \). The \( p(i) \) loop requires a reduction before the final vector scaling of operation 3, so 3 must reside in its own thread. Finally, there is only one axis of iteration in operation 3, so \( Z \) must be \( p(j) \). Therefore, the MF search produces the following organism:

\[
\{p(i)\{i\{j,1\}\{j,2\}\}\}\{p(j)\{j,3\}\}.
\]

In the worst case, each loop will try unsuccessfully to fuse with each other loop. Thus the Max-Fuse greedy heuristic is \( O(n^2) \), where \( n \) is the number of loops in the program.

### 3.3.2 Mutation

Our mutation operator applies one of the following four changes: (1) add or remove fusion, (2) add or remove a partitioning, (3) change the partition axis, or (4) change the number of threads. Mutations are constrained to the set of legal organisms; for example, attempting to further fuse an already maximally fused organism will fail, resulting in no change. However, mutations might randomly remove fusions or partitions. The following shows the removal of a partition from operation 3:

\[
\{p(i)\{i\{j,1\}\{j,2\}\}\}\{p(j)\{j,3\}\} \Rightarrow \{p(i)\{i\{j,1\}\}\{j,2\}\}\{j,3\}.
\]

### 3.3.3 Selection and Crossover

After the initial generation of organisms and each subsequent generation, we compile and empirically evaluate every organism and record its runtime to serve as the organism’s \textbf{fitness}, that is, the value the search tries to minimize. We then select 2\( N \) of the fittest organisms to be parents for the next generation, where the population size \( N \) can be user specified but defaults to 20.

**Parent Selection Method**

The population evolves through tournament selection [46]: \( k \) random organisms are chosen to be
potential parents, and the one with the best fitness becomes an actual parent. This process balances hill climbing with exploration, allowing less-fit organisms to sometimes become parents, and thus helping the algorithm escape locally optimal solutions that are not globally optimal. Larger values of $k$ cause the algorithm to converge more quickly on a solution, whereas smaller values of $k$ cause the algorithm to converge more slowly but increase exploration. BTO uses $k = 2$ to favor exploration.

**Crossover**

The crossover function takes two parent organisms and randomly chooses features of the two parents to create a child organism. The key strength of genetic algorithms is that crossover can sometimes combine the strengths of two organisms. Our crossover function generates the child recursively from the fuse-set representation of the two parents, making fusion decisions at each level and ensuring that those decisions remain valid for inner levels. Each crossover decision must also check the dependency graph to ensure the child will be a legal code variant.

Continuing with the BATAX example, consider the following two organisms $a$ and $b$:

\[
a : \{p(i)\{i\{j_1\}\{j_2\}\}\}\{j_3\}
\]

\[
b : \{p(i)\{i\{j_1\}\}\{p(i)\{i\{j_2\}\}\}\{p(j)\{j_3\}\}\}
\]

Parent $a$ partitions and partly fuses operations 1 and 2 but does not partition operation 3. Parent $b$ has all loops partitioned but has not fused operations 1 and 2.

Crossover chooses which parent to emulate for each operation, working from the outermost fuse level inward. Each step constrains the possibilities for the other operations. In our example, crossover might choose parent $a$ for the outermost level of operation 1, meaning 1 and 2 exist in the same thread (also using parent $a$’s partitioning axis $p(i)$ and thread number choice). Crossover then might choose parent $b$ for the next level, iteration $i$. This mechanism forces operations 1 and 2 not to be fused, resulting in $\{i\{j_1\}\}\{i\{j_2\}\}$.

Then the crossover moves to operation 3, and the process continues. If $b$ is chosen, the final child becomes $\{p(i)\{i\{j_1\}\}\{i\{j_2\}\}\}{p(j)\{j_3\}}$.

The genetic algorithm repeats the tournament selection process $N$ times, creating a new generation of organisms. It caches fitness values: if crossover ever produces an organism that was already tested in a
previous generation, the genetic algorithm uses the old fitness to save search time.

3.3.4 Search for Number of Threads

BTO uses a fixed number of threads to execute all of the data-parallel partitions in a kernel. We refer to this as the **global thread number** heuristic. An alternative is to use potentially different numbers of threads for each partition, which we refer to as the **exhaustive thread** search. In Section 3.4.4.3, we present data that shows that the exhaustive approach takes much more time but does not lead to significantly better performance.

BTO includes the search for the best number of threads in the MFGA algorithm. The initial number is set to the number of cores in the target computer architecture. The mutation function either increments or decrements the thread number by 2; I avoid searching over the odd numbers of threads to reduce the search space. The crossover function simply picks the thread number from one of the parents. After the genetic algorithm completes, MFGA performs an additional search for the best number of threads by testing the performance when using thread counts between 2 and the number of cores, incrementing by 2.

3.4 Performance Results

We begin this section with a comparison of 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 (Section 3.4.2). BTO generates code that is between 39% faster and 16% slower than hand-optimized code. The other automated tools and libraries achieve comparable performance for only a few of the kernels.

The later parts of this section evaluate the MFGA algorithm in more detail. We first compare MFGA to an exhaustive search, showing that MFGA finds routines that perform within 2% of the best possible routine (Section 3.4.3). We next present empirical results that explain our choices in design of the MFGA algorithm (Section 3.4.4). We defend the choice of starting with a greedy search based solely on fusion (instead of fusion and parallelism combined), we show why the genetic algorithm is needed in addition to the greedy search, and we justify using the same thread count for all of the parallel loops in a kernel.
Figure 3.10: Example sequence of BLAS calls that implement BICGK.

3.4.1 Test Environment and Kernels

This section describes the kernels that we chose for the empirical evaluation and the machines that we used to test those kernels. The kernels we chose for evaluation are listed in Table 3.2. We chose these kernels for two reasons: to test the range of the compiler and to represent real computations from numerical algorithms. VADD is an example of a particularly simple vector operation. WAXPBY, GESUMMV, and AXPYDOTE come from the updated BLAS [14], which have not been adopted by vendor-tuned BLAS libraries. The GEMV and AXPBY kernels map directly to a BLAS call, while the other kernels require two or more BLAS calls. As an example, Fig. 3.10 shows the sequence of BLAS calls that implement the BICGK kernel. The first three kernels in Table 3.2 are vector-vector kernels; the rest are matrix-vector kernels. The non-BLAS kernels were chosen from linear algebra applications: for example ATAX can be used in solving least-squares equations.

The more complicated examples are drawn from real numerical algorithms, which we show will not run efficiently using BLAS libraries or similar tools. GEMVER and GEMVT are used in bidiagonalization following the algorithm in [35]. The work in [69] extends that research and provided new kernels for reduction to Hessenberg form and for tridiagonalization: HESSLAZY, HESSBLK, and TRILAZY. In all of these cases, the authors write the best-performing versions by hand.

The computers used for testing include recent AMD and Intel multicore architectures, which we describe in Table 3.3. We ran performance experiments on square matrices of order 10,000 and vectors of dimension 10,000 for matrix-vector computations, and vectors of dimension 1,000,000 for vector-vector computations. We filled the matrices and vectors with random numbers. We chose these sssses to make the data larger than the L1 cache for the machines we tested. BTO allows the user to specify the problem size. In future work, we plan to investigate having BTO generate kernels that perform well over a range of sizes.
by using the standard approach of dispatching to differently optimized kernels based on size.

3.4.2 Comparison with Similar Tools

We place BTO performance results in context by comparing them with several state-of-the-art tools and libraries. Recall that BTO performs loop fusion and array contraction and makes use of data parallelism, but BTO relies on the native C compiler for lower-level optimizations such as loop unrolling and vectorization. In these experiments we used the Intel C Compiler (ICC) [36].

We begin by presenting detailed comparisons of the results on an Intel Westmere and then briefly summarize similar results on the AMD Phenom and Interlagos. We compare BTO to the following:

1. the best general-purpose commercial compilers: ICC and the PGI C Compiler (PGCC) [53],
2. the Pluto [15] research compiler,
3. a BLAS-based implementation combining kernels from Intel’s Math Kernel Library (MKL) [36], and
4. a hand-tuned implementation.

The input for ICC, PGCC, and Pluto was a straightforward and unoptimized version of the kernels written in C. This unoptimized version used 1d array indexing and did not use the C99 restrict keyword on inputs. The hand-tuned implementation was created by an expert in performance tuning who works in the performance library group at Apple, Inc. The expert applied loop fusion, array contraction, and data parallelism. The compiler flags we used with ICC were “-O3 -mkl -fno-alias,” and the flags for PGCC were “-O4 -fast -Mipa=fast -Mconcur -Mvect=fuse -Msafeptr” (“-Msafeptr” was not used on Interlagos). Data parallelism is exploited by ICC, PGCC, Pluto, and MKL by using OpenMP [22]. BTO and the hand-tuned versions use Pthreads [47]. Figure 3.11 shows the speedup relative to ICC on the y-axis for the linear algebra kernels in Table 3.2. (ICC performance is 1.) On the left are the three vector-vector kernels, and on the right are the nine matrix-vector kernels.

Analysis of the General Purpose Commercial Compilers

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 achieved only limited fusion.

**Analysis of the Research Compiler**

The Pluto results show speedups ranging from 0.7 to 5.7 times faster than ICC. The worst-performing kernels are AXPYDOT, ATAX, and GEMVT. 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 main 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 is otherwise identical and the difference in thread count has little effect. For the matrix-vector operations, if we enable fusion but not parallelization with Pluto’s flags, then Pluto matches BTO with respect to fusion. With both fusion and parallelization enabled, however, Pluto sometimes misses fusion and/or parallelization opportunities. For example, BICGK is parallelized but not fused. The GEMVER results depend on the loop ordering in the input file. For GEMVER, Pluto performs either complete fusion with no parallelism or incomplete fusion with parallelism; the latter provides the best performance and is shown in Figure 3.11.

**Analysis of MKL BLAS**

The MKL BLAS outperform 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. We were unable to determine why the BLAS perform so well for AXPYDOT. Surprisingly, the MKL BLAS GEMV does not perform as well as Pluto and BTO. Given the lack of fusion potential in this kernel, we speculate that differences in parallelization are the cause.

**Analysis of the Hand-Tuned Implementation**

The hand-tuned implementation is intended as a sanity check. 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 the thread
count or what appears to be a difference between Pthreads and OpenMP performance. For the matrix-vector operations, the hand-tuned version is the best for all but GEMV and GESUMMV, where it is equal to the best alternative.

**Analysis of the Results for BTO**

The BTO performance results show speedups ranging from 3.2 to 12.9 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 the hand-tuned version on WAXPBY because of a difference in thread counts. Similarly, BTO’s performance on the matrix-vector operations is close to that of the hand-tuned version. BTO fused loops the same way as the hand-tuned implementation for BICGK, GEMVER and GEMVT, with the only difference being in thread counts. For ATAX, both BTO and the hand-tuned version fused the same and selected the same number of threads, but BTO was slightly slower because of data structure initializations. In the hand-tuned version, initialization was parallel across threads, while BTO generates single-thread initialization code.

The performance for the three reduced-form kernels (HSBLK, HSLAZY, and TRILAZY) was significantly better than for the other matrix-vector operations because in each of these kernels all inner and outer loops can be fused into a single loop nest, which is the best case for our tool. For example, in GEMVER, only the outer loop can be fused.

**Results on AMD Phenom and Interlagos**

The results on AMD Phenom and AMD Interlagos
are similar to the Intel results discussed above, as shown in Table 3.4. The Pluto-generated code for the matrix-vector operations tended to perform worse than that produced for the other methods evaluated. On this computer, achieving full fusion while maintaining parallelism is of great importance. As previously discussed, Pluto tends to achieve fusion or parallelism but struggles with the combination. These results demonstrate the difficulty of portable high-performance code generation even under autotuning scenarios.

**Summary**

Compared with the best alternative approach for a given kernel, BTO performance ranges from 16% slower to 39% faster. Excluding hand-written comparison points, BTO performs between 14% slower and 229% faster. Pluto, ICC, PGCC, and BLAS all achieve near-best performance for only a few points; however, BTO’s performance is the 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.

**3.4.3 MFGA Compared with Exhaustive Searches**

In this section, we show how the performance of BTO’s MFGA search algorithm compares with the best version that can be produced by using exhaustive or nearly exhaustive search strategies on Intel Westmere. These strategies require long-running searches that can take days to complete. For the smaller kernels, an exhaustive search is possible. For larger kernels, exhaustive search was not possible, so we instead use a strategy that is exhaustive with respect to each optimization but orthogonal between optimizations, as described in Section 3.4.4.1. For the largest kernels, GEMVER and HESSLAZY, even the orthogonal approach took too much time, not completing even after weeks of running. We compared the performance of kernels produced by MFGA as percentage of the exhaustive search for smaller kernels or as a percentage of the orthogonal search for larger kernels such as GEMVT and GESUMMV. MFGA produces kernel performance within 1-2% of the best performance.
3.4.4 Evaluation of Search Methods

In this section, we examine the data that led to creating the MFGA search strategy. All the experiments in this section were performed on the Intel Westmere.

3.4.4.1 Orthogonality of Fusion and Thread Search

The MFGA strategy starts with Max-Fuse, a heuristic specifically for fusion parameters. It ends with a search only over thread count parameters. These additions significantly improve overall performance, as we show below. However, to have confidence in the usefulness of these heuristics, we want to know whether these aspects of the search are orthogonal. For example, if the best fusion combination depends heavily on the number of threads being used, using a fusion-only heuristic is ineffective. In general when designing a problem-specific search strategy, it helps to understand how heavily interdependent the search parameters are. I test this search strategy in the BTO search problem by creating an explicitly orthogonal search and comparing its performance with that of an exhaustive search for feasible kernels.

We define **orthogonal search** as first searching only the fusion parameters, then using only the best candidate, searching every viable thread count. We evaluated the effectiveness and search time of the orthogonal search as compared with an exhaustive search using the smaller kernels: ATAX, AXPYDOT, BICGK, VADD, and WAXPBY. For all kernels, orthogonal search found the best-performing version while taking 1-8% of the time of exhaustive search, demonstrating that searching the fusion space orthogonally to the thread space dramatically reduces search time without sacrificing performance. This reduction in search time results in part from the chosen orthogonal ordering. By searching the fusion space first, we often dramatically reduce the number of data-parallel loops and hence the size of the subsequent thread-count search space.

The MFGA strategy is not a strictly orthogonal search: after applying the Max-Fuse heuristic, the genetic algorithm can change fusion and partitioning parameters simultaneously. However, knowing that fusion decisions are usually correct regardless of thread parameters allows us to use heuristics like Max-Fuse that focus on fusion alone.
3.4.4.2 Fusion Search

Next, I focus on fusion strategies. In this section I analyze the choice of using a combination of a genetic algorithm and the Max-Fuse heuristic.

I compare four search strategies on our most challenging kernel, GEMVER. In particular, I test random search, the genetic algorithm without the Max-Fuse heuristic, the Max-Fuse heuristic by itself, and the combination of the Max-Fuse heuristic with the genetic algorithm (MFGA). The random search repeatedly applies the mutation operator from the genetic algorithm, essentially creating a random walk through the search space. Each step of the walk might be the fusion or unfusion of a loop, or changing the number of threads for a single operation, for example. Comparing this to the full genetic algorithm tests the importance of the crossover and selection in the search strategy.

Figure 3.12 shows the performance over time of each of the search methods. (MF is a single point near 3 GFLOPS.) Because the search is stochastic, each of the lines in the chart is the average of two runs. MFGA finds the optimal point in less than 10 minutes on average. Without the MF heuristic, GA alone eventually reaches 90% of MFGA but requires over an hour of search time. The Random search plateaus without ever finding the optimal value. The MF heuristic by itself achieves 40% of the performance of MFGA.

We conclude that a combination of GA and MF is the best strategy for the fusion portion of the search.

3.4.4.3 Thread Search

Using the MFGA heuristic described in the previous section, we explore several thread search strategies, including the global thread number and the exhaustive strategies discussed in Section 3.3.4. The baseline test is the MFGA search with number of threads set equal to the number of cores (24 for these experiments), which we refer to as the const strategy. Recall also that the global strategy starts with MFGA and then searches over a single parameter for all loop nests for the number of threads. Recall that the exhaustive search replaces the single thread parameter with the full space of possible thread counts, namely, by considering the number of threads for each loop nest individually.

The results for seven kernels are given in Fig. 3.13. The top chart shows the final performance of the
best version found in each case. Searching over the thread space improves the final performance compared with using a constant number of threads (e.g., equal to the number of cores), with negligible difference in kernel performance between the global thread count (fixed count for all threads) and fully exhaustive approaches (varying thread counts for different operations). The bottom chart in Figure 3.13 shows the total search cost of the different thread search approaches, demonstrating that global thread search improves scalability without sacrificing performance.
### Table 3.2: Kernel specifications.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXPYDOT</td>
<td>[ z \leftarrow w - \alpha v ] [ \beta \leftarrow z^T u ]</td>
</tr>
<tr>
<td>VADD</td>
<td>[ x \leftarrow w + y + z ]</td>
</tr>
<tr>
<td>WAXPBY</td>
<td>[ w \leftarrow \alpha x + \beta y ]</td>
</tr>
<tr>
<td>ATAX</td>
<td>[ y \leftarrow A^T Ax ]</td>
</tr>
<tr>
<td>BICGK</td>
<td>[ q \leftarrow Ap ] [ s \leftarrow A^T r ]</td>
</tr>
<tr>
<td>GEMV</td>
<td>[ z \leftarrow \alpha Ax + \beta y ]</td>
</tr>
<tr>
<td>GEMVT</td>
<td>[ x \leftarrow \beta A^T y + z ] [ w \leftarrow \alpha Ax ]</td>
</tr>
<tr>
<td>GEMVER</td>
<td>[ B \leftarrow A + u_1 v_1^T + u_2 v_2^T ] [ x \leftarrow \beta B^T y + z ] [ w \leftarrow \alpha B x ]</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>[ y \leftarrow \alpha Ax + \beta B x ]</td>
</tr>
<tr>
<td>HESSLAZY</td>
<td>[ y \leftarrow -Y^T u - U^T z ] [ z \leftarrow -U^T u - Z^T u ]</td>
</tr>
<tr>
<td>HESSBLK</td>
<td>[ A \leftarrow A - u_1 v_1^T + u_2 v_2^T ] [ v \leftarrow A^T x ] [ w \leftarrow Ax ]</td>
</tr>
<tr>
<td>TRILAZY</td>
<td>[ y \leftarrow -Y^T u - U^T u ]</td>
</tr>
</tbody>
</table>

### Table 3.3: Specifications of the test machines.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cores</th>
<th>Speed (GHz)</th>
<th>L1 (KB)</th>
<th>L2 (KB)</th>
<th>L3 (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Westmere 5660</td>
<td>12</td>
<td>2.66</td>
<td>12 x 32</td>
<td>12 x 256</td>
<td>2 x 12</td>
</tr>
<tr>
<td>AMD Phenom II X6 1100T</td>
<td>6</td>
<td>3.3</td>
<td>6 x 64</td>
<td>6 x 512</td>
<td>1 x 6</td>
</tr>
<tr>
<td>AMD Interlagos 6274</td>
<td>16</td>
<td>2.2</td>
<td>64 x 16</td>
<td>16 x 2048</td>
<td>8 x 8</td>
</tr>
</tbody>
</table>

### Table 3.4: Performance data for AMD Phenom and 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.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>AMD Phenom</th>
<th>AMD Interlagos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLAS</td>
<td>Pluto</td>
</tr>
<tr>
<td>AXPYDOT</td>
<td>0.97</td>
<td>1.81</td>
</tr>
<tr>
<td>VADD</td>
<td>0.84</td>
<td>1.33</td>
</tr>
<tr>
<td>WAXPBY</td>
<td>0.79</td>
<td>1.40</td>
</tr>
<tr>
<td>ATAX</td>
<td>1.27</td>
<td>0.69</td>
</tr>
<tr>
<td>BICGK</td>
<td>1.27</td>
<td>0.80</td>
</tr>
<tr>
<td>GEMV</td>
<td>1.67</td>
<td>0.71</td>
</tr>
<tr>
<td>GEMVT</td>
<td>1.67</td>
<td>0.71</td>
</tr>
<tr>
<td>GEMVER</td>
<td>1.04</td>
<td>1.61</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>1.63</td>
<td>0.63</td>
</tr>
</tbody>
</table>
Figure 3.12: GEMVER performance over time for different search strategies on Intel Westmere. MFGA finds the best version more quickly and consistently than does either search individually.

Figure 3.13: Best runtime (top) and search time (bottom) for exhaustive and global thread searches. A constant thread number (e.g., equal to the number of cores) cannot achieve the runtime performance of either global or exhaustive thread search. Searching over a global thread count results in a much shorter search time without significantly worsening kernel performance.
Chapter 4

OCTOPI

4.1 Overview

Tensors are a multidimensional generalization of matrices and are a natural way to express many computations arising in scientific computing. The rank of a tensor is the number of dimensions; a vector is a rank-1 tensor and a matrix is a rank-2 tensor. Two types of tensor computation are particularly common: tensor decompositions, a computation frequently used in data analysis, and tensor contractions, a multidimensional analog of matrix-matrix multiplication used in coupled cluster electronic structure calculations [9,64], in spectral element discretizations of partial differential equations (PDEs) [24], and as a building block for tensor decompositions. In this chapter I focus on tensor contractions, which can be expressed as summation along one or more tensor dimensions.

For convenience, I represent tensor contractions using the Einstein summation convention, where whenever the same index appears twice in an expression, once as a superscript and once as a subscript, there is an implied summation over all values of an index. Thus, the vector inner product is represented as $y = u_i v^i$, the matrix-vector product as $y^i = A^i_j x^j$, and the matrix-matrix product as $C^i_k = A^i_j B^j_k$. The contraction of a rank-3 tensor with another rank-3 tensor along one dimension results in a rank-4 tensor

$$C^{ij}_{lm} = A^{ij}_k B^k_{lm} \equiv \sum_k A^{ij}_k B^k_{lm},$$

and the contraction of a rank-3 tensor with another rank-3 tensor along two dimensions results in a rank-2 tensor

$$C^i_l = A^i_{jk} B^{jk}_l \equiv \sum_j \sum_k A^i_{jk} B^{jk}_l.$$
I am interested in computing multidimensional tensor contractions as efficiently as possible. I focus on scenarios featuring computations over thousands of identically sized small tensors (size $O(1)$–$O(10)$ in each dimension) because they occur naturally in the spectral element method [24] and provide a building block for computations with large tensors in coupled clustered computations [9, 64]. Consider the case of a $p$th-order spectral element discretization of a PDE on a mesh with $N$ elements. For each mesh element, one must compute tensor contractions of the form

\[ D_{ij} = A^l_j B^k_i U_{kl} \]

in two dimensions or

\[ D_{ijk} = A^l_k B^m_j C^n_i U_{lmn} \]

in three dimensions. A naive implementation of the two-dimensional contraction requires $O(p^4)$ operations ($O(p^2)$ for each of the $p^2$ members of $D_{ij}$). However, this approach ignores redundant subcomputations across columns and rows of $D$. One can instead compute $W^l_i = B^k_i U_{kl}$ followed by $D_{ij} = A^l_j W^l_i$ at a cost of $O(p^3)$ operations. A similar reorganization of the three-dimensional computation reduces the cost from $O(p^6)$ operations to $O(p^4)$ operations. Tools such as the Tensor Contraction Engine (TCE) [9,34] and libtensor [27] seek to reorganize tensor contractions in this fashion to minimize the number of floating point operations.
operations [34]. In this chapter, I combine such reorganizations with an autotuning compiler so that the search space of possible implementations is much richer.

4.1.1 Code Generation

D[i, j, k] += A[l, k] B[m, j] C[n, i] U[l, m, n]

(a) Tensor contraction input to OCTOPI.

ex
access: linearize
define:
N = J = M = I = L = K = 10
variables:
temp3:(J, I, L)
A:(L, K)
C:(N, I)
B:(M, J)
U:(L, M, N)
D:(I, J, K)
temp1:(I, L, M)
operations:
temp1:(i, l, m) += C:(n, i)*U:(l, m, n)
temp3:(j, i, l) += B:(m, j)*temp1:(i, l, m)
D:(i, j, k) += A:(l, k)*temp3:(j, i, l)

(b) Input to TCR.

def performance_params {
... param PERMUTE_2_TX2[] = ['m'];
param PERMUTE_2_TY2[] = ['n', '1', 'm', 'l'];
param PERMUTE_2_BX2[] = ['i', 'm', 'l'];
param PERMUTE_2_BY2[] = ['i', '1', 'm', 'l'];
param UF_2[] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
}
/*@ begin CHiLL (...
cuda(2,block=(PERMUTE_2_BX2,PERMUTE_2_BY2),
thread=(PERMUTE_2_TX2,PERMUTE_2_TY2))
registers(2,"n","D")
unroll(2,"n",UF_2)
) */
... for (i=0; i< I; i++){
for (l=0; l< J; l++){
for (m=0; m< K; m++){
for (n=0; n< L; n++){
D[i*J*K + l*K + m] = D[i*J*K + l*K + m] +
(A[n*K + m] * temp3[l*1*L + i*L + n]);
}
}
}
}

(c) Search space for CUDA-CHiLL and Orio.

Figure 4.2: Example from [24].

To generate many possible variants of a tensor computation and to identify the best-performing implementation, Nelson, Jessup, Norris et. al [49] combined four tools: OCTOPI, TCR, the CUDA-CHiLL
extension of CHiLL [41] and Orio [32, 51]. Figure 4.1 shows how these tools work in collaboration, while Figure 4.2 illustrates the input at each stage and final output, starting from a mathematical input notation.

The user inputs to OCTOPI a high-level representation of a computation that resembles mathematical tensor notation, as shown in Figure 4.2(a). This example corresponds to Equation 4.1 and is adapted from the computation $C = A_x \otimes A_y \otimes A_z$ on p. 168 of [24]. In this example $\otimes$ is the kronecker product, creating a new tensor with the indices of both operands. For example the kronecker product of two matrices is a 4-dimensional tensor with $(A \otimes B)_{ijkl} = A_{ij}B_{kl}$. To this input, OCTOPI applies tensor-specific optimizations to reorganize the computation, as described in Section 4.2, to generate a set of inputs to TCR. For this example, OCTOPI generates fifteen different versions. While six versions all perform the same amount of floating-point computation, their performance on an Nvidia GTX 980 (Maxwell) varies by as much as 9%. The TCR input for what is ultimately the best-performing version is shown in Figure 4.2(b). TCR generates code and Orio annotations specifying which transformations and autotuning search space should be explored, an excerpt of which is shown in Figure 4.2(c). Each code variant is generated automatically using CUDA-CHiLL, a source-to-source compiler transformation and code generation framework that transforms sequential loop nests to high-performance GPU code. Orio tests empirical performance, executes a search over the parameter space, and determines which parameter values produce the best version. Orio also supports code transformations, but for this work, the compiler relies on CHiLL for all code transformations and uses Orio to search the performance parameter space. The resulting tuned GPU code is excerpted in Figure 4.2(d). In the given example, three kernels are generated and individually optimized, corresponding to the three summations in the TCR input, but the data remains on the GPU across these calls. The next sections describe the implementation of OCTOPI, generation of the search space, and search space navigation for the experiments in this paper.

4.2 OCTOPI Input and Tensor Transformation

The input to OCTOPI is a sequence of summation statements, as in the 2-D spectral element computation shown in Figure 2. The argument to the sum in Figure 4.2(a) is the summation indices, and the right hand side expression is computed for the entire (implicit) range for each index. The user can optionally
Algorithm 1 Creating a valid OCTOPI algebraic transformation

Input: a set $T$ of $n$ terms, with $T_i$ having indices $I(T_i)$

1. $c \leftarrow 0$
2. $d \leftarrow n$
3. while $n > 1$ do
4.     $d \leftarrow d + 1$
5.     for $i \leftarrow$ index occurring only in $T_a$ do
6.         Create term $T_d[I(T_a) - \{i\}] = \sum_i T_a$
7.     $T \leftarrow T \cup \{T_d\} - \{T_a\}$
8.     $c \leftarrow a$
9. end for
10. Choose any $a, b$ such that $a < b, b > c$
11. To enumerate exhaustively, perform depth-first search on these choices
12. Create term $T_d[I(T_a) \cup I(T_b)] = T_a \times T_b$
13. $T \leftarrow T \cup \{T_d\} - \{T_a, T_b\}$
14. $c \leftarrow b$
15. end while

specify the index dimension or a range of dimensions for testing.

OCTOPI takes a two-stage approach to optimizing these tensor contractions. First, it analyzes the
tensor for possible high-level transformations that might improve performance; it then passes each of these
transformed variants to TCR.

The most important transformation OCTOPI applies at a high level is so called strength reduction [8],
which is explained in the next section.

The link from TCR to CUDA-CHiLL and Orio allows OCTOPI to focus on tensor-specific optimiza-
tions, while using the existing tools for what they do best: autotuning, loop-reorganization, and translation
to high-performance GPU code.

The pseudocode for the OCTOPI algebraic transformation enumeration is in Algorithm 1. The input
to the algorithm is a set of multiplication terms, and the algorithm enumerates possible reorderings of those
terms, taking advantage of commutativity and associativity. The next section gives an example of the
performance advantages that can come from this computational reordering.

4.2.1 Transformation Example

Using again the tensor example above:

\[ D[i,j] = \text{Sum}(k,l), A[l,j] \times B[k,i] \times U[k,l] \]
The naive implementation of this code creates a 4-deep nested for loop.

```plaintext
for i
  for j
    for k
      for l
```

These loops can be interchanged, creating $N!$ loop choices for $N$ indices. In the example, there are 24 total arrangements of these loops. A few of them open opportunities for strength reduction. Strength reduction is a reorganization of the sums that takes advantage of partial sums to reduce the total computation. OCTOPI carries out strength reduction by finding all subexpressions that have a smaller iteration space than the full computation. It takes advantage of commutativity to find all reorganizations. In our example, the following are the possible subexpressions:

- $A[l j] * B[k i] (i j k l)$
- $B[k i] * U[k l] (i k l)$
- $A[l j] * U[k l] (j k l)$

The set of subexpressions depends on the size of the tensor, but for the computations OCTOPI encounters it is possible to enumerate them exhaustively. The first subexpression involves all four indices and so necessarily requires order $N^4$ operations. For the other two, OCTOPI can replace the single loop nest of order $N^4$ by two $N^3$ loop nests, thus reducing the amount of computation and improving performance.

For example, using

```plaintext
B[k i] * U[k l] (i k l),
```

OCTOPI generates the following pseudocode:

```plaintext
for i
  for l
    for k
      W[i l] += B[k i] * U[k l]
```
for i
    for l
        for j
            \( D[i \ j] += A[l \ j] \ast W[i \ l] \)

OCTOPI can additionally incorporate loop fusion:

for i
    for l
        for k
            \( W[i \ l] += B[k \ i] \ast U[k \ l] \)
        for j
            \( D[i \ j] += A[l \ j] \ast W[i \ l] \)

This variant performs the same number of operations, but has better memory usage and enables more optimizations for CUDA-CHiLL.

Following these same steps, the other possible subexpression in our example: \( A[l \ j] \ast U[k \ l] \) becomes:

for j
    for k
        for l
            \( W[j \ k] += A[l \ j] \ast U[k \ l] \)
        for i
            \( D[i \ j] += B[k \ i] \ast W[j \ k] \)

All variants shown for the two subexpressions have the same operation counts, but their performance depends on data layout in memory and subsequent transformations. In our example, OCTOPI generates and sends all versions to CUDA-CHiLL for autotuning.

This type of code transformation is extremely difficult at the source-to-source level: it requires complex loop interchange, strength reduction, and loop fusion. By considering the operations at the tensor level, OCTOPI can easily enumerate the possibilities and make the transformation space searchable.
4.2.2 Addition and Multiplication

The example above deals with a product of multiple tensors. OCTOPI has also been extended to cover sums of products of tensors, e.g

\[ A[i \ j] \cdot B[j \ k] + C[i \ k] + D[i \ k] \]

This extension was necessary to handle the full nekbone code. In this case OCTOPI first generates transformations for the multiplications, then the additions. Because two tensors can only be added if they have identical index spaces, the algorithm for searching the additions is simpler than the algorithm for multiplications. OCTOPI simply chooses two terms and creates temporary variable with the same index space. For the example above the three possible addition orderings are:

\[ (A[i \ j] \cdot B[j \ k] + C[i \ k]) + D[i \ k] \]
\[ A[i \ j] \cdot B[j \ k] + (C[i \ k] + D[i \ k]) \]
\[ (A[i \ j] \cdot B[j \ k] + D[i \ k]) + C[i \ k] \]

This computation can all be done in a single fused loop, or split out into three unfused loops with temporary tensor storage, or a partially fused code with some but not all of the temporaries. This is important because some architectures may benefit from moving only some of the data to GPU, so fusing everything may not always be best.

4.3 TCR and Generating the Search Space

The variants generated by OCTOPI are expressed in an an intermediate representation that is input to a lower-level tool (TCR) that encodes a set of parameterized code variants, which are ultimately used by CUDA-CHiLL to generate GPU code. From the representation in the code example in Figure 4.2(b), TCR creates a for loop for each different loop index listed in the operation and uses the tensor equation to generate the statement. In addition, this code is accompanied by a collection of transformations to be applied in CUDA-CHiLL to define the autotuning search space for this specific variant. The search space explored is a simplification of Khan et al.’s algorithm [40,41]. The algorithm finds the thread and block decomposition
and data placement in different levels of the memory hierarchy, along with additional transformations to optimize the thread program.

Data dependence analysis is used to determine the safety of parallelization and other reordering transformations. For the purpose of tensor contractions, dependencies are carried by loops with indices present in the right-hand side but not in the left-hand side of the tensor operation. All remaining loops may be executed in parallel. Chill also analyzes the memory access patterns for each of the input tensors. This thesis refers to array references where loop indices appear in order from innermost to outermost access data in the order in which they are mapped to memory (assuming row-major layout) as contiguous tensors. Contiguous tensors are desirable, as they lead to data access orders that achieve global memory coalescing and reuse in the GPU’s caches. Nevertheless, in most tensor contraction computations, not all tensors are accessed as contiguous tensors as there does not exist a loop order that is optimal for all data.

TCR first generates the search space for thread and block decomposition on the GPU. $Thread_X$ and $Thread_Y$ refer to the $X$ and $Y$ thread dimensions on the GPU; $Block_X$ and $Block_Y$ refer to the $X$ and $Y$ block dimensions on the GPU. The $X$ dimension is the leading one, such that adjacent $X$ threads with the same $Y$ value are usually mapped to adjacent GPU cores. Therefore, TCR chooses as candidates for $Thread_X$ any loop such that adjacent elements on an input tensor are accessed by adjacent threads so as to achieve global memory coalescing. Potential choices for $Thread_Y$, $Block_X$, and $Block_Y$ are selected by the following rules:

- Select parallel loop indices from the contiguous tensors from innermost to outermost loops.
- If the contiguous tensors have fewer than four parallel loops, then start selecting parallel indices from the non-contiguous tensors from outer to inner.

The search space also consists of different loop orders, which can be realized using loop permutation. Any loops that are inside the GPU kernel that improve memory layout of inner dimensions are considered as candidates for loop permutation. A final optimization included in the search space is unrolling of inner loops to reduce control flow, enhance register reuse and increase instruction-level parallelism. A number of unroll factors are considered, but these are relatively small because of the small loop iteration counts.
Additionally, included with these optimizations, the compiler always applies scalar replacement to explicitly copy the output tensor variable to a scalar temporary so that it is accessed in a register; it is copied back to global memory only at the end of a thread’s computation to reduce accesses to global memory.

4.4 Search Space Exploration

Enumerating all possible code variants of the autotuning problem posed by OCTOPI can be computationally prohibitive: for a given tensor computation, OCTOPI can generate a number of tensor variants, where each variant with a number of parameters for thread, block decomposition, and unroll can produce a large number of code variants. Given the permutation parameters for decompositions and integer parameters for unroll, the number of code variants grow exponentially with respect to the number of parameters. For \( \text{lg3t} \) (see Table 4.1), OCTOPI has 512,000 possible tensor-code variants for empirical evaluation. A promising approach to overcome this difficulty is through the use of a search algorithm that finds high-performing code variants while examining relatively few variants. However, designing a search algorithm to navigate the search space is quite challenging from a mathematical optimization perspective because permutations do not admit a natural ordinal relationship and integer parameters cannot be relaxed.

Prasanna et al. customize and adapt the model-based search algorithm proposed in [5, 7] to the search problem. The algorithm consists of sampling a small number of parameter configurations, empirically evaluating the corresponding code variants to obtain the corresponding performance metrics, and fitting a surrogate model over the input-output space. The surrogate model is then iteratively refined by obtaining new output metrics at unevaluated input configurations predicted to be high-performing by the model. The main extension consists in handling the permutation parameters by applying a preprocessing technique called feature binarization [13], where permutation values are transformed into binary vectors to enable surrogate modeling.

Algorithm 2 shows the pseudocode of the model-based search algorithm. The algorithm takes as input a set \( X_p \) of unevaluated configurations, the stopping criterion of maximum number \( n_{\text{max}} \) of allowed evaluations, and batch size \( bs \) that determines the number of concurrent evaluations at each iteration. In the initialization phase of the algorithm, \( bs \) configurations are sampled at random and evaluated in parallel.
Algorithm 2 Pseudo-code for the model-based search (Mlsearch)

**Input:** configuration pool $X_p$, batch size $bs$, max evaluations $n_{\text{max}}$

1. $X_{\text{out}} \leftarrow$ sample min\{ $bs, n_{\text{max}}$\} distinct configurations from $X_p$
2. $Y_{\text{out}} \leftarrow \text{Evaluate}_\text{Parallel}(X_{\text{out}})$
3. $M \leftarrow \text{fit}(X_{\text{out}}, Y_{\text{out}})$
4. $X_p \leftarrow X_p - X_{\text{out}}$
5. for $i \leftarrow bs + 1$ to $n_{\text{max}}$ do
6.    $Y_p \leftarrow \text{predict}(M, X_p)$
7.    $x_{bs}^i \leftarrow$ select $bs$ configurations from $X_p$ with the best performance in $Y_p$
8.    $y_{bs}^i \leftarrow \text{Evaluate}_\text{Parallel}(x_{bs}^i)$
9.    retrain $M$ with $(x_{bs}^i, y_{bs}^i)$
10.   $X_{\text{out}} \leftarrow X_{\text{out}} \cup x_i; Y_{\text{out}} \leftarrow Y_{\text{out}} \cup y_i$ /* $\cup$ denotes set union */
11.   $X_p \leftarrow X_p - x_{bs}^i$ /* $-$ denotes set difference */
6. end for

**Output:** $x \in X_{\text{out}}$ with the best performance in $Y_{\text{out}}$

to obtain their corresponding performance metrics. These points are then used as a training set to build a predictive model for performance. The iterative phase consists of predicting the performance of all remaining unevaluated configurations using the models, evaluating $bs$ configurations with best predicted performance, and retraining the model with the evaluation results. The batching allows for a higher degree of parameter space exploration and increases the probability of finding high-quality configurations in fewer iterations [6].

The main advantage of the proposed algorithm is the reduction in time for the search needed to find high quality parameter configuration.

The OCTOPI-Chill-ORIO toolchain deploys statistical machine learning methods [13] for building surrogate models. In particular, the tool implements randomized trees [18,28], a state-of-the-art machine learning algorithm, due to their ability to handle the binarized (permutation) parameters using recursive partitioning technique and to model nonlinear interactions among the parameters. This thesis refers to this model-based search as Mlsearch.

For lg3t, our research group ran the model-based search with 100 evaluations, which took 7 mins (approximately 4 sec per variant). Assuming the same time per variant, enumeration of 512,000 variants will take approximately 23 days.
Table 4.1: Benchmarks used in this study.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEM</td>
<td>example from Figure 4.2</td>
</tr>
<tr>
<td>lg3</td>
<td>local_grad3 from nekbone</td>
</tr>
<tr>
<td>lg3t</td>
<td>local_grad3t from nekbone</td>
</tr>
<tr>
<td>nekbone</td>
<td>Mini-app using optimized Lg3 and Lg3t</td>
</tr>
<tr>
<td>TCE</td>
<td>TCE example tensor [8]</td>
</tr>
<tr>
<td>S1 (s1_1-s1_9)</td>
<td>NWCHEM excerpt: 2 objects with 2&amp;4 dimensions</td>
</tr>
<tr>
<td>D1 (d1_1-d1_9)</td>
<td>NWCHEM excerpt: 2 objects with 4 dimensions</td>
</tr>
<tr>
<td>D2 (d2_1-d2_9)</td>
<td>NWCHEM excerpt: 2 objects with 4 dimensions</td>
</tr>
</tbody>
</table>

4.5 Performance Measurements

This chapter evaluates the integrated system from Figure 4.1 on the tensor-contraction computations in Table 4.1. The kernels in that table arise in spectral element and coupled cluster electronic structure calculations. This collection of computations (SPEM, lg3, lg3t and TCE) was selected because they allow us to evaluate tensor-contraction in isolation, and the nekbone and NWCHEM computations let us consider how tensor-contractions are used in full applications. Nekbone is a 3-dimensional spectral element proxy application derived from Nek5000 [52, 67]. It performs a conjugate gradient loop that operates over a sequence of tensor contractions recast as matrix multiplications, which comprises 60% of the sequential execution time. A problem size of $12 \times 12 \times 12$ was used, because these matched the size used in our reference implementation of the Nekbone mini-application. The small dimension sizes result from the order of the discretization polynomial; as it increases, the time required to converge also increases. NWCHEM is a software package for quantum chemistry and molecular dynamics simulations [68]. Table 4.1 also includes kernels [19] extracted from the CCSD(T) (coupled cluster theory with full-treatment singles and doubles, and triples estimated by using perturbation theory) computations of NWCHEM. These kernels are representative of what executes at the socket level, with loops of 16 iterations in each dimension, so are appropriate for evaluation on a single GPU.

These experiments were performed on an Intel Haswell CPU and three generations of Nvidia GPUs: TESLA C2050 (Fermi), TESLA K20 (Kepler) and GTX 980 (Maxwell). For the OpenACC results, we used
the Portland Group compiler (PGI) version 14.3, but the PGI compiler does not yet generate code for the GTX 980. The CUDA code was compiled by using the nvcc compiler for CUDA 5.5. For each point (code variant) in the search space, I compute average execution time over 100 repetitions.

4.5.1 Individual Tensor-Contraction Computations

First consider the optimization of the individual tensor-contraction computations summarized in Table 4.2. Three of the four, lg3, lg3t and TCE, all achieve performance of more than 40 GFlops on the GTX 980 and speedups of more than $20 \times$ over the sequential Haswell implementation. Performance on the other GPU platforms is comparable for lg3 and lg3t but significantly lower for TCE. The results on the SPEM kernel are quite different. It is a computation that does not speed up compared to the Haswell, and only achieves 1.99 GFlops on the GTX 980, largely because there is insufficient work to compensate for the overhead of copying data to/from the GPU.

Search time varies from a few minutes to a few hours, depending on the computation and the architecture. The older GPU architectures typically spend more time in search, and the tiny SPEM computation spends the longest because the performance of its versions is so similar. Nevertheless, given the enormous search spaces associated with all of these variants, the Mlsearch algorithm is performing well. Axel Rivera also compared performance for some of these code versions with prior work in [58] which used a brute force search of a smaller search space. He found that the performance resulting from Mlsearch was comparable and sometimes better than the prior brute force search.

I omitted from this table S1, D1 and D2, because each is comprised of nine kernels, which are shown in more detail in Figure 4.3. To summarize our results, performance ranges from 7 to 20 GFlops for S1, from 20 to 125 GFlops for D1 and 9 to 53 GFlops for D2, as will be shown in the next subsection. Search times for each of the nine kernels range from 8 to 32 minutes per kernel.

4.5.2 GPU Code Generation Strategies in Context

The results are different for the larger kernels: Nekbone, where the optimized lg3 and lg3t have been integrated into the code, the NWChem kernels S1, D1, and D2. Performance measurements for the gener-
Table 4.2: Results summary for individual tensor contractions.

<table>
<thead>
<tr>
<th></th>
<th>GTX 980</th>
<th>K20</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speedup</strong></td>
<td>GFlops</td>
<td>Search</td>
<td>GFlops</td>
</tr>
<tr>
<td>SPEM</td>
<td>0.63×</td>
<td>1.99</td>
<td>3556.0s</td>
</tr>
<tr>
<td>lg3</td>
<td>23.74×</td>
<td>42.74</td>
<td>324.8s</td>
</tr>
<tr>
<td>lg3t</td>
<td>22.87×</td>
<td>41.11</td>
<td>356.9s</td>
</tr>
<tr>
<td>TCE</td>
<td>29.77×</td>
<td>42.72</td>
<td>276.6s</td>
</tr>
</tbody>
</table>

Table 4.3: Performance in GFLOPS using OpenACC and ML-Search for nekbone

<table>
<thead>
<tr>
<th></th>
<th>Naive OpenACC</th>
<th>OpenACC TB</th>
<th>OpenACC Registers</th>
<th>ML-Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla K20</td>
<td>2.86</td>
<td>12.30</td>
<td>12.39</td>
<td>36.47</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>1.18</td>
<td>18.09</td>
<td>19.21</td>
<td>34.65</td>
</tr>
</tbody>
</table>
ated code are shown in Figure 4.3 and for the nekbone kernel in Table 4.3. I evaluate four different strategies for generating GPU code. OpenACC refers to replacing our toolchain’s generated CUDA constructs with OpenACC directives. I produced three OpenACC versions: Naive simply includes parallelization directives but no guidance on parallelization decomposition; TB adds directives on thread and block decomposition that were derived by Mlsearch; Registers additionally performs scalar replacement on the output variable since the private designation in OpenACC does not produce the desired result. Overall the Naive OpenACC code generation is even slower than sequential execution, but the other OpenACC versions sometimes exceed performance of our generated code. Nevertheless, the autotuning is essential to achieving high performance, since even the OpenACC versions begin from an autotuned version.

4.5.3 Comparison with Manual OpenMP Code

It is interesting to also ask whether a GPU is the right architecture for these computations. With the OpenMP comparison, I use manually-coded OpenMP versions, parallelizing an outermost loop for nekbone and using the OpenMP directions provided by the author of the NWChem excerpts. Figure 4.4 shows that
the GTX 980 GPU outperforms a 4-thread OpenMP version on the Haswell in all cases for all benchmarks.

4.5.4 Performance Scaling with Problem Size

Figure 4.5 shows the runtime of the best TCE kernel versions found by the search algorithm for six problem sizes on the same three different GPUs as the previous experiments: Tesla C2050, Tesla K20, and GTX 980. The runtime for all three GPUs increases sharply at approximately $3.0 \times 10^9$ flops, although the Tesla C2050 shows degrading runtimes before the other two GPUs. This increase in runtime is due to the problem array falling out of the GPU L2 cache. These experiments are not sufficient to show scaling with additional resources: each kernel makes full use of the GPU, and the OCTOPI-Chill-Orio toolchain cannot take advantage of node communication to use multiple GPUs. Also the focus on small problem sizes in this research means that our code generation is not taking advantage of the tiling that is necessary to avoid a performance dropoff at cache size boundaries.
Figure 4.5: Runtime versus Problem Size on the TCE Kernel
Chapter 5

Conclusions and Future Work

5.1 Conclusions and Future Work

For many problems in high-performance computing, the best solutions require extensive testing and tuning. I have presented an empirical autotuning approach for dense matrix algebra and tensor computations that is reliable and scalable. The two different tools I worked on, BTO and OCTOPI, had different input domains (linear vs. tensor algebra) and different output targets (C code with pthreads vs CUDA). These differences required different code transformations and different search strategies. But, in both cases, I found an effective DSL-based autotuning approach that provided significant advantages over previous approaches.

My experiments have shown that the BTO autotuning system outperforms standard optimizing compilers and a vendor-optimized BLAS library in most cases, and our results are competitive with hand-tuned code. We also describe how we developed our search strategies and tested the usefulness of each part of the search.

We plan to implement two big expansions of functionality: distributed-memory support through MPI and extension of matrix formats to include sparse matrices. These extensions will improve the usefulness of BTO while also providing an important stress test for the scalability of the search algorithms and code generation.

OCTOPI is an autotuning system for tensor contraction computations targeting GPUs. The system uses a tensor-specific mathematical representation as input to OCTOPI and generates an autotuning search space. The OCTOPI search space is tuned for both the problem domain of tensors with small dimension sizes,
and the optimizations necessary for GPU architectures. I explored the very large search space generated by these tools using machine learning, resulting in search times that are practical. I show speedup over sequential and OpenMP execution, as high as $29 \times$, and also demonstrate the necessity of autotuning when using OpenACC to generate efficient code. Our approach was driven by a desire to improve specific tensor problems not solved by current tools, but I also view this work as an exemplar for developing highly-tuned applications specialized for individual architectures starting with a mathematical representation of the problem in a DSL.

The linking idea of my contributions in these two projects is a representation of the problem that improves the search. In BTO I created a genetic algorithm by representing the linear algebra transformations as nested sets of operations. That representation and search algorithm allowed for efficient search of a large transformation space. In OCTOPI I created a front end language that represented tensors using explicit indices for multidimensional arrays. The OCTOPI language took advantage of that high-level representation to provide tensor transformations for the tools following it in the toolchain. In both cases careful problem space representation allows for improved performance results.

In the future, this work can be extended to further prune the autotuning search space, with better understanding of where pruning will not impact quality of results, and facilitate integration of the generated code into applications. As I expand the approach to surrounding computations, such as jointly optimizing `lgrad3`, `lgrad3t` and adjacent code, the search space will grow, and pruning it will be essential to feasibility. The toolchain itself could be made more modular, so that OCTOPI could be used to apply tensor transformations for other code backends, for example generating annotated C code directly for Orio.
Bibliography


