Vectorization and Register Reuse in High Performance Computing

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Kevin Alan Stock, M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2014

Dissertation Committee:
P. Sadayappan, Advisor
Atanas Rountev
Srinivasan Parthasarathy
ABSTRACT

Automatic vectorization is crucial for attaining maximum performance on modern processors in a performance portable fashion. There is significant room for improvement over techniques in existing compilers, as demonstrated in this work, through careful synthesis of vectorized code and application of aggressive performance optimizations, especially in the domains of tensor contractions and stencils. Tensor contractions are a class of computation used extensively in computational chemistry which are essentially higher-dimension matrix multiplication. Vector code synthesis for tensor contractions has been explored, utilizing a novel in-register transpose for efficiently accessing tensors with SIMD vectors along any dimension. Aggressive use of unroll-and-jam and loop permutation optimizations is able to increase register reuse and amortize the cost of expensive vector operations.

Stencils are a critical part of many scientific codes including PDE solvers and image processing. This work presents a framework for reordering operations in regular loop computations, such as stencils, by utilizing the associativity and commutativity of operations. The freedom to reorder computations involving associative operators has been widely recognized and exploited in designing parallel algorithms and to a more limited extent in optimizing compilers. The reordering of operations enables substantially improved register reuse through reduced register pressure, resulting in up to $4 \times$ performance improvement for existing benchmarks. The multidimensional
retiming formalism characterizes the space of valid implementations in conjunction with other transformations from the vector code synthesis used for tensor contractions.

Given the transformations used to optimize high performance software, there are many parameters which create a space of possible variants from which the best, or nearly best, candidate must be found. In addition to auto-tuning over the space of synthesized vector codes, two models have been developed for selecting a high performance variant at compile time. The first model is based on estimating the total number of SIMD instructions executed and demonstrated and for a set of benchmark kernels achieved a mean performance of 73% of the best observed performance across multiple architectures and compilers. The second model used features extracted from the assembly code generated by the compiler which were then processed by machine learning algorithms. By considering a linear combination of the most successful models, a mean performance of 89% of the best observed was achieved in the same tests as the first model.

This research has shown that integration of tile code generation techniques with existing automatic tiling and parallelizing algorithms based on the polyhedral model can produce significant performance improvements on modern architectures such as Intel’s Many Integrated Core. Additionally, these techniques have been applied in production software to automatically produce high performance code on multiple architectures for the dominant computational kernel in MADNESS.
To my parents for their enduring support.
ACKNOWLEDGMENTS

Looking back to the beginning of graduate school, it seems the route in front of me should have been obvious, but I would never have thought to pursue high performance computing. However, with incredible luck I met Saday during my first quarter, and at that time he understood what I wanted even though it would take me years to fully understand it myself. Over the last five years Saday has provided me with amazing opportunities, patient mentorship, and unrelenting motivation which helped make me the person I am today and will always be remembered.

I owe a great deal of gratitude to my mentors who guided my research and studies. This dissertation would not be possible without the support of my committee members, Nasko and Srini. Louis-Noël Pouchet taught me innumerable lessons about both my research and being a researcher. My summer internship mentors Jeff Hammond, Robert Harrison, and Rajesh Bordawekar broadened my understanding of the field and gave insight into the options that await me after school.

From the lazy afternoons talking about anything but work, to the overnight marathons for paper deadlines, my labmates have been there for me and experienced every moment. Tom Henretty and Justin Holewinski as the senior students of the lab became my friends and served as guides to the world of graduate school. Naznin Fauzia, Martin Kong, Pai-Wei Lai, Mahesh Ravishankar, Naser Sedaghati, and Sanjukta Tavargeri all exhibited great patience as I constantly struck up conversations
with them while procrastinating and were still willing to help with understanding new works and meeting deadlines. As I leave, I know Venmugil Elango, Samyam Rajbhandari, and all the other students of HPCRL will continue providing a great environment for new graduate students.

These last five years would have been impossible without the immense support of my family and friends. My parents, Pete and Elena Stock, encouraged and assisted me every step of the way without hesitation. My siblings, Kirsi, Keith, Kayla, and Anna constantly reminded me of all the wonderful things in life and inspired me with the pursuit of their own dreams. Along the way, my friends have always been there to listen to everything I’ve needed to say and tell me what I’ve needed to hear.

Finally, I would like add a special thanks to Jake Tawney who took a chance on me ten years ago and taught a fantastic year of computer science which set me on the path I’ve taken.

Kevin Stock
Columbus, Ohio
August 19, 2014
VITA

January 26, 1987 ............................. Born: Columbus, OH, USA

Chemical Abstract Services

June 2008 — August 2008 ............... Scholastic Honors Intern
Federal Bureau of Investigation

June 2009 ................................. B.S.C.S.E.
The Ohio State University

January 2010 — Present ................. Graduate Research Associate
The Ohio State University

July 2010 — September 2010 .......... Intern
Argonne National Lab

June 2011 — September 2011 .......... Intern
Oak Ridge National Lab

June 2012 — August 2012 .......... Intern
Argonne National Lab

May 2013 — August 2013 .............. Research Intern
IBM

PUBLICATIONS

Research Publications

A Communication-Optimal Framework for Contracting Distributed Tensors
To Appear In Supercomputing (SC), November 2014
**CAST: Contraction Algorithm for Symmetric Tensors**
To Appear In *International Conference on Parallel Processing (ICPP)*, September 2014

K. Stock, M. Kong, T. Grosser, L.N. Pouchet, F. Rastello, J. Ramanujam and P. Sadayappan
**A Framework for Enhancing Data Reuse via Associative Reordering**
In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2014

**A Framework for Load Balancing of Tensor Contraction Expressions via Dynamic Task Partitioning**
In *Supercomputing (SC)*, November 2013

M. Kong, R. Veras, K. Stock, L.N. Pouchet, F. Franchetti and P. Sadayappan
**When Polyhedral Transformations Meet SIMD Code Generation**
In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, June 2013

K. Stock, L.N. Pouchet and P. Sadayappan
**Automatic Transformations for Effective Parallel Execution on Intel Many Integrated Core**
In *TACC-Intel Highly Parallel Computing Symposium*, April 2012

K. Stock, L.N. Pouchet and P. Sadayappan
**Using Machine Learning to Improve Automatic Vectorization**
In *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 8 Issue 4, January 2012

K. Stock, T. Henretty, I. Murugandi, R. Harrison and P. Sadayappan
**Model-Driven SIMD Code Generation for a Multi-Resolution Tensor Kernel**
In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011

T. Henretty, K. Stock, L.N. Pouchet, F. Franchetti, J. Ramanujam and P. Sadayappan
**Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures**
In *ETAPS International Conference on Compiler Construction (CC)*, March 2011
FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter/Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xvi</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xviii</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xix</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Tensor Contractions</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Multi-resolution Tensor Kernel</td>
<td>7</td>
</tr>
<tr>
<td>2.1.2 Coupled Cluster Single Double</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Stencils</td>
<td>11</td>
</tr>
<tr>
<td>2.3 SIMD CPU Architectures</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Machine Learning in Compilers</td>
<td>15</td>
</tr>
</tbody>
</table>
3. Related Work

3.1 Linear Algebra Tuning
3.2 Stencil Tuning
3.3 Compiler Techniques
  3.3.1 Automatic SIMD and MIMD Parallelism
  3.3.2 Tiling
  3.3.3 Register Reuse
3.4 Performance Modeling

4. Vector Code Synthesis

4.1 Vectorization Algorithm
4.2 Optimization Space
4.3 Examples
  4.3.1 Matrix Multiplication
  4.3.2 Tensor Contraction: ijk-lmn
  4.3.3 Jacobi 2D 5-Point Stencil
4.4 Characterization of the Optimization Space
  4.4.1 Tensor Contractions
  4.4.2 Stencils
4.5 Tile Size Selection
4.6 MADNESS: The mtxm Operator

5. Multidimensional Retiming

5.1 Motivating Example
  5.1.1 Gather/Scatter Retiming
5.2 Retiming Framework Overview
5.3 Program Representation
5.4 Formalization of Multidimensional Retiming
  5.4.1 Legality of Retiming
  5.4.2 Gather/Scatter as Retiming

6. Variant Selection

6.1 Register Reuse Modeling with Multidimensional Retiming
  6.1.1 Data Reuse Across Retimed Iterations
  6.1.2 Estimating Register Pressure and Data Movements
  6.1.3 Modeling of Complementary Optimizations
  6.1.4 Pruning the Space of Variants
  6.1.5 Complete Stencil Optimization Framework
6.2 Tensor Contraction Static Cost Model ........................................ 72
  6.2.1 Detailed Example .......................................................... 75
6.3 Machine Learning Models ......................................................... 80
  6.3.1 Assembly Features .......................................................... 82
  6.3.2 Model Training ............................................................... 86
  6.3.3 Analysis of the Generated Models ...................................... 88
  6.3.4 The Weighted-Rank Model ................................................ 94

7. Experimental Results ............................................................... 99
  7.1 A Multi-Resolution Tensor Kernel .......................................... 99
    7.1.1 Model Verification ....................................................... 100
    7.1.2 Performance with Multi-Resolution Kernel ......................... 100
  7.2 MADNESS: The mtxm Operator ............................................. 103
  7.3 CCSD Tensor Contractions .................................................. 104
    7.3.1 Performance Evaluation .............................................. 105
    7.3.2 Performance Evaluation Without Explicit Vectorization ........ 110
  7.4 Tiling for Larger Tensors ................................................. 112
    7.4.1 Tensor Contractions Tested ......................................... 112
    7.4.2 Performance Evaluation .............................................. 113
  7.5 Stencil Vectorization ....................................................... 116
  7.6 Stencil Retiming ............................................................. 119
    7.6.1 Performance Results ................................................... 123
    7.6.2 Impact of Transformations ........................................... 127

8. Ongoing and Future Research .................................................. 130
  8.1 Kernel Fusion ................................................................. 130
    8.1.1 Stencils ................................................................. 130
    8.1.2 Multidimensional Transform Operator ................................ 132
  8.2 Symmetric Tensors ........................................................... 133
  8.3 Tile Size Selection .......................................................... 135
  8.4 Variant Selection ............................................................ 136

9. Conclusion ......................................................................... 137

Appendices:

A. Code Samples ................................................................. 138
  A.1 Symmetric Matrix Multiplication ...................................... 138
  A.2 In-Register Transpose .................................................... 138
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Roofline model with performance of streaming benchmarks</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Vectorization using register level transpose</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Vectorization of matrix multiplication along $j$</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Vectorization of matrix multiplication along $i$</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Performance distribution of assorted tensor contractions</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Performance distribution across configurations</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of performance from ICC and GCC</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Performance distribution of stencils with ICC</td>
<td>38</td>
</tr>
<tr>
<td>4.8</td>
<td>Performance distribution of stencils with GCC</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Base 2D stencils</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Base and optimized 2D stencils</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>Coefficient of determination for variants of CCSD</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Prediction quality for each feature set</td>
<td>91</td>
</tr>
<tr>
<td>6.3</td>
<td>Prediction quality for $fs3$</td>
<td>93</td>
</tr>
<tr>
<td>6.4</td>
<td>Efficiency of KStar vs IBk on CCSD</td>
<td>95</td>
</tr>
</tbody>
</table>
6.5 Nehalem rank error comparison of KStar vs IBk ................. 96
6.6 Sandybridge rank error comparison of KStar vs IBk ............ 97
7.1 Double precision multi-resolution kernel ..................... 101
7.2 Single precision multi-resolution kernel ...................... 102
7.3 Relative speedup over MKL BLAS implementation .......... 103
7.4 Performance of 3d transform implemented with mtxm ....... 104
7.5 Performance of synthetic stencil benchmarks .............. 124
7.6 Performance of the Stencil Micro-Benchmarks ............. 125
7.7 Synthetic stencil benchmarks load/store counts .......... 127
7.8 Synthetic stencil benchmarks load/stores relative to FLOPS 128
7.9 Stencil performance improvement from stages of optimization 129
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>BLAS implementation of the doitgen kernel</td>
</tr>
<tr>
<td>2.2</td>
<td>SSE vector intrinsics</td>
</tr>
<tr>
<td>5.1</td>
<td>Expected cost of retiming for 2D stencils</td>
</tr>
<tr>
<td>6.1</td>
<td>Search space statistics for different pruning factors</td>
</tr>
<tr>
<td>6.2</td>
<td>Cost and implementation of abstract operations</td>
</tr>
<tr>
<td>6.3</td>
<td>Vector operations based on vectorized along dimensions</td>
</tr>
<tr>
<td>6.4</td>
<td>Average efficiency of individual models from leave-one-out analysis</td>
</tr>
<tr>
<td>7.1</td>
<td>Example loop properties for the multi-resolution kernel</td>
</tr>
<tr>
<td>7.2</td>
<td>Efficiency of the ML models on the CCSD benchmarks (SAFI)</td>
</tr>
<tr>
<td>7.3</td>
<td>Average efficiency of the ML models on the CCSD benchmarks</td>
</tr>
<tr>
<td>7.4</td>
<td>Performance of CCSD benchmarks with WeightedRank</td>
</tr>
<tr>
<td>7.5</td>
<td>Performance of WeightedRank without explicit vectorization</td>
</tr>
<tr>
<td>7.6</td>
<td>KNF performance relative to ICC for the standard dataset</td>
</tr>
<tr>
<td>7.7</td>
<td>KNF performance relative to ICC for the large dataset</td>
</tr>
<tr>
<td>7.8</td>
<td>Average efficiency of the ML models on the stencil benchmarks</td>
</tr>
</tbody>
</table>
7.9 Performance of the stencil benchmarks with WeightedRank . . . . . . 118

7.10 Practical applications of the stencil benchmarks . . . . . . . . . . . 121
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 vectorizeInstructions: Determine vector memory operation types</td>
<td>23</td>
</tr>
<tr>
<td>6.1 exploreSpace: Generate search space with retiming</td>
<td>68</td>
</tr>
<tr>
<td>6.2 optimizeProgram: End-to-end retiming and tuning</td>
<td>71</td>
</tr>
<tr>
<td>6.3 costEvaluation: Compute cost of tensor contraction variant</td>
<td>73</td>
</tr>
<tr>
<td>6.4 vectorizeTC: Tensor contraction optimization overview</td>
<td>74</td>
</tr>
</tbody>
</table>
## LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Example tensor contraction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Jacobi 2D 5-point stencil</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Triad benchmark and high arithmetic intensity variant</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Vectorization of array addition</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Example of generated code for $ijkl-imkn-jnlm$</td>
<td>33</td>
</tr>
<tr>
<td>5.1 2D stencil prototype implementation</td>
<td>43</td>
</tr>
<tr>
<td>5.2 2D stencil prototype implementation with retiming</td>
<td>47</td>
</tr>
<tr>
<td>5.3 Jacobi 1D using a weight array $W$</td>
<td>51</td>
</tr>
<tr>
<td>5.4 Jacobi 1D after statement splitting</td>
<td>51</td>
</tr>
<tr>
<td>5.5 Jacobi 1D after retiming (all-scatter)</td>
<td>52</td>
</tr>
<tr>
<td>5.6 Reduction code</td>
<td>58</td>
</tr>
<tr>
<td>6.1 Matrix-matrix multiplication</td>
<td>75</td>
</tr>
<tr>
<td>6.2 J vectorization, IKJ loop nesting</td>
<td>76</td>
</tr>
<tr>
<td>6.3 J vectorization, KJI loop nesting</td>
<td>78</td>
</tr>
<tr>
<td>6.4 I vectorization, JIK loop nesting</td>
<td>79</td>
</tr>
<tr>
<td>6.5 Example of x86 assembly code</td>
<td>83</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Developing high-performance scientific software for current and emerging systems is a tremendous challenge because of the need to optimize for high degrees of parallelism at multiple levels (MIMD and SIMD) and controlling data access costs across deep memory hierarchies. Achieving performance portability across systems is extremely difficult with currently available programming environments that do not abstract or automate the expression and management of concurrency. Further, existing tools do little to assist in managing the rapidly increasing complexity of both the computers and modern, multi-physics applications. The above issues lead to concern about the cost of developing high-end applications and the useful lifetime of that software.

This work focuses on the optimization of two classes of compute-intensive kernels. The first is tensor contractions which arise commonly in quantum chemistry codes, specifically ab initio methods for electronic structure calculations such as the coupled cluster and configuration interaction models [44, 24, 9]. Tensor contractions are essentially generalized higher dimensional matrix products where the tensors may be more than two dimensional and the contraction may be performed on multiple dimensions. The second set of kernels are stencils, a key computational pattern arising in
numerous application domains where weighted sums of values at a set of neighboring points are computed over a regular multidimensional grid. They are at the core of several large-scale scientific codes, such as those using Lattice Boltzmann methods (e.g., fluid flow simulation) [97], Finite-Difference Time Domain methods (e.g., seismic wave propagations, electromagnetic radiations) [64], image processing (e.g., edge detection) [85], and others.

One approach to generate high performance code for tensor contraction kernels is to utilize vendor optimized BLAS routines, but performance is suboptimal in many cases because at least one of the dimensions may be very small, and vendor BLAS routines are optimized for large matrices. Additionally, many computations tensor contractions cannot be performed directly, the require transposition of the data. In some cases stencils written in a naïve format can be effectively compiled by existing compilers, but for the notable case of higher-order stencils this is no longer true, and it shown to be caused by poor register reuse in existing compilers. Additionally, while most production compilers, e.g., Intel’s ICC, GNU GCC, PGI’s pgcc, IBM’s XL/C, perform automatic vectorization which outperforms scalar code, the achieved performance nevertheless is often far below the peak performance of the processor, even when the data fits within L1 cache and no cache misses are incurred. Thus in both cases it is critical that better SIMD vectorization algorithms are developed to maximize performance in many applications.

This work attempts to address the increasing availability of many CPU cores, deep memory hierarchies, and increasingly wide SIMD vector instructions. This is done in two parts: first finding the appropriate set of transformations, i.e., the vector code synthesis of Chapter 4 and the multidimensional retiming of Chapter 5, and
then applying cost models to the space of generate variants to determine the optimal parameters for the transformations as described in Chapter 6.

As manually tuning every program to take full advantage of these resources is exceptionally tedious a source-to-source compiler is proposed, capable of high-level transformations to automatically improve data locality, parallelize across cores, and vectorize with aggressive optimizations utilizing SIMD vector intrinsics to maximize their performance. The proposed code synthesizer is able to generate many variants for a single kernel, which introduces the problem of choosing the best among a huge number of possible transformations. Production compilers therefore typically use simple heuristics to guide loop transformations and back-end code generation to avoid the cost of searching the space of variants they are capable of producing. Additionally a framework implementing multidimensional retiming is presented to improve the performance of high-order stencils. A specific subset of retimings is shown to be quite effective at significantly reducing register pressure and in some cases reducing total accesses to main memory and floating point operations executed.

The experimental evaluation in Chapter 7 focuses on multiple specific problems being optimized for multiple platforms utilizing the SSE, AVX, and MIC vector instruction set architectures. First, the use of short-vector SIMD parallelism to optimize a multi-resolution kernel at the core of MADNESS [34, 45, 46, 68, 102], an advanced quantum chemistry application, is addressed. Additionally, a set of tensor contractions required in the implementation of high accuracy quantum chemistry models such as the coupled cluster method [24] are considered for evaluation of both cache sized tiles and as integrated into larger problems. For stencils a number of synthetic
benchmarks representative of real codes are tested, as well as a subset of the *Stencil Micro-Benchmarks* [29].

With the multi-resolution tensor kernel it is shown to be feasible to synthesize higher performance SSE-intrinsics-based code by developing a static model-driven compiler approach that evaluates different permutations and degrees of loop unrolling for the loops. However, the complex execution pipelines with multiple functional units in modern processors makes it extremely challenging for this analytical model to predict the execution time for an arbitrary tensor contraction. A machine learning model is built to predict the performance of the generated assembly code for the various possible transformations at compile time. After training the machine learning model using of a number of generated variants from a training set of synthetic kernels, the model is used to select optimally vectorized code for a number of tensor contraction kernels from the CCSD (Couple Cluster Singles and Doubles) method. It is shown that the code selected using the machine learning model is significantly better than that generated through auto-vectorization by GCC and ICC. Additionally, the effectiveness of the machine learning model to successfully rank-order the performance of a number of vectorized variants for stencil computations is demonstrated to show the wider applicability of the method.

It is clear from this work that careful selection transformations can be used to greatly improve the generation of SIMD vector code in multiple problem domains. In addition to the application of general compiler transformations, the development of specialized compiler transformations within specific problem domains is crucial to reaching maximum performance. Finally, heuristics used to guide parameter selection
for these transformations can be improved using machine learning models at compile time.

This work is structured as follows: Chapter 2 reviews the necessary background material, defining tensor contractions and SIMD vector architectures and their programming models. Chapter 3 presents related work in the areas of vectorization and performance modeling for compilers. Chapter 4 develops the algorithm used in this work to generate optimized variants of a kernel. Chapter 5 described the multidimensional retiming framework to further optimize stencils in conjunction with the vector code synthesis algorithm. Chapter 6 presents two schemes for selecting a variant from the space generated by the code synthesis algorithm. Chapter 7 demonstrates the effectiveness of the code synthesis and variant selection. Chapter 8 discusses possible future directions for this research. Chapter 9 concludes this work.
2.1 Tensor Constructions

A tensor is a multidimensional array of numerical values, effectively a generalization of a two dimensional matrix. Tensor contractions correspond to generalizations of matrix-matrix multiplication, in which an output tensor is formed as a multidimensional array of dot products of values in the input tensors corresponding to the indices of the output tensor. Indices which do not occur in the output tensor are the contraction indices. Equation 2.1 shows an example tensor contraction which could be implemented as in C as shown in Listing 2.1.

\[ C_{ijk} = \sum_i A_{ijk}B_{il} \]  \hfill (2.1)

In general, a tensor contraction involves \(d\)-nested loops that update a \(k\)-dimensional output tensor (\(k < d\)). The example is a four dimensional tensor contraction with a three dimensional output tensor. Each index of the nested loop appears exactly twice as an array index, each appearance being in a different tensor. In the example, \(i\) appears as an index for arrays \(A\) and \(B\) which makes it a contracted index, \(j\) and \(k\) index into arrays \(A\) and \(C\), while \(l\) indexes \(B\) and \(C\). The loop nest of any tensor
contraction is fully permutable, e.g., all 24 possible loop orderings for this example are valid, and because of this tensor contractions are also fully tileable computations [100]. As a shorthand, tensor contractions are represented in this work according to their array indices in $C, A,$ and $B$. The example tensor contraction would be represented as $ljk$-ijk-il.

2.1.1 Multi-resolution Tensor Kernel

MADNESS (Multi-resolution Adaptive Numerical Environment for Scientific Simulation) [34, 45, 46, 68, 102] is a general framework for scientific simulation that provides new capabilities for rapid computation with guaranteed precision enabled by novel mathematical tools for multi-resolution analysis in multiwavelet bases, and efficient computation in many-dimensions through the use of separated representations. Although initially motivated by applications to chemical problems, MADNESS is a general purpose framework with existing applications also in nuclear structure, material science, and atomic and molecular physics, with emerging applications in fusion, climate science, accelerator dynamics, and image processing. The combination of multi-resolution and separation representations enable fast application of many physically important integral operators, which enables direct solution of integral equations.
rather than poorly conditioned differential equations that involve operators of high-
norm with associated loss of precision.

MADNESS uses a discontinuous spectral element basis with adaptive mesh refinement. In each box of the adaptively refined mesh, the basis comprises a tensor product of the first $p$ Legendre polynomials. Relatively high-order elements are employed, typically starting at order $p = 6$ and proceeding to $p = 10$ for time-independent problems and even up to $p = 30$ for select time-dependent applications. The most common kernel is a variant on the operation

$$R_{ijk} = \sum_{i',j',k'} S_{i'j'k'} X_{i'i} Y_{j'j} Z_{k'k}$$

(2.2)

where the range of indices $i, i', j, j', k, k'$, is the order of the Legendre polynomial used, with the choice representing a trade-off between accuracy and computational efficiency. This basis leads to most kernels comprising operations on small, dense multidimension arrays, leading to good cache locality. However, it is very challenging to achieve high per-core performance from short-SIMD vector instructions.

Very often, the basis transformation matrices $X, Y, Z$ can be expressed using a separated representation, whereby each of $X, Y$ and $Z$ can be factored into the product of a pair of low-rank matrices

$$X_{i'i} = \sum_{l} X_{il}^{L} X_{li'}^{R}$$

$$Y_{j'j} = \sum_{m} Y_{jm}^{L} Y_{mj'}^{R}$$

$$Z_{k'k} = \sum_{n} Z_{kn}^{L} Z_{nk'}^{R}$$

(2.3)
where the matrix dimensions along $l$, $m$, $n$ are much smaller than the order of the Legendre polynomial $p$

\[ R_{ijk} = \sum_{i'j'k'lmn} S_{i'j'k'} X_{i'i}^L X_{l'i}^R Y_{j'm}^L Y_{mj}^R Z_{k'n}^L Z_{nk}^R \]  \hspace{1cm} (2.4)

The seven-term sum of products expression for the low-rank version of the tensor computation is most efficiently computed as a sequence of six tensor contractions as

\[
\begin{align*}
A_{ijk} &= \sum_i S_{ijk} \cdot X_{il}^L \\
B_{lmk} &= \sum_j A_{ijk} \cdot Y_{jm}^L \\
C_{lmn} &= \sum_k B_{lmk} \cdot Z_{kn}^L \\
D_{lmk'} &= \sum_n C_{lmn} \cdot Z_{nk'}^R \\
E_{lj'k'} &= \sum_m D_{lmk'} \cdot Y_{mj'}^R \\
R_{i'j'k'} &= \sum_l E_{lj'k'} \cdot X_{l'i}^R 
\end{align*}
\]  \hspace{1cm} (2.5)

It can be shown that the number of floating point operations is lower with this approach using the low rank representation if that rank is lower than \( \frac{2}{3}p \), which is very often the case. However, as shown later in the section on experimental results, performance of an naïve implementation for the above sequence is very low, even when using sophisticated vectorizing compilers like Intel’s icc.

Currently, a significant performance limitation for MADNESS is the poor quality of compiler generated code for the core computational kernels. Equally significant, the level of effort required to manually implement assembly-coded optimized kernels is excessive even for a single target family, e.g., x86 using SSSE3. A performance portable implementation of the kernels, which are all expressible as tensor expressions,
would greatly enhance both the productivity of the application developers and the performance enjoyed by the end users.

The seven-term sum of products expression can also be represented as a sequence of six BLAS class to \texttt{xgemm} using the arguments shown in Table 2.1. However, because of the low dimension sizes of the tensors, performance with vendor optimized BLAS is also low, especially when the low-rank dimension is very small. The BLAS implementation differs from the version optimized in that it is uses a different order of the $X^R, Y^R, Z^R$ arrays which combined with rotating indices shown in Table 2.1 enable the implementation with only six calls.

<table>
<thead>
<tr>
<th>TransA</th>
<th>TransB</th>
<th>M</th>
<th>N</th>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>$p \cdot p$</td>
<td>$l$</td>
<td>$p$</td>
<td>$S$</td>
<td>$X^L$</td>
<td>$A$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$p \cdot l$</td>
<td>$l$</td>
<td>$p$</td>
<td>$A$</td>
<td>$Y^L$</td>
<td>$B$</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>$l \cdot l$</td>
<td>$l$</td>
<td>$p$</td>
<td>$B$</td>
<td>$Z^L$</td>
<td>$C$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$l \cdot l$</td>
<td>$p$</td>
<td>$l$</td>
<td>$C$</td>
<td>$X^R$</td>
<td>$D$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$l \cdot p$</td>
<td>$p$</td>
<td>$l$</td>
<td>$D$</td>
<td>$Y^R$</td>
<td>$E$</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>$p \cdot p$</td>
<td>$p$</td>
<td>$l$</td>
<td>$E$</td>
<td>$Z^R$</td>
<td>$R$</td>
</tr>
</tbody>
</table>

Table 2.1: BLAS implementation of the doitgen kernel

2.1.2 Coupled Cluster Single Double

Also considered are a set of tensor contractions from CCSD (Coupled Cluster Single Double), a computational method for \textit{ab initio} electronic structure calculations [24, 50]. CCSD is composed of tensor contractions which cannot be computed using
BLAS libraries due to complex ordering of indices, thus implementations must commonly rely on manual tuning to get maximum performance. However, the overwhelming number of tensor contractions makes manual performance tuning very difficult, and thus it is desirable for a compiler to automatically vectorize and optimize the code. As such, CCSD provides a representative set of tensor contractions which occur in large-scale scientific computing and require high performance implementations. 19 unique kernels were extracted from the many tensor contractions required for CCSD. These are representative of the kinds of tensor contractions that commonly occur in computational chemistry.

2.2 Stencils

Listing 2.2 shows an example of a stencil, Jacobi 2D 5-point, which takes the average of five neighboring points to create the new value. In general a stencil is a computation of a point in an array based on the value of neighbors. However, in this work only convolution stencils, a common subclass of stencils in which each point is computed as the sum of weighted neighbors.

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    B[i][j] = 0.2 *
        (A[i-1][j] +
         A[i+1][j]);
```

Listing 2.2: Jacobi 2D 5-point stencil
This work primarily focuses on performance issues with high-order stencils. High-order stencils involve weighted averages over multiple neighboring points along each dimension. Overture [74] is a toolkit for solving partial differential equations over complex geometry, and uses high-order approximations for increased accuracy, leading to high-order stencil computations. Similarly, the Chombo library [21] uses high-order stencil operations in discretizing high-order derivatives. Additionally, low-order stencils can be \textit{unrolled} along a time dimension to create high-order stencils.

Unlike simple low-order stencils, high-order stencils feature a very high arithmetic intensity, i.e., the ratio of arithmetic operations to the number of distinct data elements accessed. It is generally the case that computations with a very low arithmetic intensity are memory bandwidth bound and achieve low compute performance, while codes with high arithmetic intensity and potential for parallel execution achieve high performance. Figure 2.1 illustrates this using a roofline [99] plot showing the peak performance possible for a problem with a given arithmetic intensity. The first two points show the simple streaming benchmarks in Listing 2.3 behave as expected, scaling in performance as the arithmetic intensity increases, and initially the stencils

\textbf{Listing 2.3:} Triad benchmark and high arithmetic intensity variant

// Triad
for (i=0; i<N; i++)
    C[i] = A[i]*X + B[i];

// High arithmetic intensity triad
for (i=0; i<N; i++)
            A[i]*X + B[i]*Y + Z;
Figure 2.1: Roofline model with performance of streaming benchmarks

follow the same pattern. However, above $5 \times 5$ the stencils no longer follow the same trend and reach maximum performance, even though there is substantial room for improvement.

The stencil benchmarks used in this work are a combination of synthetic benchmarks and a subset of the Stencil Micro-Benchmarks [29].

2.3 SIMD CPU Architectures

Vector Instruction Set Architectures enable the exploitation of data parallelism by using instructions which are applied to multiple data elements at once. This is achieved with short-vector registers that contain multiple elements, typically 2 or 4 although up to 16 on Intel Xeon Phi with single precision, and instructions that can
operate on all the elements of registers in parallel. However, there is a restriction that vectors can only operate on contiguous memory. Vector instruction sets remedy this restriction by including instructions to move data around, e.g., shuffles, or to insert scalar elements into a specific portion of a SIMD vector, enabling more complicated code than the example above to be vectorized at the cost of extra instructions.

Listing 2.4 demonstrates vectorization of adding two arrays with SIMD vectors of length four, where the notation $C[i:i+3]$ represents a vector of four elements: $<C[i], C[i+1], C[i+2], C[i+3]>$. This only requires $\frac{1}{4}$ the operations used by a non-vectorized version, clearly showing how the use of vector instructions can provide a large speedup by increasing the floating point operations per instruction.

In place of directly generating assembly instructions, the presented code synthesizer produces C code with SIMD vector intrinsics, which act as an interface to the operations for C programmers. Intrinsics appear as functions to the programmer but the compiler is able to translate them directly to the relevant assembly instruction. Using intrinsics leaves the tasks such as register allocation to the compiler, enabling the code synthesizer to focus strictly on vectorization and high level optimizations for performance. As an example of the available intrinsics, the subset of single precision SSE intrinsics used by the synthesizer are described in Table 2.2. There exist similar intrinsics for double precision operations and different architectures including AVX, Intel Many Integrated Core, and BlueGene.
<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
<th>Inst. Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_loadu_ps</td>
<td>Load four adjacent floats into a vector register</td>
<td>1</td>
</tr>
<tr>
<td>_mm_storeu_ps</td>
<td>Store a vector register of four floats to a specified location in memory</td>
<td>1</td>
</tr>
<tr>
<td>_mm_load_ps1</td>
<td>Replicate a single float from memory into all four parts of a vector register</td>
<td>2</td>
</tr>
<tr>
<td>_MM_TRANSPOSE4_PS</td>
<td>Transpose the floats in four vector registers</td>
<td>8</td>
</tr>
<tr>
<td>_mm_hadd_ps</td>
<td>Horizontal add; two applications to the same vector register will set all floats in the register to the sum of the original four values</td>
<td>1</td>
</tr>
<tr>
<td>_mm_store_ss</td>
<td>Store the first float in a vector register to memory</td>
<td>1</td>
</tr>
<tr>
<td>_mm_add_ps</td>
<td>Pointwise addition of two vector registers</td>
<td>1</td>
</tr>
<tr>
<td>_mm_mul_ps</td>
<td>Pointwise multiplication of two vector registers</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 2.2:** SSE vector intrinsics

### 2.4 Machine Learning in Compilers

Machine learning models have garnered considerable interest in the compiler community. Machine learning approaches have been used in numerous situations, for instance to optimize compiler flag settings [2, 18], choose effective loop unrolling factors [70], and optimize tile sizes [104]. However, it does not appear there is any prior use of machine learning models to assist in optimizing vector code generation. Such a model is developed in this work.
CHAPTER 3

Related Work

3.1 Linear Algebra Tuning

A significant fraction of cycles on high-end supercomputers are spent in numerical linear algebra libraries. Several efforts have addressed automatic tuning of core linear algebra routines in scientific libraries and applications [10, 98, 96, 90]. These approaches use empirical search of parameters for the given problem on a specific target machine. Well developed software is available for many dense linear algebra routines, both sequential (LAPACK [4], BLAS [98]) and parallel (ScaLAPACK [11]). GotoBLAS [39] has demonstrated impressive performance results on several architectures by using hand-tuned data structures and kernel operations. Hardware vendors often supply optimized BLAS versions [54]. Library generators for synthesizing platform-adapted high-performance code for FFT [38] and linear transforms [83] have also been developed. However, optimized libraries for tensor computations are not available. Even when some tensor computations can be expressed in terms of highly tuned matrix multiplication operations, as shown in Chapter 7, performance is unsatisfactory for the small sized operations.
3.2 Stencil Tuning

In the context of stencil codes, Dursun et al. [31] describe hand-coded optimizations that compute contributions of data in registers instead of computing stencil elements one by one. Sedaghati et al. [89] propose hardware extensions to vector instructions to reduce the IO of stencil computations. Deitz et al. [29] and Datta [27] present techniques that exploit common subexpressions across different iterations in stencil codes. While this is orthogonal from the register reuse issues addressed in this paper, techniques such as Array Subexpression Elimination [29] are emulated by the combination of multidimensional retiming with the compiler’s common subexpression elimination. This is done automatically as a single plane’s contributions to multiple points are considered in a single iteration; If the same coefficients are used for multiple points CSE removes the redundant work. Cruz et al. [25] present a technique for improving register reuse for a specific class of stencils which accumulates from neighbors along the axes. Their transformation is in fact a singular retiming, unique from the gather/scatter retiming in Chapter 5, which can be represented by the general retiming framework. Previous work has shown that pattern-specific compilation strategies for stencils are needed to address a variety of stencil specific performance bottlenecks, including parallelization, communication, data reuse, etc. [49, 48, 52, 85, 27, 89]. However no previous work on compiler optimization has addressed higher-order stencils and the unique opportunity for data locality optimization from associativity and commutativity in a systematic way.
3.3 Compiler Techniques

The transformations considered in § 4 are a superset of the SIMD-related transformations considered by Hall et al. in their auto-tuning work [19, 20], where only unroll-and-jam and loop permutation pertaining to SIMD optimization are considered. Additional dedicated optimizations such as register transpose and intrinsics code generation, considered by the presented code synthesis algorithm, can provide up to $2 \times$ additional performance improvement over the compiler’s auto-vectorization.

In addition, in contrast to previous work on auto-tuning [19, 103, 92], this work presents two models which operate at compile time and do not require the execution of any variants on the machine. It is a purely model-driven approach.

Deciding the enabling or disabling of loop unrolling was done by Monsifrot et al. [70] using decision tree learning, and was one of the early efforts on using machine learning to tune a high-level transformation. Numerous other works considered the use of machine learning to drive the optimization process [59, 22, 23, 37, 42, 2]. None of these works considered the advanced vectorization techniques that are used in this paper. Cavazos et al. address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics [18]. That work was limited to the traditional optimization space of the PathScale compiler. Further, the program to be optimized by the compiler first had to be executed (without optimization) to determine the feature vector of performance counters that were then input to the trained machine learning model to predict the best optimization sequence.

Multidimensional retiming has been been studied in the context of exploiting parallelism by a number of authors [75, 76, 26, 17].
3.3.1 Automatic SIMD and MIMD Parallelism

Automatic vectorization has been the subject of extensive study in literature [55, 101, 93, 51]. Numerous previous works have focused on generating effective code dealing with hardware alignment and stride issues [32, 72, 36, 62, 63], outer-loop vectorization [73] and multi-platform auto-vectorization [71, 51]. The automatic vectorizer of GNU GCC implements many of these techniques [71, 72, 73] and thus represents the state-of-the-art. Since Intel’s ICC is a closed source compiler, it is more difficult to assess what is currently implemented. Nevertheless its very good performance compared to GCC for many codes suggests advanced techniques for automatic vectorization have been implemented.

There have been significant advances over the last few years in compiler transformations for automatic parallelization [13, 77, 7] on SMP systems using automatic insertion of OpenMP pragmas in source-to-source compilers.

A number of works have addressed the detection and automatic parallelization of scans and reductions [12, 105, 86, 41]. Commutativity analysis has been recognized as an approach to parallelizing computations [88, 3, 82, 61].

3.3.2 Tiling

While there exist simpler solutions to tiling perfectly nested loops such as tensor contractions [40], PTile [7] is applicable to any irregular affine loop nest, and has been shown to produce comparable or better performance than other polyhedral based tiling algorithms [13, 47, 57]. Substantial other work has been done on compiler optimizations for data locality [7, 47, 87].
Goto [39] showed that generating larger tile sizes to occupy L2 cache instead of L1 can be more efficient as one of the arrays remains only L2 resident and the cost to access it is amortized over the inner loop. Tuning of the tile sizes used in this work combined with the existing transformations could benefit performance in the same way.

### 3.3.3 Register Reuse

Callahan et al. [16] and Liu and co-workers [65, 66] present a technique for register allocation of subscripted variables. They argue that most compilers do not allocate array elements to registers because standard dataflow analysis make it difficult to analyze the definitions and uses of individual array elements. They then discuss an approach of replacing subscripted variables by scalars to effect reuse. The advantage of this approach is that all array variables are replaced by scalars that are then mapped to registers which can reused in successive iterations. They do not address the issue of the resulting increase in register pressure.

Rotating registers has been discussed as a technique implemented in hardware in [53, 28, 58]. Global value numbering [56, 15] can produce similar results as the rotating registers implemented in this work for some cases.

### 3.4 Performance Modeling

Trifunovic et al. proposed an analytical cost model for evaluating the impact of loop permutation and loop strip mining [93] on vectorization. While it is applicable to tensor contractions, in contrast to this work it does not take into account the interplay of subsequent compiler passes (e.g., vector code generation, instruction selection,
scheduling and register allocation) and does not consider critical optimizations for performance, such as unroll-and-jam and register transpose.

Traditional machine learning approaches to automatically select a loop transformation sequence require explicit modeling of the sequence in the learning problem [2, 30, 23].
CHAPTER 4

Vector Code Synthesis

To generate high performance code for an input program, e.g., Listings 2.1 and 2.2, a space of optimized variants is produced using parameterized transformations introduced in this chapter. This work primarily focuses on the generation high performance cache resident tile, which can either be used as-is for small problem sizes or integrated with automated tiling tools, e.g., PTile [7], to produce effective code for larger problem sizes. To ensure the transformations used in this work are legal, the programs considered in this work are such that: Code is written with non-aliasing arrays, not general pointers; Programs are affine [35] to enable interaction with polyhedral compiler tools; Arrays are accessed without indirection which can easily break vectorization; And all loops considered for transformation are either sync-free or reduction parallel. For legal input codes, the set of possible variants is determined by the three transformations of the vector synthesis algorithm:

1. The specific loop along which vectorization is performed.

2. The loop order determined by permutation.

3. The unroll-and-jam factor applied to each loop in the program.
4.1 Vectorization Algorithm

Vectorization is done along iterations of a loop in the kernel and is not limited to the innermost loop; outer loop vectorization is considered as well. The choice of the loop to be vectorized affects how memory is loaded to and stored from the SIMD registers. All loops that are either sync-free or reduction parallel are considered for vectorization, which in the case of tensor contractions and convolution stencils implies all the loops are considered, because each loop is either fully parallel or a reduction.

**Input**: instructions, vectorize, vectorLength

**Output**: instructions

```
begin
  vectorize.unroll ← vectorLength;
  for inst ∈ instructions s.t. typeOf(inst) ∈ {load, store} do
    unitStrideDim ← getUnitStrideDim(inst);
    if unitStrideDim = vectorize then
      inst.type ← UNIT;
    else if vectorize ∈ inst then
      inst.type ← TRANSPOSE;
      unitStrideDim.unroll ← vectorLength;
    else
      if isLoad(inst) then
        inst.type ← SPLAT;
      else
        inst.type ← REDUCTION;
    end
  end
end
```

**Algorithm 4.1**: vectorizeInstructions: Determine vector memory operation types

Algorithm 4.1 demonstrates how to determine the type vectorized memory operations used for each operation in a kernel. For the arrays in the statement in which the
fastest varying dimension (referred to as the unit-stride dimension, e.g., the rightmost index in C/C++ and the leftmost index in Fortran/MATLAB) matches the loop chosen for vectorization, groups of adjacent elements in memory can be directly loaded into or stored from vector registers after applying unroll-and-jam of the vectorized loop with the unroll factor being the vector length. This is implemented as an efficient single instruction on modern architectures. If the loop chosen for vectorization does not appear at all among an array’s indices, then a replication of that element into all components of a vector register will be required, i.e. for loads a splat or broadcast operation. For loads this is implemented as a scalar load followed by a shuffle to the other elements of the SIMD register, although some architectures can perform this in a single step. If the array is being used for a write then a reduction of partial values in the vector register will be done before storing a scalar value. Finally, an array can be accessed in non-unit stride by the vectorized loop. In general, in order to vectorize the statement, consecutive elements along the vectorized loop must be gathered into a SIMD register by using multiple scalar loads. However, due to the constraints on problems considered, data can be loaded with vector loads along an array’s unit-stride dimension, followed by a register level transpose to place consecutive elements along the vectorized loop into vector registers as demonstrated in Figure 4.1. Although not considered in this work, Intel’s AVX-512 instruction set which is currently only implemented on Xeon Phi has instructions to load a vector register with elements that are non-unit-stride at the cost of maintaining another vector register with the individual offsets of each element to be loaded.

Although the necessary type of data movement is only dependent on the choice of loop to vectorize, the total data movement cost is a function of the loop which is
Figure 4.1: Vectorization using register level transpose

vectorized as well as the loop permutation and unroll factors discussed in the next section.
4.2 Optimization Space

In this section the optimizations considered for vector code synthesis are introduced, and their interactions in the space of possible variants for a kernel are discussed.

**Vectorized loop:** The choice of vectorized loop determines how the SIMD registers interact with memory as shown in Algorithm 4.1. For some tensor contractions there is a loop which can be vectorized to avoid any expensive memory interaction entirely, namely the register transpose operation, but otherwise it is useful to consider all cases as they can have different types of interaction and stride lengths. The transpose operation is a sequence of `unpack` and `permute` instructions. Stencils in contrast generally have the same accessors in the same order for all arrays, which makes the choice of vectorized dimension trivial: it is same the unit-stride loop for all the arrays. Unless an obvious solution exists with the same unit-stride dimension on all vectorized arrays, this transformation contributes a multiplicative factor equal to the loop nest depth $d$ to the size of the space of vectorized code variants considered as each loop can be vectorized. In the vectorization of stencils the coefficients can be considered a zero dimensional array, and following the vectorization algorithm they must be splatted to a register.

**Loop permutation:** Within the scope of full tiles which fit into the first level of cache, the effect of loop permutation with regards to improving spatial locality of memory accesses is minimal. However, loop permutation can enable some load and store operations to be hoisted out of the inner most loop. This enables
a significant improvement between variants with different loop orders, when
the hoisted instructions are relatively expensive (e.g., the register transpose
operations) as their cost is amortized over the entire execution of the inner
loop. For a \(d\)-dimensional loop nest, \(d!\) distinct loop orders exist. In practice
since instructions can only be hoisted out of a few loops at most, only the
permutations of each choice of inner three loops need to be considered which
contributes a factor of \(O(d^3)\) instead.

In the case of stencils the kernel is usually only two or three loops deep which
greatly reduces the size of search space relative to tensor contractions. Fur-
thermore, the benefit of being able to hoist operations out of the innermost
loop is nonexistent for stencils since all memory accesses depend on all loops.
However, loop permutation is still considered for stencils to encourage better
reuse of vector registers which is discussed in more detail in Chapter 5.

**Unroll-and-Jam:** The final transformation considered is unroll-and-jam. Applica-
tion of this optimization to a specific loop allows memory accesses which are
not dependent on that loop to be loaded once and then reused multiple times.
Additionally for stencils a loaded value can be used in the computation of un-
rolled neighbors without having to issue another load. So long as the code size
remains small enough that registers do not spill, the reuse will be of a value in a
register, which can substantially increase the arithmetic intensity of the kernel.
All loops which are parallel, which is all of the loops in tensor contractions and
stencils, are eligible for unroll-and-jam, and each loop can have one of many
unroll factors, resulting in the potential search space for this transformation
alone being very large. The possible unroll factors of each loop are restricted
to the set of divisors of the loop’s tile size, which eliminates the need to generate inefficient boundary handling code within the optimized tile code. In the context of vectorization, unroll-and-jam is also used to enable the register transpose operation described above. However, unroll-and-jam is able to negatively affect performance through increased register pressure. For register reuse to be possible all values of interest must fit into registers, but due to the limited number of named registers (e.g. 16 on x86 platforms) available, large unroll values may cause values to spill. Additionally, code size rapidly increases with increased unrolling which can reduce the effectiveness of the instruction cache. In the case of the stencil retiming framework, the unroll-and-jam transformation is augmented with domain specific code motion to improve the locality of the prologue and epilogue loops.

In the case of stencils generated with the retiming framework the rotating registers transformation is also applied, but unlike the other transformations it is not parameterized or part of the space of variants. The rotating registers [58] transformation unrolls the inner loop by the stencil size and explicitly allocates registers to avoid register copies when transferring data to the next iteration of the inner loop. In combination with the decreased register pressure from the retimed code significant improvements to spilling are possible. This optimization can be done by global value numbering [91]; the LLVM compiler has been observed to automatically apply rotating registers when the inner-loop is a gathered dimension.
4.3 Examples

4.3.1 Matrix Multiplication

The best known example of a tensor contraction is matrix-matrix multiplication, as shown in Equation 4.1. For this example, consider the problem size $0 \leq i, j, k < 16$ which is similar to practical sizes for a cache-resident tile and a SIMD vector length of four.

$$C_{ij} = \sum_k A_{ik}B_{kj} \quad (4.1)$$

There are three choices for which loop to vectorize. The most common and efficient is SAXPY-style vectorization along the $j$ loop which is illustrated in Figure 4.2. In this scheme a single element of the $A$ array is multiplied by a vector from the $B$ array and added to a vector in the $C$ array.

Since the load of $A$ is not dependent on $j$, the load of $A$ will be a *splat*. The remaining memory accessing instructions depend on $j$, and in all of those cases $j$ is a unit-stride accessor, thus they will *unit-stride operations*.

By choosing a loop order with $j$ as the inner most loop the loads and splats of $A$ can be hoisted out of the inner most loop since $A$ does not depend on $j$. Other loop orders will hoist either the loads of $B$ or the loads and stores of $C$.

Each loop can be unroll-and-jammed by a factor of 1, 2, 4, 8, or 16, except for the $j$ loop which is already effectively unrolled by a factor of four to enable vectorization and thus can only be additionally unrolled by 1, 2, or 4. Unroll-and-jam along the $k$ enables reuse of each load/store pair from $C$. If $j$ is the inner most loop then unrolling
it will not improve the arithmetic intensity as it only effects loads of $A$ which have already been hoisted outside the $j$ loop.

Alternatively, matrix multiplication could be vectorized along the $i$ loop as illustrated in Figure 4.3. In this case a single element of the $B$ array is multiplied by a vector from the $A$ array and added to a vector in the $C$ array.

When vectorizing along the $i$ loop the accesses to both $A$ and $C$ will requires the register transpose because they are accessed in non-unit stride. This incurs a penalty, but the cost of accessing $A$ or $C$ can be amortized by hoisting them out.
\[ C[i][j] += A[i][k] \times B[k][j] \]

Figure 4.3: Vectorization of matrix multiplication along \( i \)

of the innermost loop with the correct loop order. This also changes the space of possible unroll-and-jams since all loops must be unrolled by a factor of four to enable the register transposes, and thus each loop only has three possible unroll factors.

Finally, it is possible to vectorize along the \( k \) loop, which is a dot-product style vectorization which will maintain partial sums in the elements of a vector register which are reduced to a single element before being stored in \( C \). Doing so would require register transposes of the loads from the \( B \) array.
For this example the total number of variants generated by code synthesis is

\[ 3! \cdot (5^2 \cdot 3 + 3^3 + 5 \cdot 3^2) = 882 \]  

(4.2)

4.3.2 Tensor Contraction: ijkln-imknl

As a concrete example of the code generated for a more complicated tensor contraction, an optimized variant of the code for the tensor contraction ijkln-imknl using double precision AVX vector instructions is shown in Listing 4.1. The choices for the three optimizations for this example are as follows:

**Vectorized loop:** In Listing 4.1 the tensor contraction is vectorized along the \(l\) loop. Therefore the loads and stores to the result array \(C\) can be done with standard vector operations. However, loads from \(A\) require a splat operation and loads from \(B\) involve register-level transposition of the \(m\) and \(l\) loops. It’s important to note that for this tensor contraction, every variant will require some memory operations done with the register transpose because every dimension is the non-unit-stride accessor of at least one array.

**Loop permutation:** The loop order in the example variant is \(mjlkni\), which allows access to \(B\) to be hoisted from the innermost loop. Doing so is important to amortize across the \(i\) loop the cost the loads of \(B\) which require the relatively expensive transpose. A better kernel would have \(k\) and \(i\) as the inner most loops to further reduce the cost of the register transpose.

**Unroll-and-jam:** The unrolling of each loop is indicated by its iterator increment. The loops that are unrolled promote reuse of certain registers. In this example
for (m=0; m<8; m+=8)
  for (j=0; j<4; j+=1)
    for (l=0; l<8; l+=8)
      for (k=0; k<4; k+=4)
        for (n=0; n<8; n+=1) {
          Btmp0 = _mm256_loadu_pd(&B[j*512+n*64+l*8+m]);
          /* Omitting 14 similar loads into Btmp1...Btmp14 */
          Btmp15 = _mm256_loadu_pd(&B[j*512+n*64+(l+7)*8+(m+4)]);

          _t0 = _mm256_unpacklo_pd(Btmp0,Btmp1);
          _t1 = _mm256_unpackhi_pd(Btmp0,Btmp1);
          _t2 = _mm256_unpacklo_pd(Btmp2,Btmp3);
          _t3 = _mm256_unpackhi_pd(Btmp2,Btmp3);
          Btmp0 = _mm256_permute2f128_pd(_t0, _t2, 0x20);
          Btmp2 = _mm256_permute2f128_pd(_t1, _t3, 0x20);
          Btmp1 = _mm256_permute2f128_pd(_t0, _t2, 0x31);
          Btmp3 = _mm256_permute2f128_pd(_t1, _t3, 0x31);
          /* Omitting 3 similar 4x4 transposes for Btmp4...Btmp15 */

          for (i=0; i<2; i+=1) {
            Atmp0 = _mm256_broadcast_sd(&A[i*256+m*32+k*8+n]);
            /* Omitting 30 similar splats into Atmp1...Atmp30 */
            Atmp31 = _mm256_broadcast_sd(&A[i*256+(m+7)*32+(k+3)*8+n]);

            Ctmp0 = _mm256_loadu_pd(&C[i*128+j*32+k*8+1]);
            /* Omitting 6 similar loads into Ctmp1...Ctmp6 */
            Ctmp7 = _mm256_loadu_pd(&C[i*128+j*32+(k+3)*8+(1+4)]);

            Ctmp0 = _mm256_add_pd(_mm256_mul_pd(Atmp0,Btmp0),Ctmp0);
            /* Omitting 62 similar multiply adds to Ctmp1...Ctmp6 */
            Ctmp7 = _mm256_add_pd(_mm256_mul_pd(Atmp31,Btmp15),Ctmp7);

            _mm256_storeu_pd(&C[i*128+j*32+k*8+1],Ctmp0);
            /* Omitting 6 similar stores from Ctmp1...Ctmp6 to C */
            _mm256_storeu_pd(&C[i*128+j*32+(k+3)*8+(1+4)],Ctmp7);
          }
        }
      }
    }
  }
}

Listing 4.1: Example of generated code for ijkln-mlkmjn
unroll-and-jam of the \( m \) loop allows for more reuse of the registers with values from \( C \). Because of the unroll-and-jam, the resulting code is almost 300 lines long. This is shown in Listing 4.1 with repetitive sections of similar code replaced by comments.

### 4.3.3 Jacobi 2D 5-Point Stencil

Listing 2.2 shows the input code for a common five point stencil. In this code \( j \) loop is the obvious choice for vectorization because it is the unit-stride accessor of both arrays. There are only two choices for loop permutation. Unroll-and-jam can be beneficial along the \( i \) loop to enable reuse of shared elements across iterations of \( i \), but as will be shown in the next chapter, rotating registers is able to provide better reuse with minimal unrolling. Unrolling of the \( j \) loop does not increase register reuse because \( j \) is vectorized, however for larger stencils it could provide reuse.

### 4.4 Characterization of the Optimization Space

This section presents an analysis of variants of tensor contractions and stencils in order to characterize the complexity of the optimization space generated by the described vectorization and optimization algorithm.

#### 4.4.1 Tensor Contractions

Figure 4.4 shows the sorted performance distribution of all the considered variants for four representative tensor contractions from the CCSD application, for a specific processor, instruction set, data type, and compiler.

Only a very small fraction of the optimized variants, fewer than 4% of generated, attain 80% or more of the search-space optimal performance. The code generated
by ICC’s auto-vectorization on the input code, i.e., without using the code synthesis algorithm to generate the vector intrinsics-based code, only achieves 1.90 GFlop/s for the \texttt{ijkl-imln-jnkm} tensor contraction, which is worse than 97\% of the variants in this search space, where the best variant performs at 39.08 GFlop/s.

For a single tensor contraction from the CCSD code, Figure 4.5 shows the sorted performance distribution for three configurations, Nehalem with SSE, Sandy Bridge with SSE, and Sandy Bridge with AVX, where all three configurations use floats and ICC. The distribution of performance for different configurations can be quite different. For the considered tensor contraction, ICC auto-vectorization of the input code on Sandy Bridge using AVX results in performance of 2.96 GFlop/s. Here 74\% of the search space is faster, with the space optimal variant achieving 38.57 GFlop/s.

The compiler has a critical impact on the relative performance of different vectorized variants. Figure 4.6 plots, for the same representative tensor contraction, the performance of all its vectorized variants when using ICC and GCC. The variants are
sorted according to their performance when compiled with ICC. There are significant and unpredictable performance variations between ICC and GCC for a given variant, as illustrated by the numerous spikes. The best performing vectorized variant with GCC (28.1 GFlop/s) performs poorly when compiled with ICC, achieving only 11.25 GFlop/s. The converse is also true, as shown by the relatively low performance with GCC in the far right of Figure 4.6.

The choice of compiler also affects the maximal performance achievable for a particular tensor contraction. For example, for the $ijkl$-$imjn$-$nlmk$ tensor contraction on Sandy Bridge using AVX and floats, ICC attains 33.16 GFlop/s, while GCC is 13% slower at 28.94 GFlop/s. However, ICC is not always more effective; There are instances where the best among the vectorized variants compiled with GCC is faster than the best of all variants compiled with ICC.

**Figure 4.5:** Performance distribution across configurations
4.4.2 Stencils

In contrast to the performance distribution of tensor contraction variants, the performance distribution of stencils shows a much narrower gap between the best and worst vectorized variants. This section does not deal with retimed stencils or application of the rotating registers transformation and stencils are only applied to cache resident tiles. This is illustrated in Figures 4.7 and 4.8, which plot the distribution of three representative stencil benchmarks, across three distinct configurations: Nehalem with SSE, Sandy Bridge with SSE, and Sandy Bridge with AVX.

Figure 4.7 plots the distribution when using the ICC compiler. For disofive and inoisetwo, the distribution is flat as all variants have about the same performance. Nevertheless there is a significant performance improvement over ICC’s auto-vectorization on the original code: the best variant for disofive is 1.8× faster than the auto-vectorized code with ICC, which achieves 4.8 GFlop/s.
While the search space still exhibits improvement over the original code, it is clear that for all benchmarks that have a flat distribution, even purely random selection among the variant will be effective, making the process of variant selection trivial.
however, this observation is not true for all stencil benchmarks, and in particular when using the GCC compiler. Figure 4.8 plots the performance of the same benchmarks and same hardware configuration, but using GCC. A much wider performance variation is seen, for all three benchmarks. A significant performance improvement over GCC’s auto-vectorization is also observed; Sandy Bridge using single-precision AVX averages $2.86 \times$ improvement across the benchmarks.

4.5 Tile Size Selection

To select tensor tile sizes to optimize in this work, a heuristic was developed to select a tile size from the space of possible tile sizes. Each dimension of the tile starts with a power of two. As long as the data is not L1 cache resident, it halves the size of the dimension which produces the greatest reduction in total data size. In the case of a tie, the longest dimension is selected. While this may not produce optimal tile sizes it ensures that the data will fit in L1 cache and that all dimensions of the tile can be unroll-and-jammed, which is crucial for achieving good vector performance.

In the case of the mtxm kernel tiling is not implemented because the problem size is small enough to fit in cache.

In stencils tested with the multidimensional retiming, tile sizes were chosen to enable multi-core parallelism but were not tuned any further. Specifically, for a four core machine the tile size was set to half of the problem size on the outer two dimensions and the full problem size on any remaining dimensions.
4.6 MADNESS: The mtxm Operator

MADNESS heavily relies on a single operator, mtxm, which performs matrix-transpose times matrix. Although the tensors in MADNESS are commonly of dimensionality greater than two, due to patterns in the contractions it is possible to perform index fusion and reduce the problems to matrix multiplication. While this operation could be performed by BLAS libraries, the problem size is generally quite small, the largest common problems require about 160,000 multiply-add operations, which is smaller than vendor provided BLAS libraries optimize for. To ensure MADNESS effectively utilizes the machines it runs, which includes everything from laptops to some of the largest TOP500 supercomputers, the vector code synthesis algorithm of this chapter was adapted to produce high performance code for mtxm on multiple architectures.

By performing the vector code synthesis algorithm using abstract vector operations which can be implemented in any available SIMD vector instruction set, high performance code is generated for all of the architectures which run MADNESS. The only difference not abstracted in synthesis is aligned memory accesses. In the case of x86 the hardware supports unaligned loads and stores which are used exclusively during code synthesis. To manage cases where the vectorized dimension is not a multiple of the vector size, the last store of every row is done using a masked store. The BlueGene architectures however are unable to perform unaligned operations. To work around this the call to mtxm was modified to first copy the data to an aligned buffer if it could not be accessed with aligned operations, otherwise the original layout in memory was used. A small performance penalty was observed on cases which required
the copy, but the overall impact of the copy is very low because the computationally dominant problem sizes are all multiples of the vector width.

Another major difference from the previous vector code synthesis algorithm is support for more data types, specifically the generator is capable of producing code for all cases of the each operand being either real or complex using the standard representation of complex numbers as two doubles contiguous in memory. However, due to the abstract vector instructions used for synthesis it was possible to implement this as an extension of the abstract operations for each target architecture.

While the problems are not large enough to require tiling to take advantage of cache, the kernel differs from those developed in this section in that it must be adaptable to multiple problem sizes. This was managed by performing aggressive unroll-and-jam which specialized code for each case of possible remaining iterations after the primary loop. This ensures that even problems which are smaller than the primary unroll-and-jam factor are still executed quickly without requiring code generated for each problem size.

Finally, some changes were added to the code generator which instruct the compiler exactly how the pointers to the arrays should be stored and incremented. The performance benefit of this is minimal but noticeable especially on architectures which require a vendor provided compiler that may lack all of the transformations of modern compilers.
CHAPTER 5

Multidimensional Retiming

While the vector code synthesis algorithm in the previous chapter is able to effectively explore the space of possible performance for tensor contractions, the effects on stencil computations is limited. This chapter describes multidimensional retiming which is able to improve register reuse in stencils by exploiting the reduction properties of stencils in combination with the large quantities of reuse available between iterations. The complete optimization framework discussed in this chapter involves the following steps:

1. Extract an internal representation of the input code using polyhedral compilation concepts.

2. Create a space of abstract scatter/gather alternatives along with different unrolling factors for the program.

3. For each point in the space, analytically compute the expected I/O per loop iteration and the expected register count needed to exploit full reuse along the loop.

4. Prune the space of candidate variants based on their arithmetic intensity relative to the original code using an analytical model.
5. For each point remaining, scatter/gather the appropriate dimension, perform complementary optimizations and apply the vector code synthesis algorithm from Chapter 4.

6. Perform auto-tuning to find the best performing variant on the target machine.

This chapter discusses the specifics of the multidimensional retiming, items 1 and 2 above, the remaining work is discussed in § 6.1.

5.1 Motivating Example

Listing 5.1: 2D stencil prototype implementation

Listing 5.1 will be used as an example to illustrate the fundamental issues addressed in this chapter. The code in Listing 5.1 is a generic convolution stencil that sweeps over a 2D array $OUT$, where at each point $(i, j)$, a weighted sum of a $n \times n$ ($n = 2 \times k + 1$) neighborhood around $(i, j)$ in array $IN$ is computed using a weight matrix $W$ of size $n \times n$. Stencil computations are generally considered to be memory-bandwidth bound since their arithmetic intensity is not usually sufficiently greater than the machine balance parameter, the ratio of peak main memory bandwidth to
peak computational performance [99]. However, the arithmetic intensity of a stencil is directly related to its order $k$.

A $3 \times 3$ 2D stencil, that is $k = 1$ in Figure 5.1a, involves nine multiplications and eight additions to compute each point of $OUT[i][j]$ assuming all weight coefficients are distinct, i.e., 17 floating point operations. Excluding points at the boundaries, each data element $IN[i][j]$ is used in computing nine neighboring points of $OUT$. Thus if full reuse of data elements is achieved in the last level cache, possible when the cache capacity is greater than approximately $2 \times k \times N$ words, the total bandwidth requirement per floating point operation would correspond to an average of one word loaded from main memory and one word stored to memory per 17 floating point operations, or 16 bytes of data transfer across the memory bus per 17 operations, giving a bytes/flop requirement below 1. The machine balance parameter for most multi-core systems today is much lower, e.g., around 20 GB/s bandwidth and upwards of 100 GFlop/s peak performance giving a bytes to flop ratio below 0.25.

Next consider a higher order stencil. Higher order stencils arise when higher order differences are used to discretize high order derivatives in PDE solvers, for example
<table>
<thead>
<tr>
<th>Variant</th>
<th>Diagram</th>
<th>$IN_{\text{loads}}$</th>
<th>$OUT_{\text{loads}}$</th>
<th>$OUT_{\text{stores}}$</th>
<th>$REGS$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather-Gather</td>
<td><img src="Gather-Gather" alt="Diagram" /></td>
<td>$n$</td>
<td>0</td>
<td>1</td>
<td>$n^2 - n + 2$</td>
</tr>
<tr>
<td>Gather-Scatter</td>
<td><img src="Gather-Scatter" alt="Diagram" /></td>
<td>1</td>
<td>$n - 1$</td>
<td>$n$</td>
<td>$n + 2$</td>
</tr>
<tr>
<td>Scatter-Gather</td>
<td><img src="Scatter-Gather" alt="Diagram" /></td>
<td>$n$</td>
<td>0</td>
<td>1</td>
<td>$n + 2$</td>
</tr>
<tr>
<td>Scatter-Scatter</td>
<td><img src="Scatter-Scatter" alt="Diagram" /></td>
<td>1</td>
<td>$n - 1$</td>
<td>$n$</td>
<td>$n^2 - n + 2$</td>
</tr>
<tr>
<td>Compact</td>
<td><img src="Compact" alt="Diagram" /></td>
<td>$\lceil n/2 \rceil$</td>
<td>$\lfloor n/2 \rfloor$</td>
<td>$\lfloor n/2 \rfloor$</td>
<td>$2 \cdot (\lceil n/2 \rceil)^2 + 2$</td>
</tr>
</tbody>
</table>

Table 5.1: Expected cost of retiming for 2D stencils

Overture from LLNL [74]. For a convolution with a $5 \times 5$ stencil (corresponding to $k = 2$), the arithmetic intensity increases, giving a machine balance requirement of $16/50$, still memory-bandwidth bound on current multi-core systems. However, a $7 \times 7$ stencil’s machine balance requirement will be roughly half of that for the $5 \times 5$ stencil. So it is expected that as the order of the stencil increases, the computation
becomes less memory bandwidth bound, and it would therefore be expected that the achieved performance of the stencil code should monotonically increase with the order of the stencil. However, the measured performance shown in Figure 5.1b shows a different trend. While performance does indeed increase from a $3 \times 3$ ($k = 1$) stencil to a $5 \times 5$ ($k = 2$) stencil, there is a drop in performance as the stencil order is further increased. Performance was tested on an Intel i7-4770k processor using code compiled with ICC -O3, using $N = 12000$. For each value of $k$, a distinct C code is generated and compiled. This C code is obtained by fully unrolling the $ii$ and $jj$ loops so as to have the standard implementation with all neighbor points accumulated in a single statement. The same approach is used to generate Figure 5.2 from the template in Listing 5.2.

The problem is that while the burden on the memory subsystem is reduced for higher order stencils, register pressure worsens. For a $3 \times 3$ stencil, as explained in greater detail later, six registers are needed to achieve three-way register reuse in the direction of stencil movement (the $j$ loop). For a $5 \times 5$ stencil, there is an opportunity to achieve a 5-way register reuse, but 20 registers are required to implement this reuse. Greater reuse is achieved at the cost of some register spilling and the overall performance improves. Hardware counters in Figure 5.1b show the total number of load instructions executed per FLOP decreases as $k$ goes from $k = 1$ ($3 \times 3$ stencil) to $k = 2$ ($5 \times 5$ stencil).

A $7 \times 7$ stencil offers the potential for 7-way register reuse, but the register pressure is over 42. The net result is that the code generated by the Intel ICC compiler for this case is less effective in exploiting register reuse, as shown by the hardware counter measurements in Figure 5.1b. Performance continues to drop as stencil order
is further increased, while greater arithmetic intensity implies performance should be improving.

5.1.1 Gather/Scatter Retiming

```
for (i=k; i<N-k; i++) {
    for (j=0; j<2*k; j++) {
        OUT[i][j+k] = 0;
        for (ii=-k; ii<=k; ii++)
            for (jj=-k; jj<=k; jj++)
                OUT[i][j-jj] += IN[i+ii][j]*W[k+ii][k+jj];
    }

    for (j=2*k; j<N-2*k; j++) {
        OUT[i][j+k] = 0;
        for (ii=-k; ii<=k; ii++)
            for (jj=-k; jj<=k; jj++)
                OUT[i][j-jj] += IN[i+ii][j]*W[k+ii][k+jj];
    }

    for (j=N-2*k; j<N; j++)
        for (ii=-k; ii<=k; ii++)
            for (jj=j-N+k+1; jj<=k; jj++)
                OUT[i][j-jj] += IN[i+ii][j]*W[k+ii][k+jj];
}
```

Listing 5.2: 2D stencil prototype implementation with retiming

This chapter focuses on a solution to the increased register pressure for higher order stencils by exploiting the freedom to reorder the associative/commutative operations of the stencil computations. The weighted contributions from the neighboring points can be accumulated in any order. However, changing just the order of operations among the set of accumulations to a single element of $OUT$ is not useful. Instead, the individual accumulations to multiple target elements are judiciously
interleaved. A transformed code template, representative of the kind of operation retiming generated by the framework, is shown in Listing 5.2.

In contrast to the original code in Listing 5.1, which can be seen as an all-gather stencil, i.e., all contributions to a target element are gathered together in a single set of operations, the code in Listing 5.2 can be viewed as a scatter-gather stencil. The code shown in Listing 5.2 performs exactly the same set of operations as the code in Listing 5.1, but in a different order of interleaving initialization and accumulation to elements of \( OUT \). Within the loop over rows \( (i) \), the code contains a prologue loop that performs updates to some of the left columns of \( OUT \), the main middle loop that performs the bulk of updates, and a final epilogue loop that performs updates to some of the right columns of \( OUT \). Considering a \( 3 \times 3 \) stencil, for a given point \( (i, j) \) of the outer two loops there is a \( 3 \times 1 \) read-set of three elements from \( IN \) each making contributions to each element of a \( 1 \times 3 \) write-set of \( OUT \). For a \( n \times n \) stencil, the transformed version involves a \( n \times 1 \) read-set updating a \( 1 \times n \) write-set in an all-to-all fashion. The main benefit is that now the register pressure is approximately \( n \) registers instead of \( n^2 \). The performance of the modified stencil is shown in Figure 5.2:
5.2a, and is compared with the base code over which it shows substantial performance improvement. Figure 5.2b shows hardware counters for the modified code. It can be seen that the loads per flop ratio is considerably lower than the original code, while the ratio of stores per flop is slightly higher. In essence, a highly asymmetric *all-gather* stencil with minimal stores but many more loads has been transformed into a more balanced stencil that performs more stores, but is able to achieve a substantial reduction in the number of loads.

Consider again the stencil code in Listing 5.1. A rectangular iteration space over the range \([k : N - k - 1][k : N - k - 1]\) is traversed, applying a stencil operation at each point in that space. The stencil can be characterized by a read-set and a write-set. For the code in Listing 5.1, the read-set has an offset range of \([-k : k][-k : k]\) around \([i][j]\), while the write-set is a single point, with offset range \([0 : 0][0 : 0]\). In general, the stencil can be viewed as a many-to-many set of edges from points in the read-set to points in the write set. The stencil in Listing 5.1 is an *all-gather* or *gather-gather* (gather in both dimensions) stencil, i.e., at iteration point \([i, j]\), points are read from \(IN[i - k : i + k][j - k : j + k]\) and written to \(OUT[i][j]\). For the *all-scatter* or *scatter-scatter* stencil, at iteration \([i, j]\), a point is read from \(IN[i][j]\) and accumulated to all points in \(OUT[i - k : i + k][j - k : j + k]\).

For the *gather-gather* stencil the total computation may be viewed as a set of edges in a bipartite graph from \(IN[0 : N - 1][0 : N - 1]\) to \(OUT[k : N - k - 1][k : N - k - 1]\). Any order of execution of the set of computation edges in this bipartite graph is valid. This can be done by creating an arbitrary modified stencil that has the same set of edges as the original stencil shifted identically in both spaces to different pairs points.

Consider a bipartite graph with the read-set vertices on one side and the write-set
vertices on the other. Initially, for an all-gather stencil, there are \( n \times n \) points of \( \text{IN}[-k : k][k : k] \) and a single output point \( \text{OUT}[0][0] \) with an edge from each input point to the single output point. The edges can be moved around as long as the orientation is not changed, i.e., the shift between the source point on \( \text{OUT} \) and the sink point on \( \text{IN} \) is preserved. For example, the edge from \( \text{IN}[-1][-1] \) to \( \text{OUT}[0][0] \) can be shifted to go from \( \text{IN}[0][-1] \) to \( \text{OUT}[1][0] \) or from \( \text{IN}[0][0] \) to \( \text{OUT}[1][1] \) or from \( \text{IN}[1][0] \) to \( \text{OUT}[2][1] \), etc.

A gather-scatter stencil is formed by shifting the edges along one dimension so that the footprint on \( \text{IN} \) is only \([0][-k : k]\) but this changes the footprint in \( \text{OUT} \) to \([-k : k][0]\). Many other configurations are possible; the only constraint is that all stencil edges are retained with their original orientations. Table 5.1 shows different stencils equivalent to the 9-point gather-gather stencil. The read-set vertices are shown as the solid purple circles and the write-set elements are the beige annuli.

The different stencil shapes differ in their register requirements as well as the number of loads and stores from memory required assuming \( \text{REGS} \) registers are available. For the gather-gather stencil, the write-set is a single element, all of whose updates happen in a single step. Thus a single register is needed for the write-set, and the IO cost is one store per iteration space point. The read-set has \( n^2 \) elements of which \( n^2 - n \) will be reused at the next iteration point \([i][j+1]\). In order to achieve this reuse, \( n^2 - n \) registers will be needed. At each iteration space point, a new set of \( n \) input values of \( \text{IN} \) will be loaded. The register requirement and the number of loads and stores are summarized in Table 5.1 for various equivalent stencils, including scatter-scatter, gather-scatter, scatter-gather, and a non-symmetric compact stencil with a read-set and write-set of four elements in a \( 2 \times 2 \) configuration.
5.2 Retiming Framework Overview

The Jacobi 1D stencil in Listing 5.4 is used as the illustrating example throughout this chapter. It is equivalent to the code in Listing 5.3 when assuming + is an associative and commutative operator, and shows a typical program input to the transformation framework.

Listing 5.5 shows a transformed version of the code, where multidimensional retiming has been applied to realign the accesses to IN so that inside an iteration of loop i, the same element of IN is being accessed. This corresponds to the all-scatter version of the code. Retiming is the key concept used to model the generalized scatter/gather transformation.
Listing 5.5: Jacobi 1D after retiming (all-scatter)

5.3 Program Representation

For the experiments results in this work the input programs adhered to the restrictions discussed in Chapter 4, however the multidimensional retiming can be applied generally. Specifically it only requires loop-based programs whose control-flow can be statically determined and represented using only affine inequalities. This class of programs is a superset of affine programs; it is not required that array index expressions to be limited to affine functions of the surrounding loop iterators. The main motivation for requiring that the control-flow can be captured using a polyhedral representation is the ease of expression and implementation of multidimensional retiming of statements, so as to achieve different locality and register pressure trade-offs.

The first step of the framework is to convert an input program into an internal representation that is amenable to effective retiming. For maximal flexibility it is best to split a single statement as in Listing 5.3 that contains multiple associative reduction operations, e.g., +, into distinct statements so that there is only one accumulation
operation per statement as in Listing 5.4. This enables a different retiming for the operands of each reduction. There may be an = operation which needs to be changed to +=, and vice-versa, to ensure that the first statement that updates an array element in the retimed program uses = and not +=, this is discussed further in § 6.1.5. The following concepts are used to represent the program:

**Statement iteration set:** Each statement in the program, e.g., S1, S2 and S3 in Listing 5.4, is associated with a set of integer points such that (1) there is one point in this set for each runtime instance of the statement; and (2) the coordinates of each point correspond exactly to the value taken by the surrounding loop iterators when that statement instance is executed. To ensure the ability to leverage polyhedral code generators such as CLooG [8] to implement the program transformation, this is restricted to a subset of \( \mathbb{Z}^p \), if S is surrounded by \( p \) loops and this subset is bounded only using affine inequalities. For example the iteration set of \( S_1 \) in Listing 5.4 is:

\[
\mathcal{I}_{S_1} : \{ i \in \mathbb{Z} \mid 1 \leq i < N - 1 \}.
\]

If in the original code the loop bound expressions are constants but not affine expressions, e.g., \( i < \sqrt{42} + \text{pow}(x, 42) \), they can safely be replaced by a unique parametric constant \( Cst \in \mathbb{Z} \) and the equivalent loop bound in the iteration set would be \( i < Cst \), which is an affine expression.

**Data accessed by a statement:** The data accessed by a particular statement *instance* is represented by a collection of access functions, one for each array access in the statement. Scalars are viewed as zero-dimensional arrays, and pointers are forbidden. Given a \( n \)-dimensional array reference, the associated access

53
function is represented with a vector $\vec{f}$ with $n$ components. For example, the access $IN[i-1]$ of statement $S1$ is represented as $f_{S1}^{IN} : (i - 1)$ and for an access $A[i][j-2]$ it is $f_{S}^{A} : (i, j - 2)$. There is no restriction imposed on the components of this vector, but the precision of data dependence analysis and the computation of the data space touched by a loop iteration is impacted by the form of expressions used in $f$: if only affine expressions of the surrounding loop iterators and parametric constants are used, then exact polyhedral dataflow analysis [35] can be achieved. On the other hand, arbitrary expressions may lead to over-approximating the data dependences of the program, limiting the freedom to retime the code and the accuracy of the analytical model to compute the register pressure.

Program execution order: The order in which the dynamic instances of the statements are executed is specified by a scheduling function that is applied to each iteration set to obtain the timestamp at which every instance is executed. The instances will be executed according to the lexicographic ordering of the timestamps. Given a statement $S$ surrounded by $p$ iterators, the schedule $T_S$ is a vector with $2p + 1$ components, such that: (1) odd components are scalars modeling the interleaving of loops and statements; and (2) even components are affine expressions of the surrounding loop iterators. Even components are further restricted to be of the form $i + \alpha$ where $i$ is the loop iterator at that particular nesting depth and $\alpha \in \mathbb{Z}$. For example, the schedule of statement $S1$ in Listing 5.4 is $T_{S1} : (0, i, 0)$; the schedule of $S2$ is $T_{S2} : (0, i, 1)$; and for $S3$ it is $T_{S3} : (0, i, 2)$. The original schedule can be constructed from a simple AST representation of only loops and statements as nodes, where alternating
components of the schedule represent the surrounding loop iterators and the relative position of the statement within those loops.

Applying loop transformations: As the control-flow is restricted to be static and exactly captured through the iteration sets, and the execution order is restricted to be exactly captured through multidimensional affine functions, loop transformations can be applied by using polyhedral code generation. In essence, polyhedral code generation emits a C code that implements the order specified by the schedule functions when applied on each point of the iteration sets. By carefully managing the overlap between transformed iteration sets for different statements through polyhedral separation [84, 8], polyhedral code generation produces the code in Listing 5.5 simply by using the schedules $T_{S1} : (0, i - 1, 0); T_{S2} : (0, i, 1); T_{S3} : (0, i + 1, 2)$, i.e., the original schedule with only the second component updated from $i$ to either $i - 1, i$ or $i + 1$, and the iteration sets $I_{S1} = I_{S2} = I_{S3} : \{i \mid 1 \leq i < N - 1\}$.

5.4 Formalization of Multidimensional Retiming

Multidimensional retiming has been previously studied in the literature, mostly for the purpose of parallelism exposure [12, 105, 86, 41] or to fix transformations for legality [94, 79]. This work takes a different approach, seeking a characterization of different data reuse patterns and register pressure when applying arbitrary multidimensional retiming. In particular, § 6.1 shows how the formalism below can be used to analyze and optimize the register pressure for a class of stencil computations. Multidimensional retiming is in essence a shift of an iteration set by a constant quantity along one or more of its dimensions. Definition 1 captures these factors in a
vector, one for each statement in the program, which together uniquely represent a
generalized scatter/gather combination across all dimensions.

**Definition 1 (Retiming vector).** Given a statement \( S \) surrounded by \( p \) loops, and
its associated schedule \( T_S \), a retiming vector \( \vec{r}^S \in \mathbb{Z}^p \) defines the offset quantity, for
each surrounding loop, to be applied on \( S \). The new schedule of operations for \( S \) is
\[
T'_S = T_S + \text{even}(\vec{r}^S) \quad \text{where} \quad \text{even}(\vec{r}^S) = (0, r^S_1, 0, r^S_2, 0, \ldots, r^S_p, 0).
\]

By applying \( r^S_1 = (-1) \), \( r^S_2 = (0) \) and \( r^S_3 = (1) \) to the example code in Listing 5.4, one obtains the code in Listing 5.5. A candidate program transformation is represented by the retiming vectors associated with each statement. If unrolling is to be applied, the statements to be unrolled are replicated, and their schedule updated to capture the expected final order of statements. In the framework all replications of the same statement use the same retiming vector, this is only a convenience, not a requirement.

One key observation about multidimensional retiming is that one can compute seamlessly the updated access functions for each memory reference in the code after retiming without having to generate the code. This is because the retiming vectors have been constrained to constant offsets. This feature is key for designing efficient analytical solutions to capture data reuse patterns after transformation, without having to explicitly generate any transformed code. The updated access functions are defined as follows:

**Definition 2 (Access function after retiming).** Given an access function \( f^S_A \), and a
retiming vector \( \vec{r}^S \) of dimension \( p \), after retiming, each loop iterator \( i_1, i_2, \ldots, i_p \) in
\( f^S_A \) will be replaced by \( i_1 - r^S_1, i_2 - r^S_2, \ldots, i_p - r^S_p \), respectively, in the updated access function.
For example, for IN[i-1] with access function $f^{IN}_{S1}: (i - 1)$, after retiming with $r^{S1} = (-1)$ the updated access function becomes $f^{IN}_{S1} = ((i - (-1)) - 1) = i$.

### 5.4.1 Legality of Retiming

In the general case data dependence analysis is required to ensure a retiming is legal, i.e., preserves the semantics of the input program, however, by leveraging associativity of certain operations, it can be proved that any retiming preserves the program semantics without resorting to dependence analysis.

**Legality of retiming for affine programs** If the entire program region to optimize fits the polyhedral compilation model, i.e., all loops and access functions are affine, determining the legality of a particular retiming can be done using polyhedral dependence analysis, which captures exactly the set of all pairs of dynamic instances which are in dependence. This dependence information can be used to test if a particular set of retiming vectors preserves the relative order of all dependent instances.

To illustrate this point, consider the example of Listing 5.4. Array dataflow analysis would tell that there are only two flow dependences, one from $S1(i)$ to $S2(i)$ and one from $S2(i)$ to $S3(i)$. As a consequence, any retiming vectors $r^{S1}$, $r^{S2}$ and $r^{S3}$ such that $r^{S1} \leq r^{S2} \leq r^{S3}$ lead to a legal retiming. For purely affine programs, one can build the convex space of all semantics-preserving affine multidimensional schedules [80, 60], fix some particular coefficients to capture the schedule shape constraints (odd dimensions are scalars, even dimensions are of the form $i + \alpha$), and project out most dimensions keeping only those corresponding to the schedule coefficients associated with the implementation of
shifting. The resulting set contains all legal retiming factors for each statement and each loop.

**Non-affine programs** Instead considering the legal superset of affine programs, i.e., allowing array access functions to be non-affine changes. This may prevent accurate dataflow analysis from being performed using classical polyhedral analysis techniques, but it does not prevent a naïve or conservative dependence analysis, and then using the polyhedral framework to represent the computed dependences. Because retiming is an affine transformation, legality can still be checked using existing polyhedral model based techniques. To illustrate this point, consider again the example of Listing 5.4. Suppose the most conservative analysis has been used to find dependences by looking only at array names being read and written. In that case, in addition to the actual dependences, a loop-carried dependence would appear. As a consequence, any retiming vectors \( r_{S1}, r_{S2}, \) and \( r_{S3} \) such that \( r_{S1} = r_{S2} = r_{S3} \) lead to a legal retiming. However, more retimings can be considered when the associative and communicative properties of the reduction operator are considered.

Reductions amount to *accumulating* the result of a set of operations to a single memory location. An example is shown in Listing 5.6.

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
        A[i] += C[i][j];
```

**Listing 5.6:** Reduction code
In this code, classical dependence analysis captures a strict sequential order between each instance of statement $R$; every iteration depends on all the previous ones because the same value is being read and written at each iteration of the $j$ loop. However, in practice reduction operators are typically associative and commutative operations. While compilers are often limited in their ability to exploit this associativity because of the inherent limitations of the IEEE 754 Floating Point standard, numerous previous works have established the benefit of exploiting associative reordering of reduction operators for parallelism [12, 105, 86, 41]. For instance, OpenMP supports parallelization of reduction via user-provided information about the reduction operator and accumulator location.

If the artificial dependence on the reduction loop in Listing 5.6 are removed by allowing operations to be reordered based on the associative and commutative properties of the $+$ operator, then any sequential order for the loop iterations becomes valid. That is, the dynamic instances of the statement can be executed in the order desired, provided they are executed one at a time. This leads to defining a class of dependences that can be safely ignored in the context of multidimensional retiming of reduction operations.

**Definition 3** (Commutative dependences). Let $Q$ and $R$ be two statements of the form $B_Q = B_Q \circ A_{f(Q)}$ and $B_R = B_R \circ A_{g(R)}$ where: $\circ$ is an associative and commutative operator; $A_*$ do not alias with $B_*$; and $B_R$ and $B_Q$ cannot overlap partially, i.e., either they represent the same variable or they do not alias; then any dependence between $Q$ and $R$ is said to be commutative.

Clearly, if no other dependences prevent it, the order of execution of $R$ and $Q$ can be permuted. Going back to the example in Listing 5.4, all dependences turn out to
be commutative. In other words, exploiting associativity and commutativity of the addition implies that any set of retiming vectors leads to a valid retiming, assuming that $A$ and $B$ do not alias, therefore any scatter/gather combination is also valid for this code.

5.4.2 Gather/Scatter as Retiming

Convolution stencils are of special interest w.r.t. the multidimensional retiming because all dependences are commutative and thus any retiming is valid. Stencil computations implementing convolutions typically use two arrays at each time iteration. Such computation accumulates weighted neighboring pixels to form the updated value of the current pixel. This observation is key for the application of the framework: by exploiting the associativity of the operation used to accumulate the value from the different neighbor pixels, one can rewrite certain stencil computations to fit the reduction pattern described above, and therefore enable arbitrary retimings on such computations. Listing 5.5 illustrates this concept by performing a retiming that implements the all-scatter variant of the kernel. The order in which updates are performed has changed from the original code; the new order does not preserve the constraints imposed by the commutative dependences, but is valid if such dependences are ignored.

The code in Listing 5.5 shows the all-scatter version of a Jacobi 1D code which exploits retiming. In general, the scatter/gather principle is seamlessly applied on convolution stencils provided they are written in a form where each update operation is in a distinct syntactic statement. This is required in the framework because the granularity at which retiming operates is limited to syntactic statements. For
example, moving to a 2D Jacobi, one can implement a scatter/gather combination by retiming the various statements along one spatial loop, while not retiming along the other spatial loops, as illustrated in Figure 5.1.
CHAPTER 6

Variant Selection

It was shown in § 4.4 that the best vectorized variant of a code can be application-dependent, machine-dependent and even compiler-dependent. This chapter discusses two models for selecting at compile time one of the variants from the space of possible variants presented in Chapter 4. The first model is a static cost model based on high-level features of tensor contraction. The second model is based on features extracted from the generated assembly that are processed by machine learning algorithms. Additionally this chapter presents the modeling of register reuse of stencils with the multidimensional retiming of Chapter 5 and a method to prune the space of variants.

Performance predictors are developed to be independent of the algorithms used to generate the vectorized variants. This is done by building models that can predict the performance of vectorized programs without running them, which can rank programs, i.e., all the vectorized variants of the kernel, according to their predicted performance. The program that ranks first is selected as the output of a model. There is no need for the model to correctly predict the numerical value associated with the performance of a program variant; only the relative order between predicted variants is relevant.
6.1 Register Reuse Modeling with Multidimensional Retiming

This section presents an analytical estimate of the memory traffic for a code transformed with the multidimensional retiming without actually generating code.

6.1.1 Data Reuse Across Retimed Iterations

One key objective of the retiming framework is to exploit retiming to change the data reuse pattern across loop iterations, and implicitly the register pressure. One register per data element to be reused is needed to exploit all the available reuse in registers between consecutive loop iterations. Definition 4 introduces the notion of an iteration set slice to enable a generic approach to compute the set of data elements touched by consecutive loop iterations, and therefore the set of data elements to be reused between iterations. This is a generalized form of the parametric domain slice defined by Pouchet et al. [81].

**Definition 4** (Iteration set slice). *Given an iteration set $I_s$ modeling a statement surrounded by $d$ loops, the slice $I^p_s$ is*

$$\bar{I}^p_s = \{ I_s \mid i_1 = p_1 + \alpha_1, \ldots, i_d = p_d + \alpha_d \}$$

*where $p_i$ is a parametric constant unrestricted on $\mathbb{Z}$ and $\alpha_i \in \mathbb{Z}$.*

For example, $I_{S1}$ from Listing 5.4 and $\bar{p} = (p_1)$, $I^p_{S1} = \{(i) \mid 1 \leq i < N - 1 \land i = p_1 \land p_1 \in \mathbb{Z}\}$ is a set which contains only one point. Two arbitrary but consecutive iterations of $i$ are modeled using $\bar{p}^1 = (p_1)$ and $\bar{p}^2 = (p_1 + 1)$ in the two slices $I^p_{S1}$ and $I^p_{S1}$. Each set obtained contains exactly one point, and they necessarily capture two consecutive iterations (that is, $i$ and $i + 1$ for any $i$).
The data space of a reference for a particular set of iterations is presented in Definition 5.

**Definition 5** (Data space of a reference for a set of iterations). *Given an access function* $f^S_A$ *of dimension* $n$ *in a statement* $S$, *and* $I'_S \subseteq I_S$ *a subset of the iteration set of* $S$, *the data space touched by this reference is*

$$D_{f^S_A, I'_S} = \{ \vec{x} \in \mathbb{Z}^n \mid \vec{x} = f^S_A(\vec{p}), \forall \vec{p} \in I'_S \}$$

The data set that is reused between two consecutive iterations is therefore the intersection of the data spaces at iteration $i$ and $i + 1$.

**Definition 6** (Data reused between consecutive loop iterations). *Given a collection of* $k$ *references* $f^k_A$ *on array* $A$ *inside the same loop nest of depth* $d$, *the iteration set of the inner most loop body, $p^d = (p_1, \ldots, p_d)$, $p^{d-1} = (p_1, \ldots, p_{d-1}, p_d + 1)$* *with* $p_k \in \mathbb{Z}$

$$DataSpace(A, I^{p^d}_i) = \bigcup_i D_{f^A, I^{p^d}_i}$$

$$DataSpace(A, I^{p^{d-1}}_i) = \bigcup_i D_{f^A, I^{p^{d-1}}_i}$$

$$Reuse(A) = DataSpace(A, I^{p^d}_i) \cap DataSpace(A, I^{p^{d-1}}_i)$$

$$Access(A) = DataSpace(A, I^{p^d}_i)$$

It follows that the number of distinct elements reused across consecutive inner-loop iterations is simply $#Reuse(A)$ where $# is the cardinality of a set of integer points.

When the function $f$ is affine, $D_{f^A, I_S}$ can be computed analytically as the image of $f$ over the polyhedron $I_S$, and as $Reuse(A)$ is a union of convex sets of integer points $#Reuse(A)$ can be computed using tools such as the Integer Set Library [95]. More interestingly, when $f$ is limited to the form $f_k : i_k + \alpha_k$ where $i_k$ is a loop iterator, for all dimensions $k$ of $f$ as it is in typical stencil computations, the data space of one loop iteration can be simply computed from the values of $\alpha_k$. When focusing
on affine stencil codes, costly polyhedral operations to build the sets and count their number of points are not needed; the data space of the stencil for $i$ and $i + 1$ and the resulting intersection can be computed via a simple enumeration of the updated array access functions.

In the general case of programs with arbitrary access functions the data space may not be computable analytically. This is not a problem for the transformation framework, but only for the analytical model used to prune the space of possible transformations and accelerate the search for a good transformed variant. Computing the data space only for some loops in the program, instead of all loops, could be helpful to expose affine array index expressions. Intuitively, when computing the data space of one iteration of the inner most loop, all surrounding loop iterators become constants. Consequently, non-affine expressions involving only those iterators can be replaced by a parameter, in a manner similar to non-affine loop bounds as shown earlier.

To compute the data reused across iterations in the presence of retiming, one simply needs to use updated access functions according to Definition 2 in Definition 6 instead of the original access functions. As the updated access functions can be computed analytically for arbitrary retiming vectors, the computability of Definition 6 is not affected by retiming.

### 6.1.2 Estimating Register Pressure and Data Movements

Register pressure of a computation is estimated with the following assumptions:

1. Reuse is only exploited between consecutive iterations of a loop.

2. Reuse between consecutive iterations is implemented with rotating registers.

3. The register cost of computing the access functions is zero.
With these properties in mind, the register pressure is defined as the count of all reused data elements, plus two registers to ensure any 3-address operation can be performed.

**Definition 7** (Register count for a loop iteration). *The number of registers needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is*

\[ rc = \sum_{A \in \text{Arrays}} \#\text{Reuse}(A) + 2 \]

Technically, more than two registers may be needed in order to also exploit intra-iteration reuse between data elements that are not reused in the next iteration. Accounting for such cases can be done by adding one register for each set \( s \) of access functions which are identical but do not refer to a data element reused at the next iteration when \( \#s > 1 \). Furthermore, the cost of keeping coefficients in registers must be considered, which can substantially increase pressure if each point has a unique weight.

The number of memory loads required is computed with a similar reasoning. The number of loads corresponds to the number of data elements which are not reused between consecutive iterations, considering only access functions related to reads.

**Definition 8** (Load count for a loop iteration). *The number of memory loads needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is*

\[ lc = \sum_{A \in \text{ReadArrays}} (\#\text{Access}(A) - \#\text{Reuse}(A)) \]
Finally, the number of memory stores required is computed directly from the set of distinct memory locations written to at a particular loop iteration that are not also written during the next loop iteration.

**Definition 9** (Store count for a loop iteration). *The number of memory writes needed to execute a loop iteration while exploiting reuse across consecutive iterations of this loop through rotating registers is*

\[
sc = \sum_{A \in \text{WriteArrays}} (\#\text{Access}(A) - \#\text{Reuse}(A))
\]

### 6.1.3 Modeling of Complementary Optimizations

The definitions of \(rc\), \(lc\) and \(sc\) gives analytical estimates of the memory behavior of a program, under the simplification assumptions mentioned above. To optimize an input program the framework considers multiple transformations, computes the cost in terms of memory movement and registers needed for each candidate, and prunes the set of candidate transformations using these metrics.

In order to effectively exploit the available reuse, the multidimensional retiming must be complemented by the set of transformations impacting data reuse patterns presented in Chapter 4. A strength of the framework is that for all transformations considered the values of \(rc\), \(lc\) and \(sc\) can be computed analytically without having to generate the transformed code variants.

### 6.1.4 Pruning the Space of Variants

Algorithm 6.1 summarizes the analytical approach to compute an estimate of the cost of various candidate transformations for a program \(P\), removing from the search space variants with excessive register usage or excessive memory traffic.
Input: \textit{program} 
Output: \textit{variants} 
\begin{algorithm}
begin 
\textit{U} ← \textit{UCMVectors}(\textit{program}); 
\textit{R} ← \textit{retimingVectorSets}(\textit{program}); 
\textit{P} ← \textit{loopPermutations}(\textit{program}); 
\textit{variants} ← \{\}; 
\textbf{for} \ \vec{u} \ ∈ \ \textit{U} \ \textbf{do} 
\textbf{for} \ \vec{r} \ ∈ \ \textit{R} \ \textbf{do} 
\textbf{for} \ \vec{p} \ ∈ \ \textit{P} \ \textbf{do} 
\textit{program}' ← \textit{buildUpdatedRepresentation}(\textit{program}, \vec{u}, \vec{r}, \vec{p}); 
\textit{AI} ← \textit{computeCost}(\textit{program}'); 
\textbf{if} \ \textit{aboveThreshold}(\textit{AI}) \ \textbf{then} 
\textbf{continue}; 
\textbf{end} 
\textit{variants} ← \{\textit{variants}, (\vec{u}, \vec{r}, \vec{p})\} 
\textbf{end} 
\textbf{end} 
\textbf{end} 
\end{algorithm}

\textbf{Algorithm 6.1:} \textit{exploreSpace}: Generate search space with retiming

Function \textit{UCMVectors} builds a set of all possible unrolling factors to be evaluated, based on a user-defined range for each dimension. Function \textit{retimingVectorSets} builds the retiming vectors for all statements based on user-defined ranges in each dimension. Function \textit{loopPermutations} builds the set of all possible loop permutations. These three functions are restricted by the program dependences in the general case: not all combinations of possible unroll and code motion, permutations and retiming are legal. However, as shown in § 5.2, for convolutions leveraging associative reordering, all gather/scatter combinations are necessarily valid and therefore no dependence analysis is required.

Function \textit{buildUpdatedRepresentation} modifies the program representation, i.e., the iteration sets and access functions, as needed to emulate the effect of the program...
transformation. Function `computeCost` applies Definitions 7, 8 and 9 to the updated program representation and analytically computes the desired values. The arithmetic intensity is then computed from the number of floating point operations executed in one iteration of the inner most loop using the updated representation, and dividing it by $lc + sc$. Function `aboveThreshold` checks if the arithmetic intensity is above a user-defined threshold, which in the framework is a function of the arithmetic intensity of the original program, and if so the variant is discarded, otherwise the variant is stored in the list of candidates to evaluate using auto-tuning. §7.6 extensive experimental results on a collection of high-order stencils and details the parameter values used for the search space evaluation.

Table 6.1 shows the search space statistics for several 3D and 4D benchmarks for different pruning factors. The only candidates kept for tuning are those with arithmetic intensity (AI) greater or equal to the original program ($1\times$), with AI greater than $1.5\times$ the AI of the original program, and $2\times$ greater. `Perf` shows the fraction of the performance of the space-best without pruning which is achieved by auto-tuning only on the pruned space. A value of $1\times$ means the candidate achieving the best overall GF/s is still in the space after pruning. Column `#Space` reports the search space size and `#Left` its size after pruning. The benchmark names and experimental setup are detailed §7.6.

The approach for pruning is empirical with the objective is to keep the auto-tuning time tractable, by evaluating on the target machine only a reasonably small number of variants. For instance, when considering 2D stencils from the test suite, the number of variants without pruning is 24, making pruning unnecessary. On the other hand, for 3D codes the number of variants quickly grows because of the additional possible
unrolling and combinations of scatter/gather along the third dimension. Keeping only variants which achieve an estimated AI two times better than the original code leads to a performance loss of 6% over the space-optimal point, but reduces the number of variants to be evaluated during auto-tuning by up to $5 \times$. In the case of 4D stencils it may be that an even high AI factor can be considered, however, since variants can be tested in a fraction of a second the space is still tractable. Note that arithmetic intensity alone is not a good performance predictor: the quality of vectorization is key for performance, and machine-specific metrics relating to the SIMD execution engine must be taken into account to approximate actual performance.

### 6.1.5 Complete Stencil Optimization Framework

Each program variant consists of a different set of values for loop permutations, retiming vectors for each program statement, and unrolling factors. The transformation system has been implemented\(^1\) using PoCC [78]. Once the polyhedral structures

\(^1\) [http://hpcrl.cse.ohio-state.edu/wiki/index.php/HOSTS](http://hpcrl.cse.ohio-state.edu/wiki/index.php/HOSTS)
are extracted from the internal representation, the loop permutation and retiming of all statements corresponding to a particular variant is embodied in the scheduling functions of the polyhedral representation. CLooG is used to generate the code structure, automatically handling all possible boundary cases induced by the retiming. A syntactic pass then applies loop unrolling and code motion, followed by stripmining the unit-stride dimension to enable vectorization. Finally, a post-processing pass was written to explicitly vectorize the stripmined loops using SIMD intrinsics, and apply rotating registers. Algorithm 6.2 shows the step-by-step transformations applied to a single program variant.

```
Input: source, machineInfo
Output: bestVariant
begin
    IR ← convertToIR(source);
    bestVariant ← source;
    bestTime ← ∞;
    for (u̅, r̅, p̅) ∈ exploreSpace(IR) do
        P ← permuteAndRetime(IR, r̅, p̅);
        P ← polyhedralCodegen(P, r̅, p̅);
        P ← prevectorize(P, u̅);
        P ← vectorize(P, machineInfo);
        T ← compileAndRun(P);
        if T < bestTime then
            bestVariant ← P;
            bestTime ← T;
        end
    end
end
Algorithm 6.2: optimizeProgram: End-to-end retiming and tuning
```

cvtColor translates the input program to the representation described in § 5.3. permuteAndRetime applies permutation and retiming, and selects the statement with
the lexicographic largest retiming vector as initialize-and-accumulate in a single step, i.e., using = instead of += wherever needed. polyhedralCodegen generates the transformed C code. prevectorize applies loop unrolling, code motion and stripmining. vectorize generates short-vector SIMD intrinsics for each stripmined loop using the vector ISA specified in machineInfo, and applies global value numbering to enable register reuse across iterations. Finally, the transformed source is compiled and executed, and the best performing variant is kept.

6.2 Tensor Contraction Static Cost Model

<table>
<thead>
<tr>
<th>Operation</th>
<th>Implementation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Stride Load/Store</td>
<td>_mm_loadu_ps() or _mm_storeu_ps()</td>
<td>1</td>
</tr>
<tr>
<td>Splat</td>
<td>_mm_load_ps1()</td>
<td>2</td>
</tr>
<tr>
<td>Reduction</td>
<td>2× _mm_hadd_ps() and _mm_store_ss()</td>
<td>3</td>
</tr>
<tr>
<td>Transpose</td>
<td>4× (_mm_loadu_ps() or _mm_storeu_ps()) and _MM_TRANSPOSE4_PS(), for 4× registers</td>
<td>3</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>_mm_mul_ps() and _mm_add_ps()</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.2: Cost and implementation of abstract operations

The static cost model defines six abstract vector operations for which specific costs are associated based on the target architecture. This model is predominately for predicting performance for tensor contractions. Table 6.2 describes the six operations, their implementation using SSSE3 single-precision intrinsics, and the associated cost used in the model. The cost is defined as the number of assembly instructions required to implement the abstract operation. In a system with a constant cycles per
instruction ratio this would be a perfect model, and although no machine implement-
ing SSSE3 meets this criteria, it is still a reasonable approximation to find the fastest implementation as demonstrated by the model verification in § 7.1.1. Algorithm 6.3 demonstrates how the cost model is used to approximate the total cost of executing a tile, which is computed given the location of the instructions within the loop nest and a set of unrolling factors for each of the loops. The algorithm measures each instruction’s contribution to the total cost of execution. Important to notice is how unrolling along a dimension that is not accessed by a particular instruction reduces the number of times that instruction must be executed.

Since the cost is approximated by the number of instructions, determining the size of the generated code is also trivial using a variation on Algorithm 6.3. The number of instructions in a generated kernel is limited to a fixed upper bound to prevent the generator from excessively unrolling the loops.

**Input:** instructions, dimensions, unrollings  
**Output:** cost

begin
\[
\text{cost} \leftarrow 0; \\
\text{for } \text{inst} \in \text{instructions} \text{ do} \\
\quad \text{executions} \leftarrow 1; \\
\quad \text{divisor} \leftarrow 1; \\
\quad \text{for } \text{dim} \in \text{dimensions s.t. loopContainsInstruction(dim, inst)} \text{ do} \\
\quad \quad \text{executions} \leftarrow \text{executions} \cdot \text{dim.size}; \\
\quad \quad \text{if isMemoryAccess(inst) and isAnIndexOf(inst, dim) then} \\
\quad \quad \quad \text{divisor} \leftarrow \text{divisor} \cdot \text{unrollings[dim]}; \\
\quad \text{end} \\
\text{end} \\
\text{cost} \leftarrow \text{cost} + \frac{\text{executions}}{\text{divisor}} \cdot \text{instruction.cost}; \\
\text{end}
\]

**Algorithm 6.3:** costEvaluation: Compute cost of tensor contraction variant
The algorithm shown in Algorithm 6.4 provides a high level overview of the static cost model’s functioning and interaction with the underlying code generation algorithms. It attempts vectorization along each dimension, saving the solution with the lowest cost to be returned as generated code. vectorizeInstructions, shown in Algorithm 4.1 is used to initialize the cost model with the specific types of loads and stores required. orderAndUnrollLoops takes a cost model and finds a way to order the loops and unroll to minimize the cost of execution without creating code that is excessively large. This requires searching the space of all loop permutations, and for each permutation the set of possible unrollings. It is possible to do this step using an exhaustive search of all possibilities for the experiments considered in Chapter 7 since the number of loops is not too large and the loop bounds are relatively small.
6.2.1 Detailed Example

Listing 6.1 shows a simple implementation of matrix-matrix multiplication, used in this section to explain the key ideas behind the static cost model-driven code synthesis.

```c
for (i=0; i<M; i++)
    for (k=0; k<N; k++)
        for (j=0; j<P; j++)
            C[i][j] += A[j][k]*B[k][i];
```

**Listing 6.1:** Matrix-matrix multiplication

Although the loops in Listing 6.1 are all fully permutable, vectorizing compilers generally cannot effectively vectorize such codes since all three arrays have different loops indexing their fastest varying dimension, i.e., loop $j$ for array $C$, loop $k$ for array $A$, and loop $i$ for array $B$. In order for a statement to be effectively vectorized by current compilers, each array must have its fastest varying dimension indexed by a single loop so that it can be loaded in unit-stride or be invariant with respect to that loop allowing it to be loaded into each component of a vector register outside the loop.

Listings 6.2, 6.3 and 6.4 show how vectorization of Listing 6.1 is feasible using the code synthesis described in Chapter 4 with SIMD vector length of four. Listings 6.2 and 6.3 show how vectorization can be achieved along the $j$ loop. For ease of presentation, it is assumed that all dimensions are perfect multiples of the vector size so that scalar epilogue code does not clutter the example. $B$ is not indexed by $j$ and can therefore be replicated 4-ways into a vector register using the splat operation. The
```c
for (i = 0 ; i < M ; i++) {
  for (k = 0 ; k < N ; k+=4) {
    b0[0:3] = splat{B[k+0][i]};
    b1[0:3] = splat{B[k+1][i]};
    b2[0:3] = splat{B[k+2][i]};
    b3[0:3] = splat{B[k+3][i]};
    for (j = 0 ; j < P ; j+=4) {
      a0[0:3] = A[j+0][k:k+3];
      a1[0:3] = A[j+1][k:k+3];
      a2[0:3] = A[j+2][k:k+3];
      a3[0:3] = A[j+3][k:k+3];
      transpose(a0,a1,a2,a3);
      c[0:3] = C[i][j:j+3];
      c[0:3] += a0[0:3]*b0[0:3];
      c[0:3] += a1[0:3]*b1[0:3];
      c[0:3] += a2[0:3]*b2[0:3];
      c[0:3] += a3[0:3]*b3[0:3];
      C[i][j:j+3] = c[0:3];
    }
  }
}
```

**Listing 6.2:** J vectorization, IKJ loop nesting

The challenge is with $A$ which has $j$ indexing its most significant dimension. The key idea to enable vectorization in such a situation is to perform 4-way unrolling of the loop indexing the fastest varying dimension, i.e., $k$. This enables contiguous vectors of size four to be loaded four times for four successive iterations of the vectorized dimension $j$, followed by a $4 \times 4$ register transposition of the sixteen loaded $A$ elements. Following the transposition, registers hold elements that can be vectorized along $j$, as shown by the code in Listing 6.2. Since the $k$ loop was unrolled by 4, four elements of $B$, $B[k][i], B[k+1][i], B[k+2][i]$, and $B[k+3][i]$ are broadcast (splat operation) into four vector registers, denoted $b_0[0 : 3]$, $b_1[0 : 3]$, $b_2[0 : 3]$, and $b_3[0 : 3]$. The static cost...
model estimates the cost of this variant, ignoring the cost of arithmetic which is independent of vectorization and other optimizations, as

\[
\text{cost} = + \left( 2 \cdot \frac{MNP}{16} \right) \text{unit stride load/store } C \\
+ \left( \frac{MN}{4} \right) \text{splat } B \\
+ \left( \frac{MNP}{16} \right) \text{transpose load } A
\]

\[= 1408 \quad (M = N = P = 16) \quad \text{(6.1)}\]

Additionally, it can be seen from this equation how application of unroll-and-jam can reduce the total number of memory operations. For example, a factor of two unrolling on the \(i\) loop will enable each \text{splat} of \(B\) to be reused twice, turning the second term into \(\frac{MN}{8}\).

Listing 6.3 shows how vectorization can be performed along a loop even if it is not the innermost loop in the nest. It illustrates how vectorization can be performed over loop \(j\) for the \(kji\) permutation of the loops. By performing a 4-way unroll-and-jam of loop \(j\), the resulting four statement instances in the innermost loop can be combined to use vector operations. Since array \(C\) is indexed in its rightmost dimension by \(j\), the statements resulting after the unroll-and-jam on \(j\) will read, modify and write \(C[i][j]\), \(C[i][j+1]\), \(C[i][j+2]\), and \(C[i][j+3]\). This can be achieved by a vector load operation into a register denoted in the pseudocode by \(c[0 : 3]\), vector arithmetic operations, followed by a vector store operation. Array \(A\) is indexed by \(j\) in its first dimension, i.e., elements with consecutive values of \(j\) will not be physically contiguous in the memory layout. The fastest varying dimension of \(A\) is indexed by loop \(k\). In order to vectorize the computation, the \(k\) loop is unrolled/jammed so that the innermost loop now has 16 statement instances that collectively access a \(4 \times 4\) block of data.
for (k = 0 ; k < N ; k+=4) {
    for (j = 0 ; j < P ; j+=4) {
        a0[0:3] = A[j+0][k:k+3];
        a1[0:3] = A[j+1][k:k+3];
        a2[0:3] = A[j+2][k:k+3];
        a3[0:3] = A[j+3][k:k+3];
        transpose(a0,a1,a2,a3);
        for (i = 0 ; i < M ; i++) {
            b0[0:3] = splat{B[k+0][i]};
            b1[0:3] = splat{B[k+1][i]};
            b2[0:3] = splat{B[k+2][i]};
            b3[0:3] = splat{B[k+3][i]};
            c[0:3] = C[i][j:j+3];
            c[0:3] += a0[0:3]*b0[0:3];
            c[0:3] += a1[0:3]*b1[0:3];
            c[0:3] += a2[0:3]*b2[0:3];
            c[0:3] += a3[0:3]*b3[0:3];
            C[i][j:j+3] = c[0:3];
        }
    }
}

Listing 6.3: J vectorization, KJI loop nesting

$A[j : j+3][k : k+3]$, which can be accessed via four vector loads followed by a transpose operation within vector registers.

Finally, for array $B$, since the set of elements accessed by the $4 \times 4$ group of iterations due to unrolling along $j$ and $k$ will be $B[k : k+3][i]$. Each of these elements must be replicated into the elements of a vector register through splat operations. The estimated cost is
\[ \text{cost} = + \left( 2 \cdot \frac{MNP}{16} \right) \text{unit stride load/store } C \\
+ \left( \frac{MNP}{4} \right) \text{splat } B \\
+ \left( \frac{NP}{16} \right) \text{transpose load } A \]

\[ = 2608 \quad (M = N = P = 16) \]

```c
for (j = 0 ; j < P ; j+=4) {
  for (i = 0 ; i < M ; i+=4) {
    c0[0:3] = C[i+0][j:j+3];
    c1[0:3] = C[i+1][j:j+3];
    c2[0:3] = C[i+2][j:j+3];
    c3[0:3] = C[i+3][j:j+3];
    transpose(c0,c1,c2,c3);
    for (k = 0 ; k < N ; k++) {
      a0[0:3] = splat{A[j+0][k]};
      a1[0:3] = splat{A[j+1][k]};
      a2[0:3] = splat{A[j+2][k]};
      a3[0:3] = splat{A[j+3][k]};
      b[0:3] = B[k][i:i+3];
      c0[0:3] += a0[0:3]*b[0:3];
      c1[0:3] += a1[0:3]*b[0:3];
      c2[0:3] += a2[0:3]*b[0:3];
      c3[0:3] += a3[0:3]*b[0:3];
    }
    transpose(c0,c1,c2,c3);
    C[i+0][j:j+3] = c0[0:3];
    C[i+1][j:j+3] = c1[0:3];
    C[i+2][j:j+3] = c2[0:3];
    C[i+3][j:j+3] = c3[0:3];
  }
}
```

Listing 6.4: I vectorization, JIK loop nesting
Listing 6.4 shows vectorization along the \( i \) loop for the \( jik \) permutation of the loops. Because the \( C \) array is accessed along a non-unit stride dimension it requires a register transpose operation both after loading and before storing. The estimated cost is

\[
cost = + \left( 2 \cdot \frac{MP}{16} \right) \text{transpose load/store C} \\
+ \left( \frac{MNP}{4} \right) \text{splat A} \\
+ \left( \frac{MNP}{16} \right) \text{unit stride load B} \\
= 2600 \quad (M = N = P = 16)
\]  

\[ (6.3) \]

<table>
<thead>
<tr>
<th>Vectorized Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
</tr>
<tr>
<td>Array ( A_{jk} )</td>
</tr>
<tr>
<td>Array ( B_{ki} )</td>
</tr>
<tr>
<td>Array ( C_{ij} )</td>
</tr>
</tbody>
</table>

**Table 6.3:** Vector operations based on vectorized along dimensions

In the manner illustrated by Table 6.3, vectorization is possible along all three loops of the example.

### 6.3 Machine Learning Models

The development of a second cost model was motivated by following factors:
1. By developing a technique that is not tied to the specifics of the optimization algorithm and problem domain the applicability of the model is significantly increased.

2. Building a model that operates directly on a high-level transformation sequence, e.g., the parameters to be given to a vectorization algorithm, would have to take into account that the parameter space varies across kernels. Input programs to be optimized may have very different optimization spaces: for example, the decision of which loop to vectorize is made from the set of parallel and reduction loops of the program, and this set is not constant across programs. The fact that the decision space is not constant across programs makes it poorly suited to machine learning models, and severely challenges the capability of any model to be generic across arbitrary kernels.

3. Compiler optimization heuristics play a significant role in the final performance of a program. These optimizations are usually organized in passes that are successively applied. The optimization heuristics implemented in compilers are often fragile and sensitive to the order in which those passes are performed, and of course to the precise structure of the input program. The main reason for the limited success of analytical models for performance prediction is that they attempt to predict performance before the compiler optimizations are applied. Analytically modeling every optimization of a compiler is just not feasible, and even if feasible would require a redesign of the model for each compiler revision.

To address these factors the new cost model is designed to operate on the assembly code that is produced by the compiler. By working on the end result of the entire
compiler optimization process, the need to model the impact of individual optimizations is avoided. In particular, after instruction scheduling and register allocation are performed, important performance factors can be analyzed, specifically the arithmetic intensity of the vectorized loops, i.e., the ratio of arithmetic to memory movement operations, and the distance between producer and consumer operations.

The overall approach is to train machine learning models to predict the performance $P$ of a vector of input features $ASM_{features}$ that characterizes a program. This training is done off-line, ideally during the installation of the compiler. When a new program is to be optimized, the model is used to predict the performance of a number of transformed variants of the input program. This evaluation never requires actual execution of the program nor any of the transformed variants. The model determines which of the transformed variants is predicted to perform best, and this variant is the output of the optimization process.

This approach has the benefit of observing side-effects from the compiler while still not requiring any of the variants to actually be run. Since the assembly features can be statically extracted from the codes everything can take place at run-time, and could potentially be built into the compiler.

### 6.3.1 Assembly Features

Feature extraction is performed on the basic block which forms the inner most loop of a kernel, which is the dominant contributor to execution time. The features extracted are:

**Vector Operation Counts:** Five parameters counting each of the following types of vector operations: addition, multiplication, load, store, and miscellaneous,
Listing 6.5: Example of x86 assembly code

Listing 6.6: Example of x86 assembly code with Sufficient Distance annotations

e.g. shuffles; and additionally the total number of vector operations, equal to
the sum of the five counts. Other useful metrics can be derived from these values
as described below. As an example, Listing 6.5 shows a piece of assembly code
with 3 loads, 0 miscellaneous, 1 multiply, 1 add, and 1 store. Scalar operations in the assembly code are largely ignored in this model, since separate hardware outside of the vector units is available on CPUs to execute scalar operations and they are not expected to play a significant role in determining the performance of loops dominated by vector instructions. The experimental results shown later in the paper show that vector operation counts form a sound basis for the input set.

**Arithmetic Intensity:** The ratio of vector arithmetic operations to vector loads. Although this is a derived metric computed from the vector operation counts, it is a sufficiently significant metric to warrant explicit inclusion as an input feature for the machine learning models — maintaining a high arithmetic intensity is essential to high performance on modern CPUs. In the example of Listing 6.5, the arithmetic intensity is \( \frac{2}{3} \) since there are 2 arithmetic operations (1 add and 1 multiply) and 3 loads.

**Sufficient Distance:** The number of arithmetic vector operations that produce a result that is not consumed in the next four instructions. The specific distance of four was chosen as it is representative of the latency of vector arithmetic operations used on both tested x86 CPUs, however this value could be easily tuned for other architectures. The rationale for this metric is that operations with a sufficiently large distance between producer and consumer instruction are unlikely to cause pipeline stalls, while operations with limited distance between producer and consumer can be expected to be more performance limiting. Listing 6.6 shows an example of x86 of arithmetic instructions annotated with the
distance until their output is used. The right operand of each instruction is the output, and in this example there are 10 instructions with sufficient distance. Values produced that are not used until the next iteration of the loop are also considered for this feature.

**Sufficient Distance Ratio:** The percentage of arithmetic vector operations which have sufficient distance from their first consumer. Higher values of this ratio suggest that available instruction level parallelism may be better exploited by the multiple vector functional units of a processor, without pipeline stalls due to dependences. In Listing 6.6 the ratio is \( \frac{10}{13} \) since there are 13 arithmetic instructions, of which 10 have sufficient distance.

**Total Operations:** The count of the total number of instructions in the innermost loop, including non-vector operations. In Listing 6.5 this value is 10, and in Listing 6.5 this value is 16. This is important because modern processors are more effective in processing loops with a limited body size which they can cache and reuse the decoded micro operations between iterations of the loop.

**Critical Path:** An approximation of the minimum number of instructions that must be executed in serial order in the innermost loop. This metric is based on the vector operation counts and number of ports available to process each type of instruction. In example Listing 6.5 the critical path is computed as 5 instructions. This is because only vector operations are considered, of which there are 6, but the add and multiply are considered only one cycle for this metric as the CPU can issue the two of them in parallel.
These features are combined to form feature sets which are used as inputs to each of the machine learning models. The strategy followed for building an effective feature set was to start with the simplest possible set which should be able to predict performance reasonably well, and incrementally add features in order of importance based on the additional information conveyed by the features. However, it is possible for additional features to fail to provide a benefit and may even reduce the effectiveness of the model. A detailed analysis is provided in § 6.3.3.

6.3.2 Model Training

A specific model is trained for each of the configurations, i.e., for each processor type, compiler, data type (float or double) and SIMD instruction set (AVX or SSE), a dedicated model is trained. This is relevant as it has been shown that the best variant is sensitive to each of these factors. For the training of the models, a set of 30 synthetic tensor contraction kernels were generated.

In order to train the machine learning models, a set of 30 tensor contractions were randomly synthesized. These were generated using 3 to 6 indices and between 1 and 4 contracted indices. The array indices, corresponding to the contraction being performed, were chosen as to span a variety of contractions and transposition of matrices which can occur in quantum chemistry codes. The tensor contractions were named in a canonical fashion to eliminate duplicates, and any that occur in the 19 CCSD tensor contractions used in later evaluation of the models were removed from the set of synthetic tensor contractions for machine learning training. The process of generating random tensor contractions for each number of indices was repeated until 30 acceptable tensor contractions were generated: there are 4 with 3 indices, 10 with
4 indices, 5 with 5 indices, and 11 with 6 indices. Of these, only 11 have an obvious
dimension for vectorization where two of the arrays have the same unit stride loop
index, thereby avoiding the expensive register level transpose.

The total number of variants for a tensor contraction can be very large due to these
three degrees of freedom: for the 30 tensor contractions used to train the machine
learning model, the number of code variants ranged between 42 and 2497, due to
differences in the number of loops and array indices in the tensor contractions.

Since tensor contractions are fully tileable, contractions on large tensors can always
be tiled such that all data accessed by a tile fits entirely in L1 cache. Therefore, this
section focuses on L1-resident datasets for construction of the machine learning model
and their evaluation. For each tensor contraction, the sizes of the tensors were chosen
to ensure that all tensors could together fit in L1 cache, with sizes along the fastest
varying dimension of all tensors being a perfect multiple of the vector length.

A model is trained as follows. A collection of vectorized variants is generated
for each of the 30 kernels, and for each of them their feature vector $ASM_{feature}$ is
computed. Each assembly code variant is run on the target machine and its perfor-
mance $P_{actual}$ is recorded as the GFlop/s of the variant. The model is trained with
the tuple $(ASM_{feature}, P_{actual})$. In the experiments of § 6.3.3, the standard Leave
One Benchmark Out Cross-Validation procedure is used for evaluating the models on
the 30 kernel set. That is, the models are trained on all variants of $N - 1$ kernels,
approximately 20000 programs. Models are then evaluated on all the variants of the
kernel that have been left out. For each variant, the feature vector $ASM_{feature}$ is fed
to the model, which outputs $P_{predicted}$, the expected performance. This procedure is
repeated individually for each kernel: each evaluation is done such that the variants of the kernel being tested were never seen by the model during training.

The running time of the training procedure depends on the compiler used, ICC being slower than GCC for the test suite. The total training time ranges from 30 minutes to 2 hours, depending on the configuration. For the evaluation of an unseen kernel the total time is dominated by the time to compile all variants, and therefore depends on the number of variants. In experiments it ranged from about 30 seconds to 10 minutes.

**Learning Algorithms Evaluated**

Performance prediction models were implemented using six different machine learning algorithms available in Weka [14]: Perceptron is an acyclic artificial neural network trained using back propagation; \texttt{K*} and \texttt{IBk} are both instance based learning algorithms which predict based on similar instances from the training set; \texttt{M5P} generates M5 model trees which are binary trees where each internal node compares an input value to a constant determined during training; \texttt{SVM} is a support vector machine algorithm using sequential minimal optimization; finally \texttt{LR} is linear regression. All these algorithms were used with the default parameter values provided with Weka. These values have been found empirically to be generally good by the developers, however further improvements may be possible by tuning the parameters during cross validation at the risk of over fitting to the training data.

**6.3.3 Analysis of the Generated Models**

This section analyzes the feature space and evaluates choices for selection of features to be used as inputs to the models and the resulting performance.
Correlation Between Features

To characterize the feature space, a study of the coefficients of determination was performed between the features described in § 6.3.1, averaged for all vectorized variants of kernels in the CCSD application. This is summarized as a Hinton diagram shown in Figure 6.1 which graphically represents the value of the coefficient of determination between features as the size of each box: the larger the box the more strongly the two features are correlated.

The most prominent aspect of the diagram is that none of the features are well correlated with performance, motivating the need for combining multiple features to create an input set to the models. Looking at arithmetic intensity, it is among the best correlated to GFlop/s, the performance metric. The static cost model effectively measures relative arithmetic intensity between variants, and is a logical choice since the higher the arithmetic intensity, the less memory bound the computation. The performance achieved by using this feature alone is analyzed below, and is referenced as $fs_1$, feature set 1. However, using $fs_1$ as the only input to the model exhibits poor predictions.

Another category of features shows good correlation with performance, this is the set of vector operation counts. Those features are quite well correlated with each other, especially $Muls$, the number of vector multiply operations, is the most correlated to performance in this set. Feature set $fs_2$ which contains those features in addition to $fs_1$, and it greatly improved quality of the predictions over $fs_1$.

The sufficient distance ratio feature is the least correlated to other features. As such it is able to provide information to the predictors which is unavailable from the other features. Although it is also the least correlated to performance, adding this
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Coefficient of determination for variants of CCSD

feature improves the quality of the model. The set $fs3$ contains this feature in addition to those in $fs2$. 

90
Finally, the sufficient distance, critical path and total operation features are all highly correlated with each other, and show good correlation with performance. Feature set $fs_4$ contains these features in addition to $fs_3$. However, this does not improve the performance of the predictors over using only $fs_3$.

**Prediction Quality for the Feature Sets**

To determine the quality of predictions, an *efficiency metric* is defined as follows:

$$\text{Efficiency} = \frac{\text{Performance of predicted best}}{\text{Performance of actual best}}$$

The efficiency is 100% if the predicted best variant is the actual best variant for the kernel. The actual best variant was found by evaluating on the target machine the entire space of possible vectorized variants generated by the algorithm, and the one with the maximal measured performance is taken as the best variant.

![Figure 6.2: Prediction quality for each feature set](image)

Figure 6.2 quantifies the quality of predictions for each feature set across all configurations using the K* model, the number of kernels for which the predicted best
achieves less than 50% efficiency (0-0.5), those achieving 50% to 80% (0.5-0.8) and those achieving more than 80% efficiency (0.8-1.0). The poor results using fs1 illustrates how arithmetic intensity alone is not a sufficient criterion to determine performance. This is consistently observed across all ML models using fs1. On the other hand, the relatively similar performance of fs2, fs3 and fs4 indicates that arithmetic intensity coupled with vector operation counts gathers the most important information to predict performance, as would be predicted by their relatively good correlation with performance as seen in Figure 6.1. Additional features in fs3 and fs4 provide minimal benefit. A complete analysis shows that for some specific combinations of ML model and configuration, fs4 performs marginally better than fs3. However, fs3 is used in experiments, based on its ability to obtain the best average efficiency across many of the models and configurations, while limiting the required number of inputs.

Evaluation of the Models

Figure 6.3 show the result of the predictive modeling using fs3 over different machine learning models, for three different configurations using ICC with floats. The performance of ICC’s auto-vectorization on the original source code, using the same efficiency metric, and the result of an average random choice from the search space of variants.

ICC auto-vectorization is significantly outperform by considering a large search space of transformations and making a better decision about which transformation to apply. The machine learning models find performance dependencies between the transformations instead of predicting the effect of each transformation individually. Also, by considering vectorization along all dimensions instead of only those accessed
in unit-stride, they are able to utilize the SIMD units of the processors where current compilers fail to find the best transformations to optimize the vectorized code.

The efficiency achieved by Random search reflects the variation in the quantity of good vectorized variants across configurations: for Sandy Bridge w/AVX, all but 1 kernel in Random fail to exhibit more than 50% efficiency. For this configuration, the space contains fewer good variants than with Nehalem. In contrast, the machine learning models, in particular instance-based learning algorithms such as IBk and KStar, achieve 80% efficiency or more for a vast majority of the kernels. Note that
simpler classification models such as Multi-Layer Perceptron MLP and SVM, while still significantly outperforming Random selection, exhibit a lower prediction quality. However, a detailed analysis shows that while on average those models perform worse, for a few kernels they outperform the instance-based algorithms.

A similar study was conducted across all configurations, concluding that IBk, KStar and M5P are consistently the best machine learning models for accurately ranking the performance of the various vectorized variants. Table 6.4 show the efficiency of these three models across all twelve configurations. The configuration is abbreviated using four letters: N for Nehalem and S for Sandy Bridge, S for SSE and A for AVX, F for float and D for double, and I for ICC and G for GCC. When ties occur due to a model may predicting the same performance for two or more variants, the average efficiency of the variants which ranked first is reported. Table 6.4 shows that no single model is consistently best across all configurations, although all three models perform similarly within any given configuration.

6.3.4 The Weighted-Rank Model

As shown in § 7.3, it is not always the case that a single model is best. Figure 6.4 further demonstrates this for individual benchmarks by comparing the efficiency of the KStar and IBk models for the CCSD benchmarks across all configurations. While most benchmarks appear on the diagonal, i.e., both models predict the benchmark equally well, there are enough cases which are significantly better predicted by one of the models. To address this problem a second-order model was built by combining the predictions of the two best individual models, KStar and IBk.
<table>
<thead>
<tr>
<th>Model</th>
<th>IBk</th>
<th>KStar</th>
<th>M5P</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSDG</td>
<td>.88</td>
<td>.87</td>
<td>.86</td>
</tr>
<tr>
<td>NSDI</td>
<td>.94</td>
<td>.96</td>
<td>.93</td>
</tr>
<tr>
<td>NSFG</td>
<td>.88</td>
<td>.87</td>
<td>.86</td>
</tr>
<tr>
<td>NSFI</td>
<td>.96</td>
<td>.92</td>
<td>.90</td>
</tr>
<tr>
<td>SADG</td>
<td>.89</td>
<td>.91</td>
<td>.91</td>
</tr>
<tr>
<td>SADI</td>
<td>.81</td>
<td>.88</td>
<td>.85</td>
</tr>
<tr>
<td>SAFG</td>
<td>.89</td>
<td>.89</td>
<td>.89</td>
</tr>
<tr>
<td>SAFI</td>
<td>.87</td>
<td>.88</td>
<td>.89</td>
</tr>
<tr>
<td>SSDG</td>
<td>.87</td>
<td>.88</td>
<td>.87</td>
</tr>
<tr>
<td>SSDI</td>
<td>.93</td>
<td>.95</td>
<td>.94</td>
</tr>
<tr>
<td>SSFG</td>
<td>.89</td>
<td>.89</td>
<td>.83</td>
</tr>
<tr>
<td>SSFI</td>
<td>.94</td>
<td>.93</td>
<td>.87</td>
</tr>
</tbody>
</table>

**Table 6.4:** Average efficiency of individual models from leave-one-out analysis

**Figure 6.4:** Efficiency of KStar vs IBk on CCSD

Each individual model produces a predicted performance for all variants of a program, leading to a rank of the variants according to this prediction. That is, given a model $M$ and a variant $v$, $R^*_v$ is the rank of the variant according to its predicted...
performance. The variant $v$ with the best predicted performance has $R^M_v = 1$, the second predicted best has rank 2, etc. Ties are resolved by giving the tied variants the same rank. Figures 6.5 and 6.6 show for individual configurations the rank error for every variant, i.e., the difference between the expected rank and predicted rank. This is not as effective as Figure 6.4 in demonstrating the difference in final performance after prediction, but does show that across the entire space of variants there a large number of cases where the error of the best models is not correlated.

![Figure 6.5: Nehalem rank error comparison of KStar vs IBk](image)

Table 7.2 shows that the some of the models do perform significantly worse on average across all kernels. Hence not all models are considered for the composite
model, but only the best performing ones to develop the composite model. The composite model combines the ranks obtained by each variant on IBk and KStar, to obtain a composite rank \( WR \) for a variant:

\[
WR_v = \gamma_1 \cdot R_{IBk}^v + \gamma_2 \cdot R_{KStar}^v
\]

The factor \( \gamma_i \) represents the contribution of each individual models to the final vote. Intuitively, this should produce better results given that the errors in individual models are not always correlated. For instance, choosing \( \gamma_i = \frac{1}{2} \) creates a fair voting model, where both machine learning models have the same decision power. A fair

Figure 6.6: Sandybridge rank error comparison of KStar vs IBk
voting model was evaluated and observed to be no better than the best individual model, IBk.

To build the WeightedRank model, Linear Regression was used to find the optimal $\gamma_i$ coefficient values. The problem learned is

$$(R_{v}^{IBk}, R_{v}^{K*}) \rightarrow WR_{v}$$

and the model is trained using for $WR_{V}$ the actual rank of each variant. Weka’s Linear Regression model was used with the default parameter values. This is effectively a naïve boosting algorithm which demonstrates that boosting in general could be a useful technique in this domain, especially in tandem with training more models which may have individually weak performance. As such, bagging may also be an effective technique assuming the weak learners are not stable.
CHAPTER 7

Experimental Results

7.1 A Multi-Resolution Tensor Kernel

This section presents the performance of code synthesized using the static cost model for the multi-resolution tensor kernel discussed in § 2.1.1. Table 7.1 shows an example of the code that could be synthesized for each of the six tensor contractions in the kernel. The tensors given as input to this kernel are almost always small enough that tiling is not necessary, instead it adds unnecessary overhead. Experiments were done on the Intel Core i7-920 processor using the Intel ICC 11.0 compiler.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Outer loop</th>
<th>2nd loop</th>
<th>3rd loop</th>
<th>Inner loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{ijk} = \sum_i S_{ijk} X_{li}^L$</td>
<td>$l \times 4$</td>
<td>$i \times 16$</td>
<td>$j \times 1$</td>
<td>$k \times 4$</td>
</tr>
<tr>
<td>$B_{lmk} = \sum_j A_{ijk} Y_{jm}^L$</td>
<td>$1 \times 4$</td>
<td>$j \times 16$</td>
<td>$i \times 1$</td>
<td>$k \times 4$</td>
</tr>
<tr>
<td>$C_{lnm} = \sum_k B_{lmk} Z_{kn}^L$</td>
<td>$1 \times 4$</td>
<td>$j \times 4$</td>
<td>$i \times 4$</td>
<td>$k \times 4$</td>
</tr>
<tr>
<td>$D_{lmk'} = \sum_k C_{lmn} Z_{nk'}^R$</td>
<td>$j \times 4$</td>
<td>$k \times 4$</td>
<td>$i \times 4$</td>
<td>$1 \times 4$</td>
</tr>
<tr>
<td>$E_{ij'k'} = \sum_i D_{lmk} Y_{m_j'}^R$</td>
<td>$1 \times 16$</td>
<td>$j \times 4$</td>
<td>$i \times 1$</td>
<td>$k \times 4$</td>
</tr>
<tr>
<td>$R_{ij'k'} = \sum_i E_{ij'k'} X_{li}^R$</td>
<td>$1 \times 16$</td>
<td>$i \times 4$</td>
<td>$j \times 1$</td>
<td>$k \times 4$</td>
</tr>
</tbody>
</table>

Table 7.1: Example loop properties for the multi-resolution kernel
7.1.1 Model Verification

For the case of matrix-matrix multiplication for $i = j = k = 16$ the cost model predicted that vectorization along $j$ would be best. The synthesized code for performed at at 12.2 GFlop/s when vectorized on the $j$ loop, and 6.4 GFlop/s when vectorized on $k$ or $i$. Although the model cannot accurately predict the relative speedup between versions, it is sufficient that it ranks them in the correct order to achieve top performance.

7.1.2 Performance with Multi-Resolution Kernel

The synthesized code is compared to the reference implementation compiled with yacc and to MKL BLAS. There is one change in the implementation from the specification in equation (2.5): the $Z^T$ array is stored transposed. All three implementations use this transposed array, as shown by the one true value for TransB in the BLAS implementation of Table 2.1, and it is a valid transformation in practice because the array can be synthesized for the kernel in either layout. Figure 7.1 compares performance for multiple problem sizes, where $p \leq 10$ is typical for time independent problems and $p > 10$ is typically only used for time dependent problems. The absolute performance of the synthesized code manages 36% to 54% of theoretical peak on the time independent problem sizes, which is excellent given the exceptionally small size of the problems. These small problems also demonstrate the most significant improvement relative to BLAS. Figure 7.3 details the relative performance of the synthesized code to that of BLAS; While in all cases the synthesized code offers almost a $2 \times$ or better improvement, the best benefit is for smaller sizes where it is able to perform upto $6 \times$ faster than BLAS for double precision.
Performance results are also shown using code synthesized for four element single precision SSE vectors in Figure 7.2. While single precision values are rarely used in scientific codes currently, there are cases where the trade-off of precision for performance is acceptable but not implemented due to engineering difficulty. There is an
even larger speed up with the increased vector length of single precision relative to double precision, with a peak of over $17 \times$ the performance of the BLAS implementation for small sizes.
The drop in performance of the synthesized code noticeable in Figure 7.1e for larger values is due to cache misses. Implementing code tiling for larger problem sizes would improve cache performance and eliminate this decrease in performance.

It is also important to note the ubiquitous poor performance of the code synthesized by icc, which in no case exceeds 1 GFlop/s, demonstrating the inadequacy of existing automatic vectorization and optimization for non-trivial computations.

### 7.2 MADNESS: The mtxm Operator

The MADNESS mtxm operator of § 4.6 is commonly used in the transform3d operator which is further described in § 8.1.2. This can be implemented as a sequence of three calls to mtxm as shown in Listing 7.1. The most computationally dominant problem size is $N = 20$, and as such the code generated and used in MADNESS is optimized for that problem size and the smaller ones which occur less frequently. The performance of the optimized code for a Haswell processor using AVX2 instructions
is shown in Figure 7.4. For comparison the performance using MKL’s dgemm implementation in place of the optimized mtxm is also shown. For larger problem sizes MKL becomes more efficient as tiling can be more effectively used with larger problem sizes, but for the relevant sizes the optimized implementation run significantly faster. Notably, in the dominant case of $N = 20$ the optimized code is 30% faster than MKL, which has a direct impact on the overall run time of MADNESS.

### 7.3 CCSD Tensor Contractions

This section uses the CCSD tensor contractions discussed in § 2.1.2 to evaluate the effectiveness of the machine learning models developed in § 6.3 in identifying effective vectorized code variants. In order to train each machine learning model, a different set of 30 randomly generated tensor contractions was generated as described in § 6.3.2. In both this section and § 7.5, models are trained on all variants of the
void transform3d(long n,
    double *__restrict__ r,
    const double *__restrict__ t,
    const double *__restrict__ c) {
    double *__restrict__ tmp = malloc(sizeof(double)*n*n*n);
    mtxmq(n*n, n, n, r, t, c);
    mtxmq(n*n, n, n, tmp, r, c);
    mtxmq(n*n, n, n, tmp, r, c);
    free(tmp);
}

Listing 7.1: transform3d as calls to mtxm

30 tensor contraction set, and evaluated on a fully distinct test set, either the CCSD
and Stencil benchmark suites, that which does not contain any of the tensors tested
were seen during training.

In-program timing code was used to monitor the execution time. Each variant
tensor contraction was run approximately $10^5$ times to improve the accuracy of the
results, although the actual number of repetitions was set so that a total of roughly
50 million floating point operations were executed. The execution time was computed
as an average over all the repetitions. The variance among independent runs for any
specific variant was less than 4%.

7.3.1 Performance Evaluation

Evaluation in this section been performed with the following: two production
compilers: GNU GCC 4.6 and Intel ICC 12.0; two processors: Intel Core i7-920 using
the Nehalem micro-architecture and Intel Core i7-2600 using the Sandy Bridge micro-
architecture; and two data types for the tensor contractions: float and double.
For the Nehalem, the SSE SIMD instruction set was evaluated, while for the Sandy

105
Bridge both the SSE and AVX instruction sets were evaluated. The cartesian product of these possibilities results in a total of twelve different configurations.

The **Nehalem** processor runs at 2.66 GHz, with a theoretical peak per core of 10.64 GFlop/s (*double*) and 21.28 GFlop/s (*float*). The **Sandy Bridge** processor runs at 3.4 GHz, with theoretical single-threaded peak SSE performance of 13.6 GFlop/s (*double*) and 27.2 GFlop/s (*float*) and peak AVX performance of 27.2 GFlop/s (*double*) and 54.4 GFlop/s (*float*).

A careful study of the individual results shows that while some models can correctly output an 80%+ variant for a given benchmark, other models may fail to do so. Table 7.2 illustrates this with details of the performance of the various models for Sandy Bridge with AVX, using *float* data and ICC, with the fs3 feature set.

Tensor contractions are described by the indices of their three arrays. For instance, \(ij-ik-kj\) represents a contraction of two 2D tensors and corresponds to standard matrix multiplication. The \(St-m\) column indicates the efficiency obtained by using the static cost model from § 6.2 to select the vectorized variant.

While IBk and KStar are consistently the two best models, there are numerous cases where one is able to achieve a significantly better efficiency than the other. For instance, for \(ijk-ilmk-mjl\) with IBk achieves 98% efficiency while KStar is limited to 70%. The situation can also be reversed, as with \(ijkl-mink-jnlm\). In order to benefit from the ability of different models to predict best transformations for different benchmarks, The weighted-rank model (§ 6.3.4) attempts to solve this problem.

The Weighted Rank model performs on average marginally better than the best two individual models, with an average efficiency of 92%. However, its most important contribution is to improve over the worst performing benchmarks when considering
<table>
<thead>
<tr>
<th>Tensor Contraction</th>
<th>ICC</th>
<th>Rand</th>
<th>St-m</th>
<th>IBk</th>
<th>KStar</th>
<th>LR</th>
<th>M5P</th>
<th>MLP</th>
<th>SVM</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ij-ik-kj</td>
<td>.14</td>
<td>.38</td>
<td>.85</td>
<td>1.0</td>
<td>1.0</td>
<td>.70</td>
<td>1.0</td>
<td>1.0</td>
<td>.63</td>
<td>1.0</td>
</tr>
<tr>
<td>ij-ikl-ljk</td>
<td>.10</td>
<td>.32</td>
<td>.05</td>
<td>.97</td>
<td>.76</td>
<td>.81</td>
<td>.76</td>
<td>.73</td>
<td>.81</td>
<td>.97</td>
</tr>
<tr>
<td>ij-kil-lkj</td>
<td>.10</td>
<td>.31</td>
<td>.55</td>
<td>1.0</td>
<td>.89</td>
<td>.67</td>
<td>1.0</td>
<td>.67</td>
<td>.67</td>
<td>1.0</td>
</tr>
<tr>
<td>ijk-ikl-lj</td>
<td>.14</td>
<td>.47</td>
<td>.70</td>
<td>.85</td>
<td>.73</td>
<td>.63</td>
<td>.92</td>
<td>.85</td>
<td>.63</td>
<td>.73</td>
</tr>
<tr>
<td>ijk-il-jlk</td>
<td>.23</td>
<td>.29</td>
<td>.48</td>
<td>.99</td>
<td>.99</td>
<td>.55</td>
<td>.82</td>
<td>.71</td>
<td>.80</td>
<td>.99</td>
</tr>
<tr>
<td>ijk-ilk-jl</td>
<td>.25</td>
<td>.34</td>
<td>.23</td>
<td>.99</td>
<td>.99</td>
<td>.69</td>
<td>.98</td>
<td>.81</td>
<td>.80</td>
<td>.99</td>
</tr>
<tr>
<td>ijk-il-klj</td>
<td>.24</td>
<td>.43</td>
<td>.23</td>
<td>.84</td>
<td>.84</td>
<td>.65</td>
<td>.81</td>
<td>.81</td>
<td>.80</td>
<td>.84</td>
</tr>
<tr>
<td>ijk-il-mjk</td>
<td>.08</td>
<td>.25</td>
<td>.75</td>
<td>.98</td>
<td>.70</td>
<td>.66</td>
<td>.38</td>
<td>.70</td>
<td>.78</td>
<td>.98</td>
</tr>
<tr>
<td>ijk-il-nkl</td>
<td>.06</td>
<td>.35</td>
<td>.05</td>
<td>.86</td>
<td>.84</td>
<td>.73</td>
<td>.76</td>
<td>.86</td>
<td>.58</td>
<td>.86</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.10</td>
<td>.35</td>
<td>.86</td>
<td>.89</td>
<td>.96</td>
<td>.77</td>
<td>.77</td>
<td>.79</td>
<td>.57</td>
<td>.96</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.05</td>
<td>.27</td>
<td>.05</td>
<td>.96</td>
<td>.90</td>
<td>.62</td>
<td>.78</td>
<td>.67</td>
<td>.51</td>
<td>.96</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.09</td>
<td>.31</td>
<td>.28</td>
<td>.93</td>
<td>.93</td>
<td>.94</td>
<td>.88</td>
<td>.92</td>
<td>.78</td>
<td>.93</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.05</td>
<td>.28</td>
<td>.04</td>
<td>.92</td>
<td>.92</td>
<td>.70</td>
<td>.87</td>
<td>.74</td>
<td>.67</td>
<td>.92</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.05</td>
<td>.28</td>
<td>.77</td>
<td>.88</td>
<td>.92</td>
<td>.44</td>
<td>.92</td>
<td>.77</td>
<td>.44</td>
<td>.92</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.05</td>
<td>.33</td>
<td>.03</td>
<td>.83</td>
<td>.99</td>
<td>.48</td>
<td>.88</td>
<td>.75</td>
<td>.48</td>
<td>.89</td>
</tr>
<tr>
<td>ijk-il-nklm</td>
<td>.08</td>
<td>.39</td>
<td>.54</td>
<td>.78</td>
<td>.74</td>
<td>.87</td>
<td>.80</td>
<td>.83</td>
<td>.37</td>
<td>.74</td>
</tr>
<tr>
<td>ijk-injn-nklm</td>
<td>.12</td>
<td>.46</td>
<td>.44</td>
<td>.91</td>
<td>.86</td>
<td>.65</td>
<td>.90</td>
<td>.92</td>
<td>.65</td>
<td>.86</td>
</tr>
<tr>
<td>ijk-injn-nklm</td>
<td>.09</td>
<td>.37</td>
<td>.02</td>
<td>.73</td>
<td>.91</td>
<td>.46</td>
<td>.94</td>
<td>.82</td>
<td>.46</td>
<td>.91</td>
</tr>
<tr>
<td>ijk-injn-nklm</td>
<td>.09</td>
<td>.46</td>
<td>.24</td>
<td>1.0</td>
<td>1.0</td>
<td>.63</td>
<td>.97</td>
<td>.61</td>
<td>.30</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 7.2: Efficiency of the ML models on the CCSD benchmarks (SAFI)

configurations individually. In all but 3 cases, the Weighted Rank model outputs the maximal efficiency of either KStar or IBk, thus effectively selecting the model that predicts the better variant.

A complete analysis across all twelve configurations shows that the Weighted Rank model is consistently better on average than each individual model. The results are shown in Table 7.3 which reports the average efficiency of all models, across all twelve configurations, for the CCSD test set. Other composite models were considered, including a Weighted Rank model using all six individual models that uses a neural
network to learn the weight function; they did not perform as well as the 2-way
Weighted Rank model described above.

<table>
<thead>
<tr>
<th>Config</th>
<th>CC</th>
<th>Rand</th>
<th>St-m</th>
<th>IBk</th>
<th>KStar</th>
<th>LR</th>
<th>M5P</th>
<th>MLP</th>
<th>SVM</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSDG</td>
<td>.42</td>
<td>.64</td>
<td>.82</td>
<td>.86</td>
<td>.85</td>
<td>.83</td>
<td>.81</td>
<td>.84</td>
<td>.83</td>
<td>.86</td>
</tr>
<tr>
<td>NSDI</td>
<td>.37</td>
<td>.66</td>
<td>.78</td>
<td>.95</td>
<td>.96</td>
<td>.80</td>
<td>.92</td>
<td>.93</td>
<td>.93</td>
<td>.95</td>
</tr>
<tr>
<td>NSFG</td>
<td>.31</td>
<td>.53</td>
<td>.79</td>
<td>.91</td>
<td>.86</td>
<td>.64</td>
<td>.86</td>
<td>.80</td>
<td>.63</td>
<td>.90</td>
</tr>
<tr>
<td>NSFI</td>
<td>.19</td>
<td>.54</td>
<td>.84</td>
<td>.92</td>
<td>.89</td>
<td>.72</td>
<td>.89</td>
<td>.88</td>
<td>.84</td>
<td>.92</td>
</tr>
<tr>
<td>SADG</td>
<td>.27</td>
<td>.51</td>
<td>.75</td>
<td>.84</td>
<td>.89</td>
<td>.70</td>
<td>.87</td>
<td>.83</td>
<td>.72</td>
<td>.85</td>
</tr>
<tr>
<td>SADI</td>
<td>.22</td>
<td>.38</td>
<td>.44</td>
<td>.82</td>
<td>.86</td>
<td>.67</td>
<td>.88</td>
<td>.69</td>
<td>.75</td>
<td>.88</td>
</tr>
<tr>
<td>SAFG</td>
<td>.21</td>
<td>.49</td>
<td>.65</td>
<td>.81</td>
<td>.82</td>
<td>.68</td>
<td>.81</td>
<td>.81</td>
<td>.67</td>
<td>.81</td>
</tr>
<tr>
<td>SAFI</td>
<td>.11</td>
<td>.35</td>
<td>.38</td>
<td>.91</td>
<td>.89</td>
<td>.67</td>
<td>.85</td>
<td>.79</td>
<td>.62</td>
<td>.92</td>
</tr>
<tr>
<td>SSDG</td>
<td>.43</td>
<td>.67</td>
<td>.86</td>
<td>.88</td>
<td>.85</td>
<td>.83</td>
<td>.78</td>
<td>.85</td>
<td>.75</td>
<td>.87</td>
</tr>
<tr>
<td>SSDI</td>
<td>.33</td>
<td>.67</td>
<td>.79</td>
<td>.95</td>
<td>.95</td>
<td>.75</td>
<td>.93</td>
<td>.94</td>
<td>.91</td>
<td>.94</td>
</tr>
<tr>
<td>SSFG</td>
<td>.33</td>
<td>.53</td>
<td>.82</td>
<td>.88</td>
<td>.87</td>
<td>.63</td>
<td>.88</td>
<td>.78</td>
<td>.63</td>
<td>.88</td>
</tr>
<tr>
<td>SSFI</td>
<td>.20</td>
<td>.52</td>
<td>.84</td>
<td>.92</td>
<td>.89</td>
<td>.67</td>
<td>.81</td>
<td>.80</td>
<td>.78</td>
<td>.92</td>
</tr>
<tr>
<td>Average</td>
<td>.28</td>
<td>.54</td>
<td>.73</td>
<td>.88</td>
<td>.88</td>
<td>.71</td>
<td>.85</td>
<td>.83</td>
<td>.75</td>
<td>.89</td>
</tr>
</tbody>
</table>

**Table 7.3:** Average efficiency of the ML models on the CCSD benchmarks

Table 7.4 reports the performance, in GFlop/s, obtained by using the WeightedRank model to select at compile-time an effective vectorized variant for the 19 CCSD tensor contractions. As previously discussed, none of the CCSD contractions were seen during training. These results are compared against the compiler’s auto-vectorization, i.e., `icc -fast` or `gcc -O3`, on the original input code. The `avg` column reports the mean performance across all 19 contractions. To illustrate the variance across the benchmarks, `min`, the performance of the tensor contraction with the lowest GFlop/s across the benchmark suite, as well as `max`, the one with the highest GFlop/s are also reported. Table 7.4 highlights the very strong benefit in using the
machine learning model in combination with vectorizing and optimizing code synthesis algorithm, when compared to ICC's automatic vectorization. Despite the small fraction of high-performance variants in this complex search space, as illustrated in § 4.4, the \textbf{WeightedRank} model is able to achieve single-core improvements ranging from $2 \times$ up to $8 \times$ on average depending on the configuration. For some tensor contractions complex loop permutation sequences are required, along with vectorizing one of the reduction loops. At the time of writing, it appears that GCC’s and ICC’s auto-vectorization is limited to vectorizing dimensions accessed in unit-stride, and thus do not vectorize some of the tensor contractions.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Compiler (GF/s)</th>
<th>Weighted Rank (GF/s)</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>NSDG</td>
<td>1.38</td>
<td>3.02</td>
<td>8.48</td>
</tr>
<tr>
<td>NSDI</td>
<td>1.30</td>
<td>2.82</td>
<td>5.29</td>
</tr>
<tr>
<td>NSFG</td>
<td>1.39</td>
<td>4.34</td>
<td>16.70</td>
</tr>
<tr>
<td>NSFI</td>
<td>1.30</td>
<td>2.71</td>
<td>5.98</td>
</tr>
<tr>
<td>SADG</td>
<td>2.31</td>
<td>4.55</td>
<td>11.63</td>
</tr>
<tr>
<td>SADI</td>
<td>1.89</td>
<td>3.92</td>
<td>6.69</td>
</tr>
<tr>
<td>SAFG</td>
<td>2.40</td>
<td>6.87</td>
<td>24.47</td>
</tr>
<tr>
<td>SAFI</td>
<td>1.89</td>
<td>4.15</td>
<td>9.79</td>
</tr>
<tr>
<td>SSDG</td>
<td>2.31</td>
<td>4.57</td>
<td>11.62</td>
</tr>
<tr>
<td>SSDI</td>
<td>1.89</td>
<td>3.90</td>
<td>6.69</td>
</tr>
<tr>
<td>SSFG</td>
<td>2.40</td>
<td>6.89</td>
<td>24.74</td>
</tr>
<tr>
<td>SSFI</td>
<td>1.89</td>
<td>4.16</td>
<td>9.57</td>
</tr>
</tbody>
</table>

\textbf{Table 7.4}: Performance of CCSD benchmarks with WeightedRank

Interestingly, Table 7.4 also shows that GCC can outperform ICC for some configurations; this is especially the case for the max column for the original code. For the
original code, manual investigation of a few benchmarks showed that GCC applies unroll-and-jam more aggressively than ICC. The best absolute performance is found on Sandy Bridge with AVX, where ICC attains 43 GFlop/s, which is about 80% of the machine peak. However, for Sandy Bridge with SSE, GCC achieves 21.4 GFlop/s while ICC tops at 21 GFlop/s, 77% of machine peak. GCC’s ability to slightly outperform ICC for some benchmarks was observed across various configurations, but ICC performs consistently better on average when compiling vector intrinsics code for the Nehalem and Sandy Bridge systems.

### 7.3.2 Performance Evaluation Without Explicit Vectorization

To complete the evaluation of the machine learning based approach, a complementary evaluation of the predictors was performed using the CCSD benchmarks with only unroll-and-jam factors and loop permutation transformations being applied, both of which are implemented in modern compilers such as ICC and GCC. That is, the customized generation of vector intrinsics is not used, instead the compiler is provided with just variants which have had unroll-and-jam and loop permutation applied to the source code. The objective is to highlight the limitations of the current state-of-the-art heuristics implemented in the compiler to compute the unroll-and-jam factors and loop permutation, and show the improvement that can be obtained over the current state-of-the-art production compilers simply by using the machine learning model in place of the internally implemented heuristics.

Table 7.5 shows the performance obtained for the CCSD application, with variants chosen from this restricted search space, for the Intel ICC compiler, averaged for each of the six configurations.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ij-ik-kj</td>
<td>8.11</td>
<td>2.13</td>
<td>11.11</td>
<td>2.94</td>
</tr>
<tr>
<td>ijk-il-jk</td>
<td>5.18</td>
<td>2.47</td>
<td>7.40</td>
<td>3.54</td>
</tr>
<tr>
<td>ijk-kil-lkj</td>
<td>6.67</td>
<td>3.14</td>
<td>10.58</td>
<td>5.01</td>
</tr>
<tr>
<td>ijk-ikl-lj</td>
<td>6.76</td>
<td>2.09</td>
<td>11.51</td>
<td>3.84</td>
</tr>
<tr>
<td>ijk-il-jlk</td>
<td>7.31</td>
<td>1.37</td>
<td>16.87</td>
<td>2.87</td>
</tr>
<tr>
<td>ijk-ilk-jl</td>
<td>7.23</td>
<td>1.36</td>
<td>15.78</td>
<td>2.62</td>
</tr>
<tr>
<td>ijk-ilk-lj</td>
<td>7.61</td>
<td>1.43</td>
<td>14.69</td>
<td>2.48</td>
</tr>
<tr>
<td>ijk-ilmk-mjl</td>
<td>5.03</td>
<td>2.17</td>
<td>10.13</td>
<td>4.67</td>
</tr>
<tr>
<td>ijk-imjn-lnkm</td>
<td>6.77</td>
<td>4.97</td>
<td>9.52</td>
<td>7.31</td>
</tr>
<tr>
<td>ikl-imjn-nlmk</td>
<td>5.77</td>
<td>2.21</td>
<td>10.86</td>
<td>4.63</td>
</tr>
<tr>
<td>ijk-imkn-jnlm</td>
<td>6.82</td>
<td>5.29</td>
<td>9.31</td>
<td>6.93</td>
</tr>
<tr>
<td>ijk-imkn-jnjm</td>
<td>6.19</td>
<td>3.15</td>
<td>12.88</td>
<td>6.03</td>
</tr>
<tr>
<td>ijk-imln-jnkni</td>
<td>6.36</td>
<td>5.03</td>
<td>7.19</td>
<td>5.34</td>
</tr>
<tr>
<td>ijk-imln-njmk</td>
<td>5.51</td>
<td>2.12</td>
<td>9.46</td>
<td>7.26</td>
</tr>
<tr>
<td>ijk-imnnj-nlkm</td>
<td>6.33</td>
<td>2.43</td>
<td>9.58</td>
<td>7.22</td>
</tr>
<tr>
<td>ijk-imnkJ-njml</td>
<td>4.66</td>
<td>2.24</td>
<td>12.62</td>
<td>5.93</td>
</tr>
<tr>
<td>ijk-minj-nlmk</td>
<td>5.93</td>
<td>1.67</td>
<td>9.64</td>
<td>3.66</td>
</tr>
<tr>
<td>ijk-minkJ-njnm</td>
<td>6.02</td>
<td>2.42</td>
<td>10.38</td>
<td>4.14</td>
</tr>
<tr>
<td>ijk-minnj-njnmk</td>
<td>5.57</td>
<td>2.44</td>
<td>12.09</td>
<td>5.41</td>
</tr>
<tr>
<td></td>
<td>Average:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.31</td>
<td>2.64</td>
<td>11.14</td>
<td>4.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.04</td>
<td>2.77</td>
<td>19.08</td>
<td>5.91</td>
</tr>
</tbody>
</table>

**Table 7.5:** Performance of WeightedRank without explicit vectorization

The machine learning models were trained using the same feature set as above, by evaluating numerous variants where the production compiler was used to generate the final vector code. With an average improvement of $2.6 \times$ to $5.9 \times$ over the algorithm implemented in the ICC auto-vectorization pass, the weighted rank model is clearly capable of achieving good performance by using optimization parameters that are internally accessible in modern compilers. The heuristics guiding these optimization in production compilers could be improved by using the weighed rank model.

111
7.4 Tiling for Larger Tensors

This section investigates the use of PTile to perform automatic tiling and SMP parallelization in combination with the code synthesis algorithm targeting an Intel Many Integrated Core Knight’s Ferry processor for a subset of the CCSD tensor contractions.

PTile uses the polyhedral model to represent a loop nest as a system of affine inequalities, and through manipulation of those inequalities is able to create new code which maintains dependencies while extracting parameterized rectangular tiles and annotating the source code with OpenMP pragmas to enable parallelization. This enables the code to take advantage of the cache hierarchy and the many available cores. However, the cost of this code generation is more complicated loop structure and bounds which in turn makes it difficult for ICC to generate efficient SIMD vector code. Analysis of the assembly generated by ICC shows significantly less aggressive optimizations on the inner most loop when PTile is used: In the case of \(ij-ik-kj\), i.e., matrix multiplication, fused multiply-adds are not used at all in the PTile code, where as the input code when compiled by ICC uses fused multiply-adds and more aggressive loop unrolling.

The heuristic from § 4.5 is used to select the tile size used in the code produced by PTile.

7.4.1 Tensor Contractions Tested

In the case of the tensor contractions considered for this work, different vectorized dimensions do not need to be considered as all the kernels were selected to have an obvious optimal dimension for vectorization where the output tensor and one of the
input tensors are both accessed in unit-stride by the same dimension. Vectorization along this dimension for these contractions does not require in-register transpose to enables vectorization along non-unit-stride dimensions or reductions which are required for vectorization along contracted dimensions. The only instructions needed for these contractions are the vector intrinsics for unit-stride vector loads and stores, splats, i.e., a load replicating a scalar element to all elements of a vector, and fused multiply-adds. This subset was chosen due to a lack of support for other tensor contractions in the code synthesizer for the Knight’s Ferry platform. In later generations of the platform specialized instructions for reading values from non-unit-stride memory locations and better arithmetic support enable execution of the other tensor contractions.

Problem sizes where the vectorized dimension is not a multiple of the SIMD vector width are not supported in the analysis to prevent unaligned memory accesses as KNF only supports aligned loads and stores. Use of masked loads and stores would enable dimensions to be any size but could contribute a significant performance penalty as twice the loads and stores may be required. Alternatively, tensors could be padded to a multiple of the vector width to work with the generated code at the cost of additional memory usage. To select the optimal variant auto-tuning was applied across the space of variants.

### 7.4.2 Performance Evaluation

These experiments were performed on a Intel MIC Knights Ferry with 30 cores at 1.05 GHz. The kernels were compiled to natively target MIC using ICC\(^2\) (icc -mmic -O3). Kernels tested with OpenMP (icc -mmic -O3 -openmp) were set to

\(^2\)icc --version reports icc (ICC) Mainline 20120222
use 30 threads. Auto-parallelization (icc -mmic -O3 -parallel) was tested, but even when loops were successfully parallelized the performance did not differ from the single core version. All of the kernels use doubles and the tensors were sized to occupy $\sim 6MB$ and require $\sim 125M$ operations for the standard dataset shown in Table 7.6, and Table 7.7 presents the large dataset which occupied $\sim 24MB$ and required $\sim 1G$ operations. An average of 171 variants per kernel were generated, ranging from 26 for $ij-ik-kj$ to 296 for those with 6 dimensions.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>PTile</th>
<th>PTile+Intrin</th>
<th>PTile+Intrin +OpenMP</th>
<th>OpenMP Speed Up</th>
<th>OpenMP Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ij-ik-kj$</td>
<td>.08</td>
<td>1.23</td>
<td>10.34</td>
<td>8.43</td>
<td>.28</td>
</tr>
<tr>
<td>$ij-kil-lkj$</td>
<td>.46</td>
<td>1.88</td>
<td>7.08</td>
<td>3.77</td>
<td>.13</td>
</tr>
<tr>
<td>$ijk-ilj$</td>
<td>.32</td>
<td>.75</td>
<td>3.11</td>
<td>4.12</td>
<td>.14</td>
</tr>
<tr>
<td>$ijk-iljk$</td>
<td>.35</td>
<td>.81</td>
<td>16.29</td>
<td>20.07</td>
<td>.67</td>
</tr>
<tr>
<td>$ijk-il$</td>
<td>.28</td>
<td>.78</td>
<td>14.81</td>
<td>18.98</td>
<td>.63</td>
</tr>
<tr>
<td>$ijk-il$</td>
<td>.35</td>
<td>3.20</td>
<td>19.60</td>
<td>6.13</td>
<td>.20</td>
</tr>
<tr>
<td>$ijk-il$</td>
<td>.46</td>
<td>4.48</td>
<td>28.85</td>
<td>6.43</td>
<td>.21</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>.35</td>
<td>1.86</td>
<td>14.56</td>
<td>7.83</td>
<td>.26</td>
</tr>
</tbody>
</table>

**Table 7.6:** KNF performance relative to ICC for the standard dataset

Table 7.6 shows the relative performance of the synthesized code to that obtained by ICC on a single core given the input code. The first column shows a significant drop in performance by enabling PTile without parallelization or vectorization. Inspection of the generated assembly shows that the introduction of complicated loop structures by PTile hinders ICC’s ability to efficiently vectorize automatically. Introduction of intrinsics-based vectorized tiles provides a mean single core improvement of 86% over
ICC. Finally, utilization of OpenMP to take full advantage of the MIC Architecture shows a mean improvement of $14.56 \times$ over ICC single core performance. The right side of Table 7.6 shows the speed up of the code using PTile and vector intrinsics when OpenMP is enabled, and the efficiency of the 30 cores. Efficiency is defined as the ratio of the speed up to the number of processors and represents the utilizations of the cores.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>PTile</th>
<th>PTile+Intrin</th>
<th>PTile+Intrin +OpenMP</th>
<th>OpenMP Speed Up</th>
<th>OpenMP Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ij-ik-kj</td>
<td>.01</td>
<td>1.18</td>
<td>11.93</td>
<td>10.08</td>
<td>.34</td>
</tr>
<tr>
<td>ij-kil-lkj</td>
<td>.42</td>
<td>1.67</td>
<td>6.00</td>
<td>3.60</td>
<td>.12</td>
</tr>
<tr>
<td>ijk-il-jlk</td>
<td>.43</td>
<td>1.00</td>
<td>4.77</td>
<td>4.77</td>
<td>.16</td>
</tr>
<tr>
<td>ijk-ilk-jl</td>
<td>.33</td>
<td>.75</td>
<td>15.67</td>
<td>20.89</td>
<td>.70</td>
</tr>
<tr>
<td>ijk-ilk-lj</td>
<td>.29</td>
<td>.78</td>
<td>16.07</td>
<td>20.66</td>
<td>.69</td>
</tr>
<tr>
<td>ijk-ilmk-mjl</td>
<td>.19</td>
<td>2.38</td>
<td>36.44</td>
<td>15.34</td>
<td>.51</td>
</tr>
<tr>
<td>ijklnknjml</td>
<td>.10</td>
<td>2.32</td>
<td>19.35</td>
<td>8.33</td>
<td>.28</td>
</tr>
<tr>
<td>ijklnk-njml</td>
<td>.07</td>
<td>1.78</td>
<td>14.41</td>
<td>8.10</td>
<td>.27</td>
</tr>
<tr>
<td>ijklnnl-njmk</td>
<td>.10</td>
<td>2.47</td>
<td>20.50</td>
<td>8.31</td>
<td>.28</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>.15</td>
<td>1.45</td>
<td>13.78</td>
<td>9.52</td>
<td>.32</td>
</tr>
</tbody>
</table>

Table 7.7: KNF performance relative to ICC for the large dataset

Table 7.7 is structured the same as Table 7.6, except it is based on the large data set as described above. It is worth noting that the results for PTile are worse than with the standard dataset, implying that the heuristically chosen tile size is suboptimal for larger data, or the problem may require multiple levels of tiling. As such, the intrinsics and OpenMP versions are not as effective as in the standard dataset, however they are still improvements over compilation of the input code with
Furthermore, there is a notable increase in the efficiency of parallelization with OpenMP, 32\% vs. the standard dataset’s 26\%.

Due to the large number of cores available in KNF, and the relatively small number of iterations over loops in the case of tensor contractions with many dimensions, it is possible the performance of parallel code running with OpenMP is hindered by the ratio of processors to loop iterations. Merging outer parallel loops, e.g., using OpenMP collapse, as well as modifications of the tile scheduling code could allow more loop iterations to be run in parallel and thus better utilize the available cores.

### 7.5 Stencil Vectorization

To evaluate the quality of the prediction models, vectorized variants of nine stencils from the *Stencil Micro-Benchmarks* [29] were synthesized, and evaluated them with the models that were trained on the tensor contraction training set. That is, similarly to the CCSD evaluation, none of the stencil benchmarks were seen during the training of the models.

The stencils selected for the test suite were from [29] are: Partial derivatives (disofive, disothree, drowthree); Biharmonic operator (dbigbiharm, dlilbiharm); NAS MG (dresid, drprjthree); and Noise cleaning (inoiseone, inoisetwo). All are 2D or 3D stencils. The stencils were chosen with the criteria that they each have only a single pass, fewer than 25 points, and at least one non-trivial coefficient. Neither retiming or time-tiling was applied on the considered stencils.
Table 7.8 shows the average efficiency across all nine stencils of the six considered machine learning models and the Weighted Rank model, across all twelve configurations, in addition to the compiler auto-vectorization efficiency as well as random selection of a vectorized variant.

<table>
<thead>
<tr>
<th>Config</th>
<th>CC</th>
<th>Rand</th>
<th>IBk</th>
<th>KStar</th>
<th>LR</th>
<th>M5P</th>
<th>MLP</th>
<th>SVM</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSDG</td>
<td>.60</td>
<td>.81</td>
<td>.95</td>
<td>.87</td>
<td>.64</td>
<td>.80</td>
<td>.84</td>
<td>.64</td>
<td>.93</td>
</tr>
<tr>
<td>NSDI</td>
<td>1.05</td>
<td>.94</td>
<td>.95</td>
<td>.95</td>
<td>.96</td>
<td>.93</td>
<td>.94</td>
<td>.94</td>
<td>.95</td>
</tr>
<tr>
<td>NSFG</td>
<td>.32</td>
<td>.74</td>
<td>.84</td>
<td>.72</td>
<td>.60</td>
<td>.62</td>
<td>.85</td>
<td>.60</td>
<td>.89</td>
</tr>
<tr>
<td>NSFI</td>
<td>.41</td>
<td>.94</td>
<td>.95</td>
<td>.95</td>
<td>.96</td>
<td>.93</td>
<td>.93</td>
<td>.95</td>
<td>.96</td>
</tr>
<tr>
<td>SADG</td>
<td>.41</td>
<td>.80</td>
<td>.85</td>
<td>.82</td>
<td>.68</td>
<td>.75</td>
<td>.74</td>
<td>.68</td>
<td>.86</td>
</tr>
<tr>
<td>SADI</td>
<td>.79</td>
<td>.93</td>
<td>.92</td>
<td>.92</td>
<td>.92</td>
<td>.93</td>
<td>.94</td>
<td>.93</td>
<td>.92</td>
</tr>
<tr>
<td>SAFG</td>
<td>.33</td>
<td>.91</td>
<td>.90</td>
<td>.93</td>
<td>.91</td>
<td>.90</td>
<td>.91</td>
<td>.91</td>
<td>.92</td>
</tr>
<tr>
<td>SAFI</td>
<td>.41</td>
<td>.95</td>
<td>.96</td>
<td>.96</td>
<td>.94</td>
<td>.95</td>
<td>.93</td>
<td>.94</td>
<td>.96</td>
</tr>
<tr>
<td>SSDG</td>
<td>.56</td>
<td>.83</td>
<td>.97</td>
<td>.95</td>
<td>.62</td>
<td>.74</td>
<td>.73</td>
<td>.62</td>
<td>.99</td>
</tr>
<tr>
<td>SSDI</td>
<td>1.03</td>
<td>.97</td>
<td>.97</td>
<td>.97</td>
<td>.97</td>
<td>.96</td>
<td>.96</td>
<td>.96</td>
<td>.97</td>
</tr>
<tr>
<td>SSFG</td>
<td>.32</td>
<td>.80</td>
<td>.80</td>
<td>.81</td>
<td>.72</td>
<td>.72</td>
<td>.86</td>
<td>.71</td>
<td>.84</td>
</tr>
<tr>
<td>SSFI</td>
<td>.42</td>
<td>.95</td>
<td>.96</td>
<td>.96</td>
<td>.96</td>
<td>.95</td>
<td>.96</td>
<td>.96</td>
<td>.96</td>
</tr>
</tbody>
</table>

Average .55 .88 .92 .90 .82 .85 .88 .82 .93

**Table 7.8:** Average efficiency of the ML models on the stencil benchmarks

As indicated by the performance distributions in Figure 4.7, Random performs very well when using ICC. Nevertheless the Weighted Rank model does consistently outperform Random, and is on average better than any individual model. Its capability to identify a good vectorized variant is illustrated in particular when using the GCC compiler, when the performance distribution offers greater challenges to the optimization selection process. For instance, for Nehalem/SSE using `float`, the Weighted Rank model outperforms all individual model by a significant margin,
Achieving 89% efficiency for an average 2.8× performance improvement over GCC’s auto-vectorization.

Table 7.9 shows the performance, in GFlop/s, obtained when using the WeightedRank model to select at compile time an effective vectorized variant for the 9 stencils. This table follows the format of Table 7.4.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Compiler (GF/s)</th>
<th>Weighted Rank (GF/s)</th>
<th>Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>avg</td>
<td>max</td>
</tr>
<tr>
<td>NSDG</td>
<td>2.17</td>
<td>3.35</td>
<td>4.12</td>
</tr>
<tr>
<td>NSDI</td>
<td>4.26</td>
<td>5.59</td>
<td>6.65</td>
</tr>
<tr>
<td>NSFG</td>
<td>3.20</td>
<td>3.78</td>
<td>4.45</td>
</tr>
<tr>
<td>NSFI</td>
<td>2.76</td>
<td>4.20</td>
<td>5.10</td>
</tr>
<tr>
<td>SADG</td>
<td>3.41</td>
<td>4.65</td>
<td>5.52</td>
</tr>
<tr>
<td>SADI</td>
<td>6.44</td>
<td>7.89</td>
<td>9.02</td>
</tr>
<tr>
<td>SAFG</td>
<td>4.40</td>
<td>5.05</td>
<td>6.13</td>
</tr>
<tr>
<td>SAFI</td>
<td>4.17</td>
<td>5.85</td>
<td>7.02</td>
</tr>
<tr>
<td>SSDG</td>
<td>3.41</td>
<td>4.66</td>
<td>5.52</td>
</tr>
<tr>
<td>SSDI</td>
<td>6.48</td>
<td>7.87</td>
<td>8.88</td>
</tr>
<tr>
<td>SSFG</td>
<td>4.36</td>
<td>5.02</td>
<td>6.14</td>
</tr>
<tr>
<td>SSFI</td>
<td>4.17</td>
<td>5.86</td>
<td>7.02</td>
</tr>
</tbody>
</table>

Table 7.9: Performance of the stencil benchmarks with WeightedRank

There is a significant performance improvement over compiler auto-vectorization, although not as high as compared to the CCSD case. The best absolute performance is attained with Sandy Bridge/AVX, using float with the GCC compiler, reaching 19 GFlop/s (35% of machine peak). For both Nehalem/SSE and Sandy Bridge/AVX
using `double`, ICC outperforms the best vectorized variant by a small margin. However, for all other cases, a performance improvement ranging on average between $1.17 \times$ and $2.86 \times$ can be observed.

The performance of these benchmarks can be improved by using optimizations better tuned for stencils, notably the multidimensional retiming. Because the machine learning models are designed to operate on assembly code, independent of the transformation algorithm, it is expected that they can be reused as-is on code which has had the multidimensional timing applied.

### 7.6 Stencil Retiming

Experimental results presented in this section, excluding hardware counters measurements, have been obtained using a four-core Intel Core i7-4770K CPU (Haswell micro-architecture) running at 3.5GHz. It features the AVX2 SIMD instruction set including fused-multiply-add (FMA) instruction. Its theoretical peak is 112 GF/s (224GF/s if using only FMA instructions). DDR3-1600 RAM was used, the STREAM benchmark [69] has a peak performance of 17.6 GB/s on the machine. Results based on hardware counters were collected on an Intel Core i7-2600k using Intel VTune. All benchmark variants, including the reference codes, were compiled with ICC 13.1.3 using the `-std=gnu99 -Wall -O3 -fp-model fast=2 -fma -override-limits -wd13383 -xHost -openmp` flags. The `-fast` flag did not improve performance.

Two sets of benchmarks are used in this section. The first is a set of synthetic benchmarks with varying stencil size and number of neighbors with non-zero weight to evaluate the effectiveness of the retiming framework on a variety of cases. 2D, 3D and 4D stencils were generated with either a `diamond`-shaped set of non-zero coefficients.
in the weight matrix, i.e., others outside the diamond are necessarily zero, or the full weight matrix being with all non-zero coefficients within a \( n \)-cube. Benchmarks are named \( \text{xD-\{d,f\}-yy} \) where \( x \) is the dimension and \( yy \) radius, i.e., the furthest non-zero weights in any direction. Diamond stencils are represented with \( d \) and full stencils with \( f \). These synthetic benchmarks are not independent of practical applications; Table 7.10 shows applications which some of them are commonly used in.

The second set of benchmarks is a subset of stencils from the \textit{Stencil Micro-Benchmarks} \cite{29}; Only those which contain enough non-zero points for retiming to be potentially profitable are considered: \texttt{Drprj3} is a 19-point stencil from NAS MG Benchmark \cite{5}, and also corresponds to the D3Q19 Lattice Boltzmann method used in CFD applications such as \cite{97}; \texttt{Dresid} is a 21-point stencil also from \cite{5}; \texttt{Dbigbiharm} is a 25-point stencil for biharmonic operator as described in \cite{1} and is used in areas of continuum mechanics; \texttt{Inoise3} is a 49-point stencil for noise cleaning as described in \cite{43} along with the next three, which can be used as part of image pipelines, e.g. \cite{85}; \texttt{Ibiglaplace} is a 97-point stencil for gradient edge detection; \texttt{Izerocross} is a 25-point stencil for edge detection; \texttt{Inevatia} is a 20-point stencil for gradient edge detection.

For each benchmark the following implementations were generated:

**Reference:** A plain C implementation of the stencil using the standard notation; no code tested was written with points rolled together as in Listing 5.1. An OpenMP pragma was added to the outer-most parallel loop to enable multicore parallelism.

**DLT-Reference:** A plain C implementation of the DLT transformed code (described below) without any explicit reuse or vectorization.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>3d-f-1</td>
<td>D3Q27 Lattice Boltzmann method [97]</td>
</tr>
<tr>
<td>2d-d-3</td>
<td>second-order-system Upwind Schemes [6]</td>
</tr>
<tr>
<td>2d-d-1</td>
<td>Jacobi 5 point</td>
</tr>
<tr>
<td>2d-f-1</td>
<td>Jacobi 9 point</td>
</tr>
<tr>
<td>Drprj3</td>
<td>NAS MG [5]</td>
</tr>
<tr>
<td>Dresid</td>
<td>NAS MG [5]</td>
</tr>
<tr>
<td>Dbigbiharm</td>
<td>Biharmonic operator [1]</td>
</tr>
<tr>
<td>Inoise3</td>
<td>Noise cleaning [43]</td>
</tr>
<tr>
<td>Ibiglaplace</td>
<td>Gradient edge detection [43]</td>
</tr>
<tr>
<td>Izerocross</td>
<td>Edge detection [43]</td>
</tr>
<tr>
<td>Inevatia</td>
<td>Gradient edge detection [43]</td>
</tr>
</tbody>
</table>

**Table 7.10:** Practical applications of the stencil benchmarks

**Optimized:** The best of all variants generated by the algorithm shown in the previous section, operating on the original layout of the data, selected with autotuning.

**DLT-Optimized:** The best variant using the DLT transformed data layout.

An issue with the rotating registers transformation is that it unnecessarily prevents auto-vectorization in ICC because of false dependences found by the compiler. For these cases outer loop vectorization using intrinsics was implemented in the code generator to ensure all the compute hardware was utilized. ICC’s inability to vectorize the code also indicates that it may be avoiding other transformations which could further improve performance of the optimized kernels, but even with these limitations substantial improvements are possible as shown below. Outer-loop vectorization comes at the cost of forcing the transposition of the innermost two loops.
which results in a sweep of memory with a larger stride which can degrade performance due to TLB misses. The DLT codes are able to utilize the original loop order with vectorization.

Tiling is implemented but not tuned to cache size, it currently only exists to enable OpenMP parallelization in tandem with the retiming. However, there is spatial locality which could be better exploited in some cases through more careful tiling.

For the unroll and code motion transformation, each loop except the inner most was unrolled by either 1×, 2×, or 4×, although the product of all unrolling was limited to at most 8×. For the retiming vectors, only cases which correspond to gathering or scatting each dimension were considered. The threshold in the cost model was set to eliminate all candidate variants with a lower arithmetic intensity than the original code. Finally, all problems were set to sizes such that each array is approximately 1GB.

The Dimension-Lifted Transpose (DLT) by Henretty et.al. is a transformation that enables effective vectorization of stencil codes by addressing the stream-alignment conflict problem [48]. It treats a dimension of the array as a 2D array with vector width rows which is then transposed, ensuring that each vector operation in the steady state is aligned. In the standard layout it is not possible to implement rotating registers if the inner most loop is the dimension of vectorization since neighboring points will be shifted by their distance from the center of the stencil, and not by the vector length. However, the DLT allows vectors to be treated as scalars within the stencil, i.e., neighboring points in the stencil on the unit stride dimension are neighboring aligned vectors in memory. This is desirable since it enables rotating registers and vectorization along the unit-stride dimension, which is optimal for both
register and cache reuse. The DLT is tested independently of the other optimization to show results on both the standard layout and the DLT.

The code generator produced each variant of a benchmark under 3 seconds. However some instances with multiple scattered dimensions resulted in exceptionally long C code being generated, and consequently, ICC regularly took over 30 minutes to compile a single function. LLVM is able to compile the largest functions produced, \( \sim 117000 \) lines of code, with `-O3 -ffast-math` in just over 5 minutes.

### 7.6.1 Performance Results

Figure 7.5a compares the maximum performance obtained by both the original stencil C code or the naïve DLT implementation (REF), with the best performing variant found after auto-tuning (original layout – OPT, DLT – DLTOPT), for each benchmark. The shorter segment of each bar shows the sequential performance which illustrates the benefit of the approach in a best-case scenario for the original code: the entire bandwidth available is allocated to a single computation unit instead of being shared between all cores. The bar stacked on top represents the absolute performance of the same benchmark executing in parallel on all four cores.

Figure 7.6 presents a similar performance comparison for the Stencil Microbenchmarks. As many of these benchmarks are still relatively small stencils or contain many zero weight points the benefit of the retiming optimization is less significant. However, `Ibiglaplace` is an excellent example of a real world computation which can substantially benefit from the retiming: in combination with the DLT the retiming achieves a 3\( \times \) improvement over the reference code.
Figure 7.5: Performance of synthetic stencil benchmarks
All problems obtain comparable or better performance than either reference code compiled with ICC. Benchmarks with near identical performance are those with the highest proportion of zero weights and the lowest order, which is where the technique is least likely to be effective. A general trend is that the more complex the stencil, i.e., the fewer zero weights in larger stencils, the better the performance of the OPT codes. ICC is able to very effectively optimize the smaller reference stencils, but the performance does not scale with the stencil order.

A direct comparison with the Array Subexpression Elimination (ASE) technique presented in [29] is not easy because the implementation is unavailable. However, a
comparison of improvements over reference implementations for sequential codes can be made with Figure 13 of Deitz’s paper. Notable stencils to compare are \texttt{Dbigbiharm} which is 40\% faster with ASE, whereas, using different hardware, it achieves over $3 \times$ improvement using retiming for sequential execution; and both the \texttt{Ibiglaplace} and \texttt{lnoise3} stencils which are $1.8 \times$ faster with ASE, but over $4 \times$ faster with retiming.

Figure 7.5b shows performance in stencils computed per second. Based on results of the Stream benchmark, the practical peak rate for any stencil is 1100 MStencil/s. If perfect reuse of all data in the benchmarks was possible, the rate of stencils/s is expected to remain flat until the problem becomes compute bound, however the reference codes demonstrate a rapid decrease as the stencil size increases, and since the GFLOP/s of these benchmarks is not near peak performance of the machine the codes have become artificially bound. This artificial bound is explained by Figure 7.7 which shows the number of loads and stores retired for each benchmark as reported by Intel VTune.

In a perfect machine there would be about the same loads and stores for all the different stencil sizes across the given problem size, but as seen the reference codes show a steady increase of loads as stencil sizes increase. This is due to the result of high register pressure which forces each value to be reloaded many times.

As an alternative view of the hardware counters, Figure 7.8 presents the memory operations per floating point operation for each benchmark, i.e., the inverse arithmetic intensity. Because the available reuse is increasing as the stencil size increases, a corresponding increase in the arithmetic intensity would be expected, but as can be seen the reference codes generally peak at about $2\text{FLOPS}/\text{MOP}$, whereas the retimed code reaches $6\text{FLOPS}/\text{MOP}$.
7.6.2 Impact of Transformations

Figure 7.9 demonstrates the improvement gained by the various transformations for the synthetic benchmarks 2d-f-4, 3d-f-1, and 3d-f-2 as speedup over the corresponding reference implementation.

For instance, for 2d-f-4 sequential using the standard layout, vectorization and application of rotating registers improves performance by 2× over reference, choosing the optimal retiming further improves performance to 3.5× better than reference, and allowing unroll and code motion, i.e., using all optimizations, the performance increases to 3.75× better. It is clear from this figure that high-order stencils can
greatly benefit from the retiming optimization and effective reuse of registers, however unroll and code motion is less useful due to the already high register pressure in higher-order stencils. Comparing between 3d-f-1 and 3d-f-2 demonstrates that while the retiming can improve both stencils, its benefit increases as the order increases.

**Figure 7.8:** Synthetic stencil benchmarks load/stores relative to FLOPS
Figure 7.9: Stencil performance improvement from stages of optimization
CHAPTER 8

Ongoing and Future Research

This chapter presents ongoing and future research goals for improving this work.

8.1 Kernel Fusion

An active area of research is expanding the space codes considered to be a sequence of operators, either stencils or tensor contractions in this work, and optimizations across those instead of viewing them as independent kernels. This creates opportunities to improve the data structures used for intermediate values and perform aggressive fusion across kernels to better utilize the cache.

8.1.1 Stencils

Exp_CNS_NoSpec [33] is “A simple stencil-based test code for computing the hyperbolic component of a time-explicit advance for the Compressible Navier-Stokes equations using 8th order finite differences and a 3rd order, low-storage TVD RK algorithm in time.”

The benchmark is composed of multiple stencils on many arrays with opportunities to cache intermediates. The code was optimized using transformations which could be implemented in a compiler:
• Spatial tiling was applied to enable better use of cache along multiple dimensions of the arrays in stencils and allow fusion of the tiling loops to reuse arrays that are part of multiple stencils.

• In one part of the code a case was found where five output arrays were being computed one at a time from a shared set of input arrays. Complete fusion was not beneficial, however redistributing the work to apply partial contributions to all five arrays over three loop nests resulted in a noticeable performance gain.

• Following the mathematical definition of the problem many intermediate arrays were created, some of which were used multiple times, but in many cases the output would only be used once. By inlining the computation of these intermediates at their use the cache pressure was reduced and performance improved.

• Other intermediates needed to be maintained to prevent redundant computation, however by implementing a sliding window optimization a much smaller intermediate was used with data which is longer needed being over written by new values.

The end result of these transformations was a code which ran 1.8 times faster than the reference code and 1.14 times faster than Halide [85].

Preliminary work has been done investigating the use of stencil inlining in combination with the multidimensional retiming to implement effective time tiling. This comes at the cost of increasing the number of floating point operations executed, but if the stencil application rate can be maintained with the retiming then multiple time steps can be done at the cost of a single time step.
8.1.2 Multidimensional Transform Operator

Listing 8.1: transform3d as individual operators

Listing 8.1 shows a sequence of three tensor contractions for the transform3d function in MADNESS. The layout of the intermediate arrays tmp1 and tmp2 is irrelevant, they are not used later, but the shown layout enable all three contractions to be implemented with index fusion using mtxm, the matrix-transpose times matrix operator which is already well optimized. However, the intermediate arrays do not need to be the same size as the input and output arrays as shown in Listing 8.2 which uses lower dimensional arrays which store partial results. Unrolling of the i and j loops can
be performed to increase revenues at the cost of increasing the size of the intermediates which provides an interesting tradeoff as cache is heavily contested MADNESS. Additionally, optimization using code generation allows for variants working on higher dimensional arrays to be considered where the memory saving can be even more important.

```c
for (i = 0; i < N; ++i) {
    for (kp = 0; kp < N; ++kp) {
        for (jp = 0; jp < N; ++jp) {
            tmp1[jp][kp] = 0;
            for (ip = 0; ip < N; ++ip)
                tmp1[jp][kp] += t[ip][jp][kp] * c[ip][i];
        }
    }
}
for (j = 0; j < N; ++j) {
    for (kp = 0; kp < N; ++kp) {
        tmp2[kp] = 0;
        for (jp = 0; jp < N; ++jp)
            tmp2[kp] += tmp1[jp][kp] * c[jp][j];
    }
    for (k = 0; k < N; ++k) {
        result[i][j][k] = 0;
        for (kp = 0; kp < N; ++kp)
            result[i][j][k] += tmp2[kp] * c[kp][k];
    }
}
```

**Listing 8.2:** transform3d with fusion

### 8.2 Symmetric Tensors

Symmetric tensors are such that interchanging the values of two indices will get find the same values. As an example, in the case of a six dimensional tensor $A$ with
two symmetric groups of size three

\[ A^{a,b,c}_{x,y,z} = A^{a,b,c}_{z,y,x} = A^{b,a,c}_{z,y,x} \] (8.1)

More commonly however is the case of anti-symmetry, in which interchanging two indices will find opposite values, for example

\[ A^{a,b,c}_{x,y,z} = -A^{a,b,c}_{z,y,x} = A^{b,a,c}_{z,y,x} \] (8.2)

Existing solutions [67] approach these symmetries by using block sparse storage of the tensors with a middle-layer that resolves symmetry when requesting a block. However, this approach lacks a regular communication pattern which is very beneficial to highly scalable performance of dense linear algebra. S. Rajbhandari’s work to appear at ICPP’14 and SC’14 presents a communication optimal algorithm for distributed systems which supports symmetric tensor contractions. However, these approaches currently rely on either storing significant redundant data on nodes or the frequent application of data layout transformations on each node to enable the use of common BLAS operations.

Developing an efficient kernel to operate on nodes requires careful selection of the data layout used. The most compact representation that can be stored does not have any duplicate elements. In the case of a symmetric matrix this is done by storing each row with an increasing width, i.e., the first row is one element, the second row is two elements, and so forth for the lower triangle of the matrix. While this is efficient from the storage perspective, it does not enable efficient SIMD operations because most rows will not be multiples of the vector width, which at a minimum will require masking values, but additionally will make vectorization on platforms with alignment constraints nearly impossible. In contrast, another approach is to just store
the complete data which is not space efficient or even feasible for higher dimensional tensors, but is very easy for vectorization.

An effective approach is to only expand blocks with size equal to the vector width within the tensor. For example, in a two dimensional symmetric tensor with SIMD vector width 8, all $8 \times 8$ blocks on or below the diagonal of the tensor would be stored. This allows for easy access to complete SIMD vectors, and if access is needed along another dimension than the unit stride dimension the register level transpose can be used within one of the blocks. This data structure is also naturally amenable to a bricked layout in which all elements of a block, e.g. $8 \times 8$, are stored consecutively in memory to improve data locality. As a proof of concept this was implemented symmetric matrix times symmetric matrix multiplication, the code is show in Listing A.1. The code was implemented with scalar operations instead of vector intrinsics and no unrolling was done, but the performance is still up to 30% better than the naïve code operating on full matrices. A challenge with further optimization of this code is the difficulties with applying unroll-and-jam given the complicated loop structure.

8.3 Tile Size Selection

The code code synthesis discussed in this paper has thus far avoided analysis of the effect of tile size selection on the performance of kernels. In some cases, specifically the kernel analysed in § 7.1, this is a non-issue because the problem is already cache resident. However, in all other cases the simple heuristic presented in § 4.5 was used without any validation that it actually produces tiles which achieve effective cache utilization. Synthesizing high performance code for tensors which are not
cache resident will require analysis of the interactions between tile sizes and existing optimized kernels from existing research.

Ideally it will be possible to compute the best tile size based solely on properties of the target architectures cache and the tensor contraction being considered. Goto [39] did this for matrix multiplication.

8.4 Variant Selection

As shown in § 7.3, the static cost model from § 6.2 is not sufficient for selecting a high performance variant in general. However, the assembly feature based machine learning models in § 6.3 are too complicated for inclusion in production compilers due to the substantially increased compile time. An ideal solution would use a model that can be represented very simply in the run time, e.g., a decision tree, and to use higher level features as inputs to the model.
CHAPTER 9

Conclusion

The work presented has explored the synthesis of a search space of optimized variants of tensor contractions and stencils and techniques for selection of a high performance variant. The vector code synthesis enables more efficient utilization of available SIMD hardware than existing state-of-the-art auto-vectorization. The vector code synthesis is shown to be applicable to both tensor contractions and stencils, but may be too restrictive in other cases. The multidimensional retiming utilizes the associativity and commutativity to reorder the execution of stencil computations which increases register reuse. However, there are still interesting cases of stencils which do not benefit from the retiming due to either a lack of associativity and commutativity or insufficiently high order. In combination with tiling and automatic parallelization scaling to many core systems has been shown. The static cost model and machine learning models demonstrated the ability to select a variant with near space-optimal performance without executing any variants of kernel. While the cost models are effective, the static cost model is limited to a specific domain of problems and the machine learning model introduces significant compile-time overhead and compile time complexity that will prevent it from being utilized in production compilers.
APPENDIX A

Code Samples

A.1 Symmetric Matrix Multiplication

Listings A.1 and A.2 show the symmetric matrix multiplication kernel discussed in § 8.2. \( N \) is the size of the symmetric matrix and \( V \) is the length of SIMD vectors.

A.2 In-Register Transpose

Listings A.3, A.4, A.5 and A.6 are the intrinsics implementations of the in-register transpose for the SSE and AVX instruction sets using either single or double precision values.
for (i=0; i<N/V; i++) {
    for (j=0; j<i; j++) { // J < I
        float *Ak = &A[V*V*(i*i+i)/2];
        float *Bk = &B[V*V*(j*j+j)/2];

        int At = V*V; int Att = 0; int Bt = V*V; int Btt = 0;
        for (k=0; k<j; k++) {
            for (ii=0; ii<V; ii++)
                for (jj=0; jj<V; jj++)
                    for (kk=0; kk<V; kk++)
                        C[i*V*N+j*V*V+ii*V+jj] += Ak[ii*V+kk]*Bk[jj*V+kk];

                    Ak += V*V; Bk += V*V;
        }

        Bt = j*V*V; Btt = V*V;
        for (; k<i; k++) {
            for (ii=0; ii<V; ii++)
                for (jj=0; jj<V; jj++)
                    for (kk=0; kk<V; kk++)
                        C[i*V*N+j*V*V+ii*V+jj] += Ak[ii*V+kk]*Bk[kk*V+jj];

                    Bt += Btt; Ak += V*V; Bk += Bt;
        }

        At = i*V*V; Att = V*V;
        for (; k<i; k++) {
            for (ii=0; ii<V; ii++)
                for (jj=0; jj<V; jj++)
                    for (kk=0; kk<V; kk++)
                        C[i*V*N+j*V*V+ii*V+jj] += Ak[kk*V+ii]*Bk[kk*V+jj];

                    At += Att; Bt += Btt; Ak += At; Bk += Bt;
        }
    }
}

Listing A.1: Bricked symmetric matrix multiplication
```c
for (; j<N/V; j++) {  // J >= I
    float *Ak = &A[V*V*(i*i+i)/2];
    float *Bk = &B[V*V*(j*j+j)/2];

    int At = V*V; int Att = 0; int Bt = V*V; int Btt = 0;
    for (k=0; k<i; k++) {
        for (ii=0; ii<V; ii++)
            for (jj=0; jj<V; jj++)
                for (kk=0; kk<V; kk++)
                    C[i*V*N+j*V*V+ii*V+jj] += Ak[ii*V+kk]*Bk[jj*V+kk];
    
        Ak += V*V; Bk += V*V;
    }

    At = i*V*V; Att = V*V;
    for (; k<j; k++) {
        for (ii=0; ii<V; ii++)
            for (jj=0; jj<V; jj++)
                for (kk=0; kk<V; kk++)
                    C[i*V*N+j*V*V+ii*V+jj] += Ak[kk*V+ii]*Bk[jj*V+kk];
    
        At += Att; Ak += At; Bk += V*V;
    }

    Bt = j*V*V; Btt = V*V;
    for (; k<N/V; k++) {
        for (ii=0; ii<V; ii++)
            for (jj=0; jj<V; jj++)
                for (kk=0; kk<V; kk++)
                    C[i*V*N+j*V*V+ii*V+jj] += Ak[kk*V+ii]*Bk[kk*V+jj];
    
        At += Att; Bt += Btt; Ak += At; Bk += Bt;
    }
}
```

**Listing A.2:** Bricked symmetric matrix multication cont.
__t0 = _mm_unpacklo_ps(v0, v1);
__t1 = _mm_unpacklo_ps(v2, v3);
__t2 = _mm_unpackhi_ps(v0, v1);
__t3 = _mm_unpackhi_ps(v2, v3);
v0 = _mm_movehl_ps(__t0, __t1);
v1 = _mm_movehl_ps(__t1, __t0);
v2 = _mm_movehl_ps(__t2, __t3);
v3 = _mm_movehl_ps(__t3, __t2);

Listing A.3:  SSE single precision transpose

__t0 = _mm_unpackhi_pd(v0, v1);
v1 = _mm_unpacklo_pd(v0, v1);
v0 = __t0;

Listing A.4:  SSE double precision transpose
Listing A.5: AVX single precision transpose

Listing A.6: AVX double precision transpose
BIBLIOGRAPHY


145


[64] Liebig, T. openEMS - Open Electromagnetic Field Solver.


[97] Weller, H. OpenFOAM.


