RICE UNIVERSITY

Autotuning Memory-Intensive Software for Node Architectures

by

Lai Wei

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Master of Science

APPROVED, THESIS COMMITTEE:

John Mellor-Crummey, Chair
Professor of Computer Science and
Electrical and Computer Engineering

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering
Professor of Computer Science and
Electrical and Computer Engineering

Vivek Sarkar
Chair and Professor of Computer Science
E.D. Butcher Chair in Engineering

Houston, Texas
May, 2015
Autotuning Memory-Intensive Software for Node Architectures

by

Lai Wei

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

John Mellor-Crummey, Chair
Professor of Computer Science and
Electrical and Computer Engineering

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering
Professor of Computer Science and
Electrical and Computer Engineering

Vivek Sarkar
Chair and Professor of Computer Science
E.D. Butcher Chair in Engineering

Houston, Texas
May, 2015
ABSTRACT

Autotuning Memory-Intensive Software for Node Architectures

by

Lai Wei

Today, scientific computing plays an important role in scientific research. People build supercomputers to support the computational needs of large-scale scientific applications. Achieving high performance on today’s supercomputers is difficult, in large part due to the complexity of the node architectures, which include wide-issue instruction-level parallelism, SIMD operations, multiple cores, multiple threads per core, and a deep memory hierarchy. In addition, growth of compute performance has outpaced the growth of memory bandwidth, making memory bandwidth a scarce resource.

People have proposed various optimization methods, including tiling and prefetching, to make better usage of the memory hierarchy. However, due to architectural differences, code hand-tuned for one architecture is not necessarily efficient for others. For that reason, autotuning is often used to tailor high-performance code for different architectures. Common practice is to develop a parametric code generator that generates code according to different optimization parameters and then pick the best among various implementation alternatives for a given architecture.

In this thesis, we use tensor transposition, a generalization of matrix transposition, as a motivating example to study the problem of autotuning memory-intensive codes for complex memory hierarchies. We developed a framework to produce opti-
mized parallel tensor transposition code for node architectures. This framework has two components: a rule-based code generation and transformation system that generates code according to specified optimization parameters, and an autotuner that uses static analysis along with empirical autotuning to pick the best implementation scheme. We also studied how to prune the autotuning search space and perform run-time code selection using hardware performance counters. Despite the complex memory access patterns of tensor transposition, experiments on two very different architectures show that our approach achieves more than 80% of the bandwidth of optimized memory copies when transposing most tensors. Our results show that autotuning is the key to achieving peak application performance across different node architectures for memory-intensive codes.
I first offer a special thanks to my advisor Prof. John Mellor-Crummey for his inspiration and feedback in my research. This thesis was shaped by John’s input and I really appreciate his support and encouragement along the way. Also, I thank Scott Warren, a research scientist in our group, for his suggestions on my research.

I thank my fellow students Xu Liu and Karthik Murthy for their time and input on this work. I also thank people in our department for our collaborative research atmosphere.

I thank my friends for the happiness and encouragement they shared with me in my daily life. These guys keep me happy and optimistic so that I can be more energetic in research.

Finally, I express my special gratitude to my parents. I won’t be here today without their trust, support, and love.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations vii
List of Tables xi

1 Introduction 1

1.1 Motivation ................................. 1
1.2 Tensor Transposition: a Memory-intensive Primitive .............. 4
1.3 Thesis Statement .......................... 5
1.4 Contributions .............................. 6
1.5 Thesis Organization ....................... 6

2 Related Work 7

2.1 Autotuning ................................ 7
2.2 Matrix and Tensor Transposition .......................... 11

3 Background 13

3.1 Spoofax Language Workbench ..................... 13
3.2 Hardware Performance Counters ....................... 14

4 Optimization Techniques for Tensor Transposition 16

4.1 Tiling .................................... 16
4.2 In-cache Buffer ............................ 17
4.3 SIMD Instructions and Non-temporal Stores .................. 18
Illustrations

4.1 One-level tiling strategy on row-major, two-dimensional $16 \times 16$ arrays. We tile the source and destination array with $4 \times 4$ tiles. $T_r$ means the tile size of the last dimension of source array while $T_w$ stands for the tile size of the last dimension of destination array. Dotted arrows show our data access pattern within a tile while concrete arrows define the tile traversal order.

4.2 Using a $2 \times 2$ in-cache buffer on $4 \times 4$ tiles. $B_r$ means the buffer size of the last dimension of source tensor while $B_w$ stands for the buffer size of the last dimension of destination tensor. Numbers represent data elements.

4.3 Prefetch strategy within a source tile (supposing row major and no in-cache buffer optimization).

4.4 Demonstration of TLB sharing with four SMT threads per core. Inner blocks with dotted borders represent first level tiles, the outer blocks with concrete borders represent the second level tiles, and numbers on inner blocks show the thread assignment. First level tiles are assigned so that the four threads will work on neighboring tiles at the same time and thus sharing the corresponding TLB entries.
4.5 Demonstration of cache sharing with four SMT threads per core.

Inner blocks with dotted borders represent each datum, outer blocks with concrete borders represent the first level tiles, and numbers on inner blocks show the thread assignment. Data are assigned so that the four threads will work on neighboring elements in the memory and thus sharing the corresponding cache entries.

5.1 Autotuning framework structure.

5.2 An example of tensor transpose specification.

6.1 Comparison between tensor transposition, TCE array transposition, and different implementations of array copy on Intel Westmere.

6.2 Result on high dimension tensor transposition on Intel Westmere. Measured on 38400 × 38400 float tensors for 2D tensor transposition and STREAM, 1536 × 625 × 1536 float tensors for 3D tensor transposition, and 384 × 100 × 100 × 384 float tensors for 4D tensor transposition.

6.3 Comparison between tensor transposition and different implementations of array copy on IBM Power7. Measured at 32 threads, on N×N float tensors.

6.4 Performance improvement through appropriate padding on IBM Power7. Measured at 32 threads, on two-dimensional float tensors.

6.5 Comparison between the best autotuned configuration and the best setting without each technique. Measured on N × N float tensors with 6 threads on Intel Westmere.

6.6 Comparison between the best autotuned configuration and the best setting without each technique. Measured on N × N float tensors with 32 threads on IBM Power7.
6.7 Comparison between the best configuration without SMT sharing,
with SMT sharing but without thread synchronization, and with
both SMT sharing and thread synchronization. Measured on $N \times N$
float tensors with 32 threads on IBM Power7. 47

6.8 Tile size search space on our two platforms. Each point represents
the bandwidth of the best configuration without in-cache buffer
optimization (as the buffer itself is another level of tiling) at the
corresponding tile size. 48

6.9 Bandwidth and L1 cache hit rate of the best configurations (without
in-cache buffer optimization) at different tile sizes. Measured on
$38400 \times 38400$ float tensors with 6 threads on Intel Westmere. 50

6.10 Performance of different optimization configurations. Measured on $N$
$\times N$ float tensors with 32 threads on IBM Power7. 51

6.11 Performance of different optimization configurations on tensors in two
different MPT levels. The five configurations are autotuned for
different tensors as shown in Table 6.4. Measured on $N \times N$ float
tensors with 32 threads on IBM Power7. 53

6.12 Illustration of the “all data in local memory” scheme using four
sockets as an example. Both length/2 and width/2 need to be
multiples of page sizes so that each sub-portion of the tensor could be
aligned with page boundaries. 55

6.13 Illustration of the “all local loads” scheme using four sockets as an
e example. Blocks in light gray stands for local memory access; blocks
in dark gray stands for remote memory access. In this illustration, in
order to have all loads be local, we assign threads in a way that only
a quarter of stores are local. 55
6.14 Multi-socket performance results on Intel Westmere. Measured on $N \times N$ float tensors with one socket (6 threads) for the leftmost bar (the blue bar) in each case and two sockets (12 threads) for all other bars. TT is short for tensor transposition in the legend.

6.15 Multi-socket performance results on IBM Power7. Measured on $N \times N$ float tensors with one socket (32 threads) for the leftmost bar (the blue bar) in each case and four sockets (128 threads) for all other bars. TT is short for tensor transposition in the legend.
### Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 A List of Machine Parameters</td>
<td>27</td>
</tr>
<tr>
<td>5.2 A List of Optimization Parameters</td>
<td>28</td>
</tr>
<tr>
<td>5.3 Exploration of Optimization Parameters</td>
<td>30</td>
</tr>
<tr>
<td>6.1 Processor Parameters and System Configurations</td>
<td>37</td>
</tr>
<tr>
<td>6.2 Hardware Performance counter measurements on Intel Westmere</td>
<td>40</td>
</tr>
<tr>
<td>6.3 Best Configurations for Different Platforms and Tensors</td>
<td>43</td>
</tr>
<tr>
<td>6.4 Information on Configurations in Fig. 6.11</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Algorithm Description</td>
</tr>
<tr>
<td>---</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>An inefficient implementation of matrix transposition.</td>
</tr>
<tr>
<td>2</td>
<td>Implementation sketch of how generated code employs software</td>
</tr>
<tr>
<td></td>
<td>prefetching within a tile (supposing row major and no in-cache buffer</td>
</tr>
<tr>
<td></td>
<td>optimization)</td>
</tr>
<tr>
<td>3</td>
<td>A sketch of generated code. It shows where each optimization</td>
</tr>
<tr>
<td></td>
<td>technique is applied to the code.</td>
</tr>
<tr>
<td>4</td>
<td>A sketch of choosing optimization configurations for different tensors.</td>
</tr>
<tr>
<td>5</td>
<td>An implementation sketch of strided array copy.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Over the last several decades, scientific computing has become important to scientific research programs. It plays an essential role in computational modeling and simulation in different research areas, where people develop applications to solve problems such as electronic structure calculation, digital signal processing, gas combustion simulation among many others. Using simulations, scientists tackle many research challenges that can’t be approached with theory or experimentation alone.

People have been building supercomputers for large-scale applications in scientific computing. These supercomputers have complex architectures, typically consisting of a set of powerful compute nodes augmented by an interconnection network and I/O subsystem. The computational performance of the top supercomputer has increased by nearly 1000 times over the past ten years. To provide higher performance, people are organizing supercomputers in more sophisticated ways. This growing complexity makes it challenging to develop software that can keep up with the growth of hardware performance.

In large part, emerging node architectures are introducing complexity to the supercomputers. Hardware designers have pursued several approaches to provide more computational power at the node level. First, some hardware designers are placing a moderate number of powerful cores on a single processor chip. In general, these
powerful cores have multiple functional units, deep instruction pipeline, and thus can provide enhanced instruction-level parallelism. In addition, they usually provide thread-level parallelism through simultaneous multi-threading (SMT). For example, an IBM Power7 \( ^3 \) chip consists of eight processors cores. Each core can dispatch six instructions and issue eight instructions per cycle. In addition, to better utilize these compute power, Power7 has a 4-way SMT implementation. Upgraded from Power7, an IBM Power8 \( ^4 \) chip has twelve cores, where each core is capable of dispatching eight instructions and issuing ten instructions per cycle. The number of SMT threads is also doubled compared to Power7.

Second, we see many chips with a large number of light-weight cores. In most cases, these light-weight cores are in-order cores with simple pipelines. Both SMT and fine-grained multi-threading are used to provide multiple threads out of these simpler cores. For example, an IBM Blue Gene/Q \( ^5 \) chip consists of sixteen in-order cores, where each core has a 4-way SMT implementation. An Oracle’s SPARC T5 \( ^6 \) chip also has sixteen cores. Each core can provide up to eight threads through fine-grained multi-threading. The Intel Xeon Phi coprocessor \( ^7 \) provides 61 cores and 244 threads on a single chip.

Third, accelerators, such as GPUs, are introducing more computational power in node architectures. For example, the NVIDIA GeForce GTX 980 GPU \( ^8 \) can issue as many as 2048 integer operations in a single cycle. Besides, IBM Power7+ \( ^9 \) comes with multiple hardware accelerators to speed up file encryption, active memory expansion, etc. Each of these approaches, while providing more computational power, is adding complexity to the emerging node architectures.

In addition to the aforementioned complexity, memory performance significantly lags behind CPU performance, which makes it much harder to achieve high perfor-
mance for data-intensive codes. CPUs outpace memory in two ways. First, the speed at which the cores of a microprocessor can issue memory requests is faster than the memory access latency, making memory unable to provide data as fast as CPU wants. This issue is getting less important in recent years because the speed of individual CPU cores hasn’t improved much and various techniques, including instruction-level and thread-level parallelism, hardware and software prefetching, are used to hide memory access latency. Furthermore, the rate at which the cores on a single processor chip can issue memory requests exceeds the limited bandwidth that the memory system can provide, making memory unable to provide data as much as CPU needs.

Memory bandwidth is increasingly a bottleneck in today’s node architectures as the number of cores and threads on a single processor chip has been growing much faster than memory bandwidth.

To bridge this growing performance gap between CPU and memory, hardware designers employ multiple levels of cache between registers and main memory. The closer that the cache is to the registers, the lower its access latency, the higher its bandwidth, and the smaller its capacity. For data-intensive codes, rather than optimizing the computation, making good use of the memory hierarchy is the key to the best performance. Specifically, according to the Roofline model [10], low operation intensity applications, ones that have few arithmetic operations per datum, are memory-bound. Tuning the performance of such codes requires careful use of memory optimizations; other optimizations that focus on the computation, such as employing instruction-level parallelism, won’t provide much benefit.

To date, people have proposed various memory optimizations, including tiling and software prefetching, to improve utilization of the memory hierarchy. By employing these optimizations carefully, one may manually tune a piece of code for a specific
architecture. However, a code optimized for one platform is not necessarily efficient for other platforms due to architectural differences. Therefore, autotuning is often used to tailor high-performance code for two reasons. First, due to the growing complexity of emerging architectures, many hardware implementation details are not visible to programmers, making it hard to manually tailor a program for an architecture. For example, J. McCalpin \[11\] reports that there are twenty-five different kinds of locality in the memory hierarchy of AMD Operton processors; however, most programmers only know a few of them, such as cache and TLB locality. In contrast, autotuning can explore a wide optimization space to find out the best optimization setting without much knowledge about the architecture. Second, autotuning is capable of optimizing code across various platforms efficiently, which saves much time compared to hand-tuning the code for every architecture. People usually implement autotuning with two components. One component is a parametric code generator or optimizer that generates or optimizes code according to different parameters. The other component is an autotuner that picks the best among various implementation alternatives for a given architecture by analyzing empirical measurements.

1.2 Tensor Transposition: a Memory-intensive Primitive

In this thesis, we use tensor transposition, a generalization of matrix transposition, as a motivating example to study the problem of autotuning memory-intensive codes for complex memory hierarchies. Tensor transposition has low operation intensity, as there are only a few address calculations related to each datum, and thus it is a memory-bound kernel. Besides the fact that it is representative of memory-intensive codes, tensor transposition is an important primitive used when performing tensor contraction as it enables the transformation of multi-dimensional tensors to be used as
\begin{algorithm}
\begin{verbatim}
for \( i = 0 \) to \( \text{row\_size} \) do
    for \( j = 0 \) to \( \text{col\_size} \) do
        \( \text{dst}[j][i] = \text{src}[i][j] \)
\end{verbatim}
\end{algorithm}

Algorithm 1: An inefficient implementation of matrix transposition.

input to matrix multiplication and has a potential to reduce matrix multiplication cost
[12]. It accounts for around 30\% of the running time of NWChem Tensor Contraction
Engine’s CCSD module [13], which computes the ground-state energy.

Data access patterns of tensor transposition lack temporal locality, making it
hard to implement the operation efficiently. Naive implementations (Algorithm 1)
that stride through the data space in a straightforward fashion cause high cache and
TLB miss rates. They are unable to make full use of available memory bandwidth.

To accelerate tensor transposition, one can employ techniques including tiling,
SIMD instructions, and software prefetching. Efficient implementation of tensor
transposition requires careful choreography of these techniques. To achieve the best
performance, one needs to consider an architecture’s characteristics, such as its mem-
ory hierarchy and vector operations, to pick the best set of techniques and decide
their parameters accordingly.

1.3 Thesis Statement

Achieving high performance on today’s node architectures is challenging for data-
intensive applications with complex memory access patterns. Autotuning is needed
for tuning such applications to fully exploit the memory hierarchy in these systems to
achieve top performance. To effectively autotune tensor transposition, one needs to
devise a set of code optimizations that manage key machine resources, a parametric
code generator that generates optimized code versions, and an autotuner that picks the best optimized code through parameter search accelerated by feedback from timers and performance monitoring units. It is also important to use an appropriate input model for multi-version code generation so that generated code can achieve high performance across various inputs by selecting the best code version at run time.

1.4 Contributions

We developed a framework that produce optimized parallel tensor transposition code for node architectures. This framework has two components: a rule-based code generation and transformation system that generates code according to specified optimization parameters and an autotuner that uses static analysis along with empirical autotuning to pick the best implementation scheme. We studied how to prune the autotuning search space and perform run-time code selection using run time and hardware performance counters. Despite the complex memory access patterns of tensor transposition, experiments on two very different architectures show that our approach achieves more than 80% of the bandwidth of optimized memory copies when transposing most tensors.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces existing work on autotuning as well as efficient matrix and tensor transposition. Chapter 3 provides background information on some of the techniques we used. Chapter 4 discusses optimization techniques that address several implementation challenges of tensor transposition. Chapter 5 describes the structure of our autotuning framework. Chapter 6 presents our results and Chapter 7 summarizes our conclusions.
Chapter 2

Related Work

Our work builds on previous effort on autotuning as well as efficient matrix and tensor transposition. We go through these effort in the next two sections.

2.1 Autotuning

People have been using autotuning to optimize applications in scientific computing. In these applications, there could be many different implementations for the same computational work. To pick a good implementation out of this huge search space, people may build a static analysis model to predict the performance of different implementations. However, building an accurate model is generally infeasible due to hardware complexity. As a result, besides using a model to predict part of the performance or prune some of the search space, people need to build an autotuner to carry out some empirical measurements and use them to direct search space exploring and pruning.

SPIRAL \cite{14} is a framework that generates high performance code for linear digital signal processing transforms. It uses rewrite rules to break down transformation formulas recursively into base case ones that map well to hand-tuned implementations for a certain platform. During the rewriting, when a formula matches the left side of a rewriting rule, the rule may or may not apply, resulting in different ruletrees. To find a good ruletree out of its companions, the framework generates a random set of
ruletrees according to the input transform and collects the run time of each node in these ruletrees. The framework can then use collected information to predict the run time of other ruletrees and thus generate fast implementation for the input transform.

The Tensor Contraction Engine [15] is a DSL compiler that transforms tensor contraction specifications into FORTRAN implementations. To construct an optimized implementation, they decompose the computations into its constituent operations, model the cost or empirically measure the cost of these constituent operations, and then estimate the execution time of the whole computation. When measuring the cost of those constituent operations, they invoke the operations with different parameter settings to find out the best one. This result is then used to guide code optimization.

FFTW [16] is a library for computing discrete Fourier transform (DFT). It uses a planner to adapt itself for maximum performance on different hardware. After the user has specified the problem, the planner invokes rules to recursively decompose the problem into simpler sub-problems until the sub-problem can be solved with simple, pre-generated code. Since multiple rules could apply to the same problem, there are an exponential number of possible decompositions. To choose a good decomposition within reasonable time, they adopted a dynamic programming approach where they optimize each sub-problem locally through empirical measurement regardless of the context. This strategy is able to find a good solution fast, although not guaranteed to find the best one.

Halide [17] is a language and compiler developed for optimizing image processing pipelines. To achieve the best performance, the compiler needs to address the trade-off between the amount of intermediate data passed from a pipeline stage to the next, the dependency among the intermediate data within a pipeline stage, as well as amount of redundant computation performed. The compiler uses schedules, which determine
the order of allocation, communication, and execution of each pipeline stage, to make this trade-off. To achieve good performance, the compiler uses an autotuner to find a good schedule by using domain knowledge to select a good search start point and applying a genetic search algorithm thereafter.

All of the above autotuning frameworks successfully address efficient autotuning within their own application domain. However, more work is needed to extend their approach to other domains. As a result, we see some effort towards efficient autotuning framework for a spectrum of applications.

Active Harmony [18] is an automated tuning system. It tunes the code at run time by selecting among different libraries with the same or similar functionality and choosing appropriate parameter settings for the selected library. They make use of the Nelder-Mead simplex method [19] when tuning library parameters. For programs that use the same library API in a lot of iterations, the system can use run time information collected from previous iterations to help improve the performance of future ones.

Vuduc et al. [20] proposed statistical models for efficient autotuning and run-time code selection. They applied their statistical models on matrix multiplication. Their results show that their proposed autotuning model can stop the autotuning process early after getting sufficiently optimized code to cut off the autotuning time. They also proposed three statistical models for run-time code selection. However, these three models don’t yield much performance improvement.

Williams proposed a Roofline model [10] to understand the possible performance of an application for a specific architecture. First, the Roofline model helps people predict the peak performance of an application according to its flop/byte ratio. Second, the Roofline model also shows the best performance that can be achieved
without certain optimizations or in certain circumstances. This helps people analyze the application and add appropriate code optimizations to improve the performance. By taking advantage of the Roofline model, Williams successfully autotuned two important kernels, Lattice Boltzmann Magnetohydrodynamics and sparse matrix-vector multiplication, in this dissertation [21].

Pan and Eigenmann [22] looked at fast autotuning for compiler optimizations. They build three different rating methods to rate compiler optimizations for different code sections in the application. For code sections with different complexity, they are able to choose an appropriate rating method for a good balance between the accuracy of rating and the consumed amount of time. As a result, they are able to achieve good performance improvement with reduced program tuning time.

Cavazos et al. [23] proposed another way to rapidly select compiler optimization settings to improve the performance. By running many programs offline and collecting their hardware performance counter measurements, they learn a model that map a program’s hardware performance counter measurements into appropriate compiler optimization settings for this program. Their approach is able to achieve good performance improvement with only a few program executions.

Build upon the above work, we explore efficient autotuning for tensor transposition by using the feedback of timers and hardware performance counters. Our work is a step towards efficient autotuning of memory-intensive software where one can use run time and hardware performance counter measurements to learn the influence of different optimization parameters on the memory hierarchy and then select a good set of optimizations.
2.2 Matrix and Tensor Transposition

Our work on tensor transposition builds on prior research on efficient matrix and tensor transposition.

Chatterjee and Sen [24] designed six in-place matrix transposition algorithms according to different memory models to assess the relative contributions of data cache, TLB, register tiling, and array layout. They concluded that using non-linear array layouts can improve performance due to fewer cache and TLB misses. However, non-linear layouts are infeasible in many scientific computations as they slow down other operations within the application.

Hammond [13] experimented on different loop orders for 4D tensor transposition, observing that the optimal loop order improve performance by $2-8\times$. The best loop order enables cache reuse by minimizing the distance between two neighboring strided accesses, thus improving performance.

Gatlin and Carter [25] considered the special case when access patterns for matrix transpose and bit-reversal cause cache and TLB conflict misses when array dimensions are large powers of two. They proved that their in-cache buffer algorithm is optimal for cache and near-optimal for TLB in this special case.

Lu et al. [26] employed various optimization techniques and combined static analysis with empirical measurement to tune matrix transposition. Although the Tensor Contraction Engine [15] uses libraries that they generated for tensor transposition, their approach has several drawbacks. First, they only considered square tiles instead of rectangular ones. Second, their autotuning process could be time consuming as they didn’t employ any dynamic parameter search space pruning at autotuning time. Third, they didn’t address the issue that the best parameter settings could be different for different input matrices. As a result, although their generated code performed
well on Intel architectures with SSE, it achieves less than half of the bandwidth of memcpy on a PowerPC G5.

In this work, we focus on approaches for out-of-place transposition. However, there is a large body of literature on in-place matrix transposition that employs different strategies, e.g., following “cycles” of replacements; Gustavson et al. [27] provide an overview of in-place approaches.

In conclusion, while there are many work on efficient out-of-place matrix and tensor transposition, none of them addresses the problem of efficient tensor transposition across today’s node architectures. Therefore, based on these previous work, we build an autotuning framework for tensor transposition that addresses multiple issues to generate efficient tensor transposition code.
Chapter 3

Background

In our work, we implemented our parametric code optimizer using the Spoofax Language Workbench [28]. It provides us many convenient features to implement a rule-based code transformation system. We will briefly introduce it in Section 3.1.

In addition, we used hardware performance counter measurements to help prune the parameter search space during the autotuning time. We provide an overview of hardware performance counters in Section 3.2.

3.1 Spoofax Language Workbench

The Spoofax Language Workbench [28], or Stratego/XT as the old name, provides a foundation for the development of domain-specific languages (DSL). It has support for specifying the syntax of DSL and host language, parsing programs into abstract syntax trees (AST) according to the syntax, and transforming DSL ASTs into host ones.

Using Spoofax, DSL developers need to write transformation rules to transform DSL ASTs to host ones. They can express rewrite rules in plain-text instead of specifying rules in the form of AST rewritings. For example, the developer could say “(a + b) + c ⇒ a + (b + c)” instead of specifying something like “ASTPlus(ASTPlus(a,b), c) ⇒ ASTPlus(a, ASTPlus(b,c))”, which greatly eased writing transformation rules.

By combining transformation rules in different ways, one can easily transform
DSL programs into host ones with the Spoofax Language Workbench. In our work, we express rewrite rules in a way that enables us to select which optimizations to apply by invoking the rules with different parameters to generate code with different optimizations.

3.2 Hardware Performance Counters

In order to make it easier for software developers to optimize programs, hardware manufacturers build hardware counters into processors to measure information about various hardware-related activities. These activities could be thread-level activities, such as total number of cycles, number of retired floating point operations and L1 cache misses, all the way to chip-level ones such as number of memory reads and other bus traffic.

Besides the ability to count certain hardware events, IBM Power7 provides two additional performance monitoring features [29]. First, it provides support for a hierarchical break down of CPU cycles. The total running cycles can be decomposed into completion stall cycles, global completion table empty cycles and completion cycles, which can be further decomposed into more detailed categories. For example, part of the completion stall cycles can be load-store unit (LSU) stall cycles, which are broken down into data cache miss stall cycles, TLB miss stall cycles, etc. Second, Power7 provides enhanced profiling support through marked instructions. The hardware can track the execution of these marked instructions and record up to 32 events with certain information during the execution. Software developers can then use those records to pinpoint the performance bottlenecks in the program.

In our work, we use hardware performance counters to help prune autotuning search space and perform run-time code selection. Our use of hardware performance
counter measurements to guide autotuning is described in Sections 5.4 and 5.5.
Chapter 4

Optimization Techniques for Tensor Transposition

Tensor transposition is a memory bound operation with no temporal reuse, where only a few address calculations are related to each load and store. To produce optimized implementations of tensor transposition, we developed a framework that integrates support for a spectrum of optimization techniques that help improve utilization of the memory hierarchy.

4.1 Tiling

People have been using tiling \[30,31\] to improve data reuse since 1980s. For efficient implementation of tensor transposition, we employ multi-level tiling \[32\] to exploit spatial locality to avoid cache and TLB misses. We tile along the last dimension of source and destination tensors. Fig. 4.1 shows a one-level tiling where we split the source and destination arrays into four tiles. When traveling from tile to tile, we sweep down a column of tiles in the source array and across a row of tiles in the destination array. For traffic within each tile, our generated code strides down a tile column in the source tile to collect data to be written contiguously across a row in the destination tile. Such data access patterns enable the hardware to combine stores of adjacent words into a single cache line write, which is faster than issuing stride-1 loads and strided stores. In Section 6.1 experiments show that the best tiling configuration, including the number of levels in a multi-level tiling and tile sizes at
We tile the source and destination array with $4 \times 4$ tiles. $T_r$ means the tile size of the last dimension of source array while $T_w$ stands for the tile size of the last dimension of destination array. Dotted arrows shows our data access pattern within a tile while concrete arrows define the tile traversal order.

each level, is machine and tensor dependent.

### 4.2 In-cache Buffer

While tiling improves the number of cache hits in most cases, performance may degrade when tensor dimensions are large powers of two due to cache conflict misses. To address this problem, we employ an in-cache buffer optimization [25].

Fig. 4.2 shows how our in-cache buffer optimization works. The program divides each tile into panels of buffer size, transposes each panel from the source tile to the in-cache buffer, and copies each panel from the in-cache buffer to the destination tile. The program issues stride-1 loads for the source panel and strided stores for the in-cache buffer during the transpose, and stride-1 loads and stores when copying a panel.
Figure 4.2: Using a 2×2 in-cache buffer on 4×4 tiles. \( B_r \) means the buffer size of the last dimension of source tensor while \( B_w \) stands for the buffer size of the last dimension of destination tensor. Numbers represent data elements.

from the buffer to the destination tile. With appropriate buffer sizes, the program can eliminate cache conflict misses resulting from strided accesses to the source tensor.

Besides eliminating cache conflict misses, the memory access pattern when using an in-cache buffer consists of bursts of contiguous loads or stores. This could lead to better memory utilization by reducing memory turn-around overhead, since bursts consist of only loads or stores. However, using an in-cache buffer increases the number of data loads and stores in a transpose (since we load and store to the buffer in addition to the source and target tensors). Therefore, whether using an in-cache buffer is beneficial or not can depend upon tensor size and architectural characteristics. The autotuner decides it by using empirical measurements and determines the best size of the buffer as well.

4.3 SIMD Instructions and Non-temporal Stores

Most modern processors provide SIMD instructions, which have the potential to accelerate rearrangement of the data elements. Using SIMD instructions not only reduces
the number of instructions, but also provides the important capability of non-temporal stores.

Non-temporal stores go directly to memory. They don’t pollute cache and can save memory bandwidth by eliminating unnecessary loads of destination elements. For that reason, our framework considers using SIMD loads and stores as an optimization strategy when generating code. We use Eklundh’s algorithm for SIMD transposition in our vector implementation. The autotuner decides whether to exploit a processor’s SIMD and non-temporal store capabilities or not according to empirical performance measurements.

4.4 Unrolling

Even with the optimal tile size and in-cache buffer configuration, tensor transposition still suffers from cache and TLB misses. Our framework can unroll the innermost loop to provide more instruction-level parallelism to hide cache miss penalties. Our autotuner determines the appropriate unroll degree.

4.5 Software Prefetching

Most modern processors use hardware prefetchers to hide cache miss latencies. However, with our tiling strategy, strided loads usually result in too many data streams for hardware prefetchers to track. Therefore, our framework can issue software prefetch instructions to hide cache miss latencies. As shown in Fig. 1.3 and Algorithm 2, the framework prefetches the next cache line when loading the first element of a cache line. It unrolls the code in units of cache line sizes to avoid excessive branch instructions. Our framework currently doesn’t have support for cross-tile prefetches, making the prefetch strategy inefficient for tiles with a small $T_r$. The autotuner determines
Algorithm 2: Implementation sketch of how generated code employs software prefetching within a tile (supposing row major and no in-cache buffer optimization).

if it is beneficial to generate software prefetch instructions.
Figure 4.4: Demonstration of TLB sharing with four SMT threads per core. Inner blocks with dotted borders represent first level tiles, the outer blocks with concrete borders represent the second level tiles, and numbers on inner blocks show the thread assignment. First level tiles are assigned so that the four threads will work on neighboring tiles at the same time and thus sharing the corresponding TLB entries.

4.6 Resource Sharing among SMT Threads

Simultaneous multithreading (SMT) “is a technique permitting several independent threads to issue instructions to a superscalar’s multiple functional units in a single cycle.” SMT threads on the same processor core share hardware resources such as functional units, cache, TLB, etc. We have seen some effort towards efficient cache sharing among SMT threads.

For parallel executions of tensor transposition, if we have each thread work independently from other threads on the same core, they may compete against each other for hardware resources, which in turn could degrade performance. Therefore, we considered making these threads work cooperatively so that they share some hardware resources instead of competing for them.

Our framework encompasses two different resource sharing strategies. In the first
strategy, as shown in Fig. 4.4, the SMT threads on the same core share TLB entries by working on neighboring first level tiles. For large tensors, SMT threads collaborating in this way require a total of $2 \times T_{w1}$ TLB entries in the source tensor and $2 \times T_{r1}$ TLB entries in the target tensor. However, if SMT threads are working independently, each of them needs $T_{w1} + T_{r1}$ TLB entries and the number may sum up to $4 \times (T_{w1} + T_{r1})$. Having SMT threads working on neighbouring tiles can release the pressure on TLB and help reduce the number of TLB misses.

In the second strategy, as shown in Fig. 4.5, the SMT threads on the same core share the cache by working on neighboring data within the same tile. With this strategy, the four SMT threads only need a total of $T_{w1} + 4$ cache lines in the
ideal case, which is much fewer than having four threads working independently as it requires a total of $4 \times ( T_{w1} + 1 )$ cache lines. Therefore, this strategy may help reduce the number of cache misses by allowing SMT threads to share the cache lines.

In order to get the most out of the above strategies, the SMT threads may need to be well synchronized to prevent any one from getting beyond or behind the others. Therefore, we added lightweight synchronization support between SMT threads. The four threads could get synchronized at the end of the first level tile in both strategies. As the first level tiles usually contain more than one thousand elements, the synchronization cost is small compared to the total running time. We also allow some threads to work a little beyond or behind the other threads by controlling the synchronization window width. For example, if the synchronization window width is two, a thread can work two tiles beyond or behind the other threads. The autotuner determines if it is beneficial to apply these two strategies, and if it is, the autotuner tries adding synchronization among SMT threads and figure out the best synchronization window size.
Chapter 5

Autotuning Tensor Transposition

To generate high performance code for tensor transposition, we developed an autotuning framework. Fig. 5.1 shows the structure of our fully automatic tuning framework. It needs two inputs: a tensor transpose specification and a machine model. Code generation rules output an implementation sketch as machine-independent code and code transformation rules instantiate the sketch to produce machine-dependent code. These two rule sets need input parameters from the autotuner, which determines the range of parameter settings, explores the parameter space through empirical measurements, and identifies the best parameter setting.

5.1 DSL Specification and Code Generation

Using the Spoofax Language Workbench [28], we developed a small DSL and a set of rules to generate the code. As shown in Fig. 5.2 users must define the type of array element, the name of the generated function, and the number of dimensions of the tensors that the function will transpose.

The rules consist of two parts. Code generation rules generate a code sketch according to program specification and optimization parameters including tile size, unroll degree, etc. Each optimization technique in Chapter 4 is implemented with independent rules so that the rule system can choose which optimization techniques to apply by employing the corresponding rules. Code generation rules generate SIMD
**Figure 5.1**: Autotuning framework structure.

```plaintext

tensor element type = float
function = {
  function name = tensorTransposition2D
  number of dimensions = 2
}
```

**Figure 5.2**: An example of tensor transpose specification.

```plaintext
for each second_level_tile do
  // May perform data assignment among SMT threads
  for each first_level_tile do
    for each in_cache_buffer_unit do
      for each element OR simd_width_unit do
        // May unroll this loop
        // May add software prefetch instructions
        Load instructions
        // May use SIMD permutation instructions
        Stores instructions
        // May perform light-weight synchronization among SMT threads

Algorithm 3**: A sketch of generated code. It shows where each optimization technique is applied to the code.
and prefetch instructions as abstract operations. Code transformation rules then convert these abstract representations into concrete ones appropriate for the target processor. Such a division of work responsibility enables us to quickly adapt our framework to new platforms. Algorithm 3 shows a sketch of generated code as a result of applying these rules. Appendix A contains complete examples of generated code.

Our framework presently generates code for row-major tensors. It would be easy to add support for column-major ones. We have noticed that using non-linear array layouts, such as Morton order [37], could better exploit spatial locality. However, these layouts are less frequently used than row-major or column-major ones. Under most circumstances, our generated code would be provided as a library to scientific applications such as NWChem [38]. These applications generally use row-major or column-major arrays when calling tensor transposition library. As a result, we focus on generating efficient row-major tensor transposition code.

5.2 Machine Model

Our framework uses a machine model to generate machine-dependent code and prune the parameter space. Table 5.1 shows the machine parameters and their utility. Currently, the user needs to specify these parameters in a configuration file when installing the autotuning framework on a new architecture. The code transformation rules turn abstract representations into concrete ones suitable for the target processor. The tester uses the number of cores and number of threads per core to run the code in parallel. To generate appropriate parameters for optimization techniques in Chapter 4, the autotuner will need the number of threads per core, SIMD width, availability of non-temporal stores and software prefetch modes as input. The last two
### Table 5.1: A List of Machine Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target processor</td>
<td>Help decide which set of transformation rules to apply.</td>
</tr>
<tr>
<td>Number of cores</td>
<td>Help the tester run the code in parallel.</td>
</tr>
<tr>
<td>Number of threads per core</td>
<td>Help the autotuner decide whether to introduce resource sharing optimization or not.</td>
</tr>
<tr>
<td>SIMD width</td>
<td>Help the autotuner determine the parameters of various optimizations.</td>
</tr>
<tr>
<td>Availability of non-temporal stores</td>
<td>Help the autotuner prune the optimization configuration space.</td>
</tr>
<tr>
<td>Software prefetch modes</td>
<td></td>
</tr>
<tr>
<td>Cache size at different levels</td>
<td></td>
</tr>
<tr>
<td>Cache line size at different levels</td>
<td></td>
</tr>
</tbody>
</table>

set of cache parameters help the autotuner prune the optimization configuration space and we will see their usage in Section 5.4. We currently don’t require information on cache associativity as our autotuner is able to handle cache conflict misses without such information.

### 5.3 Runtime Measurements

The tester adds a function with OpenMP constructs to the machine dependent code to perform multi-threaded measurements. The tester divides the source and destination tensors evenly among the threads. When assigning data blocks to each thread, the tester avoids blocks with incomplete tiles.

The tester measures the running time and uses a processor’s hardware performance monitoring units to measure counts of important events, such as L1 cache misses. Execution time helps the autotuner choose among various configurations while hardware
Optimization Parameters

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiling</td>
<td>First level tile size $T_w$ and $T_r$</td>
</tr>
<tr>
<td></td>
<td>Second level tile size $T_{w2}$ and $T_{r2}$ if introducing two-level</td>
</tr>
<tr>
<td></td>
<td>tiling</td>
</tr>
<tr>
<td>In-cache buffer</td>
<td>Buffer size $B_w$ and $B_r$</td>
</tr>
<tr>
<td>SIMD</td>
<td>Use SIMD instructions or not</td>
</tr>
<tr>
<td>Non-temporal stores</td>
<td>Use non-temporal stores or not if SIMD is enabled</td>
</tr>
<tr>
<td>Unrolling</td>
<td>Unroll degree</td>
</tr>
<tr>
<td>Software prefetch</td>
<td>Prefetch modes or turn off</td>
</tr>
<tr>
<td>Resource sharing</td>
<td>Resource share modes or turn off</td>
</tr>
</tbody>
</table>

Table 5.2: A List of Optimization Parameters

Event counts help us understand the rationale for the result and help the autotuner prune the search space.

5.4 Autotuning Search Space Exploration

The autotuner works in the following fashion. First, it determines the appropriate configuration of various optimization techniques. A list of parameters is shown in Table 5.2. The autotuner then directs the code generation rules to generate an implementation sketch according to the optimization configuration and invokes transformation rules to produce machine-dependent code. After that, the autotuner gets the runtime data for the current configuration from the tester and picks the next optimization configuration. In the end, after the autotuner has evaluated the empirical measurements for a set of configurations, it selects the best parameter settings for the final output.

During the above process, how the autotuner explores the optimization configura-
tion search space is the key to generating good code. Due to the vastness of the search space, where there could be about \(N \times M\) different tile size configurations for an \(N \times M\) tensor, it is impossible to explore every valid optimization configuration to guarantee finding the best code. Therefore, our autotuner uses both static and dynamic search space reduction to prune the search space and picks the best configuration in the pruned search space.

### 5.4.1 Static Search Space Reduction

Our autotuner statically prunes the parameter search space according to the machine model. Suppose the source and destination tensor are cache aligned, we define \(L_1 = \) L1 cache line size, \(L_{LL} = \) last level cache line size, \(C_1 = \) L1 cache size, \(C_{LL} = \) last level cache size, and \(K = \) size of each data element, then we should have:

- \(T_r, T_w, B_r, B_w\) being multiples of \(L_1/K\), as having the boundary of each tile and each buffer unit aligned with the cache line minimizes the number of cache misses.

- \(T_w \leq C_{LL}/L_{LL}\). If not, the last level cache will be unable to hold all the cache lines introduced by strided loads, leading to an increasing number of cache misses and inefficient memory bandwidth usage.

- \(B_r \times B_w < C_1/K\). If not, the in-cache buffer will not fit in the L1 cache.

Then, instead of exploring every valid parameter in the pruned search space, our autotuner explores it as shown in Table 5.3. In this exploration scheme, although we would possibly miss the best configuration, we would still explore ones that are close enough to the best. The benefit of this exploration strategy is that it greatly
<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiling</td>
<td>Start with one-level tiling:</td>
</tr>
<tr>
<td></td>
<td>Start $T_w$ at $L_1/K$, and increase $T_w$ by factors of two up to $C_{LL}/L_{LL}$.</td>
</tr>
<tr>
<td></td>
<td>Start $T_r$ at $L_1/K$, and increase $T_r$ by factors of two up to the dimension size.</td>
</tr>
<tr>
<td></td>
<td>If observed high DTLB misses with one-level tiling, consider two-level tiling. The outer tile could be 2, 4 or 8 times larger in each dimension compared to the inner tile.</td>
</tr>
<tr>
<td>In-cache buffer</td>
<td>Try to use in-cache buffer or not.</td>
</tr>
<tr>
<td></td>
<td>Try buffer size that satisfies $B_r \times B_w &lt; C_1/K$:</td>
</tr>
<tr>
<td></td>
<td>Start $B_w$ at $L_1/K$, and increase $B_w$ by factors of two up to $T_w$.</td>
</tr>
<tr>
<td></td>
<td>Start $B_r$ at $L_1/K$, and increase $B_r$ by factors of two up to $T_r$.</td>
</tr>
<tr>
<td>SIMD</td>
<td>Try to use SIMD instructions or not.</td>
</tr>
<tr>
<td>Non-temporal stores</td>
<td>When SIMD is enabled, use non-temporal stores or not.</td>
</tr>
<tr>
<td>Unrolling</td>
<td>Start unrolling degree at 1, and increase it by factors of two five times.</td>
</tr>
<tr>
<td>Software prefetch</td>
<td>Generate prefetch instructions or not. Try different prefetch modes if available.</td>
</tr>
<tr>
<td>Resource sharing</td>
<td>Try to introduce resource sharing among SMT threads or not. Try two resource sharing strategies. Try if it is better to add synchronization among SMT threads.</td>
</tr>
</tbody>
</table>

Table 5.3 : Exploration of Optimization Parameters

reduces the number of configurations in the autotuning search space and makes the autotuning process feasible even without further dynamic pruning.
5.4.2 Dynamic Search Space Pruning

In order to further reduce the autotuning runtime, our autotuner is capable of pruning the configuration search space according to the empirical measurements. The autotuner dynamically prunes the search space in two ways.

On one hand, our autotuner dynamically reduces the search space of $T_w$ and $T_r$. For $T_w$, our static analysis shows that it should not exceed $C_{LL}/L_{LL}$ so that the all cache lines introduced by strided loads would be kept in the last level cache at least. However, the value of $T_w$ would better be smaller in most cases so that the cache lines can fit in the L1 or L2 cache. Therefore, our autotuner uses the runtime and hardware performance counts to figure out a better upper bound for $T_w$. The autotuner starts at $T_w = L_1/K$, which is the lower bound. It then increases $T_w$ by a factor of four until we get to $T_w = W$, where the autotuner observes that, for a set of configurations, the runtime and the number of cache misses at some cache level increased dramatically compared to smaller values of $T_w$. With such an observation, the autotuner knows that it is better to keep $T_w$ less than $W$ to avoid cache miss penalties of some specific cache level. In cases where $W$ is a small multiple of $L_1/K$, for example 16, the autotuner explores $T_w$ in factors of two instead of four to travel the search space in more detail.

The autotuner uses a similar strategy on $T_r$. While $T_r$ could be as large as the dimension size of the tensor, it may be better to keep it smaller on some architectures. Therefore, the autotuner starts at $T_r = L_1/K$, walking all the way through $T_r = R$, where the runtime increases dramatically compared to smaller values of $T_r$. The autotuner then verifies if the same observation holds for even larger values of $T_r$. If so, the autotuner sets $R$ as an upper bound for $T_r$ and explores $T_r$ in factors of two if $R$ is a small multiple of $L_1/K$. 
On the other hand, our autotuner reduces the search space by figuring out if some parameters are less important. By default, the autotuner will try every possible combination of those parameters. However, some of these parameters may be less important in some cases. If the autotuner observes less than 2% improvement in running time and cache hit rate when applying an optimization technique, it will mark the corresponding optimization parameters as less important. The autotuner then first explores the search space without these less important parameters. Later, the autotuner tunes these less important parameters only on a small set of best configurations selected in the first step. This strategy can prune a lot of search space without sacrificing the performance of the optimized code. Suppose there are N kinds of parameter settings for important optimizations and M kinds of parameter settings for less important ones, this strategy reduces the number of configurations from $N \times M$ to $N + M \times K$, where $K$ is the size of the small set of best configurations selected in the first step.

Dynamic pruning can greatly reduce the number of configurations in the autotuning search space and the autotuner can find the best configuration within only a few hours.

5.5 Run-time Code Selection

If we follow the autotuning process mentioned in the previous sections, we will be able to generate optimized code for a specific tensor size. However, it is possible that the optimized code for one tensor is not efficient for others. To achieve good performance across different tensors, our autotuner can generate code that encompasses a set of different optimization configurations and picks the best one for the input tensor at run time. We found out that two properties of tensors could yield performance difference
for different tensors.

First, the size of the tensors is an important factor. For example, tiling might not be important for small tensors that fit in the cache. On the other hand, for tensors that don’t fit in the last level cache, relatively smaller ones will not result in many TLB misses while sufficiently large ones will present a high pressure on the TLB. Therefore, the best configuration could differ for tensors in different size.

Second, the “multiples of powers of two (MPT) level” of the tensors is the other important factor. For a K×M tensor, if M a multiple of $2^N$ but not a multiple of $2^{N+1}$, we define the MPT level of this tensor as N. For example, the MPT level of a 24×24 tensor is 3 as 24 is a multiple of $2^3$ but not a multiple of $2^4$. With this definition, if the MPT level of a tensor is large, meaning that the tensor size is large powers of two, cache conflict misses will become an issue while we won’t observe these misses for tensors with small MPT levels. As a result, the best configuration could differ for tensors with different MPT levels as they may present distinct pressure on limited associativity of caches and TLBs.

Therefore, our autotuner is capable of finding good optimization configurations for tensors with different size and MPT levels. Algorithm 4 shows how the autotuner chooses the configurations for different tensors. For each MPT level, the autotuner first tries to apply the optimization configuration of the previous level. If it works fine, the autotuner uses it directly instead of selecting another optimization configuration through autotuning, which is usually time-consuming. If not, the autotuner used the best configuration for a random tensor that satisfies the MPT level and size range requirement and see if this configuration works well on other requirement satisfying tensors. If this new configuration can be inefficient for other tensors, the autotuner will split the size range and choose optimization configurations separately for each
half of the range.

After having figured out the best configuration for different MPT levels and size ranges, the autotuner generates code according to the `resultSet`. The generated code will be able to pick the best configuration at runtime for different tensors.
**Result:** resultSet that contains tuples where each tuple represents the chosen configuration for tensors at a specific MPT level and size range

**procedure** codeSelection(level, sizeLow, sizeHigh, initConfig):

- if checkEffectiveness(level, sizeLow, sizeHigh, initConfig) == TRUE
  - Put tuple (level, sizeLow, sizeHigh, initConfig) in resultSet
- else
  - tensorSize = generateRandomSize(level, sizeLow, sizeHigh)
  - config = findBestConfig(tensorSize)
  - if checkEffectiveness(level, sizeLow, sizeHigh, config) == TRUE
    - Put tuple (level, sizeLow, sizeHigh, config) in resultSet
  - else
    - mid = (sizeLow + sizeHigh)/2
    - codeSelection(level, sizeLow, mid, config)
    - codeSelection(level, mid, sizeHigh, config)

**main():**

- for level = 0 to maxLevel do
  - initConfig = last added configuration in resultSet
  - codeSelection(level, sizeLowBound, sizeHighBound, initConfig)

**generateRandomSize**(level, sizeLow, sizeHigh) returns a random tensor with MPT level at level and the size sits in between sizeLow and sizeHigh.

**findBestConfig**(tensorSize) returns the best autotuned configuration for the specified tensor size.

**checkEffectiveness**(level, sizeLow, sizeHigh, config) checks if the configuration is efficient across a set of tensors with MPT level at level and sits in between sizeLow and sizeHigh. Returns TRUE if efficient and FALSE otherwise.

**Algorithm 4:** A sketch of choosing optimization configurations for different tensors.
Chapter 6

Experimental Results

We carried out the measurements on a dual-socket Intel Westmere node with two Intel Xeon X5660 processors and an IBM POWER 755 with four Power7 processors. Table 6.1 shows their parameters and our configurations.

When calculating memory bandwidth, we divide the size of the source and destination tensor by running time to make it easy to compare the performance among different settings. We should notice that a tensor transpose may consume higher bandwidth than the calculation. For example, on an IBM Power7, where non-temporal stores are not available, even the best configuration reads each cache line in the target tensor before overwriting it, which increases the memory bandwidth consumed beyond the necessary minimum.

Please notice that, to get the best performance, we cache aligned the source and destination tensors whenever possible. We use row-major tensors in all of our test cases as our framework only generates row-major tensor transposition code at the moment.

In this chapter, we present and analyze experimental results on a single socket in the first four sections. We discuss multi-socket performance in the last section.


<table>
<thead>
<tr>
<th></th>
<th>Intel Westmere</th>
<th>IBM POWER 755</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Intel Xeon X5660</td>
<td>IBM Power7</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>2.8 GHz</td>
<td>3.86 GHz</td>
</tr>
<tr>
<td>Number of sockets</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Number of cores per socket</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>SMT</td>
<td>Turned off</td>
<td>4</td>
</tr>
<tr>
<td>SIMD width</td>
<td>128B</td>
<td>128B</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L3 cache size (per socket)</td>
<td>12MB</td>
<td>32MB</td>
</tr>
<tr>
<td>L3 cache organization</td>
<td>UMA</td>
<td>NUMA</td>
</tr>
<tr>
<td>L1 &amp; L3 cache line size</td>
<td>64B</td>
<td>128B</td>
</tr>
<tr>
<td>Non-temporal store availability</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Memory size per socket</td>
<td>24GB</td>
<td>64GB</td>
</tr>
<tr>
<td>Memory type</td>
<td>DDR3 1333 MHz</td>
<td>DDR3 1066 MHz</td>
</tr>
<tr>
<td>Memory bandwidth per socket</td>
<td>32GB/s</td>
<td>68.22GB/s</td>
</tr>
<tr>
<td>Page size</td>
<td>2 MB</td>
<td>64 KB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Linux 2.6.32</td>
<td>Linux 2.6.32</td>
</tr>
<tr>
<td>Compiler</td>
<td>icc 12.0.4</td>
<td>xlc 11.1</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O1 -fno-alias</td>
<td>-O2 -qalias=allptrs</td>
</tr>
</tbody>
</table>

Table 6.1 : Processor Parameters and System Configurations

6.1 Single-socket Performance

We compared the performance of generated code for tensor transposition with the STREAM benchmark \[39\]. The multithreaded version of the STREAM benchmark copy operation uses multiple threads to copy data from one array to another. This provides a realistic upper bound on memory bandwidth for multithreaded tensor

*The default page size on x86 architectures is 4 KB. However, the operating system on our test platform has enabled Transparent Huge Pages mechanism, which will automatically use larger pages for dynamically allocated memory.
for $i = 0$ to rowSize, increase by four do

for $j = 0$ to colSize do

\[ dst[i][j] \leftarrow src[i][j] \]

\[ dst[i + 1][j] \leftarrow src[i + 1][j] \]

\[ dst[i + 2][j] \leftarrow src[i + 2][j] \]

\[ dst[i + 3][j] \leftarrow src[i + 3][j] \]

Algorithm 5: An implementation sketch of strided array copy.

transposition. We hand-tuned the STREAM benchmark on each architecture to achieve better performance. On Intel Westmere, we also compared our generated code against array transposition code used by the Tensor Contraction Engine [15]. To understand the reason for the performance gap between STREAM and our tensor transposition code, we implemented an optimized strided version of 2D array copy (Algorithm 5) where the program works on four data streams in an interleaved fashion instead of just one.

Fig. 6.1 shows our results on Intel Westmere. Our generated code has very good single-thread performance on 2D double tensors, achieving a bandwidth 15% better performance than the TCE array transpose code and within 15% of multithreaded STREAM. For parallel transpositions of 2D float tensors, we achieved more than 80% of the bandwidth of multithreaded STREAM for all tensor sizes at six threads. Fig. 6.2 shows that our generated 3D and 4D tensor transposition code also worked well – achieving 94% and 82% of the bandwidth of multithreaded STREAM respectively using six threads. In 4D tensor transposition, due to the nature of the problem, we observed a much higher TLB miss rate compared to the 3D case, which we believe accounts for the performance gap between these two versions.
Figure 6.1: Comparison between tensor transposition, TCE array transposition, and different implementations of array copy on Intel Westmere.

To have a better understanding of the performance, we measured hardware performance counts for six thread executions of tensor transposition code as well as different versions of array copy. In Table 6.2, all the programs make good use of cache hier-
Figure 6.2: Result on high dimension tensor transposition on Intel Westmere. Measured on $38400 \times 38400$ float tensors for 2D tensor transposition and STREAM, $1536 \times 625 \times 1536$ float tensors for 3D tensor transposition, and $384 \times 100 \times 100 \times 384$ float tensors for 4D tensor transposition.

<table>
<thead>
<tr>
<th>Tensor size</th>
<th>STREAM</th>
<th>STREAM with -O0</th>
<th>Strided copy</th>
<th>2D tensor transposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tensor size</td>
<td>38400 × 38400</td>
<td>38400 × 38400</td>
<td>38400 × 38400</td>
<td>38400 × 38400</td>
</tr>
<tr>
<td>Tensor element type</td>
<td>float</td>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>L1 load hits</td>
<td>42471188</td>
<td>2672961660</td>
<td>137395195</td>
<td>737728002</td>
</tr>
<tr>
<td>LFB load hits</td>
<td>236767723</td>
<td>116020539</td>
<td>141507608</td>
<td>85835584</td>
</tr>
<tr>
<td>L2 load hits</td>
<td>71284561</td>
<td>53539461</td>
<td>80897249</td>
<td>44192430</td>
</tr>
<tr>
<td>L3 load hits</td>
<td>28221</td>
<td>6692</td>
<td>79268</td>
<td>34394</td>
</tr>
<tr>
<td>L3 load misses</td>
<td>18153857</td>
<td>6241137</td>
<td>9218121</td>
<td>9720860</td>
</tr>
<tr>
<td>DTLB load misses</td>
<td>4591</td>
<td>4375</td>
<td>4120</td>
<td>30577</td>
</tr>
<tr>
<td><strong>Bandwidth</strong></td>
<td>20.505 GB/s</td>
<td>20.246 GB/s</td>
<td>18.491 GB/s</td>
<td>18.909 GB/s</td>
</tr>
<tr>
<td><strong>Number of SIMD permutations</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>737280000</td>
</tr>
<tr>
<td><strong>Load streams per thread</strong></td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6.2: Hardware Performance counter measurements on Intel Westmere

... and have low L3 miss rate. There are mainly four differences between tensor transposition and STREAM:

1. although tensor transposition has more data TLB misses than array copy, we
believe such small counts won’t affect the performance.

2. tensor transposition issues more loads compared to STREAM due to data references of in-cache buffer and local variables that don’t fit in the register file. STREAM compiled with -O0 shows that more L1 hits should not cause a performance bottleneck in transposition of large tensors on this platform.

3. Eklundh’s algorithm [33] introduces in-register SIMD permutations, which may account for the performance gap between tensor transposition and STREAM.

4. multiple data streams may prevent strided copy and tensor transposition from achieving peak STREAM performance due to memory performance degradation on multiple data streams.

Fig. 6.3 shows the performance results on Power7. Our generated code achieves comparable bandwidth with STREAM for most tensor sizes. However, the perfor-
Figure 6.4: Performance improvement through appropriate padding on IBM Power7. Measured at 32 threads, on two-dimensional float tensors.

Performance dropped dramatically for a few tensors. Hardware performance counter measurements show that cache misses increase at those tensor sizes. These tensors probably introduce more cache conflict misses and thus have degraded performance. In the extreme case, we have only achieved 14.13 GB/s on 32768 × 32768 float tensors due to a large number of cache conflict misses, even though we are exploiting in-cache buffer optimization.

For tensor sizes where we have degraded performance, appropriate padding can avoid cache conflict misses and yield higher bandwidth. Fig. 6.4 shows four of our padding test cases. In the last case, bandwidth achieved on 32768 × 32768 float tensors is less than 15 GB/s. After padding it with 232 elements in each dimension, we get a bandwidth of 24.45 GB/s, which is nearly the same bandwidth we achieve on 33000 × 33000 float tensors. Therefore, we recommend adding appropriate padding.
when certain tensors degrade the performance.

In our experiments, we find that the best autotuned configurations for tensor transpose differ on our two experimental platforms as shown in Table 6.3. The best settings are the same for different tensor sizes on Westmere but vary on Power7. On Power7, the autotuner needs to generate code that performs run-time code selection to run efficiently across different tensor sizes.

Overall, our autotuning framework generates efficient code for tensor transposition for both experimental platforms. Our experiments show that code optimized for one architecture could be inefficient for the other. It is important to autotune tensor transposition for each architecture to achieve the best performance.

### 6.2 Performance Benefits of Individual Optimizations

We evaluated the performance gain of each optimization technique on our two test platforms. In Fig. 6.5 and Fig. 6.6, we compared the performance between the best
Figure 6.5: Comparison between the best autotuned configuration and the best setting without each technique. Measured on $N \times N$ float tensors with 6 threads on Intel Westmere.

autotuned configuration and the best setting without some techniques on our test platforms.

**Tiling**

On Intel Westmere, we observed a $>30\%$ bandwidth drop when removing tiling for all tensor sizes and the performance is even worse for $N = 30720$ and 38400. Hardware performance counter measurements show that most data accesses are L3 cache hits for $N \leq 24000$; however, data accesses are local memory accesses for $N = 30720$ and 38400, which explains why the performance is worse at these two tensor sizes.

On IBM Power7, removing tiling didn’t affect the performance at $N = 9600$ and 12000 much while the performance dropped a lot in the other cases. Hardware performance counter measurements show similar reasons for this difference – data accesses are L3 cache hits for $N = 9600$ and 12000 but local memory accesses for the other
Our results show that appropriate tiling can convert unnecessary memory accesses and L3 hits into L1 cache hits, which is critical for achieving good performance on this memory bound operation.

SIMD Operations and Non-temporal Stores

Fig. 6.5 shows a large benefit from SIMD operations and non-temporal stores on Intel Westmere. The program peaks at 18GB/s with SIMD and non-temporal stores, but only achieves 12GB/s without them. Non-temporal stores save memory bandwidth by not issuing loads for the destination tensor and thus improve the performance.

However, we didn’t benefit from SIMD operations a lot on IBM Power7. Fig. 6.6 shows that SIMD operations boost the performance by an average of 3% only. The reasons are twofold. First, non-temporal stores are not available on Power7 and thus
we won’t save memory bandwidth by using SIMD operations. Second, more cores and threads are available on Power7, which keep the cache and memory busy and make it less beneficial to issue fewer instructions.

In-cache buffer

On Intel Westmere, use of an in-cache buffer increases the bandwidth utilization of the best autotuned configuration by more than 10% because it organizes memory accesses as bursts of loads or stores, reducing bus turnarounds. The performance boost is more significant at $N = 15360$ and $30720$. These sizes are multiples of large power of two and an in-cache buffer can help eliminate cache conflict misses.

However, this optimization is less effective on Power7. Optimization configurations with and without in-cache buffer are usually in the same performance envelope for most tensor sizes. Hardware performance counter measurements show that while in-cache buffer cut down cache miss stall cycles by eliminating cache misses, the crowded store traffic prevent us from getting much benefit. We should notice that for $N = 32768$, which is a large power of two, an in-cache buffer is still quite useful as it help eliminate cache conflict misses.

Resource sharing among SMT threads

Fig. 6.7 shows our result on the effectiveness of SMT resource sharing. We only present results on TLB sharing here as sharing the cache result in inefficient code due to more cache and TLB stall cycles. TLB sharing boosts the performance by about 5% on $N = 7680, 9600, 12000$ and $24000$. For the other sizes, cache miss rates are relatively higher and thus we didn’t benefit from TLB sharing as it only reduces the number of TLB misses.
Figure 6.7: Comparison between the best configuration without SMT sharing, with SMT sharing but without thread synchronization, and with both SMT sharing and thread synchronization. Measured on $N \times N$ float tensors with 32 threads on IBM Power7.

The figure also shows that appropriate synchronization among SMT threads could improve the performance a bit more. Our results show that synchronization with window size = 0 is always better than other window sizes and it helps us get higher bandwidth at $N = 9600$ and 12000.

**Unrolling and Software Prefetch**

Unlike the aforementioned optimizations, we observed nearly no benefit from unrolling and software prefetch. Unrolling helps provide more instruction-level parallelism while software prefetch help reduce cache access latency. In the context of a memory-bound application, these two optimizations would offer little benefit.
Figure 6.8: Tile size search space on our two platforms. Each point represents the bandwidth of the best configuration without in-cache buffer optimization (as the buffer itself is another level of tiling) at the corresponding tile size.
6.3 Understanding the Autotuning Search Space

Our parameter search space is a multi-dimensional space. Since the choice of tile size is the most influential, we show the tile size search space on $38400 \times 38400$ tensors in Fig. 6.8. These two platforms have distinct search space shapes and best parameter settings – that’s why we need an autotuner to choose the right set of parameters across different architectures. Search space on Intel Westmere has quite a few optimal configurations while Power7 has a large plateau of good ones. We believe it is easier to hide cache access latency and keep the memory busy with more cores and threads on Power7.

Analysis on Dynamic Search Space Pruning

On one hand, our results show that dynamic tile size pruning is efficient on Intel Westmere. As shown in Fig. 6.9 when we increase $T_w$ from 16 to 32, there is a drop in both bandwidth and L1 cache hit rate for different values of $T_r$. Our autotuner will figure out that having $T_w \geq 32$ won’t help achieve the best performance due to low L1 cache hit rate and prune a bunch of configurations. The dynamic search space pruning can reduce the autotuning work by more than 80% in this case.

However, on IBM Power7, we didn’t benefit much from dynamic tile size pruning. In fact, the first level tile in the best configurations may not fit in L1 cache. Sufficient thread-level parallelism provided by more cores and threads helps hide cache access latencies, making L2 and L3 hits acceptable. Therefore, on Power7, the best $T_w$ value can vary from 32 to 512 and the autotuner will need to try a bunch of different $T_w$ values during the autotunign process.

On the other hand, the autotuner can reduce the search space as neither unrolling or software prefetch provide significant performance improvement. The autotuner
Figure 6.9: Bandwidth and L1 cache hit rate of the best configurations (without in-cache buffer optimization) at different tile sizes. Measured on $38400 \times 38400$ float tensors with 6 threads on Intel Westmere.

will only consider the other optimization techniques in the first phase of autotuning. After that, it applies unrolling and software prefetch on a group of good configurations.
selected in the first phase to pick the best one. On Intel Westmere, since there are four prefetch modes on Intel Westmere (no prefetch and prefetch to L1, L2, L3 respectively) and six different unrolling degrees, this dynamic search space pruning strategy reduced the autotuning work by more than 90%. The same thing goes for IBM Power7 where we reduced the autotuning work by more than 80%.

6.4 Analysis of Run-time Code Selection

Our test results show that the best configurations for different tensors stay the same on Intel Westmere but vary a lot on IBM Power7. Therefore, we conducted an analysis of run-time code selection on Power7.
Analysis of the Effect of Tensor Size

Fig. 6.10 shows the performance of several configurations on different tensor sizes at the same MPT level (MPT level = 3). In the result, configuration #1 and #2 remains good across all tensor sizes. The third and fourth one achieve slightly higher bandwidth than the previous two on small tensors but their performance is bad on large ones. Similar things apply to configuration #5 and #6. They have the best performance in the middle area of this size range but poor performance in other tensor sizes.

This chart clearly identifies the problem of designing a good run-time code selection mechanism. Our strategy as discussed in Section 5.5 will generate different code at different tolerance levels. For most tolerance levels, our strategy will select configuration #1 or #2 for the full tensor size range. However, if the user wants highly optimized code, he will define the tolerance level to a very low value so that our strategy will select #3 or #4 for small tensors, #5 or #6 for middle-sized tensors and #1 or #2 for large tensors.

Analysis of the Effect of MPT Level

Fig. 6.11 shows the performance of several configurations on tensors at two different MPT levels. Table 6.4 matches each configuration in Fig. 6.11 with the tensor for which it is autotuned.

At MPT level = 9, configuration #2 has the best performance for $N \leq 10752$ and configuration #3 is the best for $N \geq 11776$. Both of these configurations are autotuned for tensors at MPT level = 9. Configuration #2 is autotuned for 7680 while configuration #3 is autotuned for 38400, which explains why #2 is good for smaller tensors while #3 is better for larger ones. Other configurations that performs
Figure 6.11: Performance of different optimization configurations on tensors in two different MPT levels. The five configurations are autotuned for different tensors as shown in Table 6.4. Measured on $N \times N$ float tensors with 32 threads on IBM Power7.

well on tensors at other MPT levels are not working less efficiently here.

At MPT level = 10, configuration #5 has the best performance for $N \leq 13312$. 
<table>
<thead>
<tr>
<th>Config #</th>
<th>Best configuration for</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$N = 19200$, MPT level = 8</td>
</tr>
<tr>
<td>2</td>
<td>$N = 7680$, MPT level = 9</td>
</tr>
<tr>
<td>3</td>
<td>$N = 38400$, MPT level = 9</td>
</tr>
<tr>
<td>4</td>
<td>$N = 15360$, MPT level = 10</td>
</tr>
<tr>
<td>5</td>
<td>$N = 30720$, MPT level = 11</td>
</tr>
</tbody>
</table>

Table 6.4: Information on Configurations in Fig. 6.11

and configuration #4 is the best for $N \geq 15360$. At this time, configuration #2 and #3 is not achieving bandwidth as high as they do for MPT level = 9.

Our results show that it is important to select different configurations for tensors at different MPT level or with different sizes. Our proposed run-time code selection mechanism is capable of choosing the code generated by the best optimization configuration at run time according to the input tensor.

### 6.5 Multi-socket Performance

Previous sections show that we are able to achieve good single-socket performance through appropriate memory optimizations. However, we need to address more issues to get good multi-socket performance. This mainly involves choosing a good data placement scheme to minimize remote memory accesses and balance inter-chip traffic.

For multi-socket tensor transposition, one could divide a full tensor into multiple sub-tensors and copy these sub-tensors onto different sockets to perform tensor transposition. While tensor transpose will scale well in this case, the divide-and-copy operation could be costly. As a result, we care more about achieving good multi-socket performance without changing the original placement of the tensors. We selected five representative data placement scheme and compared their performance on our test
Figure 6.12: Illustration of the “all data in local memory” scheme using four sockets as an example. Both length/2 and width/2 need to be multiples of page sizes so that each sub-portion of the tensor could be aligned with page boundaries.

Figure 6.13: Illustration of the “all local loads” scheme using four sockets as an example. Blocks in light gray stands for local memory access; blocks in dark gray stands for remote memory access. In this illustration, in order to have all loads be local, we assign threads in a way that only a quarter of stores are local.

platforms. These schemes are:

- **All data in one socket.** In this naive scheme, we put the source and destina-
tion tensors in the local memory of one socket.

- **All data in local memory.** As shown in Fig. 6.12, we put corresponding portions of source and destination tensors to the sockets’ local memory according to thread assignment. However, since we must place one physical page of data on the same socket, this scheme require both length/2 and width/2 to be multiples of page size when using two or four sockets. Our Intel Westmere platform uses pages of 2 MB, which can hold 524288 32-bit floating point numbers, making it impossible for us to implement this data placement scheme. Our IBM Power7 platform uses pages of 64 KB, which can hold 16384 floating point numbers. As a result, both length and width for the tensors must be multiples of 32768 to implement this data placement scheme.

- **All local loads.** As shown in Fig. 6.13, assuming row-major and four-socket platforms, we assign the first quarter of both source and destination tensors to the first socket, the second quarter to the second socket, and so on. With this scheme, even if the tensors don’t align well with page boundaries, nearly all loads would be local accesses and around 1/NUM_SOCKET stores would be local accesses.

- **All local stores.** This is scheme is similar to the “all local loads” scheme except that we assign threads in a different way so that nearly all stores would be local accesses and around 1/NUM_SOCKET loads would be local accesses.

- **Interleaved memory.** We allocate source and destination tensors spread across all sockets using interleaved memory, which uses a round-robin distribution of pages. It is commonly used by applications. In this scheme, roughly
Figure 6.14: Multi-socket performance results on Intel Westmere. Measured on $N \times N$ float tensors with one socket (6 threads) for the leftmost bar (the blue bar) in each case and two sockets (12 threads) for all other bars. TT is short for tensor transposition in the legend.

$1/\text{NUM\_SOCKET}$ loads and stores would be local assuming random thread assignment.

Fig. 6.14 shows our multi-socket result on Intel Westmere. The “all data on one socket” scheme never achieves better performance than using one socket on three different tensor sizes. We believe that memory contention should be blamed for this worse performance. Bandwidth achieved by “all local reads” is around $1.4 \times$ compared to the single socket configuration while “all local writes” brings this number to $1.9 \times$, which is a near-optimal speedup. This indicates that remote stores are more expensive than remote loads on Intel Westmere, and therefore we recommend our library users to avoid remote stores on this architecture.

“Interleaved memory TT” achieves around $1.5 \times$ bandwidth compared to the single socket configuration. It is also within 95% of “interleaved memory STREAM”,
Figure 6.15: Multi-socket performance results on IBM Power7. Measured on $N \times N$ float tensors with one socket (32 threads) for the leftmost bar (the blue bar) in each case and four sockets (128 threads) for all other bars. TT is short for tensor transposition in the legend.

indicating that we are making full use of the memory hierarchy. This bandwidth is lower than “all local write” due to more remote stores; however, one should also expect it to be lower than “all local read” due to more remote loads. The answer lies in our thread assignment within a socket, which leads to the fact that half of the threads enjoy 100% local stores while the other half suffer 100% remote stores in the “all local read” scheme. In comparison, all local and remote accesses are evenly distributed among all the threads in “interleaved memory” scheme. This shows the importance of evenly balancing inter-socket traffic among all threads.

Fig. 6.15 shows our multi-socket result on IBM Power7. Again, the “all data on one socket” scheme has the worst performance. Only on $32768 \times 32768$ tensors does it achieve higher bandwidth than using one socket as single-socket performance is
low due to cache conflict misses. Bandwidth achieved by “all local reads” and “all local writes” is around $2.5 \times$ and $3 \times$ compared to the single socket one respectively. “Interleaved memory TT” only achieves about twice the bandwidth compared to the single socket configuration due to high rate of remote memory accesses. “Interleaved memory STREAM” justifies this explanation. On $32768 \times 32768$ tensors, where we are able to carry out “all data on local memory” scheme, the achieved bandwidth is $3.6 \times$ compared to the single socket configuration. However, we should notice that, due to low single-socket bandwidth, this multi-socket bandwidth is lower than the one achieved by “all local writes” scheme on the other two tensor sizes. This means that we would rather add some padding to $32768 \times 32768$ tensors and use “all local writes” scheme thereafter than applying “all data on local memory” scheme on this tensor directly.

While “all local writes” scheme achieves near-optimal speedup on Intel Westmere, it does not achieve top performance on IBM Power7. On Intel Westmere, applying “all local writes” scheme would lead to 100% local store and 50% local load. However, due to more sockets, numbers on IBM Power7 would be 100% local store and 25% local load, which means more inter-chip traffic on Power7 than Westmere. We believe that this increased inter-chip traffic prevents us from achieving higher multi-socket performance on Power7.

In conclusion, our experiment shows that our multi-socket tensor transposition code achieves more than 80% of the bandwidth of multi-socket STREAM when using the same data placement scheme. “All local writes” achieves the highest bandwidth among all five data placement schemes. It is more important to avoid remote stores than remote loads on both of our test platforms. In addition, when data placement among multiple sockets is not feasible, using only the threads in that socket would
probably be better than making use of threads in all of the sockets.
Chapter 7

Conclusions and Future Work

By combining parametric code generation and empirical autotuning, our approach achieves more than 80% of the achievable memory bandwidth bound established by the STREAM benchmark on two platforms. For tensor transposition, which is memory bound, tiling, in-cache buffer and non-temporal stores are the keys to good performance. Tiling saves bandwidth by enhancing spatial locality, in-cache buffer helps avoid cache-conflict misses, and non-temporal stores avoid unnecessary loads of the destination tensor. Plenty of thread-level parallelism is also important as one thread will be unable to keep the memory hierarchy busy. As for the multi-socket performance, we achieve the highest performance by using an “all local writes” data placement scheme. We also looked into various issues related to autotuning. Our autotuner can prune the search space on the fly using run time and hardware performance counter measurements, which help cut down the autotuning time by more than 80%. Furthermore, we proposed a strategy that performs run-time code selection to select good configurations for different inputs.

We used tensor transposition as an motivating example to study how to autotune code for memory intensive codes on today’s node architectures. Our conclusions are as follows.

• Given the complexity of today’s node architectures, it is hard to tell how much a specific optimization may help performance. Therefore, we need to try various
optimization configurations to determine the best code through autotuning.

- Due to architectural differences, we need to optimize code for each architecture separately to get the best performance. Hand tuning a piece of code for multiple architectures is too time-consuming. Autotuning makes it feasible to do so.

- Memory bandwidth is a performance bottleneck for low-operation intensity applications. For such applications, it is crucial to enhance locality and achieve better memory bandwidth usage through tiling, in-cache buffer and non-temporal stores. Plenty of thread-level parallelism is also important to keep memory busy.

- Due to the vastness of the autotuning search space, good pruning strategies are needed for efficient autotuning. Using the running time in combination with hardware performance counter measurements can provide us insights about the shape of the search space and help us figure out good pruning strategies.

- Run-time code selection is important for achieving good performance throughout different inputs. A good run-time code selection strategy would require appropriate modeling of the inputs.

In the future, we want to extend our work on the search space pruning and run-time code selection. We would like to use advanced search strategies, such as a direct search approach proposed by Lewis and Torczon [40], or take advantage of some machine-learning techniques, in search space pruning and run-time code selection to make these two jobs more automated and extensible to other memory-intensive applications.

Furthermore, we want to apply our work to more architectures. This involves work in two directions.
First, we want to apply our work to new memory architectures. A forthcoming Intel Knights Landing processor can have 16GB high-bandwidth, on-package memory that delivers five times higher bandwidth than state-of-art DDR4 memory [41]. The on-package memory is also configurable where part of it can be regarded as cache and the rest is used as scratch pad. To autotune memory-intensive code on such architectures, we will need to find a good configuration of the on-package memory and apply various memory optimizations to make full use of the improved memory bandwidth.

Second, on architectures with lots of cores, achieving peak performance for memory-intensive code may require a good balance between thread-level parallelism and memory optimizations. On an IBM Power7 core, which consists of eight super-pipelined, out-of-order cores with 4-way SMT, we observed nearly no performance difference between using 16 or 32 threads. We also find that the best configurations for different number of threads are likely to be different. Therefore, we believe it is possible that on architectures with a large number of much simpler in-order cores, such as IBM Blue Gene/Q and Intel MIC, we may need a good balance between memory optimization and thread-level parallelism to achieve peak performance. First, too much thread-level parallelism may result in memory contention and degrade performance. Second, one may need to select different memory optimization strategies for different levels of thread-level parallelism to avoid memory contentions and improve the performance.
Appendix A

Generated Tensor Transposition Code

A.1 A 2D Tensor Transposition Routine for IBM Power7

This piece of routine is generated with two-level tiling, SIMD turned on, and resource sharing among SMT threads turned on. It is one of the best configurations for 38400 × 38400 tensors on 32 threads. The interface is TensorTranspose2D().

```c
#include <builtins.h>

#define TENSOR_TRANSPOSE_ARRAY_TYPE float
#define TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1 256
#define TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_1 256
#define TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_2 2048
#define TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_2 1024
#define TENSOR_TRANSPOSE_SIMD_WIDTH 4
#define TENSOR_TRANSPOSE_UNROLL_DEGREE 1
#define TENSOR_TRANSPOSE_JAM_DEGREE 1
#define TENSOR_TRANSPOSE_NUM_CORES 8
#define TENSOR_TRANSPOSE_NUM_SMT 4
#define TENSOR_TRANSPOSE_NUM_THREAD 32
#define TENSOR_TRANSPOSE_SMT_SOURCE_STRIDE 2
#define TENSOR_TRANSPOSE_SMT_TARGET_STRIDE 2

static inline void _tensor_transposition_base_case_incomplete_tile_x
(_TENSOR_TRANSPOSE_ARRAY_TYPE source, _TENSOR_TRANSPOSE_ARRAY_TYPE target, int base_addr_source, int base_addr_target, int s0, int s1, int l0, int l1, int source_stride1, int target_stride0) {
  int i0, i1;
  _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr0;
  _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr0;
  _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr1;
  _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr1;
  vector float vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8,
  vector9, vector10, vector11;
  source_addr0 = source + base_addr_source + s0;
  target_addr0 = target + base_addr_target + s0 + target_stride0;
...
for ( i0 = s0; i0 < l0; i0 += TENSOR_TRANSPOSE_SIMD_WIDTH ) {
    source_addr1 = source_addr0 + s1 * source_stride1;
    target_addr1 = target_addr0 + s1;
    for ( i1 = s1; i1 < l1; i1 += TENSOR_TRANSPOSE_SIMD_WIDTH ) {
        vector0 = vec_xlw4(0, source_addr1);
        source_addr1 += source_stride1;
        vector1 = vec_xlw4(0, source_addr1);
        source_addr1 += source_stride1;
        vector2 = vec_xlw4(0, source_addr1);
        source_addr1 += source_stride1;
        vector3 = vec_xlw4(0, source_addr1);
        source_addr1 += source_stride1;
        vector4 = vec_mergelh(vector0, vector2);
        vector5 = vec_mergelh(vector0, vector2);
        vector6 = vec_mergelh(vector1, vector3);
        vector7 = vec_mergelh(vector1, vector3);
        vector8 = vec_mergelh(vector4, vector5);
        vector9 = vec_mergelh(vector4, vector5);
        vector10 = vec_mergelh(vector6, vector7);
        vector11 = vec_mergelh(vector6, vector7);
        vec_xstw4(vector8, 0, target_addr1 + target_stride0 * 0);
        vec_xstw4(vector9, 0, target_addr1 + target_stride0 * 1);
        vec_xstw4(vector10, 0, target_addr1 + target_stride0 * 2);
        vec_xstw4(vector11, 0, target_addr1 + target_stride0 * 3);
        target_addr1 += TENSOR_TRANSPOSE_SIMD_WIDTH;
    }
    source_addr0 += TENSOR_TRANSPOSE_SIMD_WIDTH;
    target_addr0 += TENSOR_TRANSPOSE_SIMD_WIDTH + target_stride0;
}

static inline void _tensor_transposition_base_case_incomplete_tile_y
    (_TENSOR_TRANSPOSEARRAY_TYPE source, _TENSOR_TRANSPOSEARRAY_TYPE target, int base_addr_source, int base_addr_target, int s0, int s1, int l0, int source_stride1, int target_stride0) {
    int i0, i1;
    _TENSOR_TRANSPOSEARRAY_TYPE source_addr0;
    _TENSOR_TRANSPOSEARRAY_TYPE target_addr0;
    _TENSOR_TRANSPOSEARRAY_TYPE source_addr1;
    _TENSOR_TRANSPOSEARRAY_TYPE target_addr1;
    vector float vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8, vector9, vector10, vector11;
    source_addr0 = source + base_addr_source + s0;
    target_addr0 = target + base_addr_target + s0 * target_stride0;
    for ( i0 = s0; i0 < l0; i0 += TENSOR_TRANSPOSE_SIMD_WIDTH ) {
        source_addr1 = source_addr0 + s1 * source_stride1;
        target_addr1 = target_addr0 + s1;
        for ( i1 = s1; i1 < s1 + TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_1; i1 +=
            TENSOR_TRANSPOSE_UNROLL_DEGREE * TENSOR_TRANSPOSE_SIMD_WIDTH ) {
            vector0 = vec_xlw4(0, source_addr1);
            source_addr1 += source_stride1;
        }
    }
}
vector1 = vec_xlw4(0, source_addr1);
source_addr1 += source_stride1;
vector2 = vec_xlw4(0, source_addr1);
source_addr1 += source_stride1;
vector3 = vec_xlw4(0, source_addr1);
source_addr1 += source_stride1;
vector4 = vec_mergeh(vector0, vector2);
vector5 = vec_mergeh(vector1, vector3);
vector6 = vec_mergeh(vector0, vector2);
vector7 = vec_mergeh(vector1, vector3);
vector8 = vec_mergeh(vector4, vector5);
vector9 = vec_mergeh(vector4, vector5);
vector10 = vec_mergeh(vector6, vector7);
vector11 = vec_mergeh(vector6, vector7);
vec_xstw4(vector8, 0, target_addr1 + target_stride0 * 0);
vec_xstw4(vector9, 0, target_addr1 + target_stride0 * 1);
vec_xstw4(vector10, 0, target_addr1 + target_stride0 * 2);
vec_xstw4(vector11, 0, target_addr1 + target_stride0 * 3);
target_addr1 += _TENSORTRANSPOSE_SIMD_WIDTH;
}
source_addr0 += _TENSORTRANSPOSE_SIMD_WIDTH;
target_addr0 += _TENSORTRANSPOSE_SIMD_WIDTH * target_stride0;
}

static inline void _tensor_transposition_base_case_complete_tile (_TENSORTRANSPOSE_ARRAY_TYPE source, _TENSORTRANSPOSE_ARRAY_TYPE target, int base_addr_source, int base_addr_target, int s0, int s1, int source_stride1, int target_stride0, int thread_num) {
    int i0, i1;
    _TENSORTRANSPOSE_ARRAY_TYPE source_addr0;
    _TENSORTRANSPOSE_ARRAY_TYPE target_addr0;
    _TENSORTRANSPOSE_ARRAY_TYPE source_addr1;
    _TENSORTRANSPOSE_ARRAY_TYPE target_addr1;
    vector float vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8, vector9, vector10, vector11;
    source_addr0 = source + base_addr_source + s0;
    target_addr0 = target + base_addr_target + s0 + target_stride0;
    for ( i0 = s0; i0 < s0 + _TENSORTRANSPOSE_TILE_SIZE_SOURCE_LEVEL1; i0 += _TENSORTRANSPOSE_JAM_DEGREE + _TENSORTRANSPOSE_SIMD_WIDTH ) {
        source_addr1 = source_addr0 + s1 + source_stride1;
        target_addr1 = target_addr0 + s1;
        for ( i1 = s1; i1 < s1 + _TENSORTRANSPOSE_TILE_SIZE_TARGET_LEVEL1; i1 += _TENSORTRANSPOSE_UNROLL_DEGREE + _TENSORTRANSPOSE_SIMD_WIDTH ) {
            vector0 = vec_xlw4(0, source_addr1);
            source_addr1 += source_stride1;
            vector1 = vec_xlw4(0, source_addr1);
            source_addr1 += source_stride1;
            vector2 = vec_xlw4(0, source_addr1);
            source_addr1 += source_stride1;
            vector3 = vec_xlw4(0, source_addr1);
            source_addr1 += source_stride1;
        }
    }
}
vector4 = vec_mergeh(vector0, vector2);
vector6 = vec_mergel(vector0, vector2);
vector5 = vec_mergeh(vector1, vector3);
vector7 = vec_mergel(vector1, vector3);
vector8 = vec_mergeh(vector4, vector5);
vector9 = vec_mergel(vector4, vector5);
vector10 = vec_mergeh(vector6, vector7);
vector11 = vec_mergel(vector6, vector7);
vec_xstw4(vector8, 0, target_addr1 + target_stride0 * 0);
vec_xstw4(vector9, 0, target_addr1 + target_stride0 * 1);
vec_xstw4(vector10, 0, target_addr1 + target_stride0 * 2);
vec_xstw4(vector11, 0, target_addr1 + target_stride0 * 3);
target_addr1 += TENSOR_TRANSPOSE_SIMD_WIDTH;
source_addr0 += TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr0 += target_stride0 * TENSOR_TRANSPOSE_SIMD_WIDTH;
}
}

/**
This is a 2D tensor transpose interface. Meanings of arguments are:
source — Pointer to the source tensor (assuming raw-major array)
target — Pointer to the destination tensor (assuming raw-major array)
n0, n1 — The size of each dimension for a source_tensor[n0][n1]
p0, p1 — Specifies the order of indexes. pi = k means that the i-th index in the source tensor
would be the k-th index in the destination tensor. 0 stands for the outer index while 1 stands
for the inner index.
s0, s1, e0, e1 — The library will transpose all the elements from source_tensor[s0][s1] to
target_tensor[e0][e1]. These values can be used both in thread assignment to control each
thread's work data and in padding so that the transpose won't take place in the padding area.
thread_num — The id of the thread that makes the library call.
**/
int TensorTranspose2D(_TENSOR_TRANSPOSE_ARRAY_TYPE source, _TENSOR_TRANSPOSE_ARRAY_TYPE target,
int n0, int n1, int p0, int p1, int s0, int s1, int e0, int e1, int thread_num) {
int temp[2], l0, l1, i0, i1, j0, j1, source_stride0, source_stride1, target_stride0,
target_stride1, source_addr0, source_addr1, target_addr0, target_addr1, core_num, smt_num,
smt_source_start, smt_target_start, sync_dest_round0, sync_dest_round1;
temp[0] = n1 * 1;
temp[1] = 1;
source_stride0 = temp[p0];
source_stride1 = temp[p1];
temp[0] = n0;
temp[1] = n1;
l0 = temp[p0];
l1 = temp[p1];
target_stride0 = l1 * 1;
target_stride1 = 1;
temp[0] = s0;
temp[1] = s1;
s0 = temp[p0];
s1 = temp[p1];
temp[0] = e0;
temp[1] = e1;
l0 = temp[p0];
l1 = temp[p1];
core_num = thread_num / _TENSOR_TRANSPOSE_NUM_SMT;
smt_num = thread_num % _TENSOR_TRANSPOSE_NUM_SMT;
smt_source_start = smt_num / _TENSOR_TRANSPOSE_SMT_TARGET_STRIDE;
smt_target_start = smt_num % _TENSOR_TRANSPOSE_SMT_TARGET_STRIDE;

if (p0 == 1) {
    for (i0 = s0; i0 < l0; i0 += _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_2) {
        for (i1 = s1; i1 < l1; i1 += _TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_2) {
            for (ii1 = i0 + smt_source_start * _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1;
                 (ii1 < i0 + _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_2) && (ii1 < l0); ii1 +=
                 _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1 + _TENSOR_TRANSPOSE_SMT_SOURCE_STRIDE) {
                if (jj1 = ii1 + smt_target_start * _TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_1;
                     (jj1 < l1 + _TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_2) && (jj1 < l1); jj1 +=
                     _TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_1 + _TENSOR_TRANSPOSE_SMT_TARGET_STRIDE) {
                    if (jj1 + _TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL_2 > l1) {
                        tensor_transposition_base_case_incomplete_tile_x(source, target, 0, 0, ii1, jj1, (ii1 + _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1 > l0) ? l0 : ii1 + _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1, l1, source_stride1, target_stride0);
                    } else if (ii1 + _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL_1 > l0) {
                        tensor_transposition_base_case_incomplete_tile_y(source, target, 0, 0, ii1, jj1, l0, source_stride1, target_stride0);
                    } else tensor_transposition_base_case_complete_tile(source, target, 0, 0, ii1, jj1, source_stride1, target_stride0, thread_num);
                }
            }
        }
    }
}

if (p1 == 1) {
    for (i0 = s0; i0 < l0; i0 += 1) {
        source_addr0 = i0 * source_stride0;
target_addr0 = i0 * target_stride0;
        for (i1 = s1; i1 < l1; i1 += _TENSOR_TRANSPOSE_SIMD_WIDTH) {
            source_addr1 = source_addr0 + i1 * source_stride1;
target_addr1 = target_addr0 + i1 * target_stride1;
            vector float vector0;
            vector0 = vec4_klw4(0, source + source_addr1);
            vec4_stw4(vector0, 0, target + target_addr1);
        }
    }
    return 0;
}
The following piece of code is part of a sample test program that uses the above routine to transpose a 2D tensor in parallel.

```c
#define NUM_CORE 8
#define NUM_SMT_THREAD 4
#define NUM_THREAD (NUM_CORE+NUM_SMT_THREAD)

#define N 38400
#define LENGTH N
#define WIDTH N

#pragma omp parallel num_threads(NUM_THREAD) shared(src, dst) default(none)
{
    int threadNum =omp_get_thread_num();
    int numOftThread =omp_get_num_threads();
    int coreNum = threadNum / NUM_SMT_THREAD;

#define FACTOR 2
    int coreBase = coreNum / FACTOR;
    int startL = coreBase * LENGTH / (NUM_CORE/FACTOR);
    int endL = (coreBase+1) * LENGTH / (NUM_CORE/FACTOR);

    int coreOffset = coreNum % FACTOR;
    int startW = coreOffset * WIDTH / FACTOR;
    int endW = (coreOffset + 1) * WIDTH / FACTOR;

    /*
    SMT threads on the same core would have the same startL, startW, endL, and endW value.
    The library would further perform data assignment among SMT threads.
    */
    TensorTranspose2D(src, dst, LENGTH, WIDTH,
        1, 0,
        startL, startW,
        endL, endW,
        threadNum);
}
```
A.2 A 3D Tensor Transposition Routine for Intel Westmere

This piece of routine is generated with two-level tiling, SIMD turned on, non-temporal stores turned on, and in-cache buffer turned on. It is one of the best configurations for $1536 \times 625 \times 1536$ tensors on 6 threads. The interface is $\text{TensorTranspose3D()}$.

```c
#include <xmmintrin.h>
#include <emmintrin.h>

#define TENSOR_TRANSPOSE_ARRAY_TYPE float
#define TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL 1 1536
#define TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL 1 16
#define TENSOR_TRANSPOSE_SIMD_WIDTH 4
#define TENSOR_TRANSPOSE_UNROLL_DEGREE 1
#define TENSOR_TRANSPOSE_JAM_DEGREE 1
#define TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE 16
#define TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET 16
#define TENSOR_TRANSPOSE_NUM_CORES 6
#define TENSOR_TRANSPOSE_NUM_SMT 1
#define TENSOR_TRANSPOSE_NUM_THREAD 6
#define TENSOR_TRANSPOSE_SMT_SOURCE_STRIDE 1
#define TENSOR_TRANSPOSE_SMT_TARGET_STRIDE 1

decspec(align(1024)) static float buffer[TENSOR_TRANSPOSE_NUM_THREAD]
    [TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE * TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET];

static inline void _tensor_transposition_base_case_incomplete_tile_x
    (TENSOR_TRANSPOSE_ARRAY_TYPE source, TENSOR_TRANSPOSE_ARRAY_TYPE target, int
     base_addr_source, int base_addr_target, int s0, int s1, int l0, int l1, int source_stride1,
     int target_stride0) {
    int i0, i1;
    TENSOR_TRANSPOSE_ARRAY_TYPE source_addr0;
    TENSOR_TRANSPOSE_ARRAY_TYPE target_addr0;
    TENSOR_TRANSPOSE_ARRAY_TYPE source_addr1;
    TENSOR_TRANSPOSE_ARRAY_TYPE target_addr1;
    __m128 vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8,
    vector9, vector10, vector11;
    source_addr0 = source + base_addr_source + s0;
    target_addr0 = target + base_addr_target + s0 * target_stride0;
    for (i0 = s0; i0 < l0; i0 += TENSOR_TRANSPOSE_SIMD_WIDTH) {
        source_addr1 = source_addr0 + s1 * source_stride1;
        target_addr1 = target_addr0 + s1;
        for (i1 = s1; i1 < l1; i1 += TENSOR_TRANSPOSE_SIMD_WIDTH) {
            vector0 = mm_load_ps(source_addr1);
            source_addr1 += source_stride1;
            vector1 = mm_load_ps(source_addr1);
            source_addr1 += source_stride1;
            ...
        }
    }
```
vector2 = _mm_load_ps(source_addr1);
source_addr1 += source_stride1;
vector3 = _mm_load_ps(source_addr1);
source_addr1 += source_stride1;
vector4 = _mm_unpacklo_ps(vector0, vector2);
vector6 = _mm_unpacklo_ps(vector0, vector2);
vector5 = _mm_unpacklo_ps(vector1, vector3);
vector7 = _mm_unpacklo_ps(vector1, vector3);
vector8 = _mm_unpacklo_ps(vector4, vector5);
vector9 = _mm_unpacklo_ps(vector4, vector5);
vector10 = _mm_unpacklo_ps(vector6, vector7);
vector11 = _mm_unpacklo_ps(vector6, vector7);
_mm_stream_ps(target_addr1 + target_stride0 * 0, vector8);
_mm_stream_ps(target_addr1 + target_stride0 * 1, vector9);
_mm_stream_ps(target_addr1 + target_stride0 * 2, vector10);
_mm_stream_ps(target_addr1 + target_stride0 * 3, vector11);
target_addr1 += __TENSOR_TRANSPOSE_SIMD_WIDTH;
}
source_addr0 += __TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr0 += __TENSOR_TRANSPOSE_SIMD_WIDTH * target_stride0;
}

static inline void _tensor_transposition_base_case_incomplete_tile_y

(_TENSORTRANSPOSEARRAYTYPE source, _TENSORTRANSPOSEARRAYTYPE target, int
base_addr_source, int base_addr_target, int s0, int s1, int l0, int source_stride1, int
target_stride0) {
int i0, i1;
_TENSORTRANSPOSEARRAYTYPE source_addr0;
_TENSORTRANSPOSEARRAYTYPE target_addr0;
_TENSORTRANSPOSEARRAYTYPE source_addr1;
_TENSORTRANSPOSEARRAYTYPE target_addr1;
_m128 vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8,
vector9, vector10, vector11;
source_addr0 = source + base_addr_source + s0;
target_addr0 = target + base_addr_target + s0 * target_stride0;
for (i0 = s0; i0 < l0; i0 += __TENSORTRANSPOSE_SIMD_WIDTH) {
    source_addr1 = source_addr0 + s1 * source_stride1;
target_addr1 = target_addr0 + s1;
    for (i1 = s1; i1 < s1 + __TENSORTRANSPOSE_TILE_SIZE_TARGETLEVEL1; i1 +=
__TENSORTRANSPOSE_UNROLL_DEGREE * __TENSORTRANSPOSE_SIMD_WIDTH) { 
        vector0 = _mm_load_ps(source_addr1);
        source_addr1 += source_stride1;
        vector1 = _mm_load_ps(source_addr1);
        source_addr1 += source_stride1;
        vector2 = _mm_load_ps(source_addr1);
        source_addr1 += source_stride1;
        vector3 = _mm_load_ps(source_addr1);
        source_addr1 += source_stride1;
        vector4 = _mm_unpacklo_ps(vector0, vector2);
        vector6 = _mm_unpackhi_ps(vector0, vector2);
vector5 = _mm_unpacklo_ps(vector1, vector3);
vector7 = _mm_unpackhi_ps(vector1, vector3);
vector8 = _mm_unpacklo_ps(vector4, vector5);
vector9 = _mm_unpackhi_ps(vector4, vector5);
vector10 = _mm_unpacklo_ps(vector6, vector7);
vector11 = _mm_unpackhi_ps(vector6, vector7);
_mm_stream_ps(target_addr1 + target_stride0 * 0, vector8);
_mm_stream_ps(target_addr1 + target_stride0 * 1, vector9);
_mm_stream_ps(target_addr1 + target_stride0 * 2, vector10);
_mm_stream_ps(target_addr1 + target_stride0 * 3, vector11);
target_addr1 += _TENSOR_TRANSPOSE_SIMD_WIDTH;

source_addr0 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr0 += _TENSOR_TRANSPOSE_SIMD_WIDTH * target_stride0;
}

static inline void tensor_transposition_base_case_complete_tile (_TENSOR_TRANSPOSE_ARRAY_TYPE source, _TENSOR_TRANSPOSE_ARRAY_TYPE target, int base_addr_source, int base_addr_target, int s0, int s1, int source_stride1, int target_stride0, int thread_num) {
    int i0, i1, ji0;
    _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr0;
    _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr0;
    _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr1;
    _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr1;
    _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr2;
    _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr2;
    _TENSOR_TRANSPOSE_ARRAY_TYPE source_addr3;
    _TENSOR_TRANSPOSE_ARRAY_TYPE target_addr3;
    _m128 vector0, vector1, vector2, vector3, vector4, vector5, vector6, vector7, vector8,
    vector9, vector10, vector11;
    source_addr0 = source + base_addr_source + s0;
    target_addr0 = target + base_addr_target + s0 + target_stride0;
    for (i0 = s0; i0 < s0 + _TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL; i0 +=
_TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE) {
        source_addr1 = source_addr0 + s1 + source_stride1;
        target_addr1 = target_addr0 + s1;
        source_addr2 = source_addr1;
        target_addr2 = buffer[thread_num];
        for (ji0 = 0; ji0 < _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET; ji0 +=
_TENSOR_TRANSPOSE_SIMD_WIDTH) {
            source_addr3 = source_addr2;
            target_addr3 = target_addr2;
            for (i10 = 0; i10 < _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE; i10 +=
_TENSOR_TRANSPOSE_SIMD_WIDTH * TENSOR_TRANSPOSE_UNROLL_DEGREE) {
                vector0 = _mm_load_ps(source_addr3 + source_stride1 * 0);
                vector1 = _mm_load_ps(source_addr3 + source_stride1 * 1);
                vector2 = _mm_load_ps(source_addr3 + source_stride1 * 2);
                vector3 = _mm_load_ps(source_addr3 + source_stride1 * 3);
                vector4 = _mm_unpacklo_ps(vector0, vector2);
                vector6 = _mm_unpackhi_ps(vector0, vector2);
            }
        }
    }
}
vector5 = _mm_unpacklo_ps(vector1, vector3);
vector7 = _mm_unpackhi_ps(vector1, vector3);
vector8 = _mm_unpacklo_ps(vector4, vector5);
vector9 = _mm_unpackhi_ps(vector4, vector5);
vector10 = _mm_unpacklo_ps(vector6, vector7);
vector11 = _mm_unpackhi_ps(vector6, vector7);
_mm_store_ps(target_addr3, vector8);
target_addr3 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;
_mm_store_ps(target_addr3, vector9);
target_addr3 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;
_mm_store_ps(target_addr3, vector10);
target_addr3 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;
_mm_store_ps(target_addr3, vector11);
target_addr3 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;

source_addr2 += _TENSOR_TRANSPOSE_SIMD_WIDTH * source_stride1;
target_addr2 += _TENSOR_TRANSPOSE_SIMD_WIDTH;

source_addr2 = buffer[thread_num];
target_addr2 = target_addr1;
for (i10 = 0; i10 < _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE; i10 += _TENSOR_TRANSPOSE_UNROLL_DEGREE) {
    source_addr3 = source_addr2;
target_addr3 = target_addr2;
    vector3 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector3);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector2 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector2);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector1 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector1);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector0 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector0);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    source_addr2 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;
target_addr2 += target_stride0;
}

source_addr0 += _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE;
target_addr0 += _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE * target_stride0;
}

/**
This is a 3D tensor transpose interface. Meanings of arguments are:

vector5 = _mm_unpacklo_ps(vector1, vector3);
vector7 = _mm_unpackhi_ps(vector1, vector3);
vector8 = _mm_unpacklo_ps(vector4, vector5);
vector9 = _mm_unpackhi_ps(vector4, vector5);
vector10 = _mm_unpacklo_ps(vector6, vector7);
vector11 = _mm_unpackhi_ps(vector6, vector7);

source_addr2 += _TENSOR_TRANSPOSE_SIMD_WIDTH * source_stride1;
target_addr2 += _TENSOR_TRANSPOSE_SIMD_WIDTH;

source_addr2 = buffer[thread_num];
target_addr2 = target_addr1;
for (i10 = 0; i10 < _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE; i10 += _TENSOR_TRANSPOSE_UNROLL_DEGREE) {
    source_addr3 = source_addr2;
target_addr3 = target_addr2;
    vector3 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector3);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector2 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector2);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector1 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector1);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    vector0 = _mm_load_ps(source_addr3);
    _mm_stream_ps(target_addr3, vector0);
    source_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
target_addr3 += _TENSOR_TRANSPOSE_SIMD_WIDTH;
    source_addr2 += _TENSOR_TRANSPOSE_BUFFER_SIZE_TARGET;
target_addr2 += target_stride0;
}

source_addr0 += _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE;
target_addr0 += _TENSOR_TRANSPOSE_BUFFER_SIZE_SOURCE * target_stride0;
}
**TensorTranspose3D**

```c
int TensorTranspose3D (TENSOR_TRANSPOSE_ARRAY_TYPE source, TENSOR_TRANSPOSE_ARRAY_TYPE target,
                       int n0, int n1, int n2, int p0, int p1, int p2, int s0, int s1, int s2, int e0, int e1, int e2, int thread_num)
{
    int temp[3], l0, l1, l2, i0, i1, i2, source_stride0, source_stride1, source_stride2,
    target_stride0, target_stride1, target_stride2, source_addr0, source_addr1, source_addr2,
    target_addr0, target_addr1, target_addr2, core_num, smt_num, smt_source_start,
    smt_target_start;

    temp[0] = n1 * n2 + 1;
    temp[1] = n2 + 1;
    temp[2] = 1;
    source_stride0 = temp[p0];
    source_stride1 = temp[p1];
    source_stride2 = temp[p2];
    temp[0] = n0;
    temp[1] = n1;
    temp[2] = n2;
    l0 = temp[p0];
    l1 = temp[p1];
    l2 = temp[p2];
    target_stride0 = l1 * l2 + 1;
    target_stride1 = l2 + 1;
    target_stride2 = 1;
    temp[0] = s0;
    temp[1] = s1;
    temp[2] = s2;
    s0 = temp[p0];
    s1 = temp[p1];
    s2 = temp[p2];
    temp[0] = e0;
    temp[1] = e1;
    temp[2] = e2;
    l0 = temp[p0];
    l1 = temp[p1];
    l2 = temp[p2];

    core_num = thread_num / _TENSOR_TRANSPOSE_NUM_SMT;
    smt_num = thread_num % _TENSOR_TRANSPOSE_NUM_SMT;
    smt_source_start = smt_num / _TENSOR_TRANSPOSE_SMT_TARGET2STRIDE;
    smt_target_start = smt_num % _TENSOR_TRANSPOSE_SMT_TARGET2STRIDE;

    if (p0 == 2) {
        // Code block...
    }
}
```
for ( i0 = s0; i0 < l0; i0 += TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 ) {
    for ( i1 = s1; i1 < l1; i1 += 1 ) {
        source_addr1 = i1 * source_stride1;
        target_addr1 = i1 * target_stride1;
        for ( i2 = s2; i2 < l2; i2 += TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL1 ) {
            if (i2 + TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL1 > l2) {
                _tensor_transposition_base_case_incomplete_tile_x(source, target,
                source_addr1, target_addr1, i0, i2, (i0 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 > l0) ? l0 : i0 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1, l2, source_stride2, target_stride0);
            } else if (i0 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 > l0) {
                _tensor_transposition_base_case_incomplete_tile_y(source, target,
                source_addr1, target_addr1, i0, i2, l0, source_stride2, target_stride0);
            } else _tensor_transposition_base_case_complete_tile(source, target,
                source_addr1, target_addr1, i0, i2, source_stride2, target_stride0, thread_num);
        }
    }
}

if (p1 == 2) {
    for ( i0 = s0; i0 < l0; i0 += 1 ) {
        source_addr0 = i0 * source_stride0;
        target_addr0 = i0 * target_stride0;
        for ( i1 = s1; i1 < l1; i1 += TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 ) {
            for ( i2 = s2; i2 < l2; i2 += TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL1 ) {
                if (i2 + TENSOR_TRANSPOSE_TILE_SIZE_TARGET_LEVEL1 > l2) {
                    _tensor_transposition_base_case_incomplete_tile_x(source, target,
                    source_addr0, target_addr0, i1, i2, (i1 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 > l1) ? l1 : i1 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1, l2, source_stride2, target_stride1);
                } else if (i1 + TENSOR_TRANSPOSE_TILE_SIZE_SOURCE_LEVEL1 > l1) {
                    _tensor_transposition_base_case_incomplete_tile_y(source, target,
                    source_addr0, target_addr0, i1, i2, l1, source_stride2, target_stride1);
                } else _tensor_transposition_base_case_complete_tile(source, target,
                    source_addr0, target_addr0, i1, i2, source_stride2, target_stride1, thread_num);
            }
        }
    }
}

if (p2 == 2) {
    for ( i0 = s0; i0 < l0; i0 += 1 ) {
        source_addr0 = i0 * source_stride0;
        target_addr0 = i0 * target_stride0;
        for ( i1 = s1; i1 < l1; i1 += 1 ) {
            source_addr1 = source_addr0 + i1 * source_stride1;
            target_addr1 = target_addr0 + i1 * target_stride1;
            for ( i2 = s2; i2 < l2; i2 += TENSOR_TRANSPOSE_SIMD_WIDTH ) {
                source_addr2 = source_addr1 + i2 * source_stride2;
                target_addr2 = target_addr1 + i2 * target_stride2;
            }
        }
    }
}
\_m128 vector0;
vector0 = \_mm_load_ps(source + source_addr2);
\_mm_stream_ps(target + target_addr2, vector0);
}
}
}

return 0;
}
The following piece of code is part of a sample test program that uses the above routine to transpose a 3D tensor in parallel.

```c
#define NUM_THREAD 6

#define DEPTH 1536
#define LENGTH 625
#define WIDTH 1536

#pragma omp parallel num_threads(NUM_THREAD) shared(src, dst) default(none)
{
    int threadNum = omp_get_thread_num();
    int numOfThread = omp_get_num_threads();

    int start = DEPTH/numOfThread*threadNum;
    int end = DEPTH/numOfThread*(threadNum+1);

    TensorTranspose3D(src, dst, DEPTH, LENGTH, WIDTH,
                      2, 1, 0,
                      start, 0, 0,
                      end, LENGTH, WIDTH,
                      threadNum);
}
```
Bibliography


[6] “Oracle’s SPARC T5-2, SPARC T5-4, SPARC T5-8, and


[32] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout,


