Autotuning Divide-and-Conquer Matrix-Vector Multiplication

by

Payut Pantawongdecha

S.B., C.S. M.I.T., 2015

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2016

Author: _____

Department of Electrical Engineering and Computer Science
May 2016

Certified by: _____

Professor Charles E. Leiserson
Thesis Supervisor
May 2016

Accepted by: _____

Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee
May 2016

Autotuning Divide-and-Conquer Matrix-Vector Multiplication [1]

by

Payut Pantawongdecha

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of Master of
Engineering in Electrical Engineering and Computer Science

Abstract

Divide and conquer is an important concept in computer science. It is used ubiquitously to simplify and speed up programs. However, it needs to be optimized, with respect to parameter settings for example, in order to achieve the best performance. The problem boils down to searching for the best implementation choice on a given set of requirements, such as which machine the program is running on. The goal of this thesis is to apply and evaluate the Ztune approach [14] on serial divide-and-conquer matrix-vector multiplication.

We implemented Ztune to autotune serial divide-and-conquer matrix-vector multiplication on machines with different hardware configurations, and found that Ztune-optimized codes ran 1%-5% faster than the hand-optimized counterparts. We also compared Ztune-optimized results with other matrix-vector multiplication libraries including the Intel Math Kernel Library and OpenBLAS.

Since the matrix-vector multiplication problem is a level 2 BLAS, it is not as computationally intensive as level 3 BLAS problems such as matrix-matrix multiplication and stencil computation. As a result, the measurement in matrix-vector multiplication is more prone to error from factors such as noise, cache alignment of the matrix, and cache states, which lead to wrong decision choices for Ztune. We explored multiple options to get more accurate measurements and demonstrated the techniques that remedied these issues.

Lastly, we applied the Ztune approach to matrix-matrix multiplication, and we were able to achieve 2%-85% speedup compared to the hand-tuned code.

This thesis represents joint work with Ekanathan Palamadai Natarajan.

Thesis Supervisor: Professor Charles E. Leiserson
Title: Edwin Sibley Webster Professor in Electrical Engineering and Computer Science

[1] This research was supported in part by NSF Grants 1314547 and 1533644.

Acknowledgments

I would like to express my sincerest gratitude to my adviser, Professor Charles E. Leiserson, for his guidance and support over the past years. Despite his busy schedule, he has been giving useful advice and ensuring that I was on track to a fruitful goal. He has been not only an admirable mentor but also an aspiring academic role model. The opportunity to follow in his footsteps has been an honor for me. I would like to extend my thanks to graduate student mentor and colleague, Ekanathan, for working with me to achieve the results presented here. He has given numerous pieces of advice, without which this thesis would not have reached completion.

I would like to thank all my friends at MIT who share good and bad times with me, and make years at MIT one of the most memorable moments in my life.

Lastly, my accomplishment today is only possible because of the support from my loving family. I would like to thank mom, dad and sister–Jeerawan, Tanakorn, and Raktapa–for always being by my side.

Contents

CONTENTS

1 Introduction

Improving performance is one of the most crucial objectives in code development. We often employ the same set of codes in contexts that may be vastly different, but variations in factors ranging from machine specifications to the nature of the computational problem can influence the performance of a program. Under new circumstances, the program might benefit from using an alternative implementation strategy, which is the developer's task to identify.

We can think of this objective as a search problem over choices of all possible implementations. Traditionally, we can find the best implementation choice using various approaches. One example is trial and error based on prior knowledge of the algorithms and computer architectures, but this iterative approach can become tedious and labor-intensive. Moreover, performance is not portable, which means that the best implementation for one machine might not yield the best performance on another. Developers would have to hard-code the programs on each individual machine to achieve the optimal result–an inefficient and arduous process, especially for commonplace codes that are going to be used in millions of machines.

Sorting is an example of algorithms whose performance depends on the choice of implementation. Two of the most popular sorting algorithms are insertion sort, which rearranges the numbers one-at-a-time, and merge sort, which divides the

array into halves before recursively sorting the subarrays and merging the results. In theory, merge sort has an asymptotic running time of $O(n \log n)$, which is faster than insertion sort's $O(n^2)$. In practice, however, insertion sort may run faster on small arrays because it does not have the extra recursive function-call overhead that merge sort needs. The preferred choice among these two sorting methods would therefore depend on the array size.

Figure 1.1 shows a piece of code in `stl_algo.h` from the `gcc` library that has the cutoff of 15; insertion sort is used for arrays of size smaller than or equal to 15, or merge sort will be run otherwise. This number 15 was hard-coded, presumably for the best performance on computers from decades ago. The best cutoff for today's computers can be a lot bigger due to improvements in factors such as processing speed, memory size, and architecture optimizations that may benefit insertion sort. As a consequence, this code will run merge sort in some cases for which insertion sort would perform better. Such discrepancies between the optimal and the actual implementation choices prevent programs from achieving their full potential.

The inefficiency, as exemplified by the sorting problem, calls for a more powerful means of determining the optimal implementation choice. This quest has led to an effort to automate the searching process, which culminates in program autotuners. A program autotuner generally searches for the best implementation choice within the search space and can tune for each machine individually. Specifically, when programs are ported from one machine to another, autotuners determine the most suitable implementation in the new context and ensure that the programs still achieve the best performance. Because autotuners already take into account the characters of the programs and the machine's hardware configurations, code developers would not need to re-examine these factors later on. Autotuners can be easily employed by anybody and obviate the need for the original program developer to

gcc/blob/master/libstdc++-v3/include/bits/stl_algo.h

```
2762 /// This is a helper function for the stable sorting routines.
2763 template<typename _RandomAccessIterator, typename _Compare>
2764   void
2765   __inplace_stable_sort(_RandomAccessIterator __first,
2766                         _RandomAccessIterator __last, _Compare __comp)
2767   {
2768     if (__last - __first < 15)
2769       {
2770         std::__insertion_sort(__first, __last, __comp);
2771         return;
2772       }
2773     _RandomAccessIterator __middle = __first + (__last - __first) / 2;
2774     std::__inplace_stable_sort(__first, __middle, __comp);
2775     std::__inplace_stable_sort(__middle, __last, __comp);
2776     std::__merge_without_buffer(__first, __middle, __last,
2777                                 __middle - __first,
2778                                 __last - __middle,
2779                                 __comp);
2780   }
```

Figure 1.1: Hard-coded optimization for stable merge sort with constant 15 in `std::stable_sort`.

be present during the re-tuning process.

Divide and conquer is a type of program that can take great advantage of program autotuners because its search space is relatively limited and explicit. The most common pattern in divide and conquer is to first break a problem into smaller subproblems; when the size of each subproblem becomes sufficiently small, we then execute a more straightforward code called the base case to solve it. Divide and conquer, like merge sort, is usually most beneficial for large-sized problems, but might slow down codes for smaller problems because of several reasons such as extra recursive function-call overhead. Figure 1.2 shows an example of a divide-and-conquer matrix-matrix multiplication code with the base-case cutoff at 64, which is optimized for AWS1. [1]

The goal of this thesis is to study how to autotune serial divide-and-conquer

[1]Machine specifications can be found in Appendix A.

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B, C, n, row_size)
 1  if n ≤ 64
 2      SQUARE-MATRIX-MULTIPLY-BASE-CASE(A, B, C, n, row_size)
 3  else partition A, B, and C into four n/2 × n/2 matrices:
 4      A11, A12, A21, A22, B11, B12, B21, B22, C11, C12, C21, C22.
 5      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2, row_size)
 6      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2, row_size)
 7      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2, row_size)
 8      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2, row_size)
 9      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2, row_size)
10      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2, row_size)
11      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2, row_size)
12      SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2, row_size)
13
SQUARE-MATRIX-MULTIPLY-BASE-CASE(A, B, C, n, row_size)
1  for i = 1 to n
2      for j = 1 to n
3          for k = 1 to n
4              C[i · row_size + j] += A[i · row_size + k] · B[k · row_size + j]
```

Figure 1.2: Matrix-matrix multiplication pseudocode where A, B, C are square matrices of a power-of-two size with the assumption that they are stored in row-major order.

matrix-vector multiplication. We choose this topic because considerable research has already been done on *Basic Linear Algebra Subprogram* (**BLAS**) level 3 problems such as matrix-matrix multiplication, while studies into BLAS level 2 problems such as matrix-vector multiplication are still lacking.[2] We also focus on the serial version, in which the programs use only one thread on a single-core processor, because autotuning parallel codes would introduce complications such as difficulty in time measurement and work scheduling nondeterminism that could confound our results. In the following subsections, we summarize the existing approaches and the Ztune approach to tuning.

[2]More information on BLAS can be found in [9, p. 13].

Existing Approaches

There are several existing methodologies that can be used to tune programs. In the first so-called *model-based* approach, we build a model that describes the problem we are trying to tune. Based on this model, we can unequivocally determine the most suitable implementation choice that will yield the optimal result. The extent of performance improvement is limited by our ability to construct a reasonable model. This is not always achievable because it may be difficult to build or it may not comprehensively address every aspect of the problem. We demonstrate the use of a model to hand-tune matrix-vector multiplication in Chapter 2.

The next two approaches–*exhaustive* search and later developed *heuristic* search–are used in application-specific autotuners. An exhaustive autotuner tries every possible implementation and therefore will always find the most suitable choice, but it can take a large amount of time to iterate through the whole search space [7, 6, 19]. A heuristic autotuner, on the other hand, uses heuristics and machine learning to help determine an optimal choice, although this might not be the best possible outcome within the search space. [1, 3, 4, 5, 11, 12, 13, 18].

There are two other non-autonomous strategies for improving performance of a related problem, matrix-matrix multiplication, that are also worth consideration. The first strategy, *tiling* [17, 16], divides the multiplying matrices into submatrices whose size fits exactly in cache. Tiling requires knowledge of the cache size; as a consequence, the result will be specific to only one machine. It also introduces a few tuning parameters for multilevel cache environments as well as several levels of nested loops. The second strategy, *cache-oblivious* matrix-matrix multiplication [8], divides the matrices along the largest dimension and recurses in a typical divide-and-conquer fashion independently of cache size. Cache-oblivious algo-

rithms, in contrast to tiling, use an optimal amount of work and automatically perform well in multilevel cache environments without any knowledge of hardware parameters.

Contributions

This thesis is the result of joint work with Ekanathan Palamadai Natarajan. We apply the Ztune routine, originally developed in [14], on matrix-vector multiplication. Contrary to the popular trend in the field, which was leaning toward heuristic autotuners, Ztune was designed to be a domain-specific exhaustive autotuner. It searches for the best divide choice at each step of the matrix-vector multiplication recursion; i.e., at every step, Ztune determines the best option among

1. dividing the multiplying matrix and the resulting vector by row,
2. dividing the multiplying matrix by column and the multiplying vector by row, or
3. executing the base case, i.e., a nested loop.

The divide choice for each subproblem is then stored in its corresponding node. After identifying all the optimal choices, we build a plan, which is a tree node for the original problem. When we execute matrix-vector multiplication, we walk from the root of the plan down to the leaves, following the best implementation choices as established by Ztune. We discuss Ztune and how we apply it to matrix-vector multiplication in more detail in Chapter 3. We then employ supplemental techniques to reduce Ztune's running time in Chapter 4.

Although applying Ztune to matrix-vector multiplication was rather straightforward, there were several obstacles that would have prevented us from obtaining

the optimal result. In particular, attempts to reduce the tuning time caused Ztune to output a suboptimal program as a result of inconsistent cache states, inconsistent cache alignments and noise in the measurement. We discuss these issues and provide additional techniques to improve Ztune's reliability in Chapter 5.

	Hand-Tuned	*Ztune*			*Otune*	
Machine	*Runtime*	*Runtime*	*Tuning Time*	*Speedup*	*Runtime*	*Speedup*
AWS1	2.707	2.635	0.119	1.03	2.665	1.02
AWS2	2.655	2.603	0.197	1.02	2.598	1.02
C9	3.302	3.245	0.010	1.02	3.243	1.02

Figure 1.3: Running times (in seconds) of hand-tuned, optimized Ztune, and Otune implementations of matrix-vector multiplication of size 25000×100000. Each value is the geometric mean of 10 repeated runs. The Speedup columns refer to the ratios of running times *Hand-Tuned*/*Ztune* and *Hand-Tuned*/*Otune*. The Tuning Time column is the time Ztune uses for tuning. Otune is given 60 seconds.

The overall results of applying Ztune to matrix-vector multiplication are shown in Figure 1.3. We ran all tests on 3 different machines, and their hardware configurations are described in Appendix A. We are showing the results of only one representative problem size since the magnitude of the speedups turned out to be comparable regardless of the input size. We hand-tuned the problem using a cache-based model described in Chapter 2. The hand-tuned code gave reasonably good performance and we used it as the baseline for all future comparisons. The findings in this thesis support the hypothesis that hand-tuned codes do not always give the best performance, further highlighting the benefit of autotuners.

Ztune-optimized codes ran 1%-5% faster than the hand-optimized counterparts. The Tuning Time column shows that Ztune tuned extremely quickly and usually finished tuning within a small fraction of the time needed to execute the problem itself. We also used OpenTuner [2] to implement an autotuner called *Otune* that optimized the row size and the column size cutoffs for the base-case execution, and

was given 60 seconds to tune. Ztune and Otune yielded roughly the same performance speedups, but Ztune finished tuning within one second whereas Otune often took more than 10 seconds to reach its optimal cutoffs.

Thesis Organization

This thesis has 7 chapters in addition to this introduction:

Chapter 2 describes the matrix-vector multiplication problem and builds a "model" for hand-tuning, which we use as the baseline for comparing all future performance. We also state the assumptions about the matrix-vector multiplication problem that apply throughout this thesis.

Chapter 3 describes the Ztune algorithm and its method of estimating the execution cost. We also estimate the tuning time of basic Ztune and outline methods to implement Ztune's helper functions for specific uses on matrix-vector multiplication.

Chapter 4 describes the application of two strategies, equivalence and divide subsumption from [14], to improve Ztune's tuning time. We also demonstrate that when using these strategies, Ztune returns a suboptimal plan as a result of a reduced tuning time and other possible factors.

Chapter 5 investigates why Ztune gives a suboptimal plan and provides a few strategies to resolve the issue.

Chapter 6 compares the performance of Ztune-optimized codes with other matrix-vector multiplication libraries including the Intel Math Kernel Library and OpenBLAS.

Chapter 7 applies Ztune with equivalence and divide subsumption to the matrix-matrix multiplication problem. We show that under certain circumstances, Ztune can yield an usually large speedup.

Chapter 8 suggests ways to improve on our work and Ztune in the future.

Finally, Appendix A describes the detailed specifications of the hardware that we use for our experiments.

1. INTRODUCTION

2 Tuning Matrix-Vector Multiplication

Matrix-vector multiplication is a common problem in computer science. In this chapter, we explain the general characteristics of matrix-vector multiplication, as well as the divide-and-conquer strategies to execute it. We also state the assumptions about the problem, along with the notations that we use in this thesis, for consistency. Lastly, we describe a tuning "model" based on cache as a foundation for hand-tuning the problem. This model suggests that dividing the vectors in a certain way will accelerate the code–a hypothesis that we verified experimentally.

Figure 2.1 gives a quick revision of matrix-vector multiplication. Each element in the resulting vector is simply the dot product between the corresponding row of the matrix and the multiplying vector.

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Figure 2.1: Matrix-vector multiplication $y = Ax$, where $y_i = \sum_{j=1}^{n} a_{ij}x_j$ for $i = 1, \ldots, m$.

Matrix-vector multiplication can be computed using the base-case nested loop, shown as MATRIX-VECTOR-MULTIPLY-BASE-CASE in Figure 2.2. Alternatively, it

can be solved with a divide-and-conquer approach, which carries out one of the following actions at each step of the recursion:

1. dividing the multiplying matrix and the resulting vector by row,
2. dividing the multiplying matrix by column and the multiplying vector by row, or
3. executing the base case, i.e., a nested loop.

Note that dividing the matrix down to 1×1 submatrices would impair rather than improve performance in practice because the extra function-call overhead would exceed the cache advantage of divide and conquer. A base-case cutoff is required to avoid such a scenario. Figure 2.2 shows a serial divide-and-conquer matrix-vector multiplication code that only executes the base case on a 1×1 submatrix. We may change line 1 to

1 **if** $m \leq 64$ and $n \leq 64$

to execute the base case whenever both dimensions of the matrix go below 64. If either dimension is larger than 64, the code divides the matrix along the bigger dimension.

Assumptions

Before moving forward, we dedicate this section to describing the general assumptions upon which this thesis is based.

- We focus on the serial version of divide-and-conquer matrix-vector multiplication, which means that we use only one thread on one processor core to execute the whole program. Whenever we mention a matrix-vector multiplication problem, we refer to it in a divide-and-conquer context, as outlined in Figure 2.2.

```
MATRIX-VECTOR-MULTIPLY-RECURSIVE(A, x, y, m, n, column_size)
 1  if m ≤ 1 and n ≤ 1
 2      MATRIX-VECTOR-MULTIPLY-BASE-CASE(A, x, y, m, n, column_size)
 3  elseif m > n // divide by row
 4      f = ⌊m/2⌋
 5      MATRIX-VECTOR-MULTIPLY-RECURSIVE(A, x, y, f, n, column_size)
 6      MATRIX-VECTOR-MULTIPLY-RECURSIVE(
 7          A + f · column_size, x, y + f, m − f, n, column_size)
 8  else // divide by column
 9      f = ⌊n/2⌋
10      MATRIX-VECTOR-MULTIPLY-RECURSIVE(A, x, y, m, f, column_size)
11      MATRIX-VECTOR-MULTIPLY-RECURSIVE(
12          A + f, x + f, y, m, n − f, column_size)
MATRIX-VECTOR-MULTIPLY-BASE-CASE(A, x, y, n, column_size)
1  for i = 1 to m
2      for j = 1 to n
3          y[i] += A[i · column_size + i] · x[j]
```

Figure 2.2: Matrix-vector multiplication pseudocode for computing $y = Ax$, assuming that the matrix and the vector are stored in row-major order. The parameter *column_size* is the original column size, and m, n are the row size and the column size of the matrix, respectively.

- We assume that matrices and vectors always start in DRAM when they are allocated on the heap for the first time and we have not accessed it yet, and are not stored in L1, L2 or L3 cache.

- Matrices and vectors are stored in row-major order; i.e., each element in the array is located in memory following the previous element from the same row. The first element on each row is located after the last element from the previous row. Examples are given in Figure 2.3.

- We use `double` as the data type for values in matrices and vectors. The size of `double` is 8 bytes for our machine architectures and compilers.

- All machines that we use have a cache line of 64 bytes, and hence exactly 8 consecutive elements of type double from a matrix or a vector can fit in one

cache line.

- The first element of the matrix and the vector is cache-aligned. Specifically, its address in memory is a multiple of 8.
- We assume that the dimensions of all input matrices and vectors are multiples of 8 for consistency and for simplicity of the analysis. This means that the start of each matrix row is cache-aligned. (Despite such specific requirements of this assumption, our approach is still generalizable because we can always pad any arbitrarily-sized matrix or vector into multiples of 8.)
- In the context of matrix-vector multiplication, we use m to denote the row size of the multiplying matrix (and hence the size of the resulting vector), and n to denote the column size of the multiplying matrix (and hence the size of the multiplying vector). The variable m is also called the *first dimension* of the problem, and n the *second dimension*.
- Speedup of an implementation means the ratio

$$\frac{\text{the running time of the hand-tuned version}}{\text{the running time of the implementation}},$$

unless stated otherwise. Greater speedup indicates better performance.

- We define the top-left corner of a matrix to be coordinates $(0, 0)$. The first and the second coordinates increase as we move along rows and columns, respectively. The bottom-right coordinates are exclusive; i.e., the bottom-right coordinates of an $m \times n$ matrix are denoted (m, n), not $(m - 1, n - 1)$. We also use this notation to specify a submatrix. For example, a submatrix may start at coordinates $(8, 16)$, inclusive, and end at $(12, 24)$, exclusive. We call this submatrix $(8, 16, 12, 24)$.

- A matrix A' is a *submatrix* of A if and only if A' is the first argument to the call of MATRIX-VECTOR-MULTIPLY-RECURSIVE from Figure 2.2 at some point during the execution of well-defined MATRIX-VECTOR-MULTIPLY-RECURSIVE with A as the first argument and a special condition that it nondeterministically divides by row or column instead of having deterministic check in line 3 and 8. A submatrix of A is *not* defined as a matrix that can be obtained by removing a collection of rows and columns from A. A *subvector* of a vector is defined similarly.

- A matrix-vector multiplication problem $y' = A'x'$ is a *subproblem* of a matrix-vector multiplication problem $y = Ax$ if and only if A', x', y' are the first, second, and third arguments to the call of MATRIX-VECTOR-MULTIPLY-RECURSIVE at some point during the execution of well-defined MATRIX-VECTOR-MULTIPLY-RECURSIVE with A, x, y as the first, second, and third arguments and a special condition that it nondeterministically divides by row or column instead of having deterministic check in line 3 and 8.

$$A[12] \Rightarrow \begin{pmatrix} A[0] & A[1] & A[2] \\ A[3] & A[4] & A[5] \\ A[6] & A[7] & A[8] \\ A[9] & A[10] & A[11] \end{pmatrix}$$

$$A[mn] \Rightarrow \begin{pmatrix} A[0] & A[1] & \cdots & A[n-1] \\ A[1 \cdot n + 0] & A[1 \cdot n + 1] & \cdots & A[1 \cdot n + (n-1)] \\ \vdots & \vdots & \ddots & \vdots \\ A[(m-1) \cdot n + 0] & A[(m-1) \cdot n + 0] & \cdots & A[(m-1) \cdot n + (n-1)] \end{pmatrix}$$

Figure 2.3: Examples of matrices in row-major order.

How to Hand-Tune Divide-and-Conquer Matrix-Vector Multiplication

Matrix-vector multiplication makes the most use of caching if the size of the vector fits exactly in cache. This is because, unlike in matrix-matrix multiplication, there is no temporal locality in the matrix–each element in the matrix is used only once. On the other hand, each element of the multiplying vector is used repetitively to compute the resulting vector, suggesting that the vector can take advantage of caching. The effect of cache on performance will be particularly evident when the vector is large.

Using the above "model," we hypothesize that the multiplying vector should fit in cache to take the greatest advantage of it. We therefore keep dividing the vector in half until the size of the subvectors just fits in cache, and then execute the base-case. There is no use dividing it any further; not only would this not increase cache performance, but it could also slow the program down by adding extra function-call overhead. We tested this hypothesis using following experiment. We used machines that had a 32KB L1 cache, which could store 4096 doubles. We ran three implementations of a 1000×3000000 matrix-vector multiplication:

1. The first implementation was a straight nested loop.
2. The second implementation executed the base case when the size of the multiplying vector was smaller than 4096–the cache size. (While there might not be enough space to store both a row of the matrix and the multiplying vector of size, say, 4095, we found that using a smaller cutoff in the range of 1024-4096 yielded similar results.)
3. The third implementation executed the base case when both dimensions of the matrix were smaller than 64. This represented the over-dividing scenario.

Machine	*Nested Loop*	$n < 4096$	$m, n < 128$
AWS1	3603.32	3260.27	6660.81
AWS2	3635.51	3234.35	6764.46
C9	5205.75	4110.46	9938.36

Figure 2.4: Running times (in milliseconds) using three different choices of base-case cutoff when multiplying a matrix and a vector of size 1000×3000000. Each value is the minimum of 5 repeated runs.

Figure 2.4 corroborates our claim: the second implementation, with the base-case cutoff corresponding to the cache size, performed the best. We therefore conclude that the hand-tuned divide-and-conquer matrix-vector multiplication implementation is to divide the multiplying vector until it just fits in cache. We will use this result as the baseline against which we compare all future Ztune performance.

3 Ztune

The primary autotuner strategy that we employ in this thesis is Ztune, which was originally described in [14]. We dedicate the first part of this chapter to explaining Ztune, describing its tuning mechanisms, and outlining a method to evaluate its performance with the *cost* function. We also estimate the theoretical cost for tuning and set goals to improve it in the next chapter.

In this first part, we describe how Ztune works and how it correctly captures the program running time. We borrow notations from [14], and readers can find more details there.

Ztune is a domain-specific exhaustive autotuner that we apply to matrix-vector multiplication. In the original paper by Natarajan et al., Ztune was applied to stencil computation problems, but here we adapt it to matrix-vector multiplication problems. Ztune either identifies the best divide choice or decides to execute the base case for each problem Z by building a plan. A *plan* is an ordered tree of *plan nodes* where the root node corresponds to the input problem. Each plan node z contains information about how to execute that specific problem; i.e., z contains:

- $z.choice$: the divide choice for z. In the context of matrix-vector multiplication, there are only three choices: division by row (of the matrix, and row of the resulting vector), division by column (of the matrix, and row of the multiplying

vector), or execution of the base case (also known as nested-loop multiplication).

- z.*cost*: the running time for z following z.*choice*. This value accounts for all the time spent from the start to the end of the execution if we follow z.*choice*.
- z.*children*: the list of Z's children in case the choice is to divide, or NIL for the base case. Suppose the best divide choice for problem Z is to divide it into subproblems $Z_{c_1}, Z_{c_2}, \ldots, Z_{c_{k_c}}$. Then, z.*children* will be the list of $z_{c_1}, z_{c_2}, \ldots, z_{c_{k_c}}$ where z_{c_i} is the plan node for Z_{c_i}. In the context of matrix-vector multiplication, a child of Z is a multiplication between a submatrix and a subvector. The list z.*children* has either exactly two elements from dividing a problem along a dimension, or nothing for the base case.

There are four helper functions in the ZTUNE pseudocode in Figure 3.1:

- LOOKUP(Z): returns the plan node z for Z, or NIL if empty.
- CHOICE(Z): returns the set of possible divide choices for Z.
- BASE-CASE(Z, z): executes the base-case code on Z.
- INSERT(Z, z): inserts the plan node z for Z into the lookup table.

The actual implementation of these functions varies depending on the problem and the programmer. This subsection only touches upon some of these functions, but we will describe their specific use for matrix-vector multiplication in greater detail in a later section.

As a means to evaluate performance, we hereby discuss how Ztune measures *cost*, i.e., the time used to execute the code on a machine. Getting accurate cost is usually tricky in divide-and-conquer codes because we need to account for function-call and function-return overhead. Ztune nonetheless is able to measure cost accurately.

To demonstrate this, let us first define the following notations that Ztune uses for a timer:

- TIC(): starts the timer
- TOC(): stops the timer and returns the elapsed time since the last TIC().

By placing TIC() and TOC() at appropriate places, we will be able to obtain the cost of the process in between.

```
ZTUNE(Z)
 1  call-cost = TOC()
 2  z = LOOKUP(Z)
 3  if z == NIL
 4        Allocate plan node z
 5        z.cost = ∞
 6        z.children = NIL.
 7        for each choice c ∈ C
 8              TIC()
 9              Divide Z using choice c into k_c subproblems Z_{c_1}, Z_{c_2}, ..., Z_{c_{k_c}}
10              rec-cost = TOC()
11              for i = 1 to k_c
12                    TIC()
13                    (z_{c_i}, call-cost_{c_i}) = ZTUNE(Z_{c_i})
14                    ret-cost_{c_i} = TOC()
15                    rec-cost += call-cost_{c_i} + z_{c_i}.cost + ret-cost_{c_i}
16              if rec-cost < z.cost
17                    z.cost = rec-cost
18                    z.choice = c
19                    z.children = [Z_{c_1}, Z_{c_2}, ..., Z_{c_{k_c}}]
20        BASE-CASE(Z, z)
21        INSERT(Z, z)
22  TIC()
23  return (z, call-cost)
```

Figure 3.1: ZTUNE pseudocode from [14].

These functions enable ZTUNE to capture the total cost of solving a problem Z using any specific divide choice. The total cost comprises four portions that we ought to take into account:

```
BASE-CASE(Z, z)
1  TIC()
2  execute base-case
3  base-cost = TOC()
4  if z.cost ≥ base-cost
5        z.cost = base-cost
6        z.choice = −1 // base case
7        z.children = NIL
```

Figure 3.2: BASE-CASE pseudocode from [14].

1. The cost to solve the problem itself. This is the cost to execute a plan or to solve problem Z, stored in $z.cost$.
2. The cost to divide the problem into subproblems. This is the cost of executing line 9 in Figure 2.2. In the context of matrix-vector multiplication, this cost accounts for dividing along a dimension and calculating the locations of the submatrices and the subvectors.
3. Function-call cost. For example, the cost of recursively calling MATRIX-VECTOR-MULTIPLY-RECURSIVE or MATRIX-VECTOR-MULTIPLY-BASE-CASE in Figure 2.2. This cost does not include the execution cost of the rest of the function.
4. Function-return cost. The cost of callee function return. For example, the cost when MATRIX-VECTOR-MULTIPLY-RECURSIVE or MATRIX-VECTOR-MULTIPLY-BASE-CASE in Figure 2.2 exits and returns to the caller. This cost does not include the execution cost prior to the return.

In line 2 of Figure 3.1, Ztune calls LOOKUP(Z) to check whether it has already evaluated Z before and if so, it will use the recorded data. If not, Ztune has to evaluate Z by examining all divide choices and measuring their running times, as shown in line 7. In our case of matrix-vector multiplication, there are only two choices: dividing along the rows or diving along the columns of the matrix. Ztune measures the cost of dividing Z using a specific choice c, and stores this cost in *rec-cost* in

line 10. The variable *rec-cost* is reused as the total cost for each divide choice to potentially be later stored in z.*cost*. Ztune then executes the subproblems $Z_{c_1}, Z_{c_2}, \ldots, Z_{c_{k_c}}$, taking into account the call cost and the return cost to each ZTUNE(Z_{c_i}), and adds the overall cost to *rec-cost* in line 15.

We now describe how Ztune measures function-call and function-return costs. We assume that the caller always calls TIC() right before calling ZTUNE. We place TOC() at the beginning of ZTUNE to measure the function-call cost, which we assign to *call-cost* in line 1. The value *call-cost* is then returned in line 23 along with its plan node z. Similarly, we place TIC() right at the end of ZTUNE in line 22, and the caller is responsible for calling TOC() after ZTUNE ends to measure the function-return cost, which we assign to *ret-cost*. Lines 12-14 show an example of how the caller wraps ZTUNE(Z_{c_i}) with TIC() and TOC(), effectively measuring the call cost and the return cost of ZTUNE(Z_{c_i}). In line 15, *call-cost* returned from ZTUNE is added to *rec-cost* to penalize recursive function calls, and similarly *ret-cost* in line 14 is added to *rec-cost* to penalize recursive function returns.

The variable *rec-cost* therefore represents the total cost of a specific divide choice c, consisting of the costs to divide Z and to call, execute and return ZTUNE on all of Z's children. In lines 16-19, Ztune compares this newly obtained *rec-cost* with the cost of the best possible divide choice so far, z.*cost*. If the new choice performs better, Ztune then updates z.*cost*, z.*choice* and z.*children*.

After Ztune has tried all divide choices, it evaluates the base case by calling BASE-CASE. The function BASE-CASE simply calls TIC(), executes the base-case nested loop, and finally calls TOC(). Ztune updates the plan node z if performance improves.

Application to Matrix-Vector Multiplication

This section narrows the application of Ztune down to the matrix-vector multiplication problem. We will give specific details on implementing routines such as LOOKUP(Z) and INSERT(Z, z).

Let us first investigate the most basic case where LOOKUP(Z) and INSERT(Z, z) are not utilized, which means we have no lookup table. In this case, we define the functions as follows:

1. LOOKUP(Z): always returns NIL.

2. INSERT(Z, z): does nothing.

Theorem 3.1. *Basic Ztune with no lookup table has $O(m^2n^2)$ tuning time.*

Proof. Let $T(m, n)$ be the time Ztune with no lookup table uses to tune an $m \times n$ matrix-vector multiplication problem. Ztune has to spend cmn on evaluating the base case of this problem, where $c > 0$ is a constant. It then divides the matrix by row or by column and recursively calls ZTUNE on the subproblems. Therefore, we have the following recurrence:

$$T(m, n) = 2T\left(\frac{m}{2}, n\right) + 2T\left(m, \frac{n}{2}\right) + cmn.$$

We will prove that $T(m, n) = O(m^2n^2)$ by the substitution method. We want to show that $T(m, n) \leq dm^2n^2 - cmn$ holds for all $m, n \in \mathbb{Z}^+$, for an appropriate choice of the constant $d > 0$, by induction on $m + n$.

For $m + n = 2$, it is clear that $m = n = 1$, and $T(1, 1)$ is a constant. The inequality holds for an appropriate choice of d. Next, assume that for all $m, n \in \mathbb{Z}^+$ such that

$m + n < K$, the inequality $T(m,n) \leq dm^2n^2 - cmn$ holds. For any $m + n = K$, we have:

$$\begin{aligned} T(m,n) &= 2T\left(\frac{m}{2}, n\right) + 2T\left(m, \frac{n}{2}\right) + cmn \\ &\leq 2\Big(d\Big(\frac{m}{2}\Big)^2n^2 - c\frac{m}{2}n\Big) + 2\Big(dm^2\Big(\frac{n}{2}\Big)^2 - cm\frac{n}{2}\Big) + cmn \\ &= dm^2n^2 - cmn. \end{aligned}$$

The induction is complete and we conclude that $T(m,n) = O(m^2n^2)$. □

Theorem 3.1 shows that basic Ztune with no lookup table has $O(m^2n^2)$ tuning time, which is $O(mn)$ factor larger than the matrix-vector multiplication time. The large tuning time is due to the fact that Ztune is wasting resources re-tuning the subproblems that it has already tuned. While having no lookup table saves space usage, the resulting tuning time is too large.

To improve tuning performance, let us make the first attempt to use a lookup table. A subproblem of a matrix-vector multiplication is defined by a submatrix multiplying a subvector equal to a subvector. A subproblem can in fact be identified by the location and the shape of its submatrix alone, and the two subvectors are determined automatically. Therefore, we can specify a submatrix by a triplet (row size, column size, starting location). We then build a map M between a triplet and a plan node z and utilize M in the following helper functions:

1. LOOKUP(Z): uses the row size, column size, and starting location of Z as a key to find its plan node z in M.

2. INSERT(Z, z): inserts a map between a triplet (row size, column size, starting location) and z into M.

However, implementing such a map without a helper library can be challenging. We will provide an alternative approach that has the same functionality as a map: a lookup table. A simple way to build a lookup table for an $m \times n$ matrix-vector multiplication problem is to create a three-dimensional array where the index of the first dimension specifies the row size, the index of the second dimension specifies the column size, and the index of the last dimension specifies the location of the top-left corner of the submatrix.

The index of the first dimension runs from $0, 1, \ldots, m-1$, representing the submatrix row size of $1, 2, \ldots, m$. The index of the second dimension similarly runs from $0, 1, \ldots, n-1$. Since we use row-major ordering, the top-left corner can be specified by a single number rather than two dimensional coordinates, and thus the index of the third dimension runs from $0, 1, \ldots, mn-1$. The original problem, for example, is stored at index $(m-1, n-1, 0)$. To construct a plan after ZTUNE finishes, we use the data at index $(m-1, n-1, 0)$ to define the root node z of the plan, and then we recursively visit $z.children$ and build the tree of plan nodes. The lookup table is no longer needed after the plan is complete.

This lookup table is simple but extremely inefficient. The three-dimensional array for a 1000×1000 matrix-vector multiplication problem would require $1000 \times 1000 \times 1000000 = 1000000000000$ (1 trillion) slots, which would be too big to fit in memory, and we might be better off without a lookup table. This problem underscores the need for a better approach.

We can reduce the size of the lookup table by realizing the fact that the first and the second dimensions do not need to run all the way from $0, 1, \ldots, m-1$ and $0, 1, \ldots, n-1$. Since we divide the matrix along the row or the column in half when executing, there are only certain submatrix sizes that we will encounter. For in-

stance, if $m = 18$, then the only possible submatrix row sizes are $1, 2, 3, 4, 5, 9, 18$, and we will never encounter $6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17$.

The following theorem, which we borrow from [14], can be used to determine the size of the lookup table.

Theorem 3.2. *Consider the following program:*

DIVIDE-TWO(s, k)

1 *Repeat k times:*

2 *Nondeterministically assign* $s = \lfloor s/2 \rfloor$ *or* $s = \lceil s/2 \rceil$

3 ***return*** *s.*

Let n and k be positive integers. Then, DIVIDE-TWO(n, k) *returns either* $\lfloor n/2^k \rfloor$ *or* $\lceil n/2^k \rceil$.

Proof. First, we will prove the following lemma.

$$\lfloor x + y \rfloor \geq \lfloor x \rfloor + \lfloor y \rfloor$$
$$\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$$

Let $x = a_1 + b_1$ and $y = a_2 + b_2$, where $a_1, a_2 \in \mathbb{Z}$ and $0 \leq b_1, b_2 < 1$. Then,

$$\begin{aligned} \lfloor x + y \rfloor &= \lfloor a_1 + b_1 + a_2 + b_2 \rfloor \\ &\geq a_1 + a_2 \\ &= \lfloor x \rfloor + \lfloor y \rfloor. \end{aligned}$$

The second inequality can be proven similarly. □

Next, we will prove the theorem by induction on k.

Base Case: $k = 0$, the return value of DIVIDE-TWO$(n, 0)$ is n, which is equal to $\lfloor n/2^0 \rfloor$.

Inductive Step: Suppose calling DIVIDE-TWO(n, l) returns a value r, which is either $\lfloor n/2^l \rfloor$ or $\lceil n/2^l \rceil$. Note that DIVIDE-TWO$(n, l+1)$ nondeterministically returns $\lfloor r/2 \rfloor$ or $\lceil r/2 \rceil$.

From the induction hypothesis, we have

$$\begin{aligned} r &\geq \left\lfloor \frac{n}{2^l} \right\rfloor \\ \frac{r}{2} &\geq \frac{1}{2}\left\lfloor \frac{n}{2^l} \right\rfloor \\ &\geq \frac{1}{2}\left(\left\lfloor \frac{n}{2^{l+1}} \right\rfloor + \left\lfloor \frac{n}{2^{l+1}} \right\rfloor\right) && \text{(from the lemma)} \\ &= \left\lfloor \frac{n}{2^{l+1}} \right\rfloor \\ \left\lfloor \frac{r}{2} \right\rfloor &\geq \left\lfloor \frac{n}{2^{l+1}} \right\rfloor. \end{aligned}$$

Similarly, we can show that

$$\left\lceil \frac{r}{2} \right\rceil \leq \left\lceil \frac{n}{2^{l+1}} \right\rceil.$$

Therefore, DIVIDE-TWO$(n, l+1)$ returns either $\lfloor r/2 \rfloor$ or $\lceil r/2 \rceil$, which is equal to either $\lfloor n/2^{l+1} \rfloor$ or $\lceil n/2^{l+1} \rceil$, and the induction is complete. □

Theorem 3.2 shows that we will only encounter submatrices of row sizes

$$m, \left\lfloor \frac{m}{2} \right\rfloor, \left\lceil \frac{m}{2} \right\rceil, \left\lfloor \frac{m}{2^2} \right\rfloor, \left\lceil \frac{m}{2^2} \right\rceil, \ldots, 1$$

and column sizes

$$n, \left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil, \left\lfloor \frac{n}{2^2} \right\rfloor, \left\lceil \frac{n}{2^2} \right\rceil, \ldots, 1.$$

There are only $\Theta(\log m)$ row sizes and $\Theta(\log n)$ column sizes, so the first and the second dimensions of the lookup table only need $\Theta(\log m)$ and $\Theta(\log n)$ slots respectively, down from m and n. The size of the three-dimensional lookup table

now reduces to $\Theta(mn \log m \log n)$, which is substantially smaller than before despite still requiring a large memory.

Theorem 3.3. *Basic Ztune with a lookup table has $\Theta(mn \log m \log n)$ tuning time.*

Proof. For any matrix, Ztune spends constant time dividing it into subproblems and recursively calling ZTUNE on the subproblems. It spends the majority of the tuning time, in regards to this particular matrix, evaluating the base case. Therefore, we will find the time spent evaluating the base case.

Based on the fact that we have $\Theta(\log m)$ row sizes and $\Theta(\log n)$ column sizes, there are $\Theta(\log m \log n)$ possible submatrix sizes. All submatrices of one specific size fit in the original matrix with no overlap. Therefore, for each size, we spend $\Theta(mn)$ time in total on the base case, giving the overall tuning time of $\Theta(mn \log m \log n)$.

We also provide an alternative proof. Consider any element in the original matrix. How much time does Ztune spend on this element? We may answer the question by finding the number of distinct submatrices that contain the element. Let us begin by considering only the first dimension. In the original matrix, the indices of the rows are in the range $[0, m)$. We use the term "segment" to denote a range of indices. When we divide the matrix by row, the element must be in either the top or the bottom submatrix; i.e., the element belongs to a new smaller segment, either the top $[0, m/2)$ or the bottom $[m/2, m)$. Since we can divide by row at most $\Theta(\log m)$ times, there are $\Theta(\log m)$ different row segments that may contain the element. Similarly, there are $\Theta(\log n)$ different column segments that may contain the element. The combination of a row segment and a column segment defines a submatrix that Ztune encounters during the tuning process. Since there are $\Theta(\log m \log n)$ combinations of row and column segments, there are $\Theta(\log m \log n)$ submatrices that contain a particular element. Ztune spends $\Theta(1)$ on an element executing the base

case of a submatrix, so it spends $\Theta(\log m \log n)$ on the element over all submatrices. Since there are mn elements, the tuning time is $\Theta(mn \log m \log n)$. □

From Theorem 3.3, the tuning time (and space) of basic Ztune is $\Theta(mn \log m \log n)$, which is substantially smaller than before, but still not as satisfying as it can be. The tuning time is a factor of $\Theta(\log m \log n)$ larger than the matrix-vector multiplication time, and the space usage for the lookup table is also $\Theta(\log m \log n)$ times larger than the space to store the matrix. Cutting down the row and the column sizes is clearly not enough. We also need to reduce the tuning time and memory usage to an acceptable level. One problem that could arise from working with faster codes, however, is that noise would have a bigger influence on time measurements, which could potentially mislead the tuner to pick a suboptimal plan. Our goal in the next two chapters is to improve the tuning time while arriving at the optimal plan.

4 Improving Tuning Time

The previous chapter concludes that basic Ztune needs too much memory to store the lookup table and takes too long to tune. We utilize Ztune's pruning properties– equivalence and divide subsumption, which are originally described in [14], to provide improvements. These two concepts dramatically reduce the tuning time, which can become as small as 1/10000 of the time used to compute the multiplication itself. Shorter tuning times, however, come at the expense of accuracy in cost measurements. A suboptimal plan may appear to be much faster than it actually is, and the tuner wrongly picks this plan over the optimal one. We will inspect the causes of this problem and propose solutions to improve Ztune's accuracy and reliability in the next chapter.

Equivalence

The *equivalence* (**EQ**) property [14] is intuitive. It states that matrix-vector multiplication problems of the same size have the same cost and the same optimal plan. Equivalence allows us to evaluate one matrix of a particular size and generalize the result across all matrices of the same size, resulting in a massive decrease in tuning time. The number of dimensions of the lookup table decreases from three to two as we can forgo the third dimension that specifies the starting location of the

submatrices. In other words, all submatrices having the same size (as specified by the first and the second dimensions) will have the same cost and the same plan node regardless of their starting position. The size of the lookup table now becomes $\Theta(\log m \log n)$, reduced from the previous $\Theta(mn \log m \log n)$.

Hand-Tuned	*Ztune with EQ*	*Speedup*	*Tuning Time with EQ*
8.35	8.17	1.02	46.60

Figure 4.1: Running times (in seconds) of hand-tuned and basic Ztune with EQ implementations of matrix-vector multiplication of size 75000×100000 on AWS2. Each value is the geometric mean of two runs. The Speedup column refers to the ratio of running times *Hand-Tuned/Ztune with EQ*. The Tuning Time with EQ column is the time Ztune with EQ uses for tuning.

Theorem 4.1. *Ztune with EQ has $\Theta(mn)$ tuning time.*

Proof. The submatrix row sizes are values in the set $M = \{m, \lfloor m/2 \rfloor, \lceil m/2 \rceil, \lfloor m/2^2 \rfloor, \lceil m/2^2 \rceil, \ldots, 1\}$ and column sizes are values in the set $N = \{n, \lfloor n/2 \rfloor, \lceil n/2 \rceil, \lfloor n/2^2 \rfloor, \lceil n/2^2 \rceil, \ldots, 1\}$. Using the EQ property on each submatrix of size $a \times b$, we spend $\Theta(ab)$ on running the base case and storing its plan node in the lookup table. Therefore, the tuning time is

$$\begin{aligned}
\sum_{a \in M} \sum_{b \in N} \Theta(ab) &= \Theta\left(\sum_{a \in M} \sum_{b \in N} ab \right) \\
&= \Theta\left(4 \sum_{a=0}^{\log m} \sum_{b=0}^{\log n} 2^a 2^b \right) \\
&= \Theta\left(4 \sum_{a=0}^{\log m} 2^a \sum_{b=0}^{\log n} 2^b \right) \\
&= \Theta(16mn) \\
&= \Theta(mn).
\end{aligned}$$

□

The tuning time of Ztune with EQ is $\Theta(mn)$ from Theorem 4.1, which is asymptotically the same as the time to multiply the matrix and the vector itself, and is better than basic Ztune's $\Theta(mn \log m \log n)$. Figure 4.1 shows that we are now able to tune a 75000×100000 matrix-vector multiplication in 46.60 seconds whereas the running time of the multiplication itself is 8.35 seconds. This is an upgrade compared to basic Ztune alone, which takes over one hour just to tune a much smaller 1000×1000 problem.

Divide Subsumption

Ztune spends the majority of its tuning time executing and evaluating the base case, which is pointless if the choice is certainly to divide. We can speed up Ztune by pruning unnecessary base-case evaluations using the *divide subsumption* (**DS**) property [14]. The DS property states that if all subproblems choose to divide (rather than to execute the base case), then executing the base case cannot be the optimal choice for the current problem. To put it another way, if every child of a problem Z chooses to divide, then Z itself must also choose to divide, and thus we can skip evaluating the base case on Z to reduce tuning time. We replace BASE-CASE(Z, z) in line 20 of the ZTUNE pseudocode (Figure 3.1) with DIVIDE-SUBSUMPTION(Z, z) from Figure 4.2.

```
DIVIDE-SUBSUMPTION(Z, z)
1  measure-base = FALSE
2  for z' ∈ z.children
3      if z'.choice == −1 // child chooses base-case
4          measure-base = TRUE
5  if measure-base
6      BASE-CASE(Z, z)
```

Figure 4.2: DIVIDE-SUBSUMPTION, modified from [14].

In the context of a matrix-vector multiplication problem, we first evaluate the cost of dividing by row and by column. For the choice with the lower cost, we then check if both of the subproblems choose to divide further as their optimal choice. If so, we skip the base-case evaluation on the original multiplication.

The base-case evaluation is often the most time-consuming part of Ztune on matrix-vector multiplication. Skipping this step can therefore reduce the tuning time massively. Figure 4.3 shows that Ztune's tuning time decreases from 46.60 seconds when using EQ alone (Figure 4.1) to only 0.015 seconds when combining both EQ and DS–a thousand-fold acceleration. Moreover, the tuning time is only 1/1000 of the multiplication time itself.

Hand-Tuned	*Ztune with EQ + DS*	*Speedup*	*Tuning Time with EQ + DS*
8.35	8.15	1.02	0.015

Figure 4.3: Running times (in seconds) of hand-tuned and Ztune with EQ + DS implementations of matrix-vector multiplication of size 75000×100000 on AWS2. Each value is the geometric mean of two runs. The Speedup column refers to the ratio of running times *Hand-Tuned/Ztune with EQ + DS*. The Tuning Time with EQ + DS column is the time Ztune with EQ + DS uses for tuning.

The significant speedup of the tuning time comes at the cost of inaccurate measurements. We have tried over 300 different matrix-vector multiplication problems and noticed that the actual cost of a Ztune-optimized code may exceed that of the hand-tuned code by a factor of 2. For example, Figure 4.4 shows that the Ztune-optimized code for a 20000×75000 matrix-vector multiplication problem on AWS2 is 2.1 times more costly than the hand-optimized counterpart. Such an anomaly appears only sporadically; we have to repeat Ztune with EQ + DS several times to witness one bad result. We also find that Ztune's predicted cost, which can be derived from *z.cost*, does not always match the actual cost. The strategies to shorten the tuning time seem to impair Ztune's ability to consistently pick the optimal choice

and to accurately measure the cost.

Size	*AWS1*	*AWS2*	*C9*
20000×75000	0.49	0.47	0.60
35000×25000	0.63	0.61	0.98
85000×40000	1.01	0.50	0.97

Figure 4.4: Worst speedup ratio of running times *Hand-tuned*/*Ztune*, repeated 10 times for different sizes. Ztune uses EQ + DS properties.

Despite these issues, we will continue to incorporate EQ and DS in Ztune because of the immense reduction in the tuning time. In the next chapter, we will investigate what can go wrong with EQ and DS that consequently damages Ztune's reliability, and propose slight modifications to our strategies to fix this issue.

5 Improving Reliability

Equivalence and divide subsumption tremendously lower Ztune's tuning time, but they also compromise the accuracy in time measurements and hence cost evaluation. A bad choice might appear to be much less costly than it actually is, so Ztune might mistakenly pick this choice as the preferred plan. Execution of this suboptimal plan causes Ztune to underperform hand-tuning. This chapter investigates the factors that impair Ztune's accuracy: noise in the measurement, cache states during the execution, and cache alignment of the matrix and the vector. We then propose a few solutions to minimize the inconsistency arising from these causes, and make sure that Ztune reliably achieves the optimal plan and has a short tuning time.

Noise

Noise in the measurements can come from many sources such as I/O interrupts, CPU/memory usage, CPU temperature, clock function delay, hyperthreading and turbo-boost. In Figure 5.1, we measured the running times of the nested-loop execution of a 2000×2000 matrix-vector multiplication, repeated 20 times. The running times varied from 4.6 to 5.9 milliseconds on AWS1, and from 4.1 to 5.0 milliseconds on AWS2. This variability in the running time was observed even though these machines were quiescent, and hyperthreading and turbo-boost were disabled.

Machine	*Mean*	*Min*	*Max*	*Variance*
AWS1	5.572	4.627	5.923	0.381
AWS2	4.772	4.100	5.018	0.197
C9	5.202	5.180	5.244	0.015

Figure 5.1: Statistics of running times, in milliseconds, of nested loops on 20 runs of matrix-vector multiplication of size 2000×2000.

Noise is especially critical in Ztune with EQ and DS because the error can propagate from a child to its ancestors. Suppose we have a large matrix-vector multiplication problem, such as that of size 100000×100000. Ztune recursively tunes the subproblems until it gets to a 48×48 submatrix, which is just one tiny component of the entire multiplication. It evaluates the base-case cost of this submatrix and obtains a false running time that is much shorter than what it is supposed to be. Ztune records this underestimated cost in the lookup table, which affects the decision of all of its ancestors. To make matters worse, the implementation of EQ means that an inaccurate measurement in one submatrix will be copied over to all other submatrices of the same size. One small error may jeopardize the accuracy of the whole plan.

Unfortunately, there is not much we can do to eradicate noise, but we can suppress it to the minimum. Some of the measures that we take include making sure that the machine is in a quiescent state (i.e., no other jobs are running and no other users are using the machine) and turning off features such as hyperthreading and turbo-boost. As stated earlier, the running times will still vary even when these measures are in place (Figure 5.1).

Cache State

An inconsistent cache state during tuning or execution may also impair Ztune's ability to measure costs correctly. Recall that A, B, C are stored in row-major order. Consider Figure 5.2 where we tune a 4×4 matrix-vector multiplication, $A \times B$. By calling ZTUNE$(A \times B)$, Ztune evaluates the division of A by row and by column, and then evaluates the base case. Let us closely examine what happens to the cache state of A during this process. Suppose Ztune divides A by row and recursively calls ZTUNE$(A[0:8] \times B[0:2])$ and ZTUNE$(A[8:16] \times B[2:4])$. At the end of both recursive calls, the base case is executed on $A[0:8] \times B[0:2]$ and $A[8:16] \times B[2:4]$, so the CPU reads $A[0:8]$ and $A[8:16]$ in from memory, and they are now stored in L1 cache. When ZTUNE later proceeds to evaluate the base case for $A \times B$, the whole matrix A is already stored in L1 cache. As a result, Ztune underestimates the base-case cost of $A \times B$, because the CPU does not have to spend time reading matrix A from DRAM.

$$\begin{pmatrix} A[0] & A[1] & A[2] & A[3] \\ A[4] & A[5] & A[6] & A[7] \\ A[8] & A[9] & A[10] & A[11] \\ A[12] & A[13] & A[14] & A[15] \end{pmatrix} \begin{pmatrix} B[0] \\ B[1] \\ B[2] \\ B[3] \end{pmatrix} = \begin{pmatrix} C[0] \\ C[1] \\ C[2] \\ C[3] \end{pmatrix}$$

Figure 5.2: Matrix-vector multiplication diagram demonstrating inconsistent cache states when assuming EQ.

Figure 5.3 shows the base-case statistics of running Ztune with EQ and DS on a 35000×40000 matrix-vector multiplication. The optimal choice for most submatrices, according to Ztune, is to recursively divide the matrix down to 273×20 and 273×19 submatrices and execute the base case. Ztune predicts the total time spent nested-looping on these 273×20 matrices to be 376.48 milliseconds (computed from

z.cost $\times$ *count*) but the actual execution takes 1037.29 milliseconds, nearly thrice the prediction. Moreover the 273×20 base-case running times vary from 0.005 to 0.057 milliseconds. This statistics shows that the predicted base-case cost does not truly represent the actual base-case cost. We hypothesize that this variation arises from the inconsistency in cache states during tuning and execution (noise is another factor, but there is nothing we can do to solve it).

Base-Case Size	*Count*	*Actual Min*	*Actual Max*	*Predicted Total*	*Actual Total*
273×20	78336	0.005	0.057	376.48	1037.29
273×19	69120	0.004	0.042	321.96	712.88
17×1250	21504	0.023	0.047	438.66	538.30
35×625	7168	0.022	0.044	151.28	182.41

Figure 5.3: Statistics of base-case execution using Ztune with EQ + DS on matrix-vector multiplication of size 35000×40000 on AWS1. Base-Case Size is the size that gets executed as the base case in the plan (i.e., it is not divided further). Count is the number of times each base-case size is executed. Actual Min and Actual Max are the minimum and the maximum of the running times of the actual base-case execution. Predicted Total is the predicted total running time spent on computing all base-case executions of a particular size. The prediction is based on *z.cost* of a plan node. Actual Total is the total running time from the actual execution. All running times are in milliseconds.

We suggest two solutions to remedy the inconsistency in cache states. The first is to randomize the location of the submatrix when evaluating the base case (**RAN**). Specifically, instead of evaluating the base case on a submatrix by calling BASE-CASE in line 6 of Figure 4.2 right away, we randomly pick a new submatrix of the same size within the original matrix to call BASE-CASE on. What we want to achieve from this strategy is to eliminate the bias on cache being warm rather than cold. It is highly likely that we have a cold cache because the randomized submatrix can fall anywhere within the original matrix, especially if the former is much smaller than the latter. Randomization allows us to consistently evaluate a submatrix that is not yet in cache, avoiding the cache-state problem.

We also consider the benefit of randomizing subvector locations, but we avoid it for the following reason. In a matrix-vector multiplication, we likely read in each element of the matrix once from DRAM and there is no temporal locality, so we must account for cold misses on the submatrix when evaluating a subproblem. On the other hand, there is temporal locality in the vector since we reuse its elements multiple times. We choose to preserve this temporal locality by not randomizing subvector locations when tuning.

```
2-DIVIDE-SUBSUMPTION(Z, z)
 1  measure-base = FALSE
 2  for z' ∈ z.children
 3       if z'.choice == −1 // child chooses base-case
 4            measure-base = TRUE
 5       else
 6            for z'' ∈ z'.children
 7                 if z''.choice == −1 // grandchild chooses base-case
 8                      measure-base = TRUE
 9  if measure-base
10       BASE-CASE(Z, z)
```

Figure 5.4: 2-LEVEL DIVIDE-SUBSUMPTION, modified from [14].

The second solution does not solve the problem at its cause–the cache state–but raises the robustness of divide subsumption. We define *n-level divide subsumption* (*n***DS**) as follows: if all subproblems choose to divide down to n levels, then executing the base case cannot be the optimal choice for the current problem. The original DS property from Chapter 4 skips evaluating the base case on a problem if all of its immediate children choose to divide, so it is a 1-level divide subsumption. Figure 5.4 shows a 2-level divide subsumption pseudocode that checks both the children and the grandchildren of Z.

The motivation behind this strategy is that in our experience, Ztune with EQ alone does not introduce noticeable inaccuracy but Ztune with EQ + 1DS does, sug-

gesting that the reliability problem may arise from a shallow DS. Deepening the level of DS ensures that any error in the measurement cannot cause the ancestors to prematurely rule out the base case from the optimal choice. Only when the children choose to divide at multiple levels can we be sure that we can prune the base-case evaluation. However, increasing the depth of DS means that we have to evaluate more base cases, which increases tuning time. We need to find a good balance: the fewest levels of DS that avoid inaccurate measurements.

Cache Alignment

The EQ property assumes that two submatrices of the same size share the same cost, but this assumption may fall apart especially when the submatrices have different cache alignments.

Consider Figure 5.5 in which two submatrices have the same size but different starting positions. Recall that a 64-byte cache line can store 8 doubles. The top row of submatrix $(24, 24, 30, 30)$ is cache-aligned; i.e., the top-left element is stored in the first location of a cache line and the remaining elements follow. Therefore, when we read the top-left element of $(24, 24, 30, 30)$, the rest of its row is automatically stored in L1 cache. However, the top row of submatrix $(0, 12, 6, 18)$ is not cache-aligned; its top-left element is not stored in the first location of a cache line. When we read the top-left element of $(0, 12, 6, 18)$, we only store $(0, 12), (0, 13), (0, 14), (0, 15)$ in L1 cache (along with $(0, 8), (0, 9), (0, 10), (0, 11)$), and we need to fetch $(0, 16)(0, 17)$ from DRAM later.

Even though the two submatrices share the same size, the base-case execution time on the cache-aligned $(24, 24, 30, 30)$ will presumably be smaller than that of $(0, 12, 6, 18)$ because of the number of cold misses. This violates the EQ property.

We need to make sure that whenever we assume EQ between submatrices, they not only share the same size but also have the same cache alignment.

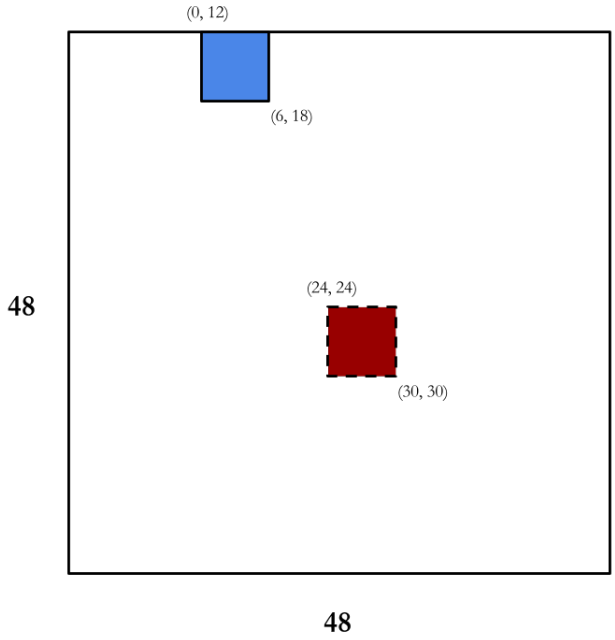

Figure 5.5: Submatrices that have different cache alignments.

We hereby describe two methods to enforce the same cache alignments among equivalent matrices. The first is to divide the problem into the largest smaller power of 2 (**P2**) instead of into halves. The motivation behind this approach comes from the fact that we never observe an anomaly when using Ztune with EQ + 1DS to tune matrix-vector multiplication of power-of-2 sizes.

Figure 5.6 shows a 24×24 matrix-vector multiplication, which we would normally divide along the rows or the columns in half down to 12. In P2, however, the largest power of 2 that is smaller than 24 is 16. We therefore divide the rows or the columns from 24 into 16 and 8. In an actual implementation, one only needs to replace $f = \lfloor m/2 \rfloor$ and $f = \lfloor n/2 \rfloor$ in line 4 and line 9 of Figure 2.2 with $f = 2^{\lfloor \log(m-1) \rfloor}$

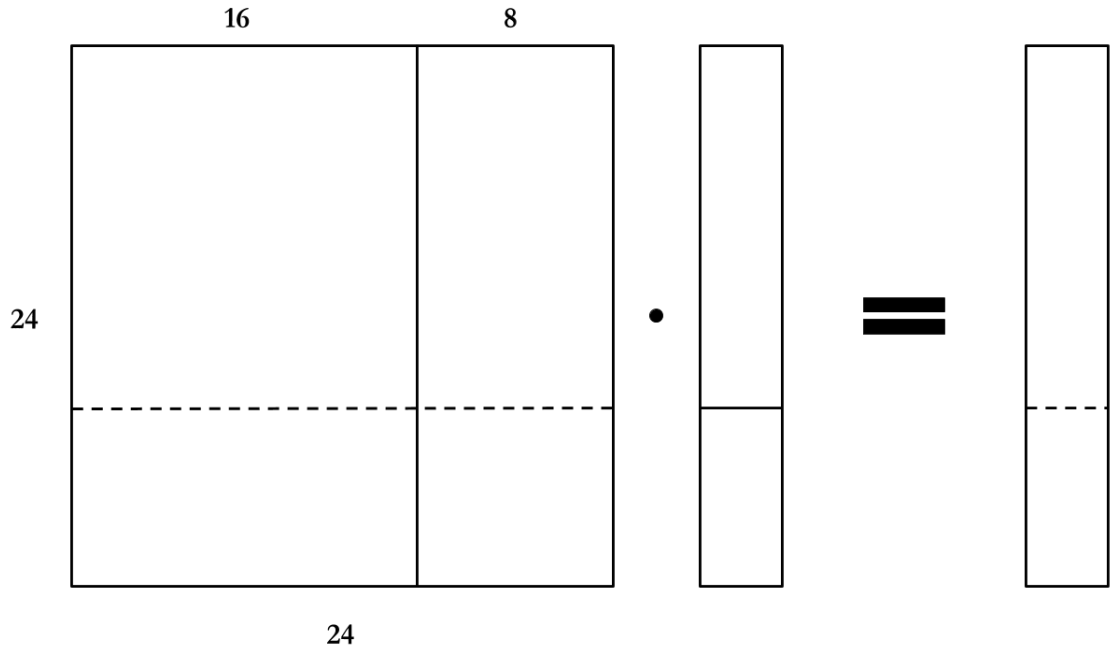

Figure 5.6: P2 division strategy.

and $f = 2^{\lfloor \log(n-1) \rfloor}$ respectively. One might worry that the lookup table would need extra slots to accommodate new row and column sizes. Fortunately, Theorem 5.1 implies that the number of possible row sizes and column sizes remains $O(\log m)$ and $O(\log n)$, unchanged from basic Ztune with EQ. The size of the lookup table therefore also remains unchanged.

Theorem 5.1. *Consider the following program:*

DIVIDE-POW-TWO(s)

1 ***while*** $s > 1$

2 *Let r be the largest power of 2 that is smaller than s*

3 *Nondeterministically do exactly one of the following:*

4 *1. $s = r$*

5 *2. $s = s - r$*

6 *3.* ***return*** *s*

7 ***return*** *s*

Let n be a positive integer. There are $O(\log n)$ distinct values that DIVIDE-POW-TWO(n) *can return.*

Proof. Let k be the number of digits in the binary representation of s. Then, $k = O(\log n)$. Let us observe the following facts:

1. If s is not a power of 2, option 1 in line 4 replaces s by maintaining only the highest set bit in s and disregarding the rest of the bits. If s is a power of 2, it shifts s to the right by 1 bit.

2. After option 1 is executed, s will always be in the form $10\ldots0_2$, which is also a power of 2.

3. If s is not a power of 2, option 2 in line 5 replaces s by taking away the highest set bit of s .

4. If s is a power of 2, option 2 has the same effect as option 1, and we disregard option 2. After this point, the program can either choose option 1 or return, so s will always be a power of 2.

5. If s has been through option 1, it is a power of 2, and hence we disregard option 2.

Next, let us consider three cases that can happen during DIVIDE-POW-TWO(n):

1. We never modify n. There is one possible outcome, the original n itself.
2. The last modification to n comes from option 1. Then n is in the form $10\ldots0_2$. Since n always decreases in each iteration, there are only k possible values of such a form that do not exceed the original n.
3. the last modification to n comes from option 2. From facts 4 and 5, the value n before the last modification is not a power of 2, and n has never been through option 1, which implies that n has been through option 2 only. In other words, we have repeatedly taken away the highest set bit in n and returned. Since we can take away the highest set bit at most k times, there are only k possible outcomes.

From the three cases, there are at most $1 + k + k = 2k + 1 = O(\log n)$ possible values of DIVIDE-POW-TWO(n). □

How does P2 enforce the same cache alignment among equivalent matrices? It makes sure that the starting location (the top-left element) of each submatrix is always the first element in a cache line. In the original matrix, the top-left element is always cache-aligned. Suppose we divide by column into the left and the right halves using P2. The top-left element of the left half is still cache-aligned because it is the same as that of the original matrix. Since the column size of the left half is a power of 2, the offset between the top-left elements of the left and the right halves is a multiple of 8; therefore, the top-left element of the right half is also cache-aligned. We can recursively apply this argument to the subproblems to show that all submatrices are cache-aligned.

What if the row size of the left half is smaller than 8? We assume that dividing the problem down to submatrices whose row size or column size < 8 is never an

optimal option. These submatrices would be so small that there would be no use dividing to that extent. As a result, we only consider the cases in which the submatrix dimensions are greater than 8.

In conclusion, P2 makes sure that all submatrices are cache-aligned in the sense that their starting position, i.e., the top-left element, is always the first element in a cache line. Some might argue that dividing by powers of 2 is unorthodox, so we propose an alternative approach that would work with dividing in half, which is more common.

The second strategy is to impose a new condition onto EQ. The original EQ assumes that two submatrices of the same size will have the same cost, but this premise is broken when their cache alignments differ. A simple fix to this problem is, roughly speaking, to assume EQ between two matrices only when

1. they have the same size, and
2. they share the same starting column.

Figure 5.7 illustrates this strategy with submatrices of size 6×6. The blue matrices with solid lines $(0, 12, 6, 18)$, $(24, 12, 30, 18)$, $(36, 12, 42, 18)$ start at the same column 12 and are considered equivalent to one another, but not equivalent to the red matrices with dashed lines $(24, 24, 30, 30)$ and $(24, 36, 30, 42)$ which start at columns 24 and 36 respectively.

This intuitive idea eliminates the issue of cache alignments–equivalent submatrices must start from the same column and hence must have the same cache alignment. Nevertheless, it comes at the cost of longer tuning times. We now have to recompute the base-case cost for the submatrices that have the same size but start at different columns. Experimental results suggest that the damage is greater than the benefit. In Figure 5.8, we try to tune a 75000×100000 matrix-vector multiplication

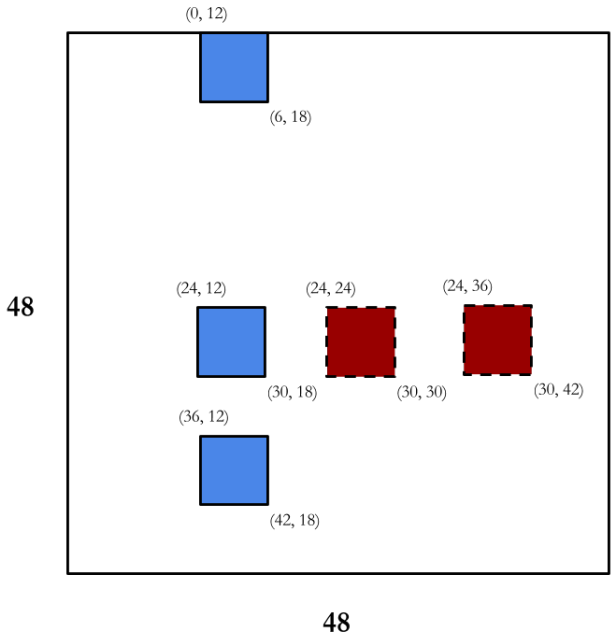

Figure 5.7: Assuming EQ only when submatrices start at the same column. The blue matrices with solid lines are considered equivalent to one another, but not equivalent to the red matrices with dashed lines.

on AWS2. Ztune with EQ + 1DS takes 0.015 seconds, while Ztune with this idea and 1DS takes 56 seconds. We are doing much worse than before in terms of tuning speed, so we will further refine this idea.

EQ	*EQ + 1DS*	*EQ on same column + 1DS*	*CA + 1DS*
46.601	0.015	56.551	0.095

Figure 5.8: Tuning times (in seconds) of different Ztune implementations of matrix-vector multiplication of size 75000×100000 on AWS2.

Since we care more about cache alignment rather than the starting column of the matrix per se, we can relax this requirement slightly. The key is to realize that submatrices that start at different columns may still share the same alignment. We therefore assume EQ between two submatrices only when

1. they have the same size, and
2. they share the same cache alignment; i.e., their starting elements are located at the same position in a cache line.

This is our formal second strategy to tackle the cache alignment issue. We call this strategy **CA**. Figure 5.9 shows submatrices of size 6×6. The blue matrices with solid lines $(0, 12, 6, 18)$, $(36, 12, 42, 18)$, $(24, 36, 30, 42)$ start at the columns which have the same cache alignment (starting columns are 4 mod 8) and are considered equivalent to one another, but are not equivalent to the red matrices with dashed lines $(18, 0, 24, 0)$ and $(24, 36, 30, 42)$ which start at different columns modulo 8.

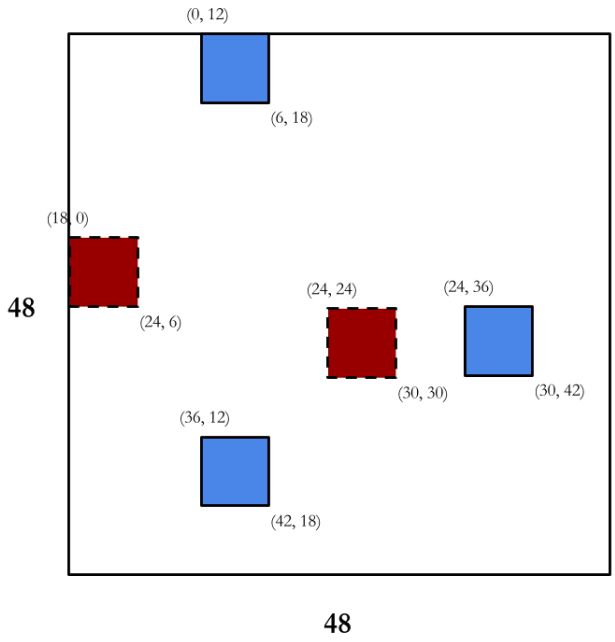

Figure 5.9: Assuming EQ only when submatrices start at the same position in cache. The top-left label of each submatrix indicates its starting column. The blue matrices with solid lines are considered equivalent to one another, but not equivalent to the red matrices with dashed lines.

CA is tremendously faster than the previous idea because it avoids evaluating same-sized submatrices for every different starting column. We only need to eval-

uate same-sized submatrices at most 8 times because there are 8 possible starting positions in cache. Because of this reason, the tuning time of Ztune with CA + 1DS should theoretically be around 8-fold slower than that of Ztune with EQ + 1DS. Our experimental results are somewhat in line with these speculations. In Figure 5.8, where we tune a 75000×100000 matrix-vector multiplication on AWS2, Ztune with CA + 1DS takes only 0.095 seconds. This is over 500 times faster than the previous idea, and about 6 times slower than the 0.015 seconds of Ztune with EQ + 1DS. It is still very fast compared to the multiplication time itself, which is about 8 seconds according to Figure 4.3.

Combinations that Work

We have described many techniques to make Ztune fast and reliable. We first employ equivalence (EQ) and divide subsumption (1DS) to shorten Ztune's tuning time. Ztune + EQ + 1DS, however, suffers from two cache-related problems for which we propose the following solutions:

1. We tackle the issue of inconsistent cache states with randomization (RAN) and n-level divide subsumption (nDS).
2. We address the issue of inconsistent cache alignments by dividing into the largest smaller power of 2 (P2) and by imposing an additional cache-alignment requirement on equivalence (CA).

We have tried several different combinations by picking two strategies, each of which solves one aspect of the cache problems, and appending them onto Ztune + EQ + 1DS (except for the first combination that does not include a strategy to resolve the issue of inconsistent cache states). The top four candidates that perform the best are

- EQ + 1DS + P2
- EQ + 2DS + P2
- EQ + 1DS + P2 + RAN
- CA + 1DS + RAN.

Note that for CA + 1DS + RAN, we do not let the randomization space of the starting location be anywhere in the original matrix, but we limit it to the locations that would share the same cache alignment as the original submatrix (i.e., the original submatrix and the randomized submatrix have the same starting location mod 8).

We test each configuration on 300 different $m \times n$ matrix-vector multiplications, where $m \in \{5000, 10000, \ldots, 75000\}$ and $n \in \{5000, 10000, \ldots, 100000\}$ as long as they fit in memory. We examine the worst-case speedup from each configuration as a measure of robustness: if it is robust, it should not cause a large negative speedup within these 300 trials.

Machine	*EQ+1DS+P2*	*EQ+2DS+P2*	*EQ+1DS+P2+RAN*	*CA+1DS+RAN*
AWS1	0.49	0.95	0.97	0.91
AWS2	0.48	0.97	0.95	0.88
C9	0.57	0.95	0.96	0.92

Figure 5.10: Worst speedup ratio of running times *Hand-tuned*/*Ztune* for each implementation from running 300 matrix-vector multiplications of size $m \times n$ where $m \in \{5000, 10000, \ldots, 75000\}$ and $n \in \{5000, 10000, \ldots, 100000\}$, excluding the sizes that do not fit in the memory of the machine on which this operation is tested.

The results in Figure 5.10 suggest that all of the above configurations are robust except for EQ + 1DS + P2, which have a 1% chance of anomalies yielding around 0.50 speedup. This is expected, considering that we do not include any strategy

to tackle the inconsistent cache state issue. Although these anomalies are quite infrequent, we consider them unacceptable and regard this configuration as a failure. The worst-case speedups for the other three configurations are only about 0.91, and their average speedups are good. We attribute the small negative speedups to noise in the measurements for which we do not have a solution. We consider these three strategies robust and we will evaluate their performance in the next subsection.

We have also tried other configurations, but they either are not robust or take too long to tune, so we do not mention them here.

Results after Improvements

The previous subsection has established three heuristics that can tune matrix-vector multiplication reliably: EQ + 2DS + P2, EQ + 1DS + P2 + RAN and CA + 1DS + RAN. We can now closely inspect different aspects of their performance.

We first examine their tuning times. All of these strategies tune really fast. For a 25000×100000 matrix-vector multiplication on AWS2, the nested loop takes around 2,650 milliseconds while the three strategies only take 26.22, 36.49 and 91.54 milliseconds (Figure 5.11). From the data, it turns out that EQ + 1DS + P2 + RAN tunes the fastest, followed by EQ + 2DS + P2 and CA + 1DS + RAN. The fastest tuning time is only four-fold faster than the slowest one, and is as much as 100 times faster than the nested loop.

We next look at the performance of these heuristics. We run each heuristic on five repeats of a 25000×100000 matrix-vector multiplication and compute its speedup compared to that of the hand-tuned version. The average speedups vary from 1.00-1.04.

Machine	*EQ+2DS+P2*	*EQ+1DS+P2+RAN*	*CA+1DS+RAN*
AWS1	70.19	28.79	106.95
AWS2	26.22	36.49	91.54
C9	34.55	26.24	73.40

Figure 5.11: Average tuning time of running times (in milliseconds) *Hand-tuned/Ztune* for robust implementations from five repeated runs of matrix-vector multiplication of size 25000×100000. The average is computed using the geometric mean.

Machine	*EQ+2DS+P2*	*EQ+1DS+P2+RAN*	*CA+1DS+RAN*
AWS1	1.03	1.03	1.01
AWS2	1.02	1.03	1.01
C9	1.01	1.02	1.00

Figure 5.12: Average speedup ratio of running times *Hand-tuned/Ztune* for robust implementations from five repeated runs of matrix-vector multiplication of sizes 25000×100000. The average is computed using the geometric mean.

By comparing tuning times and speedups among different heuristics in Figures 5.11 and 5.12, Ztune with EQ + 1DS + P2 + RAN not only tunes the fastest but also gives the best speedup of 1.02-1.03. We therefore regard this strategy as the "optimized" Ztune for matrix-vector multiplication. To summarize, the best strategy for matrix-vector multiplication is to run Ztune with

- Equivalence
- 1-level divide subsumption
- Dividing into the largest smaller power of 2.
- Randomized submatrix location for the base case.

Lastly, we give an experimental result on the base-case statistics of applying Ztune with EQ + 1DS + P2 + RAN on a 35000×40000 matrix-vector multiplication. The resulting plans differ between different Ztune runs, and Figure 5.13 shows the base-case execution statistics of one of the plans. This particular plan results in

Base-Case Size	*Count*	*Actual Min*	*Actual Max*	*Predicted Total*	*Actual Total*
256×4096	1224	1.097	1.232	1325.78	1374.05
32×3136	1088	0.105	0.135	118.39	118.23
184×40000	1	7.807	7.807	7.79	7.81

Figure 5.13: Statistics of base-case execution from using Ztune with EQ + 1DS + P2 + RAN on matrix-vector multiplication of size 35000×40000 on AWS1. Base-Case Size is the size that gets executed as the base case in the plan (i.e., it is not divided further). Count is the number of times each base-case size is executed. Actual Min and Actual Max are the minimum and the maximum of the running times of the actual base-case execution. Predicted Total is the predicted total running time spent on computing all base-case executions of a particular size. The prediction is based on *z.cost* of a plan node. Actual Total is the total running time from the actual execution. All running times are in milliseconds.

a 1.03 speedup compared to the hand-tuned version. Comparing this plan to that of Ztune with EQ + 1DS in Figure 5.3, we can see a clear improvement in the accuracy of the predicted time. Values in the Predicted Total and Actual Total columns in Figure 5.13 are close to each other, showing that Ztune with EQ + 1DS + P2 + RAN is able to achieve accurate measurements.

6 Comparison to Existing Approaches

Optimized Ztune always performs at least as well as the hand-tuned codes at matrix-vector multiplication. In this chapter, we take it a step further and compare Ztune to the well-established BLAS libraries, which include the Intel Math Kernel Library (MKL) and OpenBLAS. Ztune is, not surprisingly, still slower than both MKL and OpenBLAS. These libraries are able to attain superior performance because they perform optimization at a lower level that is beyond the scope of this thesis. We also compare optimized Ztune to an implementation of OpenTuner and find that both have roughly the same performance.

Machine	*Ztune*	*MKL*	*OpenBLAS*
AWS1	1.04	1.89	2.04
AWS2	1.03	1.82	1.89
C9	1.11	1.16	2.13

Figure 6.1: Average speedup ratio of running times *Hand-tuned*/*Ztune* for EQ + 1DS + P2 + RAN from 5 repeated runs of matrix-vector multiplication of size 10000×100000, compared to speedups from the Intel Math Kernel Library and OpenBLAS. The average is computed using the geometric mean.

Figure 6.1 compares the speedup of optimized Ztune with that of the DGEMV routines from the Intel Math Kernel Library and OpenBLAS on a matrix-vector multiplication of size 10000×100000, repeated 5 times. The DGEMV routines clearly give a much greater speedup, which is not surprising because these routines incorporate

low-level optimizations that are beyond the scope of this thesis.

Machine	*Ztune*	*Otune*
AWS1	1.03	1.02
AWS2	1.02	1.02
C9	1.02	1.02

Figure 6.2: Average speedup ratio of running times *Hand-tuned*/*Ztune* for EQ + 1DS + P2 + RAN from 10 repeated runs of matrix-vector multiplication of size 25000×100000, compared to speedups of Otune. Otune is given 60 seconds. The average is computed using the geometric mean.

We then compare optimized Ztune with the implementation *Otune* of the OpenTuner framework [2]. Otune optimizes the row and the column size cutoffs for base-case execution, and is given 60 seconds to tune. We then use those cutoffs, which are different for each machine, to replace the value in line 1 of the divide-and-conquer matrix-vector multiplication code in Figure 2.2. Otune timeout is set at 60 seconds to find the cutoff parameter. Figure 6.2 shows that optimized Ztune and Otune have similar performance speedups. Both implementations yield 2%-3% speedup.

7 Results on Matrix-Matrix Multiplication

This thesis has focused heavily on the application of Ztune in matrix-vector multiplication. In this chapter, we translate our findings to a related problem, matrix-matrix multiplication. The program is similar to Ztune for matrix-vector multiplication, except with an additional dimension. We first describe a strategy that tunes fast and is reliable, before moving to square matrices of power-of-two sizes. Interestingly, these power-of-two-sized matrices suffer from conflict misses when tuned with a typical divide and conquer with a constant cutoff [10, 15]. Ztune does not suffer from the same problem and is able to give dramatic improvements in computation speed.

Ztune with EQ + 1DS has reliability issues when run on matrix-vector multiplication, but it does not have any issue with matrix-matrix multiplication because the latter is more computationally intensive and less prone to small errors in the measurement. Therefore, we use Ztune with EQ + 1DS on matrix-matrix multiplication.

We first implement the hand-tuned version of divide-and-conquer matrix-matrix multiplication by recursively dividing along the largest dimension until we execute

Machine	*Hand-Tuned*	*Ztune*	*Speedup*	*Tuning Time*
AWS1	83.47	79.85	1.04	0.35
AWS2	79.72	76.48	1.04	0.16
C9	147.41	146.75	1.01	0.13

Figure 7.1: Running times (in seconds) of hand-tuned and Ztune with EQ + 1DS implementations of matrix-matrix multiplication of size $5000 \times 5000 \times 5000$. Each value is the geometric mean of 5 repeated runs. The Speedup column refers to the ratio of running times *Hand-Tuned*/*Ztune*. The Tuning Time column is the time Ztune uses to tune.

the base case. We choose the base-case cutoff by trying different cutoffs for the first, the second and the third dimensions, and choose the one that gives the best performance. We find that using the cutoff size 64 for all three dimensions gives a reasonably good performance and we use this implementation as our baseline.

Figure 7.1 shows the results of Ztune with EQ + 1DS on matrix-matrix multiplication of size $5000 \times 5000 \times 5000$, geometrically averaged from 5 repeated runs. We achieve a 1.01 average speedup on C9, and up to 1.04 on AWS1 and AWS2. The tuning time is fast as expected, only 1/500 of the time used to run the hand-tuned version of matrix-matrix multiplication.

Size	*Hand-Tuned*	*Ztune*	*Speedup*	*Tuning Time*
$512 \times 512 \times 512$	109.24	82.90	1.32	9.44
$1024 \times 1024 \times 1024$	992.31	654.56	1.52	10.04
$2048 \times 2048 \times 2048$	8099.72	5259.37	1.54	12.50
$4096 \times 4096 \times 4096$	65132.70	45275.70	1.43	21.35

Figure 7.2: Running times (in milliseconds) of hand-tuned and Ztune with EQ and DS implementations of multiplication between square matrices with power-of-two sizes on AWS2.

We observe much greater speedup on power-of-two matrix sizes. Figure 7.2 shows that we are able to achieve up to 1.54 speedup on AWS2 on different power-of-two matrix sizes. It is well known that multiplying power-of-two matrices suffers heavily from cache conflict misses [10, 15] and hence requires a very lengthy

running time. Ztune does not suffer from conflict misses, which results in a large speedup.

7. RESULTS ON MATRIX-MATRIX MULTIPLICATION

8 Future Work

There are several interesting concepts based on Ztune to be explored. The first suggestion is the application of Ztune on parallel divide-and-conquer matrix-vector multiplication and matrix-matrix multiplication. There are complications in measuring time and how the computer schedules parallel jobs, which are beyond the user's control. We can still compute the work and the span of a Ztune plan and find the optimized parallelism based on the work and the span.

The second suggestion is to extend Ztune to any general divide-and-conquer problem. Currently, Ztune is applied to serial matrix-vector multiplication and serial matrix-matrix multiplication in this thesis, and serial stencil computations in its original paper [14]. However, it will be interesting to investigate whether we can achieve a general divide-and-conquer autotuner based on Ztune where users can feed in a piece of code, specifying the recursive parts and the base-case parts, and the Ztune-based autotuner outputs the best plan to execute.

8. FUTURE WORK

A Appendix

Machine Specifications

The specifications of the hardware that we use for experiments are shown in Figure A.1. Machines with AWS in their name are virtual cloud servers provided by Amazon Web Services. The machine *C9* is a cloud platform provided by MIT Computer Science and Artificial Intelligence Laboratory.

	AWS1	*AWS2*	*C9*
Manufacturer	Intel	Intel	Intel
CPU	Xeon E5-2666 v3	Xeon E5-2666 v3	Xeon X5650
Clock	2.90GHz	2.90GHz	2.67GHz
Turbo Boost	Disabled	Disabled	Disabled
Processor cores	16	36	12
Sockets	1	2	2
L1 data cache/core	32KB	32KB	32KB
L2 cache/core	256KB	256KB	256KB
L3 cache/socket	25MB	25MB	12MB
DRAM	30GB	60GB	48GB
Compiler	ICC 13.1.1	ICC 13.1.1	ICC 13.1.1
Operating system kernel	Linux 3.13.0	Linux 3.13.0	Linux 3.13.0
Operating system	Ubuntu Server 14.04.3	Ubuntu Server 14.04.3	Ubuntu 14.04.4

Figure A.1: Specifications of the machines used for benchmarking. Turbo-boost is disabled for reliability of time measurements.

APPENDIX A. APPENDIX

Bibliography

[1] ANSEL, J., AND CHAN, C. PetaBricks: Building adaptable and more efficient programs for the multi-core era. *XRDS 17*, 1 (2010).

[2] ANSEL, J., KAMIL, S., VEERAMACHANENI, K., O'REILLY, U.-M., AND AMARASINGHE, S. OpenTuner: An extensible framework for program autotuning. Tech. Rep. TR-2013-026, MIT CSAIL, 2013.

[3] CHRISTEN, M., SCHENK, O., AND BURKHART, H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS* (2011), IEEE, pp. 676–687.

[4] ŢĂPUŞ, C., CHUNG, I.-H., AND HOLLINGSWORTH, J. K. Active Harmony: Towards automated performance tuning. In *SC* (2002), ACM/IEEE, pp. 1–11.

[5] DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC* (2008), ACM/IEEE, pp. 4:1–4:12.

[6] FRIGO, M. A fast Fourier transform compiler. *ACM SIGPLAN Notices 34*, 5 (May 1999), 169–180.

[7] FRIGO, M., AND JOHNSON, S. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231.

[8] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *FOCS* (1999), IEEE, pp. 285–297.

[9] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*. Johns Hopkins University Press, 1989.

[10] GUSE, M., AND RISTOV, S. Performance gains and drawbacks using set associative cache. *JNIT, Journal of Next generation Information Technology 3* (2012), 87–98.

[11] KAMIL, S., CHAN, C., OLIKER, L., SHALF, J., AND WILLIAMS, S. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS* (2010), IEEE, pp. 1–12.

[12] KAMIL, S. A. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, University of California, Berkeley, 2012.

[13] MOURA, J. M. F., SINGER, B., XIONG, J., JOHNSON, J., PADUA, D., VELOSO, M., AND JOHNSON, R. W. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. of High Perf. Comp. Appl. 18*, 1 (2004), 21–45.

[14] NATARAJAN, E. P., DEHNAVI, M. M., AND LEISERSON, C. E. Autotuning divide-and-conquer stencil computations. Unpublished manuscript, 2016.

[15] RISTOV, S., AND GUSE, M. Achieving maximum performance for matrix multiplication using set associative cache. *Proceedings of the IEEE* (2012), 542–547.

[16] RIVERA, G., AND TSENG, C. Tiling optimizations for 3D scientific computations. In *SC* (2000), ACM/IEEE, pp. 32:1–32:23.

[17] SONG, Y., AND LI, Z. New tiling techniques to improve cache temporal locality. In *PLDI* (1999), ACM, pp. 215–228.

[18] VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. OSKI: A library of automatically tuned sparse matrix kernels. In *J. of Phys.* (2005), vol. 16, p. 521.

[19] WHALEY, R. C., AND DONGARRA, J. Automatically tuned linear algebra software. In *SC* (1998), ACM, pp. 1–27.